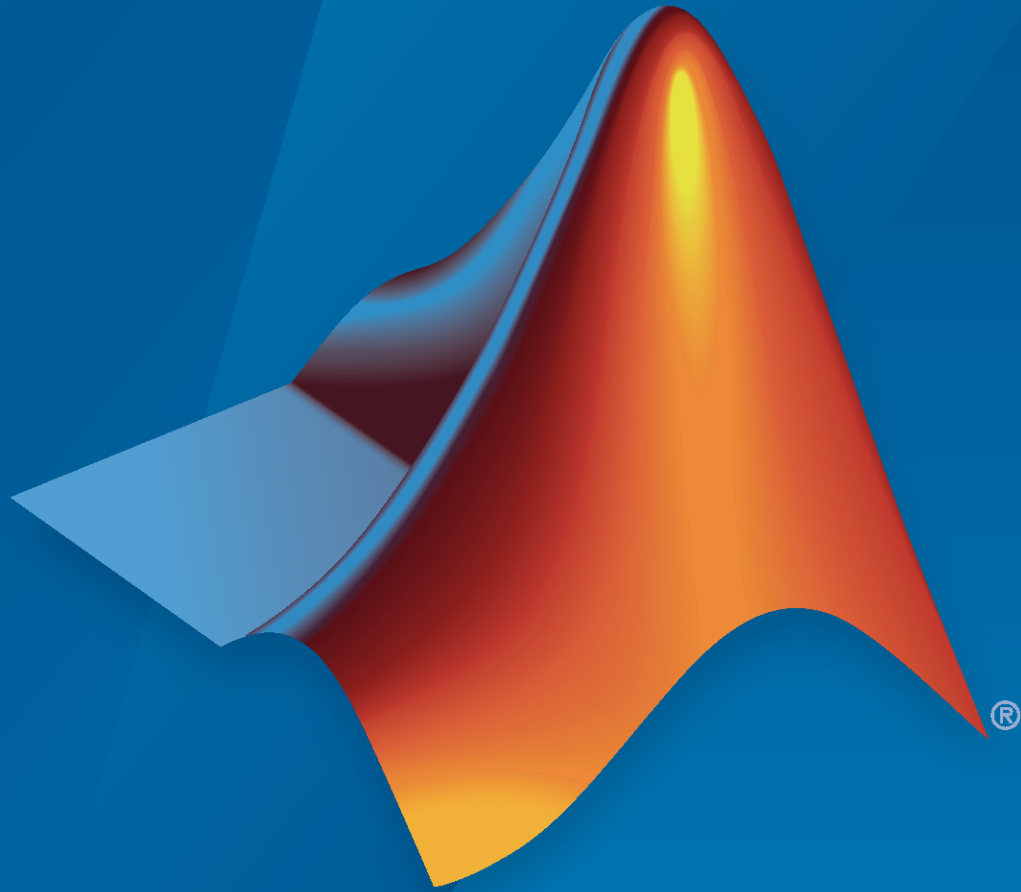


**DSP System Toolbox™**

Reference



**MATLAB® & SIMULINK®**

R2023a



## How to Contact MathWorks



Latest news: [www.mathworks.com](http://www.mathworks.com)  
Sales and services: [www.mathworks.com/sales\\_and\\_services](http://www.mathworks.com/sales_and_services)  
User community: [www.mathworks.com/matlabcentral](http://www.mathworks.com/matlabcentral)  
Technical support: [www.mathworks.com/support/contact\\_us](http://www.mathworks.com/support/contact_us)



Phone: 508-647-7000



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

### *DSP System Toolbox™ Reference*

© COPYRIGHT 2012–2023 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

### **Trademarks**

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

### **Patents**

MathWorks products are protected by one or more U.S. patents. Please see [www.mathworks.com/patents](http://www.mathworks.com/patents) for more information.



## Revision History

|                |             |                                   |
|----------------|-------------|-----------------------------------|
| April 2011     | Online only | Revised for Version 8.0 (R2011a)  |
| September 2011 | Online only | Revised for Version 8.1 (R2011b)  |
| March 2012     | Online only | Revised for Version 8.2 (R2012a)  |
| September 2012 | Online only | Revised for Version 8.3 (R2012b)  |
| March 2013     | Online only | Revised for Version 8.4 (R2013a)  |
| September 2013 | Online only | Revised for Version 8.5 (R2013b)  |
| March 2014     | Online only | Revised for Version 8.6 (R2014a)  |
| October 2014   | Online only | Revised for Version 8.7 (R2014b)  |
| March 2015     | Online only | Revised for Version 9.0 (R2015a)  |
| September 2015 | Online only | Revised for Version 9.1 (R2015b)  |
| March 2016     | Online only | Revised for Version 9.2 (R2016a)  |
| September 2016 | Online only | Revised for Version 9.3 (R2016b)  |
| March 2017     | Online only | Revised for Version 9.4 (R2017a)  |
| September 2017 | Online only | Revised for Version 9.5 (R2017b)  |
| March 2018     | Online only | Revised for Version 9.6 (R2018a)  |
| September 2018 | Online only | Revised for Version 9.7 (R2018b)  |
| March 2019     | Online only | Revised for Version 9.8 (R2019a)  |
| September 2019 | Online only | Revised for Version 9.9 (R2019b)  |
| March 2020     | Online only | Revised for Version 9.10 (R2020a) |
| September 2020 | Online only | Revised for Version 9.11 (R2020b) |
| March 2021     | Online only | Revised for Version 9.12 (R2021a) |
| September 2021 | Online only | Revised for Version 9.13 (R2021b) |
| March 2022     | Online only | Revised for Version 9.14 (R2022a) |
| September 2022 | Online only | Revised for Version 9.15 (R2022b) |
| March 2023     | Online only | Revised for Version 9.16 (R2023a) |



|          |  |
|----------|--|
| <b>1</b> | <b>Apps</b>  |
| <b>2</b> | <b>Blocks</b>  |
| <b>3</b> | <b>Analysis Methods for Filter System Objects</b>              |
|          | Analysis Methods for Filter System Objects . . . . . 3-2       |
| <b>4</b> | <b>System Objects</b>  |
| <b>5</b> | <b>Functions</b>   |
| <b>6</b> | <b>Reference for the Properties of Filter Objects</b>          |
|          | Multirate Filter Properties . . . . . 6-2                      |
|          | Compatibility . . . . . 6-2                                    |
|          | Property Summaries . . . . . 6-2                               |
|          | Property Details for Multirate Filter Properties . . . . . 6-5 |
|          | References for Multirate Filters . . . . . 6-12                |



# Apps

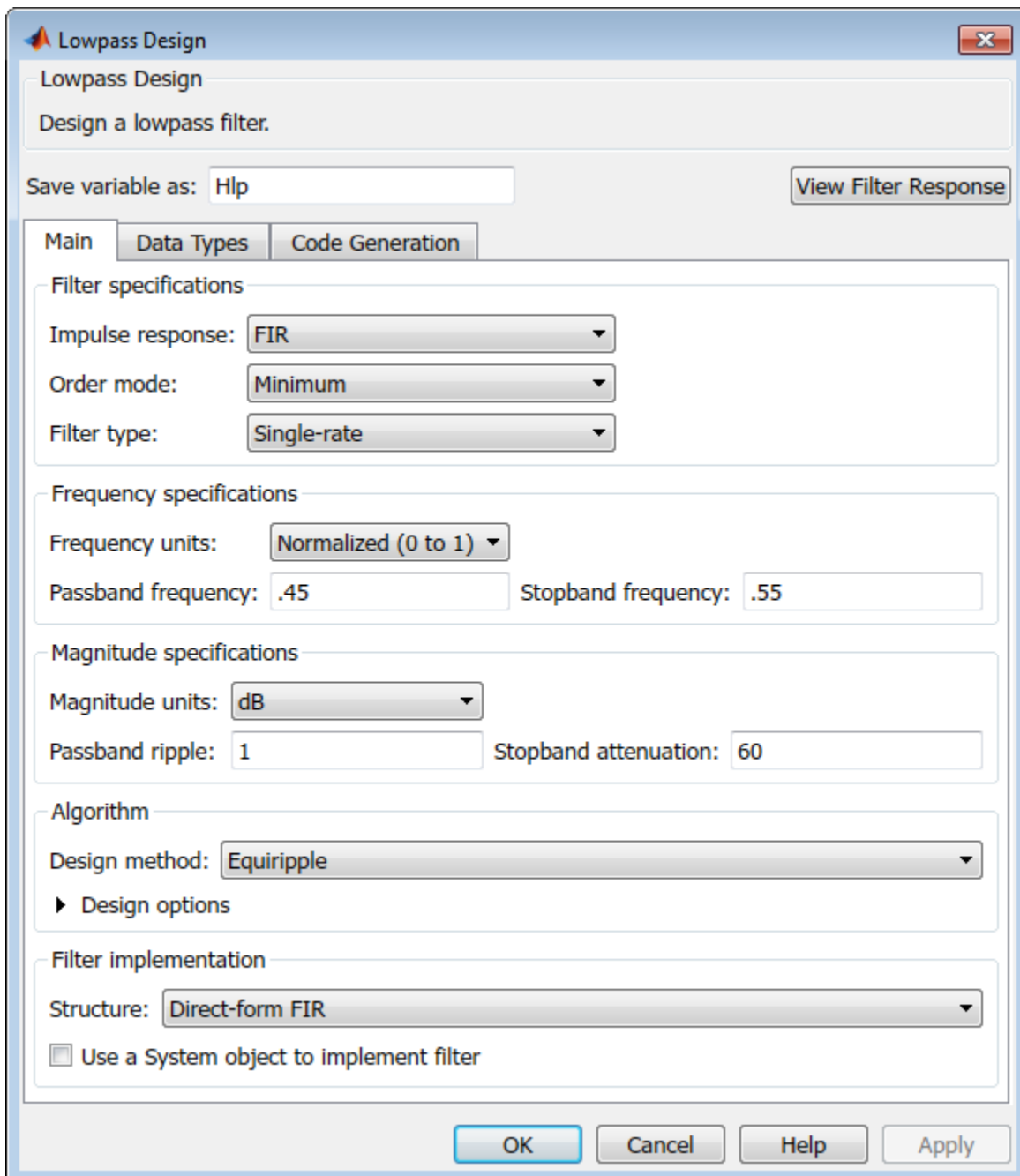
---

## Filter Builder


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

### Description

The **Filter Builder** app provides a graphical user interface to design filters using the `fdesign` object. The first step is to choose the filter response. Based on the response you choose, the algorithm, constraints, and the design parameter settings appear on the **Main** tab of the user interface. You can further specify the precision and data types in the **Data Types** tab. The **Code Generation** tab contains options for various implementations of the completed filter design. Once you specify all the filter parameters and the design algorithm, you can visualize the filter response by clicking on the **View Filter Response** button. When you click on this button, **FVTool** opens to display the magnitude response of the filter. When you have achieved the desired filter response through iterations of design and analysis, click **OK**. When you click **OK**, the app exports the filter object to the base workspace. If you select the **Use a System object to implement filter** check box in the user interface, the app exports a filter System object™. For more information on the design process using the **Filter Builder** app, see “Filter Builder Design Process”. For details on each of the response methods and settings of all the associated parameters, see `filterBuilder`.



## Open the Filter Builder App

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

- MATLAB command prompt: Enter `filterBuilder(obj)`, where `obj` is an existing filter object. For example, if `obj` is a bandpass filter, `filterBuilder(obj)` opens the bandpass filter design dialog. The `obj` object must have been created using `filterBuilder` or using `fdesign`.
- MATLAB command prompt: Enter `filterBuilder('response')` to replace *response* with a supported response method. MATLAB opens a filter design dialog that corresponds to the specified response. For details on the supported response methods, see `filterBuilder`.

## Examples

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

## Version History

Introduced before R2006a

## See Also

### Apps

[Window Designer](#) | [Filter Designer](#)

### Functions

[filterBuilder](#) | [designfilt](#)

### Tools

[FVTool](#) | [WVTool](#)

### Topics

[“Filter Builder Design Process”](#)

[“Lowpass FIR Filter Design”](#)



# Logic Analyzer

Visualize, measure, and analyze transitions and states over time

## Description

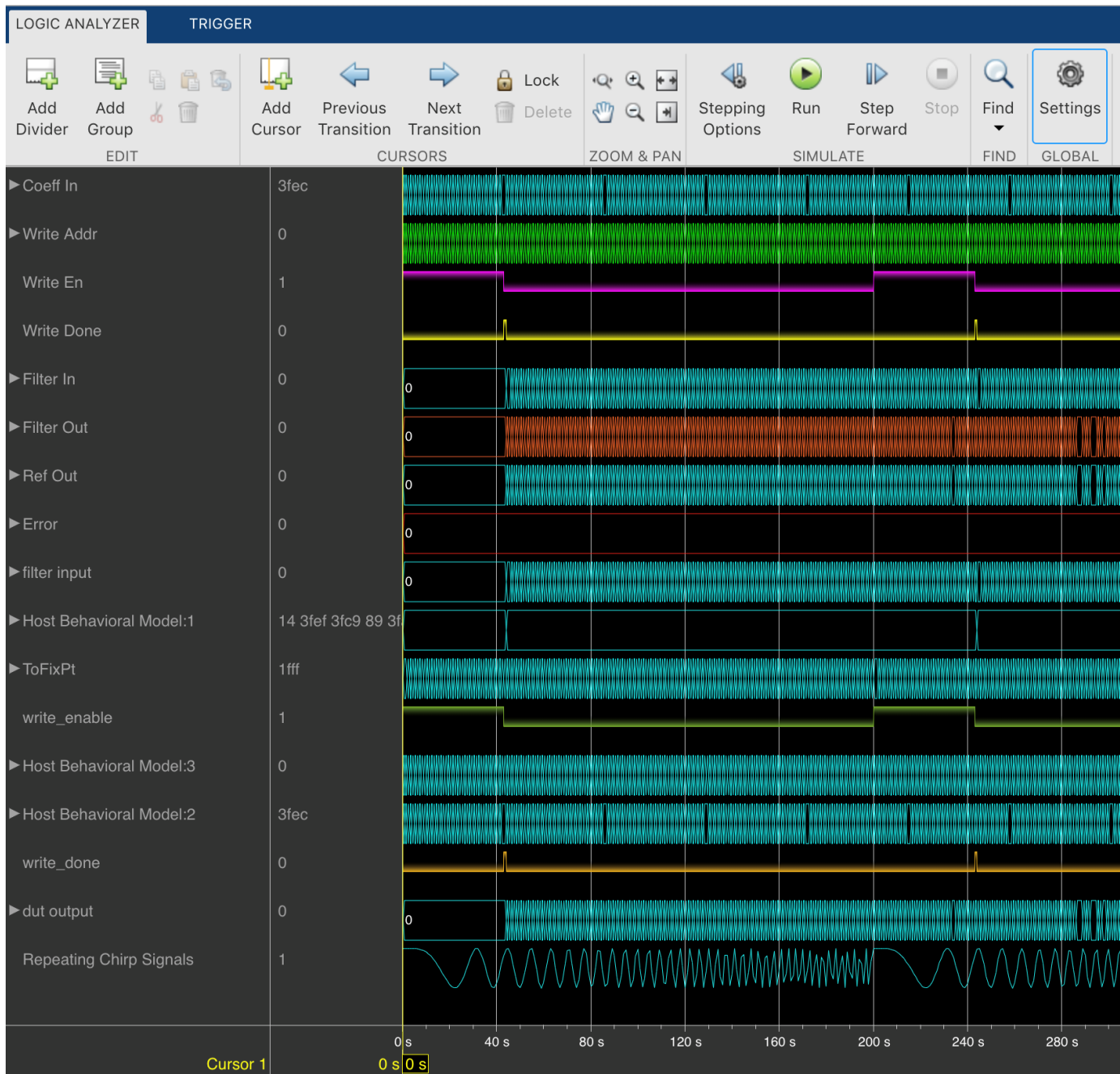
The **Logic Analyzer** is a tool for visualizing and inspecting signals and states in your Simulink® model. Using the **Logic Analyzer**, you can:

- Debug and analyze models
- Trace and correlate many signals simultaneously
- Detect and analyze timing violations
- Trace system execution
- Detect signal changes using triggers

**For keyboard shortcuts, click [More](#).**

### Keyboard Shortcuts

| Actions                      | Description                   | Applicable When                         |
|------------------------------|-------------------------------|---|
| <b>Ctrl+X</b>                | Cut                           | Wave is selected                        |
| <b>Ctrl+C</b>                | Copy                          | Wave is selected                        |
| <b>Ctrl+V</b>                | Paste                         | Wave is selected                        |
| <b>Delete</b>                | Delete                        | Wave is selected                        |
| <b>Ctrl+-</b>                | Zoom out                      | Always                                  |
| <b>Shift+Ctrl+-</b>          | Zoom out around active cursor | Always                                  |
| <b>Ctrl++</b>                | Zoom in                       | Always                                  |
| <b>Shift+Ctrl++</b>          | Zoom in around active cursor  | Always                                  |
| <b>Shift+Ctrl+C</b>          | Move display to active cursor | When cursor is not in the display range |
| <b>Space</b>                 | Zoom out full                 | Always                                  |
| <b>Tab, Right Arrow</b>      | Next transition               | Digital format wave is selected         |
| <b>Shift+Tab, Left Arrow</b> | Previous transition           | Digital format wave is selected         |
| <b>Ctrl+A</b>                | Select all waves              | Always                                  |
| <b>Up Arrow</b>              | Select wave above selected    | Wave is selected                        |
| <b>Down Arrow</b>            | Select wave below selection   | Wave is selected                        |
| <b>Ctrl+Up Arrow</b>         | Move selected waves up        | Wave is selected                        |
| <b>Ctrl+Down Arrow</b>       | Move selected waves down      | Wave is selected                        |
| <b>Escape</b>                | Unselect all signals          | Wave is selected                        |
| <b>Page Up</b>               | Scroll up                     | Always                                  |
| <b>Page Down</b>             | Scroll down                   | Always                                  |



## Open the Logic Analyzer App

On the Simulink toolstrip Simulation tab, click the **Logic Analyzer** app button. If the button is not displayed, expand the review results app gallery. Your most recent choice for data visualization is saved across Simulink sessions.

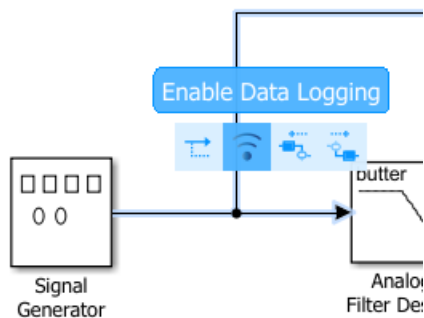
To visualize referenced models, you must open the Logic Analyzer from the referenced model. You should see the name of the referenced model in the Logic Analyzer toolbar.

## Examples

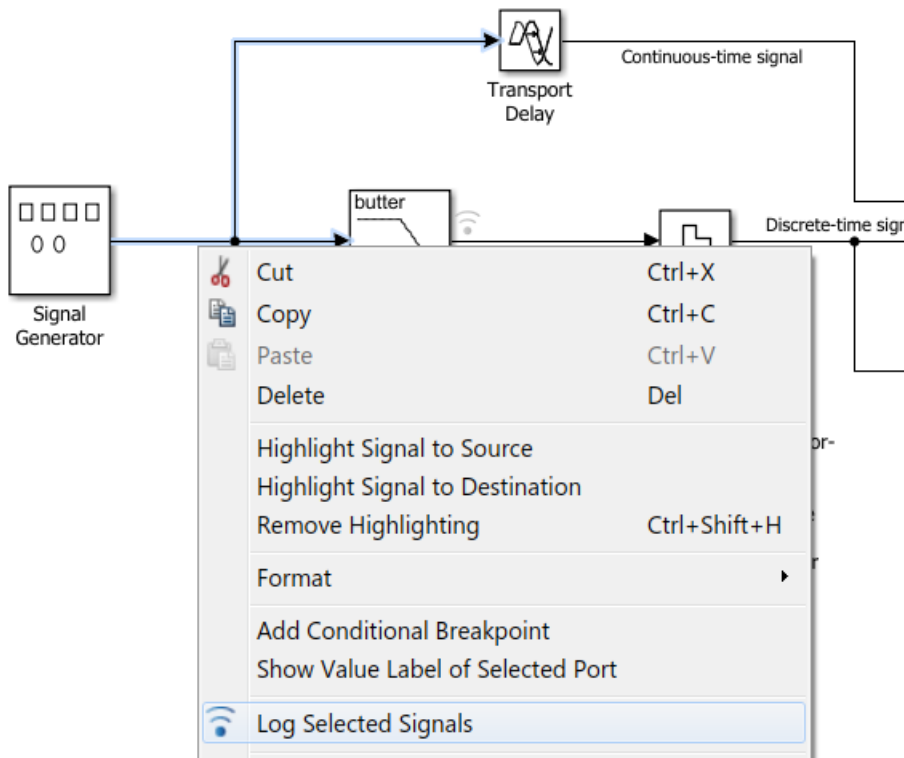
### Select Signals to Analyze

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

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



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



- Use any method to select multiple signal lines in your model. For example, use **Shift**+click to select multiple lines individually or **CTRL+A** to select all lines at once. Then, on the **Signal** tab, select the **Log Signals** button.



To visualize data in the Logic Analyzer, you must enable signal logging for the model. (Logging is on by default.) To enable signal logging, open **Model Settings** from the toolstrip, navigate to the **Data Import/Export** pane, and select **Signal logging**.

When you open the **Logic Analyzer**, all signals marked for logging are listed. You can add and delete waves from your **Logic Analyzer** while it is open. Adding and deleting signals does not disable logging, only removes the signal from the Logic Analyzer.

### Modify Individual Wave Settings

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

The screenshot shows the Logic Analyzer interface with the **WAVE** tab selected. The **PROPERTIES** section includes:

- Name: Discrete-time signal
- Height: 12
- Radix: (dropdown menu)
- Font Size: 10
- Format: Digital
- Wave Color: (color selector)

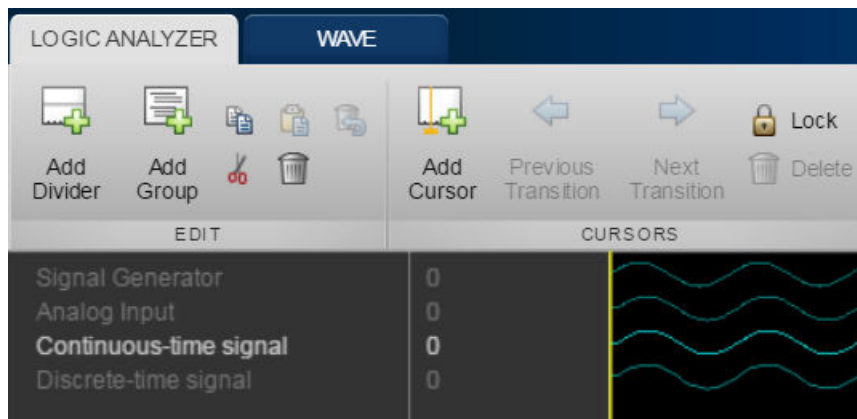
The **ACTIONS** section includes:

- Move: (up/down arrows)
- Reset: (refresh icon)
- Delete: (trash icon)

The main display area shows a signal waveform with a cursor. The signal is a discrete-time signal with values 0, 1, 0, 1, 0, 1, 0, 1, 0, 1. The cursor is positioned at 0 us. The time axis is marked with 7340800 us, 7341000 us, and 7341200 us.

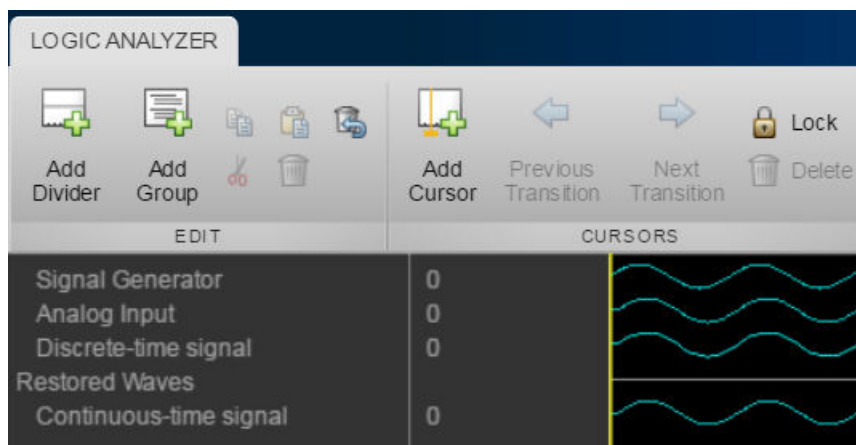
### Delete and Restore Waves

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



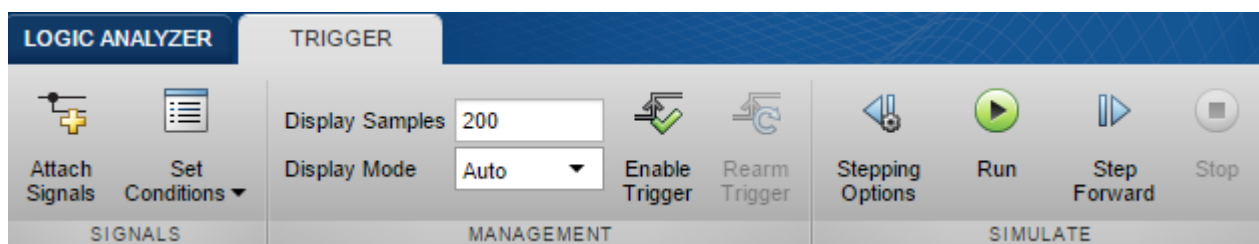
- 2 From the **Logic Analyzer** toolbar, click . The wave is removed from the **Logic Analyzer**.
- 3 To restore the wave, from the **Logic Analyzer** toolbar, click .

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



## Add Trigger

- 1 Open the **Logic Analyzer** and select the **Trigger** tab.



- 2 To attach a signal to the trigger, select **Attach Signals**, then select the signal you want to trigger on. You can attach up to 20 signals to the trigger. Each signal can have only one triggering condition.
- 3 By default, the trigger looks for rising edges in the attached signals. You can set the trigger to look for rising or falling edges, bit sequences, or a comparison value. To change the triggering conditions, select **Set Conditions**.

If you add multiple signals to the trigger, control the trigger logic using the **Operator** option:

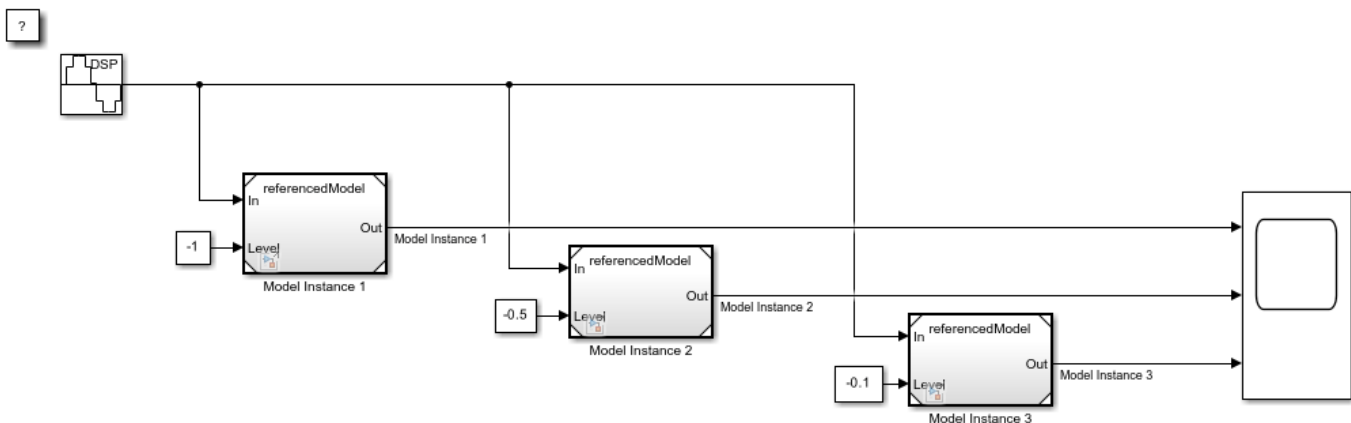
- AND - match all conditions.
  - OR - match any condition.
- 4 To control how many samples you see before triggering, set the **Display Samples** option. For example, if you set this option to 500, the **Logic Analyzer** tries to give you 500 samples before the trigger. Depending on the simulation, the **Logic Analyzer** may show more or fewer than 500 samples before the trigger. However, if the trigger is found before the 500th sample, the Logic Analyzer still shows the trigger.
  - 5 Control the trigger mode using **Display Mode**.
    - Once - The **Logic Analyzer** marks only the first location matching the trigger conditions and stops showing updates to the Logic Analyzer. If you want to reset the trigger, select **Rearm Trigger**. Relative to the current simulation time, the **Logic Analyzer** shows the next matching trigger event.
    - Auto - The **Logic Analyzer** marks every location matching the trigger conditions.
  - 6 Before running the simulation, select **Enable Trigger**. A blue cursor appears as time 0. Then, run the simulation. When a trigger is found, the **Logic Analyzer** marks the location with a locked blue cursor.

### Choose Visible Instance of Multi-Reference Model Block

The **Logic Analyzer** can stream only a single instance of a multi-instance Model block. If the same model is opened across different windows, those models will share the same Logic Analyzer. This example shows how to select an instance of a multi-instance Model block for logging on the **Logic Analyzer**.

Open the `multipleModelInstances` model.

```
open_system('multipleModelInstances')
```

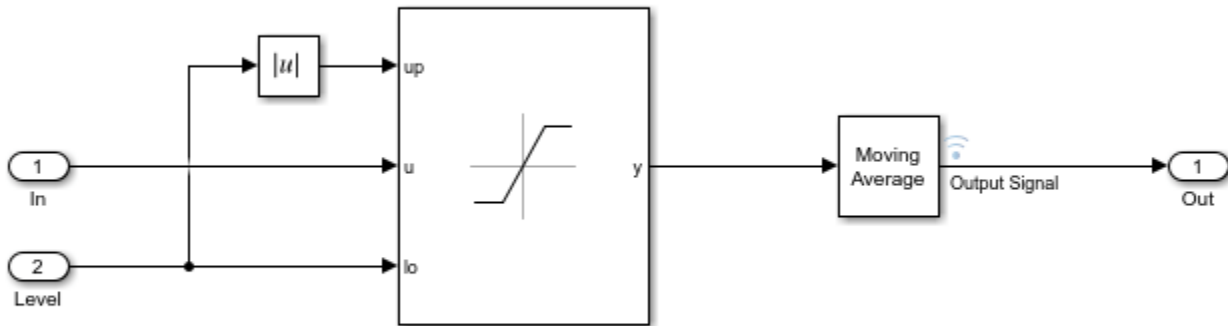


Copyright 2016 The MathWorks, Inc.

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

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

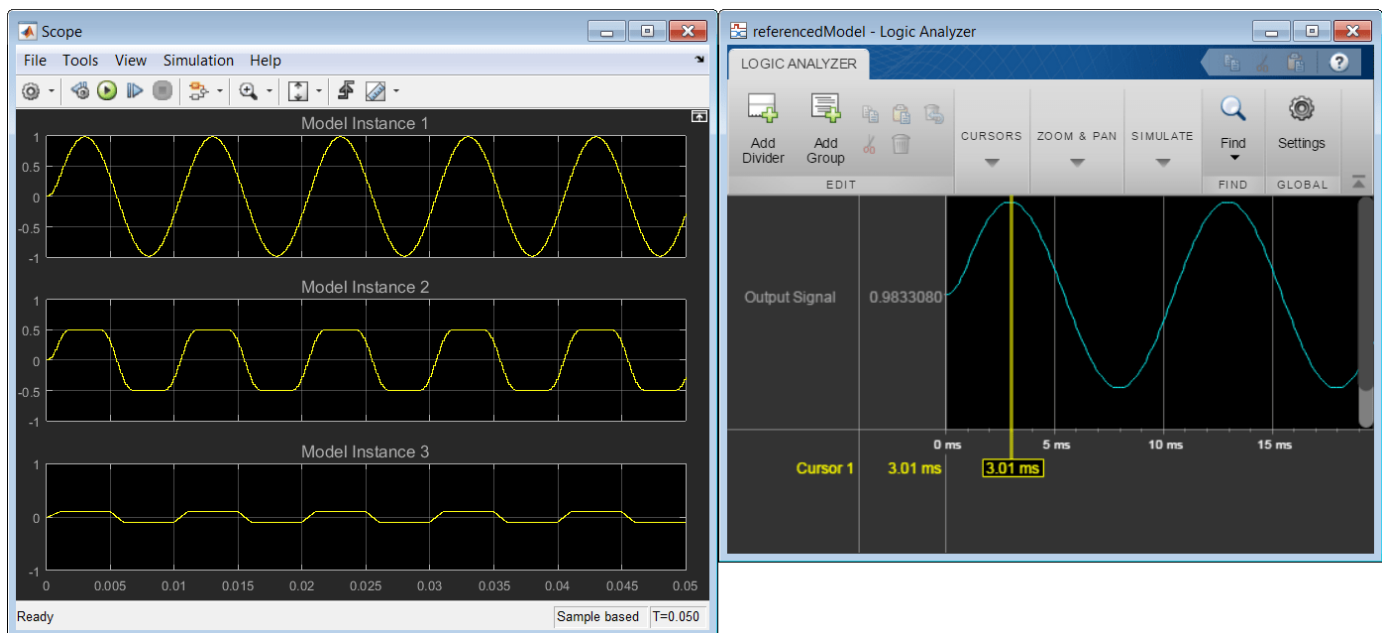
```
open_system('referencedModel')
```



Copyright 2016 The MathWorks, Inc.

Open the Logic Analyzer in the referenced model by double-clicking the logging symbol next to the MovingAverage block. You should see `referencedModel - [multipleModelInstances]` in the toolbar of the Logic Analyzer.

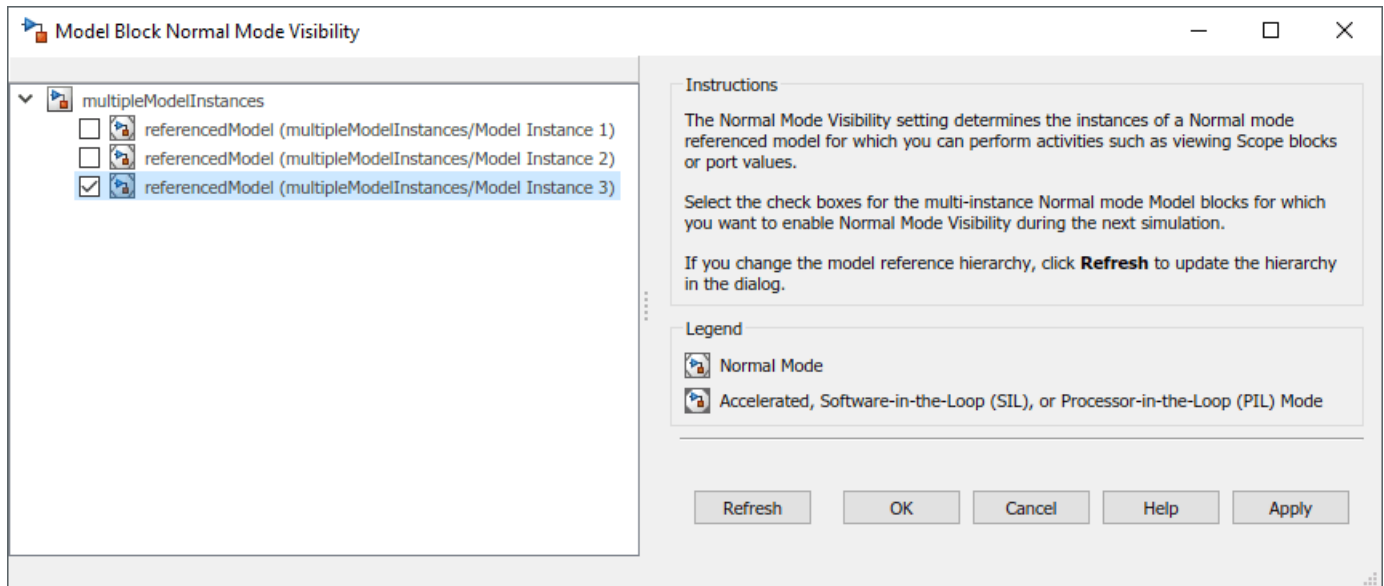
From the Logic Analyzer window, run the model. By running the simulation from a referenced version of `referencedModel`, Simulink runs the top model (`multipleModelInstances`) and referenced models (`referencedModel`). The **Logic Analyzer** displays a single instance of a multi-instance Model block.



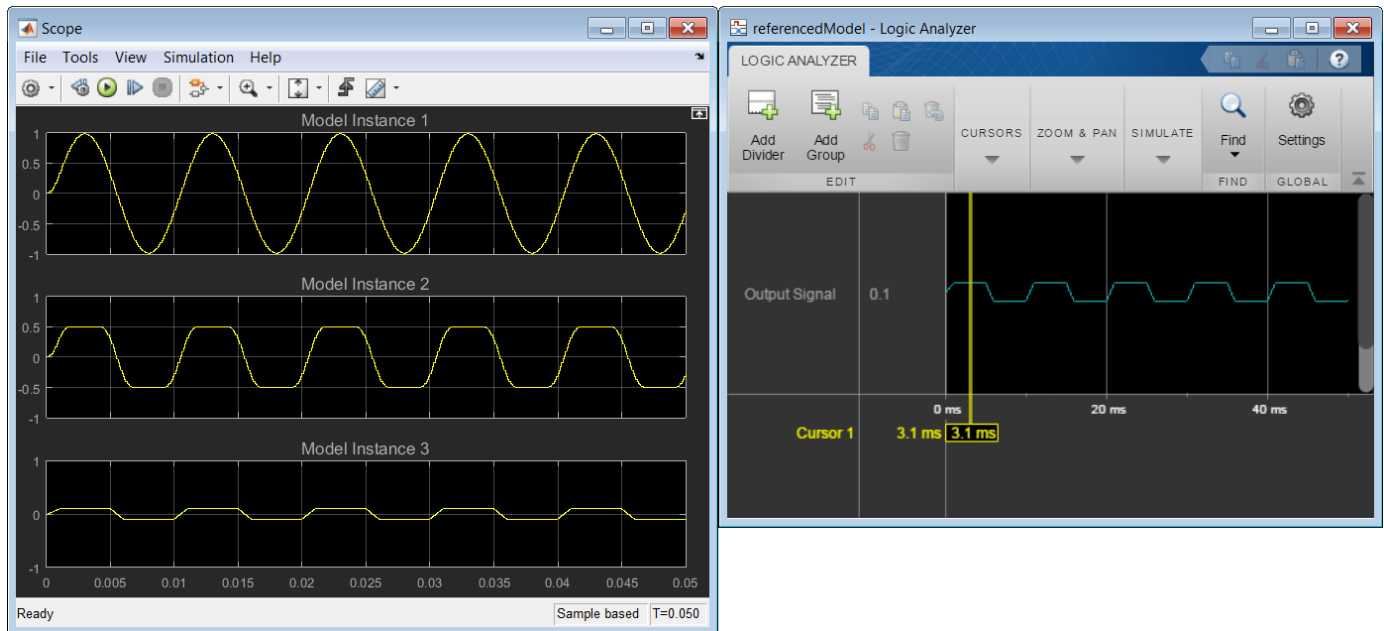
When you run a simulation, the logic analyzer runs the model listed in the Logic Analyzer toolbar. If this model is a referenced model, the toolbar also lists the top model and you will see results from

running the top model. To view results from the referenced model in isolation, you must open the referenced model as a top model.

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



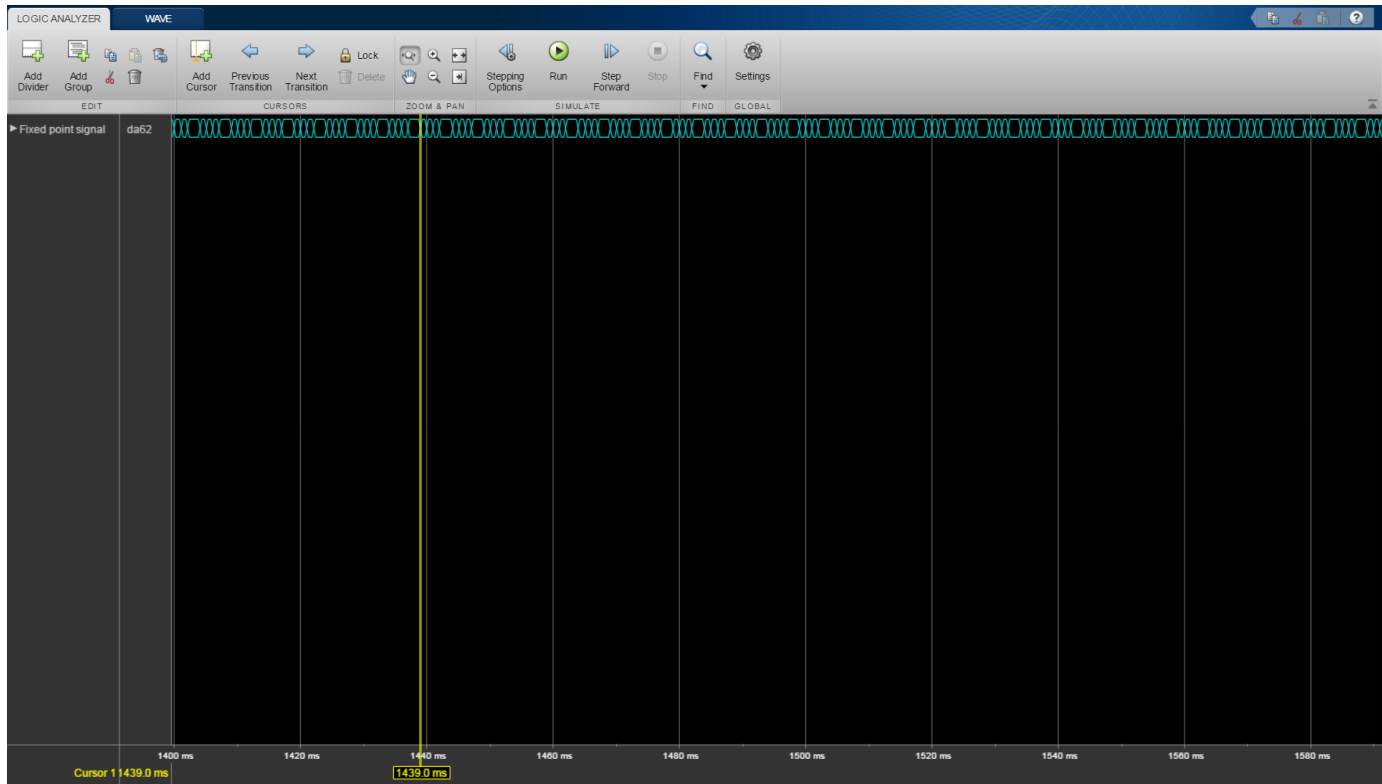
Run the `multipleModelInstances` model again. The **Logic Analyzer** displays `Model Instance 3` data.





## View Bit-Expanded Wave and Reverse Display Order of Bits

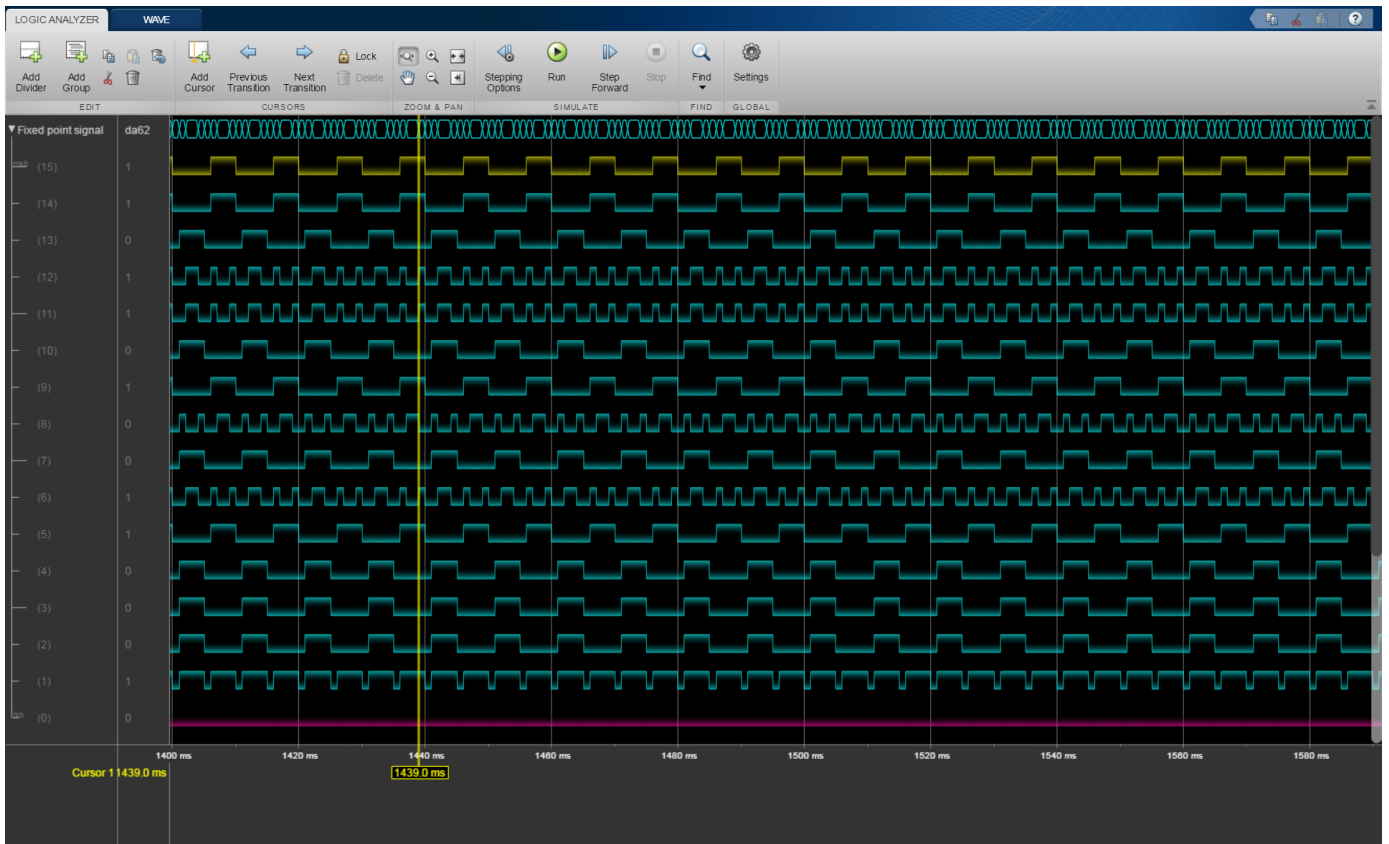
The **Logic Analyzer** enables you to bit-expand fixed-point and integer waves.



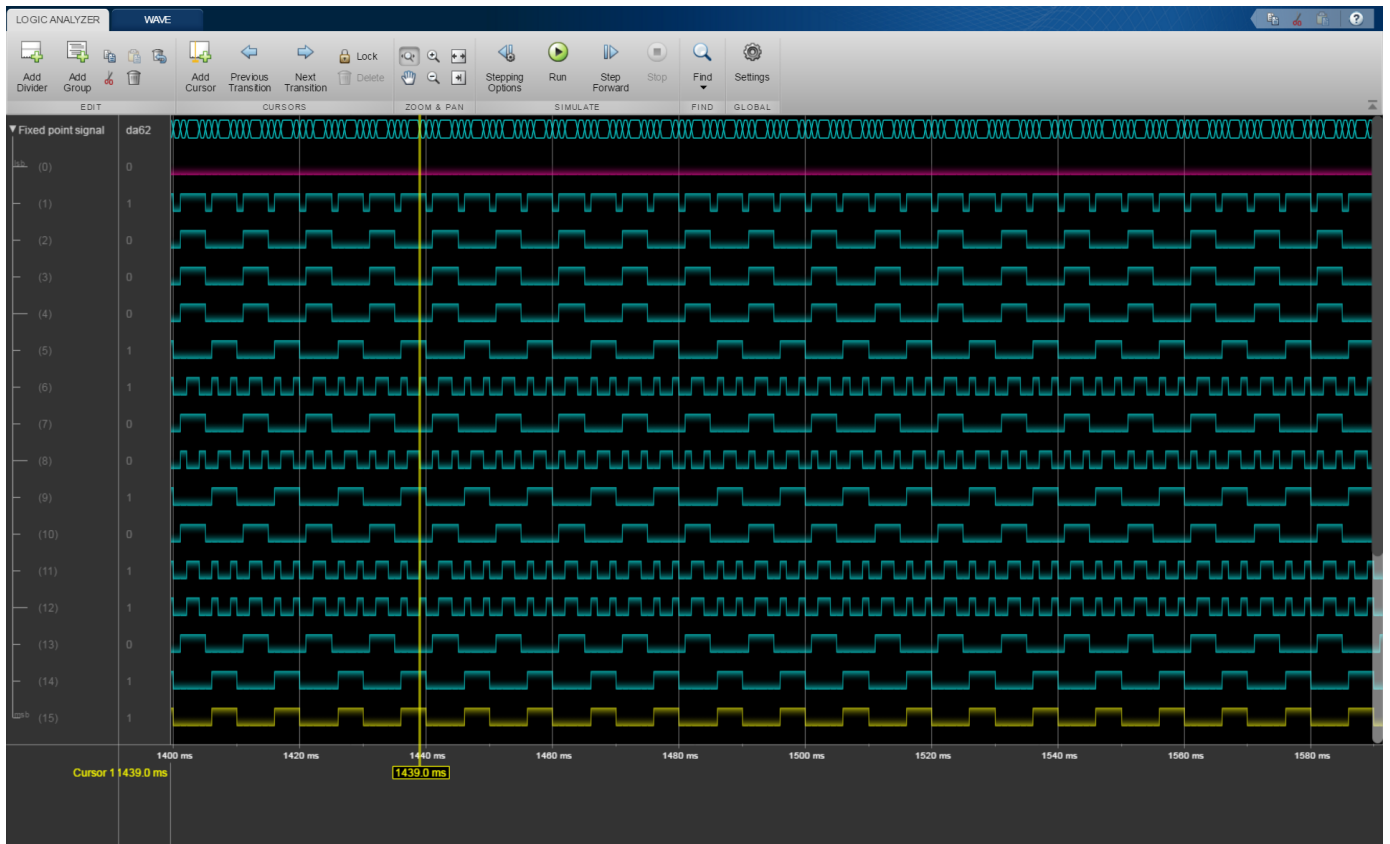
- 1 In the **Logic Analyzer**, click the arrow next to a fixed-point or integer wave to view the bits.

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

# 1 Apps

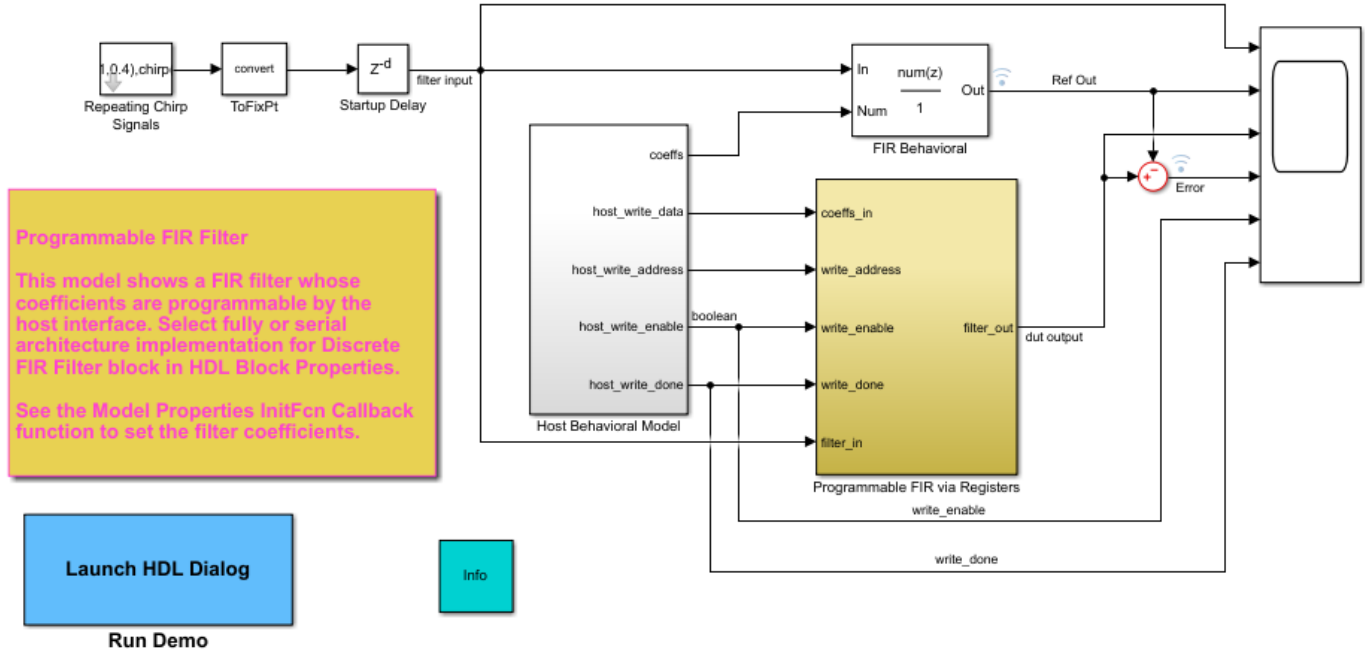


- 2 Click Settings, and then select **Display Least Significant bit first** to reverse the order of the displayed bits.



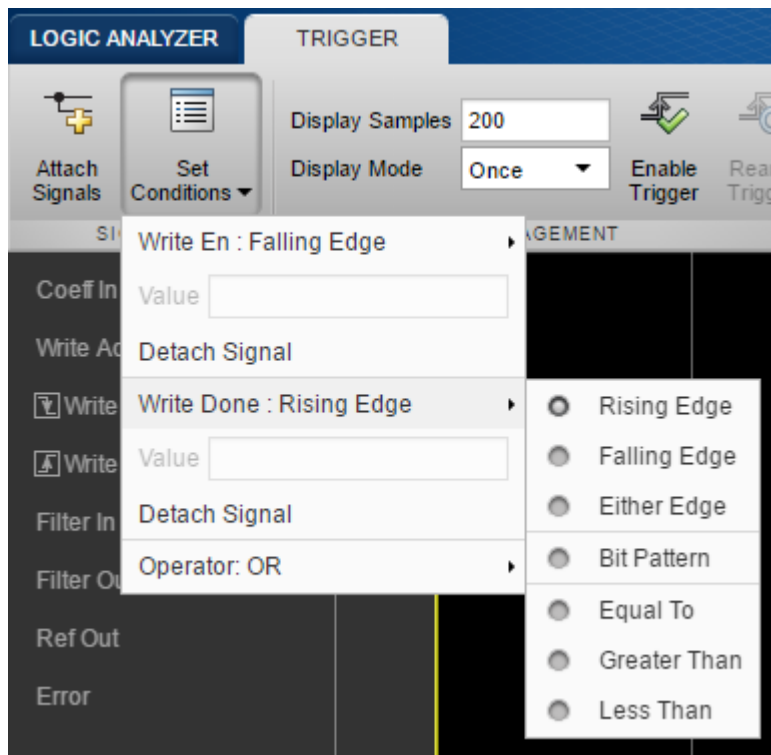
### Add Triggers to Verify Write Operation

- 1 Open the Programmable FIR Filter model (dspprogfirhdl).



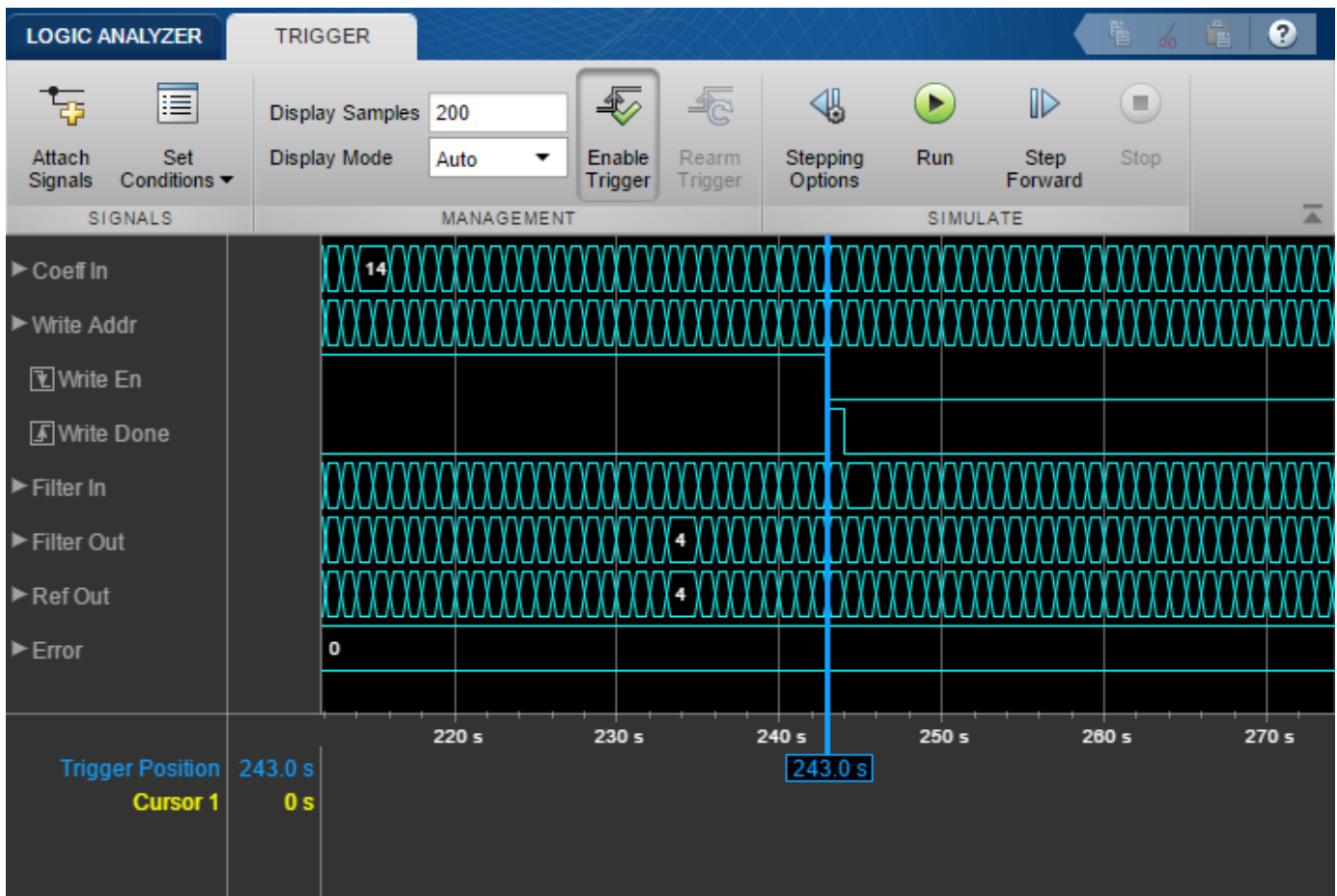
- 2 Open the **Logic Analyzer** and select the **Trigger** tab.
- 3 To add a trigger, in the toolstrip, select **Attach Signals** and attach the write enable `Write En` signal. An icon appears in front of the signal name to indicate it is attached to a trigger. The icon changes depending on the type of trigger.
- 4 Select **Set Conditions** and change the trigger condition for the `Write En` signal to **Falling Edge**. The trigger will show when the write enable signal was sent.
- 5 Attach the `Write Done` signal to the trigger. Keep the trigger condition for this signal as the default, **Rising Edge**. Now, the trigger will also show when the write was completed.

If you open the **Set Conditions** drop down, you see an **Operator** field. This field appears when multiple signals are attached to the trigger. Change the operator to **OR** so that the trigger will show instances where a write was started or completed.



- 6 Set the **Display Mode** to Auto. With this setting, the **Logic Analyzer** marks all locations where the trigger conditions are met.
- 7 Select **Enable Trigger** and run the simulation.

Each time the trigger conditions are met, the **Logic Analyzer** marks the time with a locked blue cursor. At each marked location, `Write En` is 0 and `Write Done` is 1. If you examine each location marked by a trigger, you can verify that each time a write is sent, it is also completed.



- “Inspect and Measure Transitions Using the Logic Analyzer”
- “Visualizing Multiple Signals Using Logic Analyzer”
- “Partly Serial Systolic FIR Filter Implementation” (DSP HDL Toolbox)
- “Fully Parallel Systolic FIR Filter Implementation” (DSP HDL Toolbox)
- “Configure Logic Analyzer”
- “Programmable FIR Filter for FPGA” (HDL Coder)
- “Log Simulation Output for States and Data” (Stateflow)
- “View Stateflow States in the Logic Analyzer” (Stateflow)

## Limitations

### Logging Settings

- If you enable the configuration parameter **Log Dataset data to file**, you cannot stream logged data to the **Logic Analyzer**.
- Signals marked for logging using `Simulink.sdi.markSignalForStreaming` or visualized with a Dashboard Scope do not appear on the **Logic Analyzer**.
- You cannot visualize Data Store Memory block signals in the **Logic Analyzer** if you set the **Log data store data** parameter to on.

## Input Signal Limitations

- Signals marked for logging for the **Logic Analyzer** must have fewer than 8000 samples per simulation step.
- The **Logic Analyzer** does not support frame-based processing.
- For 64-bit integers and fixed-point numbers greater than 53 bits, if the numbers are greater than the maximum value of double precision, the transitions between numbers might not display correctly.
- You may see performance degradation in the **Logic Analyzer** for large matrices (greater than 500 elements) and buses with more than 1000 signals.
- The **Logic Analyzer** does not support Stateflow data output.

## Graphical Settings

- While the simulation is running, you cannot zoom, pan, or modify the trigger.
- To visualize constant signals, in the settings, you must set the **Format** to **Digital**. Constants marked for logging are visualized as a continuous transition.

## Supported Simulation Modes

| Mode                        | Supported | Notes and Limitations   |
|-----------------------------|-----------|---|
| Normal                      | Yes       |   |
| Accelerator                 | Yes       | You cannot use the <b>Logic Analyzer</b> to visualize signals in Model blocks with <b>Simulation mode</b> set to Accelerator.   |
| Rapid Accelerator           | Yes       | Data is not available in the <b>Logic Analyzer</b> during simulation.<br><br>If you simulate a model with the simulation mode set to rapid accelerator, after simulation the following signals cannot be visualized in the <b>Logic Analyzer</b> : <ul style="list-style-type: none"> <li>• Multi-instance model reference signals</li> <li>• Nonvirtual bus signals</li> </ul> |
| Processor-in-the-loop (PIL) | No        |   |
| Software-in-the-loop (SIL)  | No        |   |
| External                    | No        |   |

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

## Version History

Introduced in R2016b

## See Also

### Objects

`dsp.LogicAnalyzer`

**Topics**

“Inspect and Measure Transitions Using the Logic Analyzer”

“Visualizing Multiple Signals Using Logic Analyzer”

“Partly Serial Systolic FIR Filter Implementation” (DSP HDL Toolbox)

“Fully Parallel Systolic FIR Filter Implementation” (DSP HDL Toolbox)

“Configure Logic Analyzer”

“Programmable FIR Filter for FPGA” (HDL Coder)

“Log Simulation Output for States and Data” (Stateflow)

“View Stateflow States in the Logic Analyzer” (Stateflow)



# Blocks

---

# Allpass Filter

Single-section or multiple-section allpass filter



## Libraries:

DSP System Toolbox / Filtering / Filter Implementations

## Description

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

In minimum multiplier form, the block uses the minimum number of required multipliers,  $n$ , with  $2n$  delay units and  $2n$  adders. In wave digital filter form, the block uses only  $n$  multipliers and  $n$  delay units, at the expense of  $3n$  adders. The lattice structure uses  $2n$  multipliers,  $n$  delay units, and  $2n$  adders. For more details on these structures, see “Algorithms” on page 2-7.

## Input/Output Ports

### Input

**x** — Input data

column vector | row vector | matrix

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

This port is unnamed until you set **Internal allpass structure** to Minimum multiplier or Lattice, and select the **Specify coefficients from input port** parameter.

Data Types: single | double

**coeffs** — Allpass filter coefficients

column vector | row vector | matrix

This port inputs the coefficients of the allpass filter. When you set **Internal allpass structure** to Minimum multiplier, the **coeffs** port accepts matrices of size  $N$ -by-1 or  $N$ -by-2. When you set **Internal allpass structure** to Lattice, the **coeffs** port accepts an  $N$ -by-1 column vector or an 1-by- $N$  row vector.

### Dependencies

This port appears when you set **Internal allpass structure** to Minimum multiplier or Lattice, and select the **Specify coefficients from input port** parameter.

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

## Output

**Port\_1** — Output of the allpass filter  
column vector | row vector | matrix

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

Data Types: single | double

## Parameters

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

**Internal allpass structure** — Filter structure

Minimum multiplier (default) | Wave Digital Filter | Lattice

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

For more details on these structures, see “Algorithms” on page 2-7.

**Specify coefficients from input port** — Flag to specify allpass polynomial coefficients

off (default) | on

When you select this check box and set **Internal allpass structure** to **Minimum multiplier**, the allpass polynomial coefficients are input through the **coeffs** port. When you clear this check box, the allpass polynomial coefficients are specified on the block dialog through the **Allpass polynomial coefficients** parameter.

When you select this check box and set **Internal allpass structure** to **Lattice**, the lattice allpass coefficients are input through the **coeffs** port. When you clear this check box, the lattice allpass coefficients are specified on the block dialog through the **Lattice allpass coefficients** parameter.

## Dependencies

This parameter applies when you set **Internal allpass structure** to **Minimum multiplier** or **Lattice**.

**Allpass polynomial coefficients** — Coefficients in minimum multiplier form

$[-2^{(-1/2)}, 1/2]$  (default) |  $N$ -by-1 matrix |  $N$ -by-2 matrix

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

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

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

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

Tunable: Yes

#### Dependencies

To enable this parameter, set **Internal allpass structure** to Minimum multiplier and clear the **Specify coefficients from input port** parameter.

**Wave Digital Filter allpass coefficients** — Coefficients in wave digital filter form  
 $[1/2, -2^{(1/2)}/3]$  (default) |  $N$ -by-1 matrix |  $N$ -by-2 matrix

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

Tunable: Yes

#### Dependencies

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

**Indicate if last section is first order** — Is last section first order  
 off (default) | on

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

#### Dependencies

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

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

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

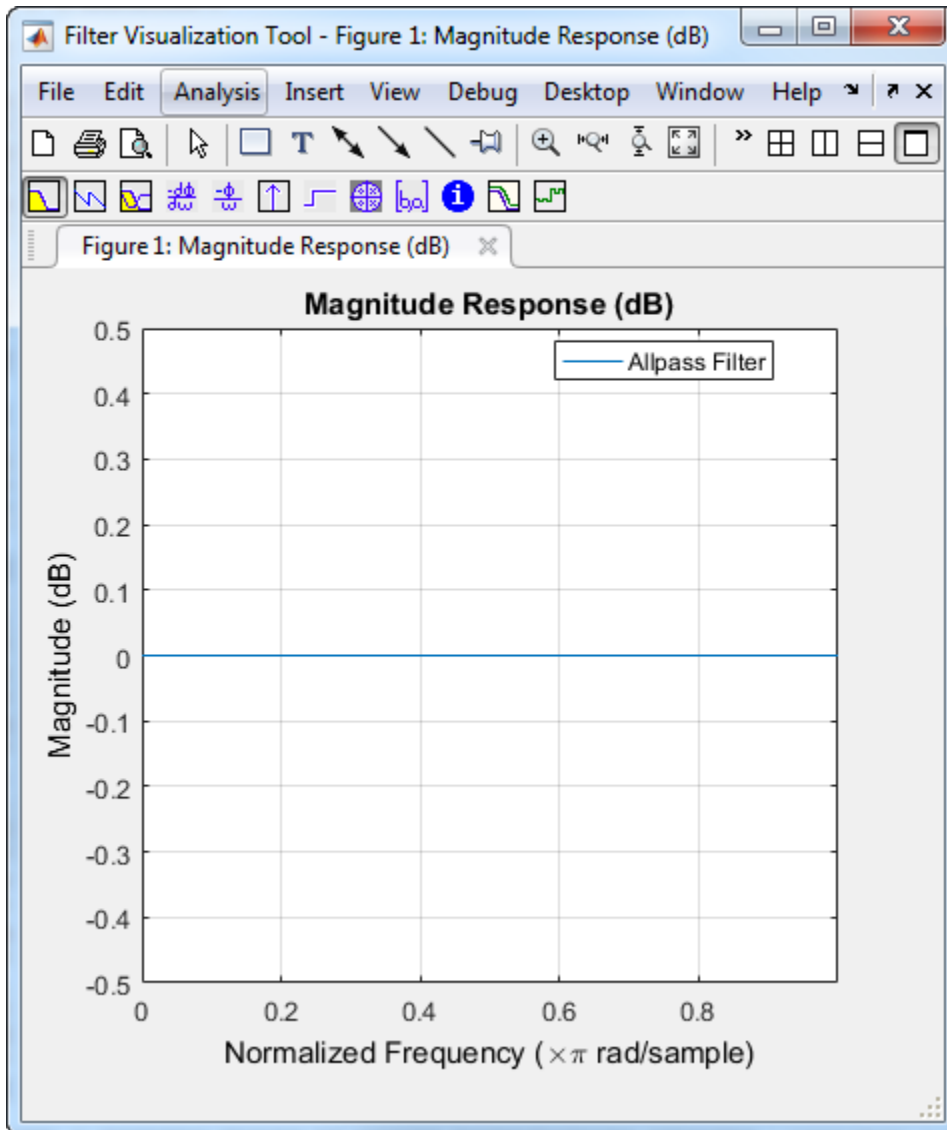
Tunable: Yes

#### Dependencies

To enable this parameter, set **Internal allpass structure** to Lattice and clear the **Specify coefficients from input port** parameter.

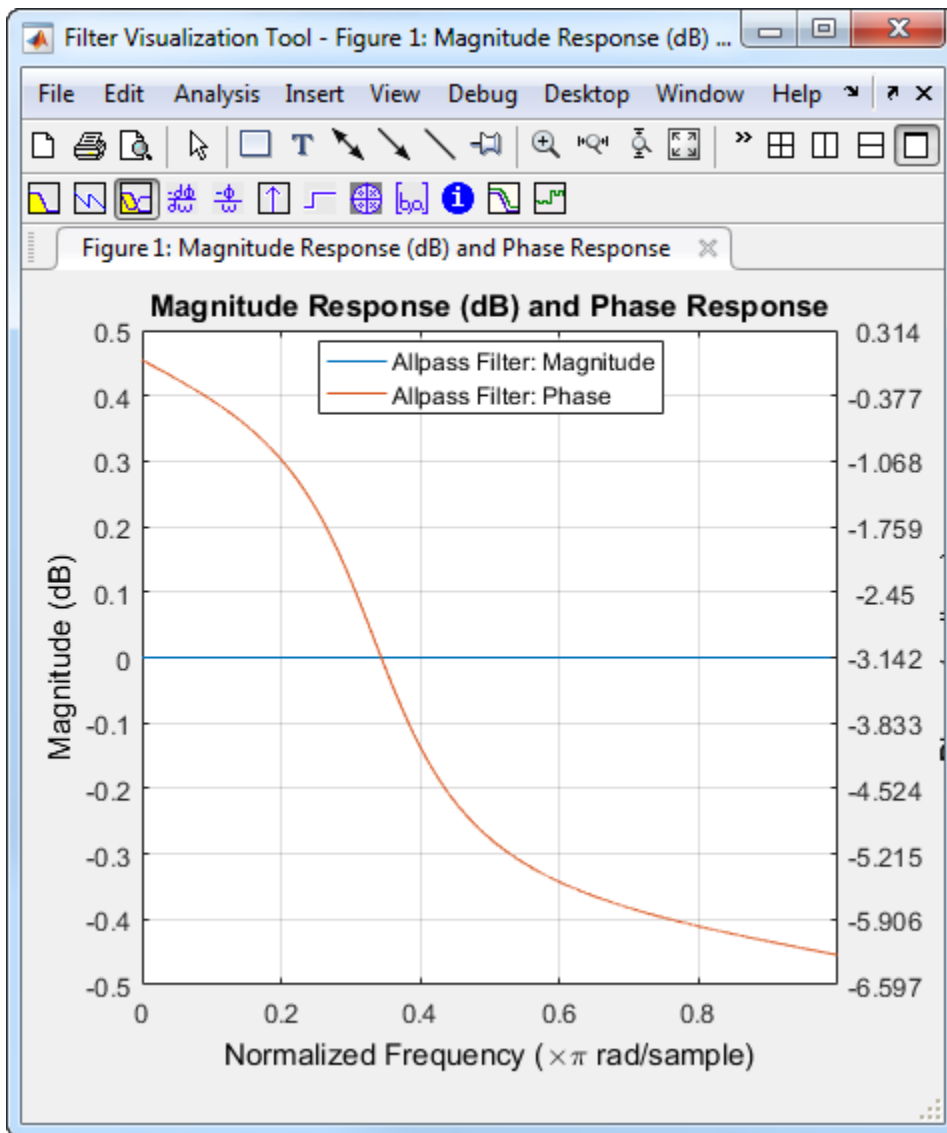
**View Filter Response** — Visualize filter response  
button

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



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

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



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

**Simulate using** — Type of simulation to run

Code generation (default) | Interpreted execution

Specify the type of simulation to run as one of the following:

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

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

## Block Characteristics

|                                 |                 |
|---------------------------------|-----------------|
| <b>Data Types</b>               | double   single |
| <b>Multidimensional Signals</b> | No              |
| <b>Variable-Size Signals</b>    | Yes             |

## Algorithms

The transfer function of an allpass filter is given by

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

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

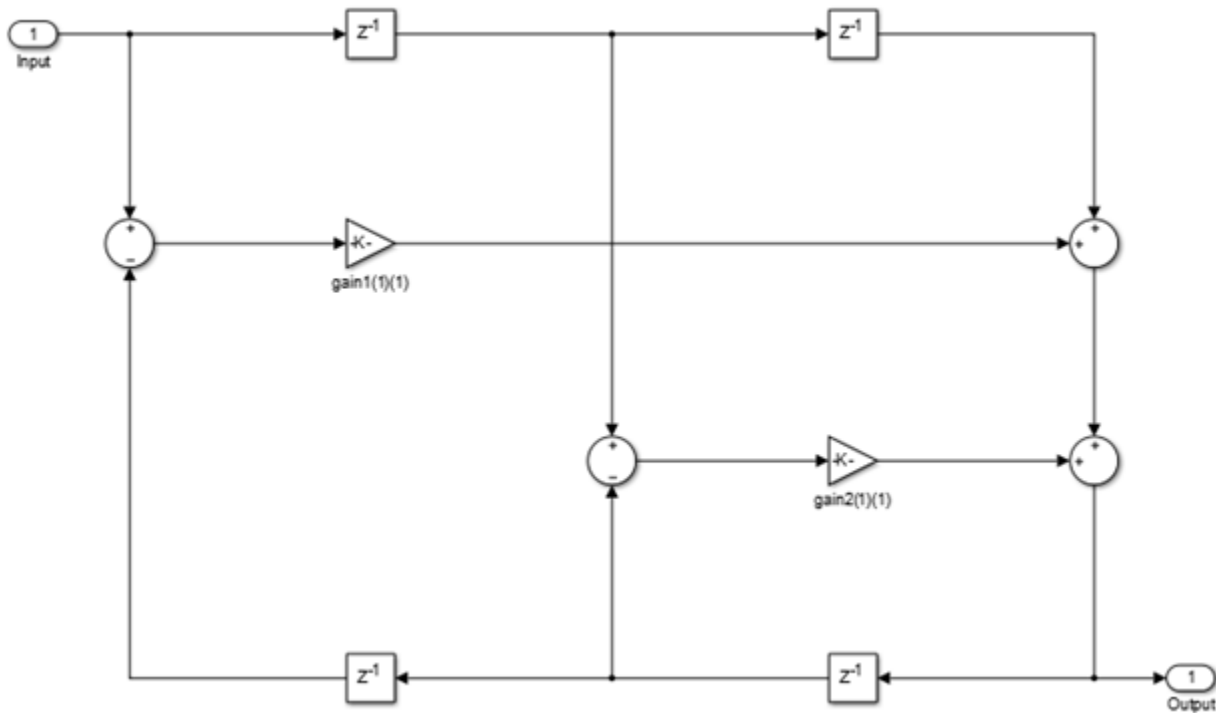
In the minimum multiplier form and wave digital form, the allpass filter is implemented as a cascade of either second-order (biquad) sections or first-order sections. When the coefficients are specified as an  $N$ -by-2 matrix, each row of the matrix specifies the coefficients of a second-order filter.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

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

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

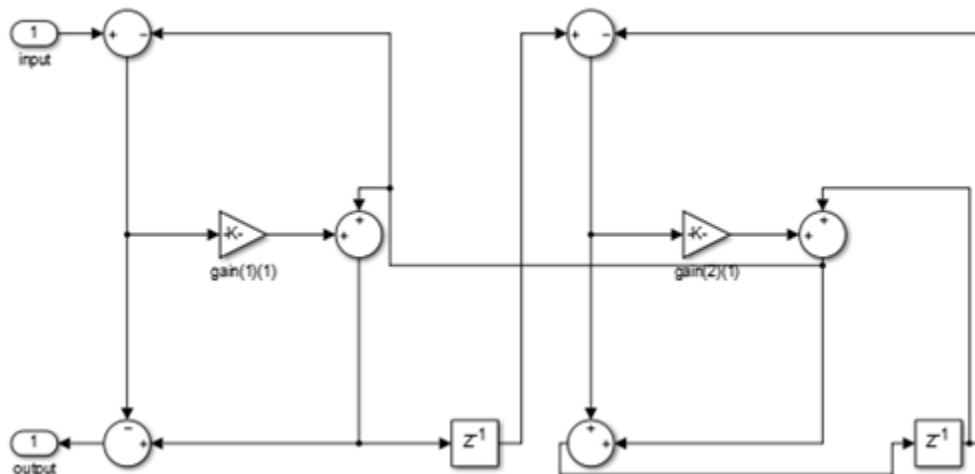
### Minimum Multiplier

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



### Wave Digital Filter

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

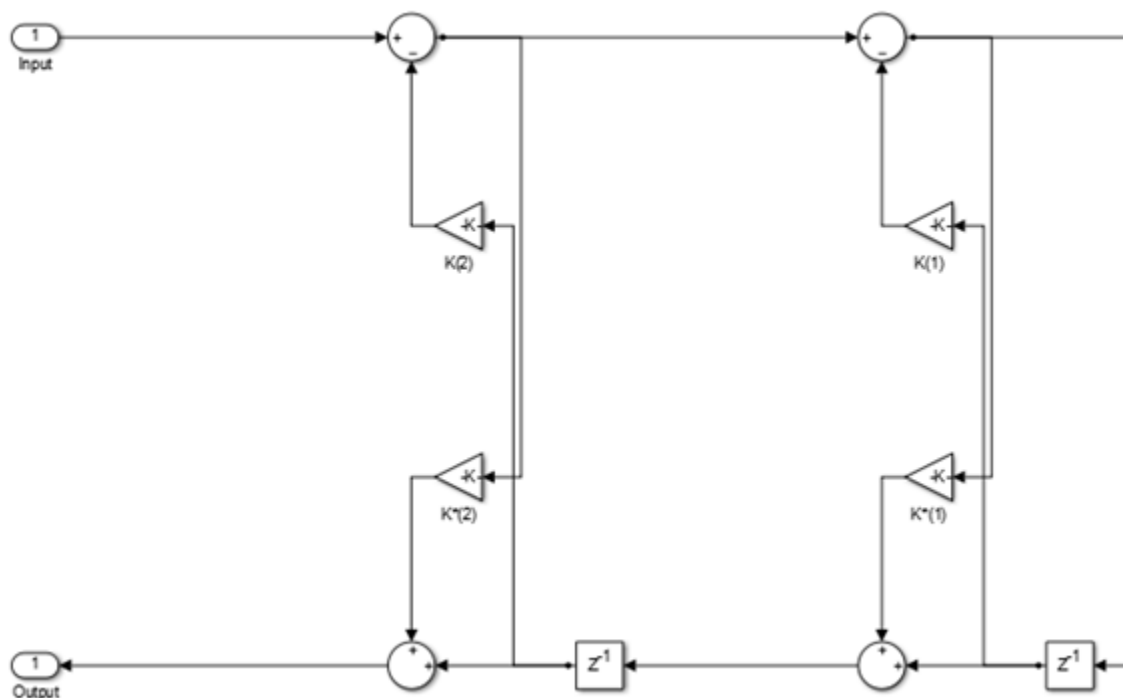




## Lattice

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

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



## Version History

Introduced in R2016b

## References

- [1] Regalia, Philip A., Sanjit K. Mitra, and P.P.Vaidyanathan. "The Digital All-Pass Filter: A Versatile Signal Processing Building Block." *Proceedings of the IEEE*. 76, no. 1 (1988): 19-37.
- [2] Lutovac, M., D. Tomic, and B. Evans. *Filter Design for Signal Processing Using MATLAB and Mathematica*. Upper Saddle River, NJ: Prentice Hall, 2001.

## **Extended Capabilities**

### **C/C++ Code Generation**

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

## **See Also**

### **Blocks**

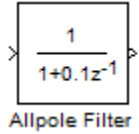
Biquad Filter | IIR Halfband Decimator | IIR Halfband Interpolator

### **Objects**

`dsp.AllpassFilter` | `dsp.SOSFilter` | `dsp.CoupledAllpassFilter` | `dsp.IIRFilter` | `dsp.IIRHalfbandDecimator` | `dsp.IIRHalfbandInterpolator`

# Allpole Filter

Model allpole filters



## Library

Filtering / Filter Implementations

dsparch4

## Description

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

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

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

### Filter Structure Support

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

- Direct form
- Direct form transposed
- Lattice AR

### Specifying Initial States

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

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

### Valid Initial States

| Initial Condition   | Description   |
|---|---|
| Scalar  | The block initializes all delay elements in the filter to the scalar value.   |
| Vector or matrix<br>(for applying different delay elements to each channel) | <p>Each vector or matrix element specifies a unique initial condition for a corresponding delay element in a corresponding channel:</p> <ul style="list-style-type: none"> <li>• The vector length equals the product of the number of input channels and the number of delay elements in the filter, <code>#_of_filter_coeffs-1</code> (or <code>#_of_reflection_coeffs</code> for Lattice AR).</li> <li>• The matrix must have the same number of rows as the number of delay elements in the filter, <code>#_of_filter_coeffs-1</code> (<code>#_of_reflection_coeffs</code> for Lattice AR), and must have one column for each channel of the input signal.</li> </ul> |

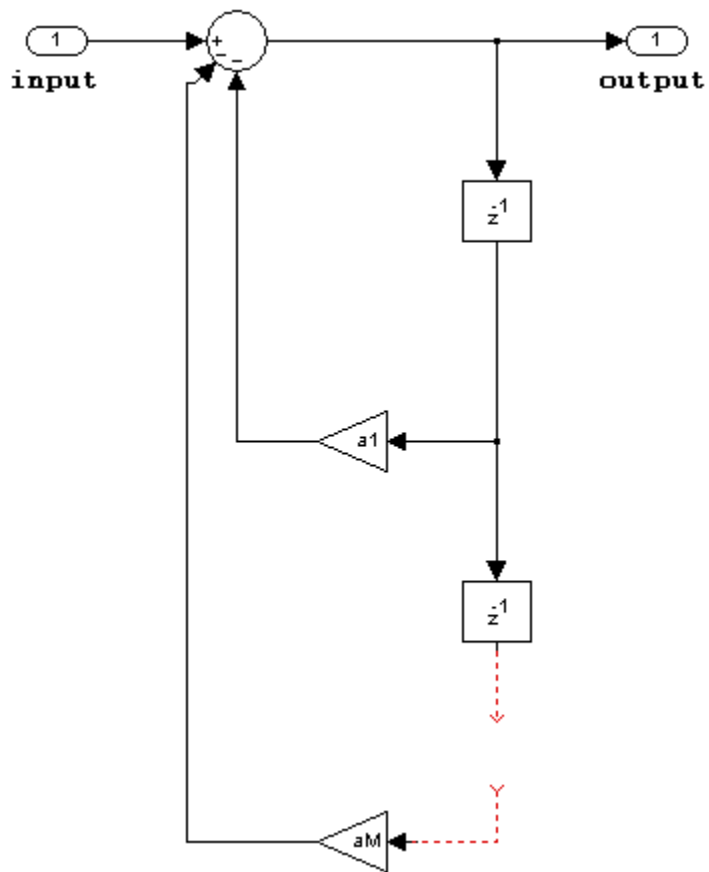
### Data Type Support

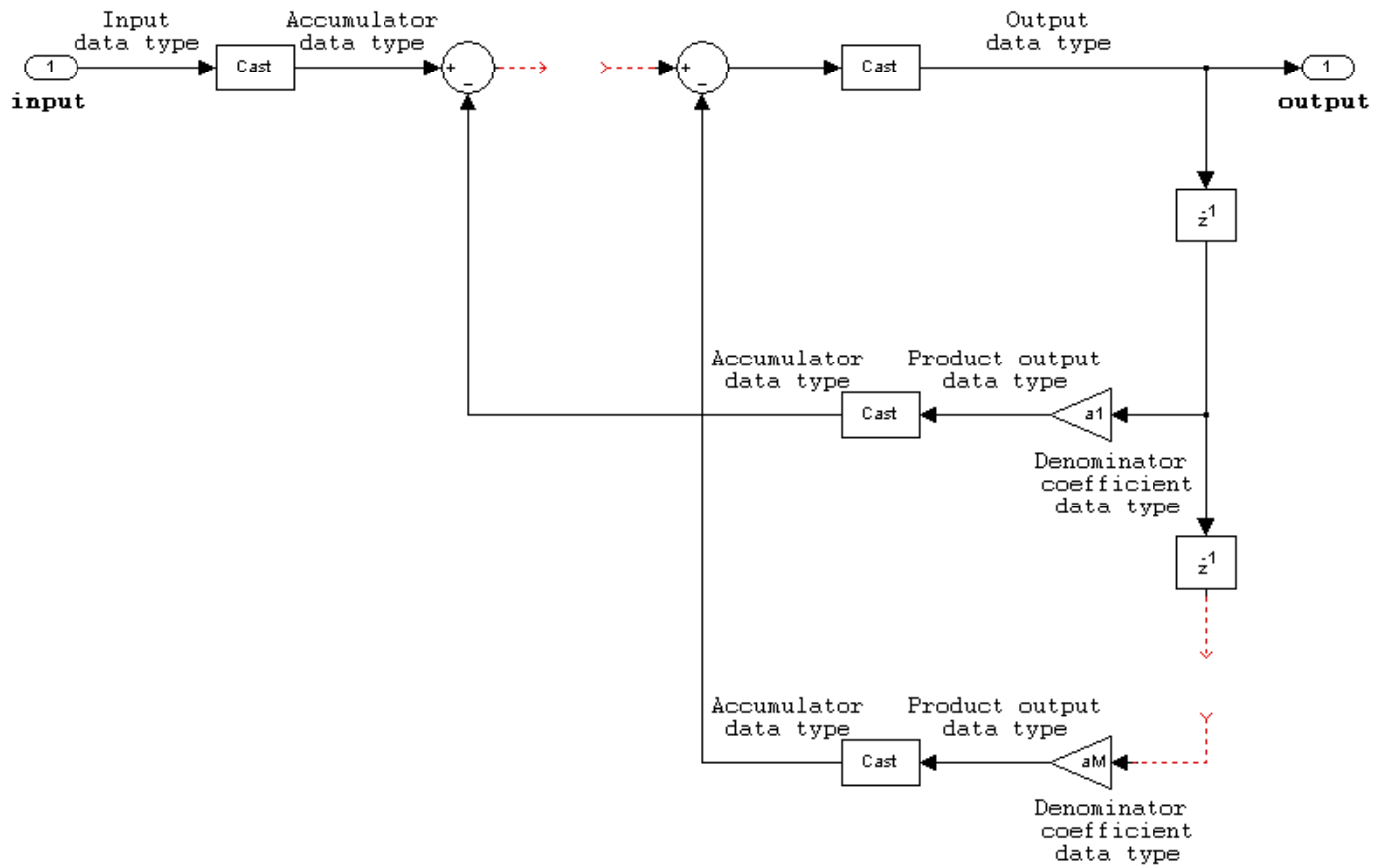
The Allpole Filter block accepts and outputs real and complex signals of any numeric data type supported by Simulink. The block supports the same types for the coefficients.

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

#### Direct Form

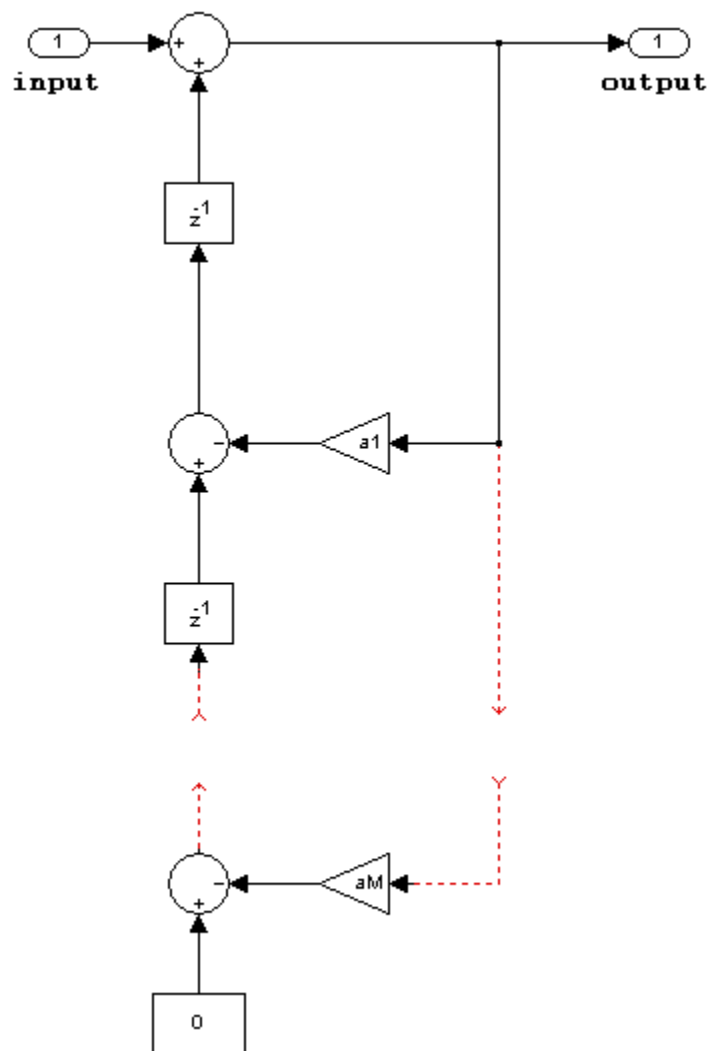
You cannot specify the state data type on the block mask for this structure because the output states have the same data types as the output.





**Direct Form Transposed**

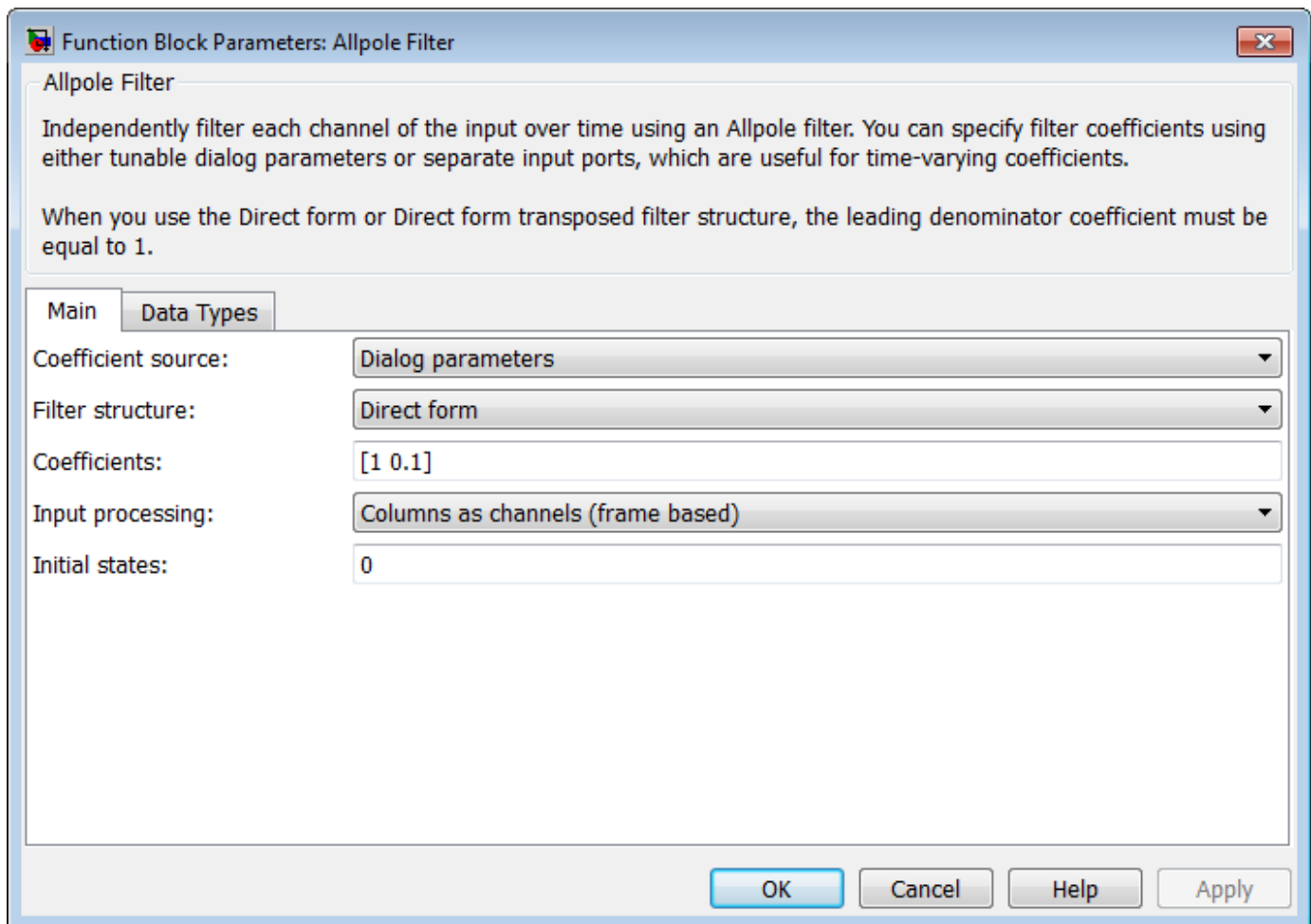
States are complex when either the inputs or the coefficients are complex.











### Coefficient source

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

### Filter structure

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

### Coefficients

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

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

### Input processing

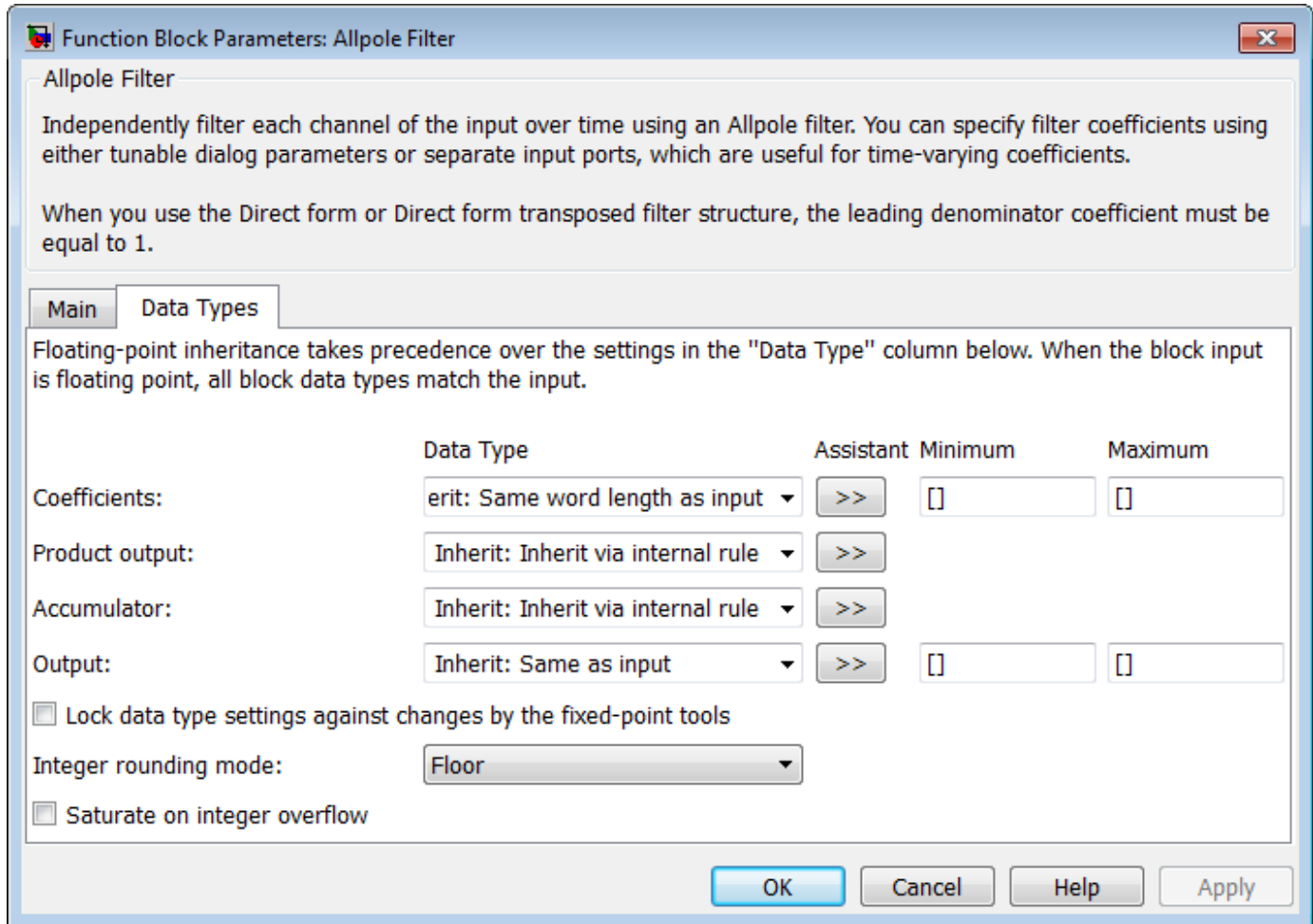
Specify whether the block performs sample- or frame-based processing. You can select one of the following options:

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

### Initial states

Specify the initial conditions of the filter states. To learn how to specify initial states, see “Specifying Initial States” on page 2-11.


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



### Coefficients

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

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

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

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

**Coefficients minimum**

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

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

**Coefficients maximum**


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

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

**Product output**

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

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


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

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

**Accumulator**

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

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

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

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


**State**

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

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

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

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


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

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

### Output

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

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

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

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

### Output minimum

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

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

### Output maximum

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

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

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

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

### Integer rounding mode

Specify the rounding mode for fixed-point operations.

### Saturate on integer overflow

| Action                        | Reasons for Taking This Action   | What Happens for Overflows  | Example   |
|-------------------------------|--|---|---|
| Select this check box.        | Your model has possible overflow and you want explicit saturation protection in the generated code.  | Overflows saturate to either the minimum or maximum value that the data type can represent. | An overflow associated with a signed 8-bit integer can saturate to -128 or 127. |
| Do not select this check box. | You want to optimize efficiency of your generated code.<br><br>You want to avoid overspecifying how a block handles out-of-range signals. For more information, see "Troubleshoot Signal Range Errors" (Simulink). | Overflows wrap to the appropriate value that is representable by the data type.             | The number 130 does not fit in a signed 8-bit integer and wraps to -126.        |

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

### Supported Data Types

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

### See Also

|                           |                           |
|---------------------------|---------------------------|
| Discrete FIR Filter       | DSP System Toolbox        |
| Filter Realization Wizard | DSP System Toolbox        |
| filterDesigner            | DSP System Toolbox        |
| <b>FVTool</b>             | Signal Processing Toolbox |

### Version History

Introduced in R2011b

### Extended Capabilities

#### C/C++ Code Generation

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

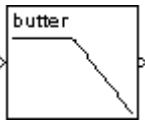
#### Fixed-Point Conversion

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

If the input is fixed point, it must be a signed integer or a signed fixed point value with a power-of-two slope and zero bias.

# Analog Filter Design

Design and implement analog filters



## Libraries:

DSP System Toolbox / Filtering / Filter Designs

## Description

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

You select the design and band configuration of the filter from the **Design method** and **Filter type** drop-down lists in the dialog box. For each combination of design method and band configuration, an appropriate set of secondary parameters is displayed. For more details, see “Design/Band Combination Table” on page 2-29.

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

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

---

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

---

## Ports

### Input

**Port\_1** — Input signal  
scalar

Input signal to filter, specified as a sample-based, continuous-time, real-valued scalar value.

Data Types: double

### Output

**Port\_1** — Output signal  
scalar

Filtered signal, returned as a real double scalar value.

Data Types: double



## Parameters

### Design method — Filter design

Butterworth (default) | Chebyshev I | Chebyshev II | Elliptic | Bessel

The filter design methods are:

- **Butterworth** — The magnitude response of a Butterworth filter is maximally flat in the passband and monotonic overall. The Butterworth design uses the toolbox function `butter`.
- **Chebyshev I** — The magnitude response of a Chebyshev I filter is equiripple in the passband and monotonic in the stopband. The Chebyshev I design uses the toolbox function `cheby1`.
- **Chebyshev II** — The magnitude response of a Chebyshev II filter is monotonic in the passband and equiripple in the stopband. The Chebyshev II design uses the toolbox function `cheby2`.
- **Elliptic** — The magnitude response of an elliptic filter is equiripple in both the passband and the stopband. The elliptic design uses the toolbox function `ellip`.
- **Bessel** — The magnitude response of a Bessel filter is maximally flat in the passband and monotonic overall. The filter has a maximally flat linear phase response. The Bessel design uses the function `bessel`.

**Tunable:** Yes

### Filter type — Filter type

Lowpass (default) | Highpass | Bandpass | Bandstop

The type of filter to design:

- Lowpass
- Highpass
- Bandpass
- Bandstop

**Tunable:** Yes

### Filter order — Filter order

8 (default) | scalar value

The order of the filter when `Filter type` is set to:

- Lowpass
- Highpass

The order of the final filter is *twice* the value of **Filter order** when **Filter type** is set to:

- Bandpass
- Bandstop

**Passband edge frequency (rad/s)** — Passband edge frequency

30 (default) | scalar value

The passband edge frequency, in rad/s.

**Tunable:** Yes**Dependencies**This parameter appears only when `Design method` is set to:

- Butterworth
- Chebyshev I
- Elliptic
- Bessel

and `Filter type` is set to:

- Lowpass
- Highpass

**Lower passband edge frequency (rad/s)** — Lower passband edge frequency

30 (default) | scalar value

The lower passband edge frequency, specified in rad/s.

**Tunable:** Yes**Dependencies**This parameter appears only when `Design method` is set to:

- Butterworth
- Chebyshev I
- Elliptic
- Bessel

and `Filter type` is set to:

- Bandpass
- Bandstop

**Upper passband edge frequency (rad/s)** — Upper passband edge frequency

80 (default) | scalar value

The upper passband edge frequency, specified in rad/s.

**Tunable:** Yes**Dependencies**This parameter appears only when `Design method` is set to:

- Butterworth
- Chebyshev I
- Elliptic
- Bessel

and `Filter type` is set to:

- Bandpass
- Bandstop

**Stopband edge frequency (rad/s)** — Stopband edge frequency

30 (default) | scalar value

The stopband edge frequency, specified in rad/s.

**Tunable:** Yes

**Dependencies**

This parameter appears only when `Design method` is set to:

- Chebyshev II

and `Filter type` is set to:

- Lowpass
- Highpass

**Lower stopband edge frequency (rad/s)** — Lower stopband edge frequency

30 (default) | scalar value

The lower stopband edge frequency, in rad/s.

**Tunable:** Yes

**Dependencies**

This parameter appears only when `Design method` is set to:

- Chebyshev II

and `Filter type` is set to:

- Bandpass
- Bandstop

**Upper stopband edge frequency (rad/s)** — Upper stopband edge frequency

80 (default) | scalar value

The upper stopband edge frequency, specified in rad/s.

**Tunable:** Yes

**Dependencies**

This parameter appears only when `Design method` is set to:

- Chebyshev II

and `Filter type` is set to:

- Bandpass
- Bandstop

**Passband ripple in dB** — Passband ripple

2 (default) | scalar value

The passband ripple, specified in dB.

**Tunable:** Yes

**Dependencies**

This parameter appears only when `Design method` is set to:

- Chebyshev I
- Elliptic

**Stopband attenuation in dB** — Stopband attenuation

40 (default) | scalar value

The stopband attenuation, in dB.

**Tunable:** Yes

**Dependencies**

This parameter appears only when `Design method` is set to:

- Chebyshev II
- Elliptic

**Block Characteristics**

|                                 |        |
|---------------------------------|--------|
| <b>Data Types</b>               | double |
| <b>Direct Feedthrough</b>       | no     |
| <b>Multidimensional Signals</b> | no     |
| <b>Variable-Size Signals</b>    | no     |
| <b>Zero-Crossing Detection</b>  | no     |

## More About

### Design/Band Combination Table

This table lists the available parameters for each design/band combination. For lowpass and highpass band configurations, these parameters include:

- The passband edge frequency  $\Omega_p$ .
- The stopband edge frequency  $\Omega_s$
- The passband ripple  $R_p$
- The stopband attenuation  $R_s$

For bandpass and bandstop configurations, the parameters include:

- The lower and upper passband edge frequencies,  $\Omega_{p1}$  and  $\Omega_{p2}$
- The lower and upper stopband edge frequencies,  $\Omega_{s1}$  and  $\Omega_{s2}$
- The passband ripple  $R_p$
- The stopband attenuation  $R_s$

Frequency values are in rad/s, and ripple and attenuation values are in dB.

| Filter Design Type       | Lowpass                           | Highpass                          | Bandpass   | Bandstop   |
|--------------------------|-----------------------------------|-----------------------------------|--|--|
| <b>Butterworth</b>       | Order, $\Omega_p$                 | Order, $\Omega_p$                 | Order, $\Omega_{p1}$ , $\Omega_{p2}$                 | Order, $\Omega_{p1}$ , $\Omega_{p2}$                 |
| <b>Chebyshev Type I</b>  | Order, $\Omega_p$ , $R_p$         | Order, $\Omega_p$ , $R_p$         | Order, $\Omega_{p1}$ , $\Omega_{p2}$ , $R_p$         | Order, $\Omega_{p1}$ , $\Omega_{p2}$ , $R_p$         |
| <b>Chebyshev Type II</b> | Order, $\Omega_s$ , $R_s$         | Order, $\Omega_s$ , $R_s$         | Order, $\Omega_{s1}$ , $\Omega_{s2}$ , $R_s$         | Order, $\Omega_{s1}$ , $\Omega_{s2}$ , $R_s$         |
| <b>Elliptic</b>          | Order, $\Omega_p$ , $R_p$ , $R_s$ | Order, $\Omega_p$ , $R_p$ , $R_s$ | Order, $\Omega_{p1}$ , $\Omega_{p2}$ , $R_p$ , $R_s$ | Order, $\Omega_{p1}$ , $\Omega_{p2}$ , $R_p$ , $R_s$ |
| <b>Bessel</b>            | Order, $\Omega_p$                 | Order, $\Omega_p$                 | Order, $\Omega_{p1}$ , $\Omega_{p2}$                 | Order, $\Omega_{p1}$ , $\Omega_{p2}$                 |

## Version History

Introduced before R2006a

## References

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

## Extended Capabilities

### C/C++ Code Generation

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

This block is not recommended for production code generation.

Consider using the “Model Discretizer” (Simulink) to map this continuous block into a discrete equivalent that supports code generation. To access the Model Discretizer, on the **Apps** tab, under **Apps**, under **Control Systems**, click **Model Discretizer**.

## **See Also**

### **Functions**

besself | butter | cheby1 | cheby2 | ellip

### **Blocks**

Digital Filter Design

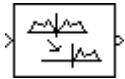
### **Topics**

“Filter Design”

“Filter Analysis”

# Analytic Signal

Compute analytic signals of discrete-time inputs



**Libraries:**  
DSP System Toolbox / Transforms

## Description

The Analytic Signal block computes the complex analytic signal  $y$  corresponding to each channel of the real input  $u$ .

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

where  $j = \sqrt{-1}$  and  $H\{ \}$  denotes the Hilbert transform.

This block supports SIMD code generation. For details, see “Code Generation” on page 2-33.

## Ports

### Input

**Input** — Data input  
vector | matrix

Specify the data input  $u$  as a vector or a matrix of size  $M$ -by- $N$ .

Data Types: single | double

### Output

**Output** — Analytic signal output  
vector | matrix

Analytic signal output  $y$ , returned as a vector or a matrix.

The block computes the analytic signal for each channel. The real part of the output in each channel is a replica of the real input in that channel. The imaginary part of the output is the Hilbert transform of the input. In the frequency domain, the Fourier transform of the analytic signal doubles the positive frequency content of the original signal while zeroing out the negative frequencies and retaining the DC component.

The output has the same size and data type as the input.

Data Types: single | double

## Parameters

**Filter order (must be even)** — Filter order

100 (default) | even positive integer

Specify the order of the equiripple FIR filter as an even positive integer. The FIR filter computes the Hilbert transform of the real input signal.

**Input processing** — Method to process input signals

Columns as channels (frame based) (default) | Elements as channels (sample based)

Specify how the block should process the input.

You can set this parameter to one of these options:

- **Columns as channels (frame based)** — The block performs frame-based processing. In this mode, the block treats an  $M$ -by- $N$  matrix input as  $N$  independent channels containing  $M$  sequential time samples. The block computes the analytic signal for each channel over time.
- **Elements as channels (sample based)** — The block performs sample-based processing. In this mode, the block treats an  $M$ -by- $N$  matrix input as  $M*N$  independent channels and computes the analytic signal for each channel (matrix element) over time.

## Block Characteristics

|                                 |                 |
|---------------------------------|-----------------|
| <b>Data Types</b>               | double   single |
| <b>Multidimensional Signals</b> | No              |
| <b>Variable-Size Signals</b>    | No              |

## More About

### Analytic Signal

The analytic signal  $x = x_r + jx_i$ , where the real part  $x_r$  is the original data and the imaginary part  $x_i$  contains the Hilbert transform. The imaginary part is a version of the original real sequence with a 90° phase shift. Sines are therefore transformed to cosines, and conversely, cosines are transformed to sines. The Hilbert-transformed series has the same amplitude and frequency content as the original sequence. The transform includes phase information that depends on the phase of the original.

The Hilbert transform is useful in calculating instantaneous attributes of a time series, especially the amplitude and the frequency. The instantaneous amplitude is the amplitude of the complex Hilbert transform. The instantaneous frequency is the time rate of change of the instantaneous phase angle. For a pure sinusoid, the instantaneous amplitude and frequency are constant. The instantaneous phase, however, is a sawtooth, reflecting how the local phase angle varies linearly over a single cycle.

## Algorithms

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

## Version History

Introduced before R2006a



## References

- [1] Claerbout, Jon F. *Fundamentals of Geophysical Data Processing with Applications to Petroleum Prospecting*. Oxford, UK: Blackwell, 1985.
- [2] Marple, S. L. "Computing the Discrete-Time Analytic Signal via FFT." *IEEE® Transactions on Signal Processing*. Vol. 47, 1999, pp. 2600-2603.
- [3] Oppenheim, Alan V., Ronald W. Schaffer, and John R. Buck. *Discrete-Time Signal Processing*. 2nd Ed. Upper Saddle River, NJ: Prentice Hall, 1999.

## Extended Capabilities

### C/C++ Code Generation

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

Generated code relies on the `memcpy` or `memset` function (`string.h`) under certain conditions.

The Analytic Signal block supports SIMD code generation using Intel AVX2 technology under these conditions:

- **Input processing** is set to `Columns as channels (frame based)`.
- Input signal has to be real-valued.
- Input signal has a data type of `single` or `double`.

The SIMD technology significantly improves the performance of the generated code.

## See Also

### Functions

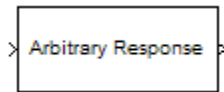
`hilbert`

### Objects

`dsp.AnalyticSignal`

# Arbitrary Response Filter

Design arbitrary response filter



## Library

Filtering / Filter Designs

dspfdesign

## Description

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

This block supports SIMD code generation. For details, see “Code Generation” on page 2-38.

## Dialog Box

See “Arbitrary Response Filter Design — Main Pane” on page 5-594 for more information about the parameters of this block. The **Data Types** and **Code Generation** panes are not available for blocks in the DSP System Toolbox Filter Designs library.

### View filter response

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

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

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

### Filter Specifications

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

#### Impulse response

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

#### Order mode

Select Minimum or Specify from the drop-down list. Selecting Specify enables the **Order** option so you can enter the filter order. When you set the **Impulse response** to IIR, you can

specify different numerator and denominator orders. To specify a different denominator order, you must select the **Denominator order** check box.

### Order

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

### Denominator order

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

### Filter type

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

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

### Decimation Factor

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

### Interpolation Factor

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

### Response Specification

#### Number of Bands

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

#### Specify response as

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

#### Frequency units

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

#### Input sample rate

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

### Band Properties

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

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

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

### Algorithm

#### Design Method

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

#### Design Options

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

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

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 16 represents a reasonable trade-off between the accurate approximation to the ideal filter and the time to design the filter.

- **Phase constraint** — Valid when the **Design method** is **Equiripple**, you have the DSP System Toolbox installed, and **Specify response as** is set to **Amplitudes**. Choose one of **Linear**, **Minimum**, or **Maximum**.
- **Weights** — Valid when the **Design method** is **Equiripple**. Uses the weights in **Weights** to weight the error for a single-band design. If you have multiple frequency bands, the **Weights** design option changes to **B1 Weights**, **B2 Weights** to designate the separate bands.

### Filter Implementation

#### Structure

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

#### Use basic elements to enable filter customization

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

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

- **Optimize for zero gains** — Terminate chains that contain Gain blocks with a gain of zero.
- **Optimize for unit gains** — Remove Gain blocks that scale by a factor of one.

- **Optimize for delay chains** — Substitute delay chains made up of  $n$  unit delays with a single delay by  $n$ .
- **Optimize for negative gains** — Use subtraction in Sum blocks instead of negative gains in Gain blocks.

### Optimize for unit-scale values

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

### Input processing

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

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

### Rate options

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

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

### Use variable names for coefficients

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

## Supported Data Types

| Port   | Supported Data Types   |
|--------|--|
| Input  | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> </ul> |
| Output | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> </ul> |

## Version History

Introduced in R2009b

## Extended Capabilities

### C/C++ Code Generation

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

The Arbitrary Response Filter block supports SIMD code generation using Intel AVX2 technology under these conditions:

- **Filter type** is set to Single-rate, Decimator, or Interpolator.
- For **Filter type** that is set to Single-rate, **Structure** is set to Direct-form FIR or Direct-form FIR transposed.
- For **Filter type** that is set to Decimator, **Structure** is set to Direct-form FIR polyphase decimator and **Rate options** is set to Enforce single-rate processing.
- For **Filter type** that is set to Interpolator, **Rate options** is set to Enforce single-rate processing.
- **Input processing** is set to Columns as channels (frame based).
- Input signal has a data type of single or double.

The SIMD technology significantly improves the performance of the generated code.

# Array Plot

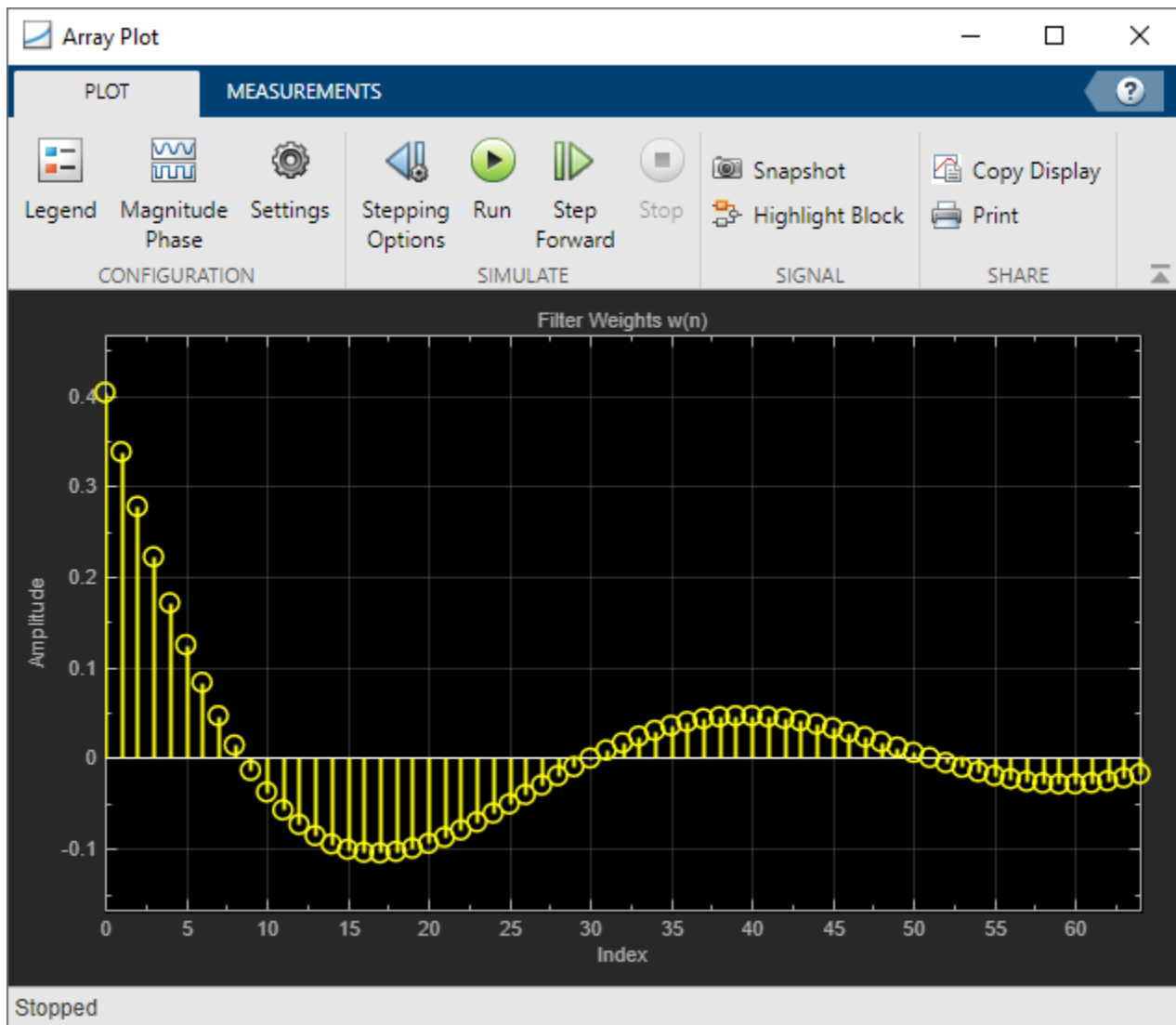
Display vectors or arrays



**Libraries:**  
DSP System Toolbox / Sinks

## Description

The Array Plot block plots vectors or arrays of data.



The Array Plot is a vector plot where data is uniformly spaced along the x-axis. To modify the spacing, change the Sample Increment property.

Measurements available:

- Data Cursors — Measure signal values using vertical and horizontal cursors.
- Signal Statistics — Display the maximum, minimum, peak-to-peak difference, mean, median, and RMS values of a selected signal.
- Peak Finder — Find maxima, showing the x-axis values at which they occur.

## Ports

### Input

**Port\_1** — Signal or signals to visualize

scalar | vector | matrix | array

Connect the signals you want to visualize. You can have up to 96 input ports. Input signals must have these characteristics:

- Fixed number of channels, but size can be variable
- Discrete, continuous, or constant sample time
- Real or complex values
- Floating- or fixed-point data type
- 2-D and nonscalar

The Array Plot block supports inputs with different frame lengths.





This block accepts fixed-point input, but converts it to `double` for display.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point` | `enumerated`

## Properties

For information about how to configure the Array Plot and use the toolstrip, see “Configure Array Plot”.

### Plot

**Legend** — Display signal legend

`off` (default) | `on`

Select this option to show the legend on the plot. The names listed in the legend are taken from the `Channel Names` property.

From the legend, you can control which signals are visible. This control is equivalent to changing the visibility in the **Style** properties. In the scope legend, click a signal name to hide the signal in the scope. To show the signal, click the signal name again. To show only one signal, right-click the signal name, which hides all other signals. To show all signals, press **Esc**.

---

**Note** The legend only shows the first 20 signals. Any additional signals cannot be controlled from the legend.

---

**Tunable:** Yes

### Programmatic Use

**Block Parameter:** `ShowLegend`

**Type:** logical

**Magnitude and Phase** — Split display into magnitude and phase plots

`off` (default) | `on`

- **On** — Display magnitude and phase plots. If the signal is real, the scope plots the absolute value of the signal for the magnitude. The phase is 0 degrees for positive values and 180 degrees for negative values. This feature is useful for complex-valued input signals. If the input is a real-valued signal, selecting this check box returns the absolute value of the signal for the magnitude.
- **Off** — Display signal plot. If the signal is complex, the scope plots the real and imaginary parts on the same y-axis.

**Tunable:** Yes

### Programmatic Use

**Block Parameter:** `PlotAsMagnitudePhase`

**Type:** logical

### Measurements

For more information about the measurements, see “Use Array Plot Measurements”.

## Settings

### Data and Axes

**X-Data Mode** — Type of x-axis spacing

**Sample increment and X-offset** (default) | **Custom**

Select the type of spacing to use between x-axis data values.

- **Sample increment and X-offset** — Use the “Sample Increment” on page 2-0 and “X-Offset” on page 2-0 values to specify x-axis data.
- **Custom** — Specify a custom spacing between data values using the “Custom X-data” on page 2-0 property.

### Programmatic Use

**Block Parameter:** XDataMode

**Type:** character vector or string scalar

**Values:** 'Sample increment and X-offset' | 'Custom'

**Sample Increment** — x-axis spacing

1 (default) | finite scalar

Specify the spacing between samples along the x-axis as a finite numeric scalar. The input signal is only y-axis data. x-axis data is set automatically based on both the **Sample Increment** and “X-Offset” on page 2-0 values.

Example: When **X-Offset** is 0 and **Sample Increment** is 1, the x-axis values are set to 0, 1, 2, 3, 4, ... .

Example: When **X-Offset** is -1 and **Sample Increment** is 0.25, the x-axis values are set to -1, -0.75, -0.5, -0.25, 0, ... .

**Tunable:** Yes

### Dependency

To use this property, set “X-Data Mode” on page 2-0 to **Sample increment and X-offset**.

### Programmatic Use

**Block Parameter:** SampleIncrement

**Type:** character vector or string scalar

**Values:** scalar

**X-Offset** — x-axis offset

0 (default) | scalar

Specify the offset to apply to the x-axis, as a numeric scalar. x-axis data is set automatically based on both the “Sample Increment” on page 2-0 and **X-Offset** values. The x-offset represents the first value on the x-axis.

Example: When **X-Offset** is 0 and **Sample Increment** is 1, the x-axis values are set to 0, 1, 2, 3, 4, ... .

Example: When **X-Offset** is -1 and **Sample Increment** is 0.25, the x-axis values are set to -1, -0.75, -0.5, -0.25, 0, ... .

**Tunable:** Yes

**Dependency**

To use this property, set “X-Data Mode” on page 2-0 to Sample increment and X-offset.

**Programmatic Use****Block Parameter:** XOffset

**Type:** character vector or string scalar

**Values:** scalar

**Custom X-data** — x-axis data values

empty vector (default) | vector with length equal to the input frame length

Specify the x-axis data values as a vector equal in length to the frame length of the inputs. If you use the default (empty vector) value, the x-axis data is uniformly spaced over the interval  $(0:L-1)$ , where  $L$  is the frame length.

Example: A custom logarithmic x-axis data scaling is `[0:log10(44100/2):1024]`

**Tunable:** Yes

**Dependency**

To use this property, set “X-Data Mode” on page 2-0 to Custom

**Programmatic Use****Block Parameter:** CustomXData

**Type:** character vector or string scalar

**Values:** scalar

**Default:** ' [] '

**X-Axis Scale** — x-axis scale

Linear (default) | Log

Select Linear or Log as the x-axis scale.

**Tunable:** Yes

**Dependency**

If **X-Offset** is a negative value, you cannot set this parameter to Log.

**Programmatic Use****Block Parameter:** XScale

**Type:** character vector or string scalar

**Y-Axis Scale** — y-axis scale

Linear (default) | Log

Select Linear or Log as the y-axis scale.

**Tunable:** Yes

**Programmatic Use****Block Parameter:** YScale

**Type:** character vector or string scalar

**Y-limits** — y-axis limits

`[-10,10]` (default) | `[ymin, ymax]`

Specify the y-axis limits as a two-element numeric vector, `[ymin, ymax]`.

If **Magnitude and Phase** is off, the default is `[-10,10]`. If the magnitude/phase plot is on, the default is `[0,10]`.

**Tunable:** Yes

#### **Dependencies**

When `PlotAsMagnitudePhase` is `true`, this property specifies the y-axis limits of only the magnitude plot. The y-axis limits of the phase plot are always `[-180,180]`.

#### **Programmatic Use**

**Block Parameter:** `YLimits`

**Type:** character vector or string scalar

**Value:** two-element numeric vector

**Plot Type** — Type of plot

`Stem` (default) | `Stairs` | `Line`

- `Stem` — Stem graph displayed as circles at the input value with vertical lines to the x-axis.
- `Line` — Line graph
- `Stairs` — Stair-step graph. A stair-step graph is made up of only horizontal and vertical lines. Each horizontal line represents the signal value for a discrete sample period and is connected to two vertical lines. Each vertical line represents the change in the signal value occurring at a specific sample time.

**Tunable:** Yes

#### **Programmatic Use**

**Block Parameter:** `PlotType`

**Type:** character vector or string scalar

**Grid** — Show internal grid lines

`on` (default) | `off`

Select this option to show grid lines on the plot.

**Tunable:** Yes

#### **Programmatic Use**

**Block Parameter:** `ShowGrid`

**Type:** logical

#### **Display and Labels**

**Title** — Display name

`none` (default) | string

Specify a title for display. To use the input signal name for the title, enter %<SignalLabel>.

**Tunable:** Yes

**Programmatic Use**

**Block Parameter:** Title

**Type:** character vector or string scalar

**Default:** ''

**X-Label** — x-axis label

none (default) | string

Specify the text for the scope to display below the x-axis.

**Tunable:** Yes

**Programmatic Use**

**Block Parameter:** XLabel

**Type:** character vector or string scalar

**Default:** ''

**Y-Label** — Y-axis label

none (default) | string

Specify the text to display on the y-axis. To display signal units, add (%<SignalUnits>) to the label. At the beginning of a simulation, Simulink replaces (%<SignalUnits>) with the units associated with the signals.

Example: For a velocity signal with units of m/s, enter Velocity (%<SignalUnits>).

**Tunable:** Yes

**Dependency**

If you click on **Magnitude and Phase**, this property does not apply. The y-axes are labeled Magnitude and Phase.

**Programmatic Use**

**Block Parameter:** YLabel

**Type:** character vector or string scalar

**Default:** ''

**Property Inspector Only**

**Number of input ports** — Number of input ports

1 (default) | integer between 1 and 96

The number of input ports to the block, specified as an integer between 1 and 96. To change the number of input ports, drag a new input signal line to the block and the block will auto-create new ports.

**Programmatic Use**

**Block Parameter:** NumInputPorts

**Type:** character vector or string scalar

**Values:** scalar between 1 and 96

**Open at start of simulation** — Automatically open scope at start  
on (default) | off

Select this parameter to automatically open the Array Plot window when you run the simulation.

This parameter is only available from the Property Inspector.

#### Programmatic Use

**Block Parameter:** OpenAtSimulationStart

**Type:** logical

**Channel Names** — Input channel names

[] (default) | character vector | string | array of strings or character vectors.

Input channel names, specified as a character vector, string, or array of either. The names appear in the legend, **Settings**, and **Measurements** panels. If you do not specify names, the channels are labeled as Channel 1, Channel 2, etc.

Example: ["A","B"]

#### Dependency

To see channel names, set “Legend” on page 2-0 to true.

#### Programmatic Use


**Block Parameter:** ChannelNames

**Type:** cell array of character vectors or string array

**Maximize axes** — Maximize size of plots

Auto (default) | Off | On

- Auto — If “Title” on page 2-0 and “Y-Label” on page 2-0 properties are not specified, maximize all plots.
- On — Maximize all plots. Values in **Title** and **Y-label** are hidden.
- Off — Do not maximize plots.

Hover over the array plot to see the maximize axes button .

**Tunable:** Yes

#### Programmatic Use

**Block Parameter:** MaximizeAxes

**Type:** character vector or string scalar

**Axes scaling** — Y-axis scaling mode

OnceAtStop (default) | Manual | Auto | Updates

- OnceAtStop — Scale y-axis after the simulation is finished.
- Manual — Manually scale y-axis range with the **Scale Y-axis Limits** toolbar button.

- Auto — Scale y-axis range during and after simulation.
- Updates — Scale y-axis after the number of time steps specified in the “Number of updates” on page 2-0 text box (100 by default). Scaling occurs only once during each run.

**Tunable:** Yes

#### Programmatic Use

**Block Parameter:** AxesScaling

**Type:** character vector or string scalar

**Number of updates** — Number of updates before scaling  
10 (default) | integer

Set this property to delay auto scaling the y-axis.

**Tunable:** Yes

#### Dependency

To enable this property, set “Axes scaling” on page 2-0 to AfterNUpdates.

#### Programmatic Use

**Block Parameter:** AxesScalingNumUpdates

**Type:** character vector or string scalar

**Values:** scalar

## Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | Boolean   double   enumerated   fixed point   half   integer   single |
| <b>Direct Feedthrough</b>       | no  |
| <b>Multidimensional Signals</b> | yes   |
| <b>Variable-Size Signals</b>    | yes   |
| <b>Zero-Crossing Detection</b>  | no  |

## Version History

Introduced in R2015b

## Extended Capabilities

### C/C++ Code Generation

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

This block can be used for simulation visibility in systems that generate code, but is not included in the generated code.

## **See Also**

### **Blocks**

Time Scope | Spectrum Analyzer

### **Objects**

`dsp.ArrayPlot`

### **Topics**

“Configure Array Plot”

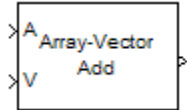
Array Plot with Apple iOS Devices (Simulink Support Package for Apple iOS Devices)

“Array Plot with Android Devices”



## Array-Vector Add

Add vector to array along specified dimension



## Library

Math Functions / Matrices and Linear Algebra / Matrix Operations

dspmtx3

## Description

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

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

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

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

where

$$1 \leq i \leq M$$

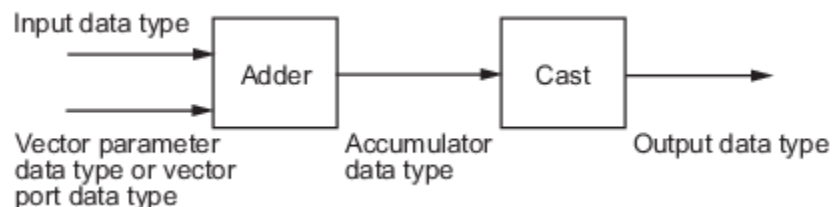
$$1 \leq j \leq N$$

$$1 \leq k \leq P$$

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

## Fixed-Point Data Types

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



When you specify the vector  $V$  on the **Main** pane of the block mask, you must specify the data type and scaling properties of its elements in the **Vector (V)** parameter on the **Data Types** tab. When the

vector comes in through the block port, its elements inherit their data type and scaling from the driving block.

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

## Parameters

### Main Tab

#### Add along dimension

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

#### Vector (V) source

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

#### Vector (V)

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

### Data Types Tab

---

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

---

### Rounding mode

Specify the rounding mode for fixed-point operations as one of the following:

- Floor
- Ceiling
- Convergent
- Nearest
- Round
- Simplest
- Zero

For more details, see rounding mode.

If **Accumulator** data type is `Inherit: Inherit via internal rule` and **Output** data type is `Inherit: Same as accumulator`, the value of **Rounding mode** does not affect the numerical results.

---

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

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

---

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

---


### Saturate on integer overflow

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

### Vector (V)

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

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

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

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

---


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

---

### Accumulator

Specify the accumulator data type. See “Fixed-Point Data Types” on page 2-49 for a diagram showing the use of the accumulator data type in this block. You can set this parameter to:

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


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

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

### Output

Specify the output data type. See “Fixed-Point Data Types” on page 2-49 for illustrations depicting the use of the output data type in this block. You can set it to:

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

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

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

### Minimum

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

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

### Maximum

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

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

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

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

### Vector (V) Data Type

#### Mode

Select how you would like to specify the data type properties of the **Vector (V)** data type. You can choose:

- **Inherit** — Lets you specify a rule for inheriting a data type, for example, `Inherit: Inherit via internal rule`
- **Fixed point** — Lets you specify the fixed-point attributes of the data type.
- **Expression** — Lets you specify an expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

#### Signedness

Specify the **Signedness** for the **Vector (V)** data type.

#### Scaling

Specify the **Scaling** for the **Vector (V)** data type.

For more information see “Scaling” in the DSP System Toolbox User's Guide.

#### Word length

Specify the **Word length** for the **Vector (V)** data type.

**Fraction length**

Specify the **Fraction length** for the **Vector (V)** data type.

**Data type override**

Specify the data type override mode. You can select one of the following options:

- **Inherit** — Inherits the data type override setting specified for the model.
- **Off** — Ignores the data type override setting specified for the model and uses the fixed-point data type you specify

This parameter appears only when you set the **Mode** parameter to **Built in** or **Fixed Point**. For more information, see “Specify Data Types Using Data Type Assistant” (Simulink).

**Accumulator Data Type Assistant Parameters****Mode**

Select how you would like to specify the data type properties of the **Accumulator** data type. You can choose:

- **Inherit** — Lets you specify a rule for inheriting a data type, for example, **Inherit: Inherit via internal rule**
- **Fixed point** — Lets you specify the fixed-point attributes of the data type.
- **Expression** — Lets you specify an expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

**Signedness**

Specify the **Signedness** for the **Accumulator** data type.

**Scaling**

Specify the **Scaling** for the **Accumulator** data type.

For more information see “Scaling” in the DSP System Toolbox User's Guide.

**Word length**

Specify the **Word length** for the **Accumulator** data type.

**Fraction length**

Specify the **Fraction length** for the **Accumulator** data type.

**Data type override**

Specify the data type override mode. You can select one of the following options:

- **Inherit** — Inherits the data type override setting specified for the model.
- **Off** — Ignores the data type override setting specified for the model and uses the fixed-point data type you specify

This parameter appears only when you set the **Mode** parameter to **Built in** or **Fixed Point**. For more information, see “Specify Data Types Using Data Type Assistant” (Simulink).

### **Output Data Type Assistant Parameters**

#### **Mode**

Select how you would like to specify the data type properties of the **Output** data type. You can choose:

- **Inherit** — Lets you specify a rule for inheriting a data type, for example, `Inherit: Inherit via internal rule`
- **Fixed point** — Lets you specify the fixed-point attributes of the data type.
- **Expression** — Lets you specify an expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

#### **Signedness**

Specify the **Signedness** for the **Output** data type.

#### **Scaling**

Specify the **Scaling** for the **Output** data type.

For more information see “Scaling” in the DSP System Toolbox User's Guide.

#### **Word length**

Specify the **Word length** for the **Output** data type.

#### **Fraction length**

Specify the **Fraction length** for the **Output** data type.

#### **Data type override**

Specify the data type override mode. You can select one of the following options:

- **Inherit** — Inherits the data type override setting specified for the model.
- **Off** — Ignores the data type override setting specified for the model and uses the fixed-point data type you specify

This parameter appears only when you set the **Mode** parameter to **Built in** or **Fixed Point**. For more information, see “Specify Data Types Using Data Type Assistant” (Simulink).

## Supported Data Types

| Port   | Supported Data Types   |
|--------|--|
| A      | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed only)</li> <li>• 8-, 16-, and 32-bit signed integers</li> </ul> |
| V      | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed only)</li> <li>• 8-, 16-, and 32-bit signed integers</li> </ul> |
| Output | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed only)</li> <li>• 8-, 16-, and 32-bit signed integers</li> </ul> |

## Version History

Introduced in R2007b

## Extended Capabilities

### C/C++ Code Generation

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

### Fixed-Point Conversion

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

If the input is fixed point, it must be a signed integer or a signed fixed point value with a power-of-two slope and zero bias.

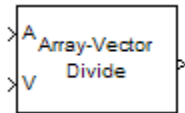
## See Also

### Blocks

Array-Vector Divide | Array-Vector Multiply | Array-Vector Subtract

## Array-Vector Divide

Divide array by vector along specified dimension



## Library

Math Functions / Matrices and Linear Algebra / Matrix Operations

dspmtrx3

## Description

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

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

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

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

where

$$1 \leq i \leq M$$

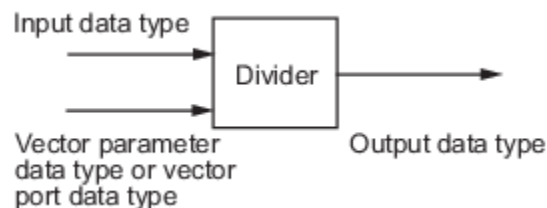
$$1 \leq j \leq N$$

$$1 \leq k \leq P$$

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

## Fixed-Point Data Types

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





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

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

## Parameters

### Main Tab

#### Divide along dimension

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

#### Vector (V) source

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

#### Vector (V)

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

### Data Types Tab

---

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

---

### Rounding mode

Specify the rounding mode for fixed-point operations as one of the following:

- Floor
- Ceiling
- Convergent
- Nearest
- Round
- Simplest
- Zero

For more details, see rounding mode.


### Saturate on integer overflow

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

### Vector (V)

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

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

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

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

---


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

---

## Output

Specify the output data type. See “Fixed-Point Data Types” on page 2-56 for illustrations depicting the use of the output data type in this block. You can set it to:

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

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

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

## Minimum

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

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

## Maximum

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

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

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

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

## Vector (V) Data Type

### Mode

Select how you would like to specify the data type properties of the **Vector (V)** data type. You can choose:

- **Inherit** — Lets you specify a rule for inheriting a data type, for example, `Inherit: Inherit via internal rule`
- **Fixed point** — Lets you specify the fixed-point attributes of the data type.
- **Expression** — Lets you specify an expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

### **Signedness**

Specify the **Signedness** for the **Vector (V)** data type.

### **Scaling**

Specify the **Scaling** for the **Vector (V)** data type.

For more information see “Scaling” in the DSP System Toolbox User's Guide.

### **Word length**

Specify the **Word length** for the **Vector (V)** data type.

### **Fraction length**

Specify the **Fraction length** for the **Vector (V)** data type.

### **Data type override**

Specify the data type override mode. You can select one of the following options:

- **Inherit** — Inherits the data type override setting specified for the model.
- **Off** — Ignores the data type override setting specified for the model and uses the fixed-point data type you specify

This parameter appears only when you set the **Mode** parameter to **Built in** or **Fixed Point**. For more information, see “Specify Data Types Using Data Type Assistant” (Simulink).

## **Output Data Type Assistant Parameters**

### **Mode**

Select how you would like to specify the data type properties of the **Output** data type. You can choose:

- **Inherit** — Lets you specify a rule for inheriting a data type, for example, `Inherit: Inherit via internal rule`
- **Fixed point** — Lets you specify the fixed-point attributes of the data type.
- **Expression** — Lets you specify an expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

**Signedness**

Specify the **Signedness** for the **Output** data type.

**Scaling**

Specify the **Scaling** for the **Output** data type.

For more information see “Scaling” in the DSP System Toolbox User's Guide.

**Word length**

Specify the **Word length** for the **Output** data type.

**Fraction length**

Specify the **Fraction length** for the **Output** data type.

**Data type override**

Specify the data type override mode. You can select one of the following options:

- **Inherit** — Inherits the data type override setting specified for the model.
- **Off** — Ignores the data type override setting specified for the model and uses the fixed-point data type you specify

This parameter appears only when you set the **Mode** parameter to **Built in** or **Fixed Point**. For more information, see “Specify Data Types Using Data Type Assistant” (Simulink).

**Supported Data Types**

| <b>Port</b> | <b>Supported Data Types</b>  |
|-------------|--|
| A           | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed only)</li> <li>• 8-, 16-, and 32-bit signed integers</li> </ul> |
| V           | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed only)</li> <li>• 8-, 16-, and 32-bit signed integers</li> </ul> |
| Output      | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed only)</li> <li>• 8-, 16-, and 32-bit signed integers</li> </ul> |

## Version History

Introduced in R2007b

### Extended Capabilities

#### **C/C++ Code Generation**

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

#### **Fixed-Point Conversion**

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

If the input is fixed point, it must be a signed integer or a signed fixed point value with a power-of-two slope and zero bias.

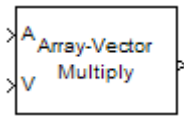
### See Also

#### **Blocks**

Array-Vector Add | Array-Vector Multiply | Array-Vector Subtract

## Array-Vector Multiply

Multiply array by vector along specified dimension



### Libraries:

DSP System Toolbox / Math Functions / Matrices and Linear Algebra / Matrix Operations

### Description

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

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

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

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

where

$$1 \leq i \leq M$$

$$1 \leq j \leq N$$

$$1 \leq k \leq P$$

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

### Ports

#### Input

**A** — Array input  
vector | matrix

Specify the array input as a vector or a matrix.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

**V** — Vector input  
vector

Specify the vector input as a vector.

#### Dependencies

This port appears only when you set the **Vector (V) source** parameter to Input port.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

## Output

**Port\_1** — Multiplied output

vector | matrix

Output of the Array-Vector Multiply block, returned as a vector or a matrix.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

## Parameters

### Main Tab

**Multiply along dimension** — Dimension along which to multiply input array

2 (default) | positive integer

Specify the dimension along which to multiply the input array  $A$  by the elements of vector  $V$  as a positive integer.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**Vector (V) source** — Source of vector

Input port (default) | Dialog parameter

Set the source of the vector  $V$  to one of these options.

- Input port — Vector comes from the input port.
- Dialog parameter — Vector comes from the dialog parameter **Vector (V)**.

**Vector (V)** — Vector parameter

[0.5 0.25] (default) | vector

Specify the vector  $V$ .

### Dependencies

This parameter appears only when you set the **Vector (V) source** parameter to Dialog parameter.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

### Data Types Tab

---

**Note** Floating-point inheritance takes precedence over the data type settings defined on the **Data Types** tab. When inputs are floating point, the block ignores these settings and all internal data types are floating point.

---

**Rounding mode** — Method of rounding operation

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

Set the rounding mode for fixed-point operations to one of these options.

- Ceiling
- Convergent
- Floor
- Nearest
- Round
- Simplest
- Zero

For details, see rounding mode.

### **Saturate on integer overflow** — Method of overflow action

off (default) | on


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

### **Vector (V)** — Word and fraction lengths for elements of vector

Inherit: Same word length as input (default) | `fixdt(1,16,0)` | <data type expression>

Use this parameter to specify the word and fraction lengths for the elements of the vector, *V*. Select this parameter as one of these options.

- **Inherit: Same word length as input** — Inherits the data type so it is the same as that of the input array *A*.
- `fixdt(1,16,0)` — Specifies a signed, binary-point, scaled, fixed-point data type with a word length of 16 bits and a fraction length of 0.
- <data type expression> — Lets you specify an expression that evaluates to a valid data type.

Alternatively, you can set the **Vector (V)** data type by using the **Data Type Assistant**. To use the assistant, click the **Show data type assistant** button .

For more information on the data type assistant, see “Specify Data Types Using Data Type Assistant” (Simulink).

---

**Note** When the vector comes in through the input port of the block, the data type and scaling of its elements are inherited from the driving block.

---

### **Dependencies**

This parameter appears only when you set the **Vector (V) source** parameter on the **Main** tab to **Dialog** parameter.

### **Vector (V) Minimum** — Minimum value for elements of vector

[] (unspecified) (default) | scalar



Specify the minimum value for the elements of vector  $V$ . The default value is [] (unspecified). Simulink uses this value to perform:

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

#### Dependencies

This parameter appears only when you set the **Vector (V) source** parameter on the **Main** tab to Dialog parameter.

**Vector (V) Maximum** — Maximum value for elements of vector  
[] (unspecified) (default) | scalar

Specify the maximum value for the element of vector  $V$ . The default value is [] (unspecified). Simulink uses this value to perform:

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

#### Dependencies

This parameter appears only when you set the **Vector (V) source** parameter on the **Main** tab to Dialog parameter.

**Product output** — Product output data type

Inherit: Inherit via internal rule (default) | Inherit: Same as first input |  
fixdt([],16,0) | <data type expression>

Specify the data type of the output of a product operation in the Array-Vector Multiply block. For illustrations depicting the use of the product output data type in this block, see “Fixed Point” on page 2-67 and “Multiplication Data Types”. Set this parameter to one of these options.

- **Inherit: Inherit via internal rule** — Inherits the data type based on an internal rule. For more information on this rule, see “Inherit via Internal Rule”.
- **Inherit: Same as first input** — Inherits the data type so it is the same as that of the input array  $A$ .
- **fixdt([],16,0)** — Lets you specify the fixed-point attributes of the data type.
- **<data type expression>** — Lets you specify an expression that evaluates to a valid data type.

Alternatively, you can set the **Product output** data type by using the **Data Type Assistant**. To use

the assistant, click the **Show data type assistant** button .


For more information on the data type assistant, see “Specify Data Types Using Data Type Assistant” (Simulink).

**Accumulator** — Accumulator data type

Inherit: Inherit via internal rule (default) | Inherit: Same as first input |  
Inherit: Same as product output | fixdt([],16,0) | <data type expression>

Specify the data type of the output of an accumulation operation in the Array-Vector Multiply block. For illustrations depicting the use of the accumulator data type in this block, see “Fixed Point” on page 2-67. Set this parameter to one of these options.

- `Inherit: Inherit via internal rule` — Inherits the data type based on an internal rule. For more information on this rule, see “Inherit via Internal Rule”.
- `Inherit: Same as first input` — Inherits the data type so it is the same as that of the input array *A*.
- `Inherit: Same as product output` — Inherits the data type so it is the same as the product output data type.
- `fixdt([],16,0)` — Lets you specify the fixed-point attributes of the data type.
- `<data type expression>` — Lets you specify an expression that evaluates to a valid data type.

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

For more information on the data type assistant, see “Specify Data Types Using Data Type Assistant” (Simulink).


### Output — Output data type

`Inherit: Same as product output (default) | Inherit: Same as first input | Inherit: Same as first input | fixdt([],16,0) | <data type expression>`

Specify the output data type. For illustrations depicting the use of the output data type in this block, see “Fixed Point” on page 2-67. Set this parameter to one of these options.

- `Inherit: Same as first input` — Inherits the data type so it is the same as that of the input array *A*.
- `Inherit: Same as product output` — Inherits the data type so it is the same as the product output data type.
- `fixdt([],16,0)` — Lets you specify the fixed-point attributes of the data type.
- `<data type expression>` — Lets you specify an expression that evaluates to a valid data type.

For more information, see “Control Data Types of Signals” (Simulink).

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

For more information on the data type assistant, see “Specify Data Types Using Data Type Assistant” (Simulink).

### Output Minimum — Minimum value of block output

`[] (unspecified) (default) | scalar`

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

- Simulation range checking. For more information, see “Specify Signal Ranges” (Simulink).

- Automatic scaling of fixed-point data types.

**Output Maximum** — Maximum value of block output

[ ] (unspecified) (default) | scalar

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

- Simulation range checking. For more information, see “Specify Signal Ranges” (Simulink).
- Automatic scaling of fixed-point data types.

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

off (default) | on

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

## Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | double   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no                                      |
| <b>Multidimensional Signals</b> | no                                      |
| <b>Variable-Size Signals</b>    | yes                                     |
| <b>Zero-Crossing Detection</b>  | no                                      |

## Version History

Introduced in R2007b

## Extended Capabilities

### C/C++ Code Generation

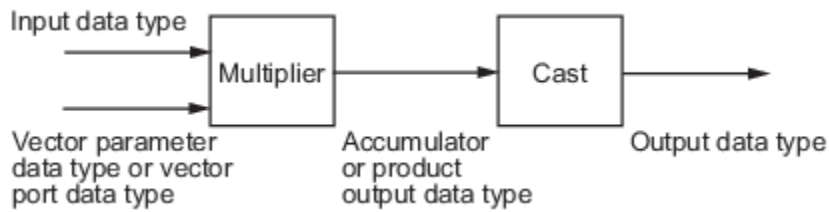
Generate C and C++ code using Simulink® Coder™.

### Fixed-Point Conversion

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

If the input is fixed point, it must be a signed integer or a signed fixed point value with a power-of-two slope and zero bias.

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



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

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

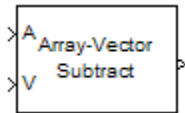
You can set the vector, accumulator, product output, and output data types in the block dialog.

## See Also

Array-Vector Add | Array-Vector Divide | Array-Vector Subtract

# Array-Vector Subtract

Subtract vector from array along specified dimension



## Library

Math Functions / Matrices and Linear Algebra / Matrix Operations

dspmtrx3

## Description

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

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

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

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

where

$$1 \leq i \leq M$$

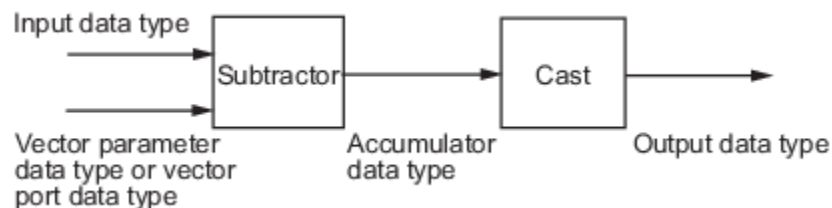
$$1 \leq j \leq N$$

$$1 \leq k \leq P$$

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

## Fixed-Point Data Types

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



When you specify the vector  $V$  on the **Main** pane of the block mask, you must specify the data type and scaling properties of its elements in the **Vector (V)** parameter on the **Data Types** tab. When the

vector comes in through the block port, its elements inherit their data type and scaling from the driving block.

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

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

## Parameters

### Main Tab

#### Subtract along dimension

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

#### Vector (V) source

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

#### Vector (V)

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

### Data Types

---

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

---

### Rounding mode

Specify the rounding mode for fixed-point operations as one of the following:

- Floor
- Ceiling
- Convergent
- Nearest
- Round
- Simplest
- Zero

For more details, see rounding mode.

---

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

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

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

---


### Saturate on integer overflow

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

### Vector (V)

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

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

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

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

---


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

---

### Accumulator

Specify the accumulator data type. See “Fixed-Point Data Types” on page 2-69 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

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

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

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

### Output

Specify the output data type. See “Fixed-Point Data Types” on page 2-69 for illustrations depicting the use of the output data type in this block. You can set it to:

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

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

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

**Minimum**

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

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

**Maximum**

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

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

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

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

**Vector (V) Data Type****Mode**

Select how you would like to specify the data type properties of the **Vector (V)** data type. You can choose:

- **Inherit** — Lets you specify a rule for inheriting a data type, for example, `Inherit: Inherit via internal rule`
- **Fixed point** — Lets you specify the fixed-point attributes of the data type.
- **Expression** — Lets you specify an expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

**Signedness**

Specify the **Signedness** for the **Vector (V)** data type.

**Scaling**

Specify the **Scaling** for the **Vector (V)** data type.

For more information see “Scaling” in the DSP System Toolbox User's Guide.

**Word length**

Specify the **Word length** for the **Vector (V)** data type.

**Fraction length**

Specify the **Fraction length** for the **Vector (V)** data type.



**Data type override**

Specify the data type override mode. You can select one of the following options:

- **Inherit** — Inherits the data type override setting specified for the model.
- **Off** — Ignores the data type override setting specified for the model and uses the fixed-point data type you specify

This parameter appears only when you set the **Mode** parameter to **Built in** or **Fixed Point**. For more information, see “Specify Data Types Using Data Type Assistant” (Simulink).

**Accumulator Data Type Assistant Parameters****Mode**

Select how you would like to specify the data type properties of the **Accumulator** data type. You can choose:

- **Inherit** — Lets you specify a rule for inheriting a data type, for example, `Inherit: Inherit via internal rule`
- **Fixed point** — Lets you specify the fixed-point attributes of the data type.
- **Expression** — Lets you specify an expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

**Signedness**

Specify the **Signedness** for the **Accumulator** data type.

**Scaling**

Specify the **Scaling** for the **Accumulator** data type.

For more information see “Scaling” in the DSP System Toolbox User's Guide.

**Word length**

Specify the **Word length** for the **Accumulator** data type.

**Fraction length**

Specify the **Fraction length** for the **Accumulator** data type.

**Data type override**

Specify the data type override mode. You can select one of the following options:

- **Inherit** — Inherits the data type override setting specified for the model.
- **Off** — Ignores the data type override setting specified for the model and uses the fixed-point data type you specify

This parameter appears only when you set the **Mode** parameter to **Built in** or **Fixed Point**. For more information, see “Specify Data Types Using Data Type Assistant” (Simulink).

### **Output Data Type Assistant Parameters**

#### **Mode**

Select how you would like to specify the data type properties of the **Output** data type. You can choose:

- **Inherit** — Lets you specify a rule for inheriting a data type, for example, `Inherit: Inherit via internal rule`
- **Fixed point** — Lets you specify the fixed-point attributes of the data type.
- **Expression** — Lets you specify an expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

#### **Signedness**

Specify the **Signedness** for the **Output** data type.

#### **Scaling**

Specify the **Scaling** for the **Output** data type.

For more information see “Scaling” in the DSP System Toolbox User's Guide.

#### **Word length**

Specify the **Word length** for the **Output** data type.

#### **Fraction length**

Specify the **Fraction length** for the **Output** data type.

#### **Data type override**

Specify the data type override mode. You can select one of the following options:

- **Inherit** — Inherits the data type override setting specified for the model.
- **Off** — Ignores the data type override setting specified for the model and uses the fixed-point data type you specify

This parameter appears only when you set the **Mode** parameter to **Built in** or **Fixed Point**. For more information, see “Specify Data Types Using Data Type Assistant” (Simulink).

## Supported Data Types

| Port   | Supported Data Types   |
|--------|--|
| A      | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed only)</li> <li>• 8-, 16-, and 32-bit signed integers</li> </ul> |
| V      | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed only)</li> <li>• 8-, 16-, and 32-bit signed integers</li> </ul> |
| Output | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed only)</li> <li>• 8-, 16-, and 32-bit signed integers</li> </ul> |

## Version History

Introduced in R2007b

## Extended Capabilities

### C/C++ Code Generation

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

### Fixed-Point Conversion

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

If the input is fixed point, it must be a signed integer or a signed fixed point value with a power-of-two slope and zero bias.

## See Also

### Blocks

Array-Vector Add | Array-Vector Divide | Array-Vector Multiply

## Audio Device Writer

Play to sound card



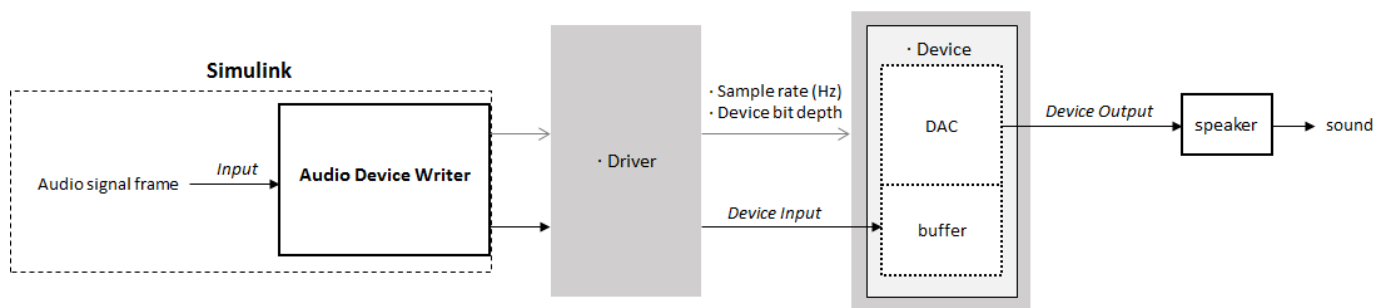
### Libraries:

Audio Toolbox / Sinks  
DSP System Toolbox / Sinks

### Description

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

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



### Data Flow of Audio Device Writer Block

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

### Ports

#### Input

**Port\_1** — Input signal

scalar | vector | matrix

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

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

## Output

**Port\_1** — Number of samples underrun  
scalar

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

## Dependencies

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

Data Types: uint32

## Parameters

### Main Tab

**Driver** — Driver used to access your audio device

DirectSound (default) | ASIO | WASAPI

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

---

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

---

- WASAPI drivers are supported for exclusive-mode only.

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

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

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

**Device** — Device used to play audio samples

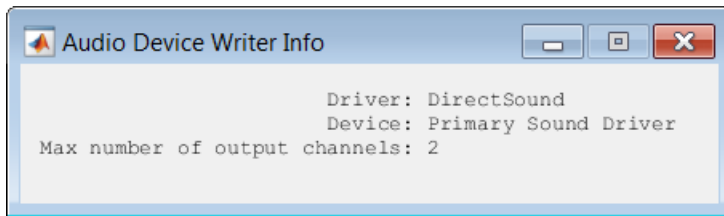
default audio device (default)

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

**Info** — View information about your audio output configuration

button

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



**Inherit sample rate from input** — Specify source of input sample rate  
on (default) | off

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

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

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

#### Dependencies

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

#### Advanced Tab

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

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

---

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

---

**Use default channel mapping** — Toggle channel mapping source  
on (default) | off

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

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

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

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

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

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

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

---

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

---

### Dependencies

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

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

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

### Block Characteristics

|                                 |  |
|---------------------------------|--|
| <b>Data Types</b>               | double   integer <sup>a</sup>   single |
| <b>Direct Feedthrough</b>       | no                                     |
| <b>Multidimensional Signals</b> | no                                     |
| <b>Variable-Size Signals</b>    | yes                                    |
| <b>Zero-Crossing Detection</b>  | no                                     |

<sup>a</sup> Supports 16- and 32-bit signed and 8-bit unsigned integers.

### Version History

Introduced in R2016a

### Extended Capabilities

#### C/C++ Code Generation

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

The following code generation limitations apply:

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

**See Also**

Binary File Reader | `audioDeviceWriter` | `sound` | `audioplayer`

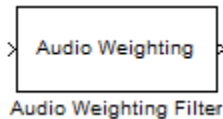
**Topics**

“How To Run a Generated Executable Outside MATLAB”



# Audio Weighting Filter

Design audio weighting filter



## Compatibility

---

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

---

## Library

Filtering / Filter Designs

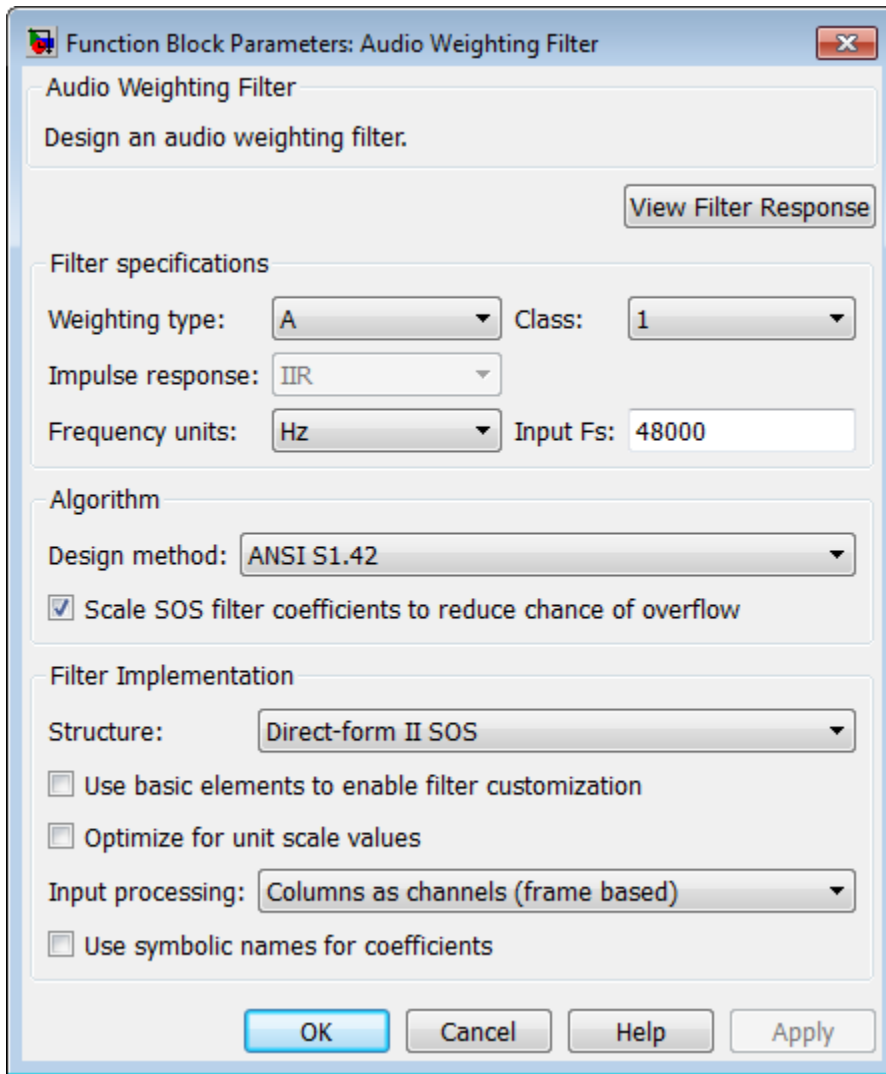
dspfdesign

## Description

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

## Dialog Box

See “Audio Weighting Filter Design — Main Pane” on page 5-597 for more information about the parameters of this block. The **Data Types** and **Code** panes are not available for blocks in the DSP System Toolbox Filter Designs library.



### View Filter Response

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

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

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

### Filter Specifications

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

**Weighting type**

The weighting type defines the frequency response of the filter. The valid weighting types for this filter are A, C, C-message, ITU-R 468-4, and ITU-T 0.41.

**Class**

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

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

**Impulse response**

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

**Frequency units**

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

**Input sample rate**

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

**Algorithm****Design Method**

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

**Scale SOS filter coefficients to reduce chance of overflow**

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

**Filter Implementation****Structure**

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

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

### Use basic elements to enable filter customization

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

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

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

### Optimize for unit-scale values

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

### Input processing

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

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

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

### Use variable names for coefficients

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

## Supported Data Types

| Port   | Supported Data Types   |
|--------|--|
| Input  | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> </ul> |
| Output | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> </ul> |

## Version History

Introduced in R2011b

## References

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

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

## Extended Capabilities

### C/C++ Code Generation

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

## See Also

### Functions

filterBuilder

### Tools

FVTool

# Autocorrelation

Autocorrelation of  $N$ -D array



**Libraries:**  
DSP System Toolbox / Statistics

## Description

The Autocorrelation block computes the autocorrelation along the first dimension of an  $N$ -D input array. The computation can be done in the time domain or frequency domain. You can specify the domain through the **Computation domain** parameter. In the time domain, the input signal is convolved with its time-reversed complex conjugate. In the frequency domain, the block computes the autocorrelation by taking the Fourier transform of the input signal, multiplying the Fourier transform with its conjugate, and computing the inverse Fourier transform of the product. In this domain, depending on the input length, the block can require fewer computations. For information on these two computation methods, see “Algorithms” on page 2-91.

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

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

## Ports

### Input

**Port\_1** — Data input

vector | matrix |  $N$ -D array

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

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fixed point

### Output

**Port\_1** — Autocorrelated output

vector | matrix |  $N$ -D array

Autocorrelated output of the data input.

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

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fixed point

## Parameters

### Main Tab

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

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

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

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

### Dependencies

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

**Scaling** — Scaling of the output  
None (default) | Biased | Unbiased | Unity at zero-lag

Scaling applied to the output.

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

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

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

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

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

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

**Computation domain** — Domain in which the block computes the autocorrelation  
Time (default) | Frequency

- **Time** — Computes the convolutions in the time domain, which minimizes the memory usage.
- **Frequency** — Computes the autocorrelation in frequency domain. For more information, see “Algorithms” on page 2-91.

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

### Data Types Tab

---

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

---

#### **Rounding mode** — Method of rounding operation

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

Specify the rounding mode for fixed-point operations as one of the following:

- Floor
- Ceiling
- Convergent
- Nearest
- Round
- Simplest
- Zero

For more details, see rounding mode.

---

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

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

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

---

#### **Saturate on integer overflow** — Method of overflow action

off (default) | on

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

---

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

- **Product output** data type is `Inherit`: `Inherit via internal rule`.
- **Accumulator** data type is `Inherit`: `Inherit via internal rule`.

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

---



**Product output** — Product output data type

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

**Product output** specifies the data type of the output of a product operation in the Autocorrelation block. For more information on the product output data type, see “Multiplication Data Types” and the ‘Fixed-Point Conversion’ section in “Extended Capabilities” on page 2-0 .

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

Alternatively, you can set the **Product output** data type by using the **Data Type Assistant**. To use

the assistant, click the **Show data type assistant** button .

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

**Accumulator** — Accumulator data type

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

**Accumulator** specifies the data type of the output of an accumulation operation in the Autocorrelation block. For illustrations on how to use the accumulator data type in this block, see the ‘Fixed-Point Conversion’ section in “Extended Capabilities” on page 2-0 .

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

Alternatively, you can set the **Accumulator** data type by using the **Data Type Assistant**. To use the

assistant, click the **Show data type assistant** button .


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

**Output** — Output data type

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

**Output** specifies the data type of the output of the Autocorrelation block. For more information on the output data type, see the 'Fixed-Point Conversion' section in “Extended Capabilities” on page 2-0 .

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

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

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

**Output Minimum** — Minimum value the block can output  
[] (default) | scalar

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

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

**Output Maximum** — Maximum value block can output  
[] (default) | scalar

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

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

**Lock data type settings against changes by the fixed-point tools** — Prevent fixed-point tools from overriding data types  
off (default) | on

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

## Block Characteristics

|                                 |  |
|---------------------------------|--|
| <b>Data Types</b>               | double   single   base integer   fixed point |
| <b>Multidimensional Signals</b> | No   |

|                              |    |
|------------------------------|----|
| <b>Variable-Size Signals</b> | No |
|------------------------------|----|

## More About

### Autocorrelation

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

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

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

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

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

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

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

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

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

## Algorithms

### Time-Domain Computation

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

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

$$y_{i,j} = \sum_{k=0}^{M-l-1} u_{k,j}^* u_{(k+i),j} \quad 0 \leq i \leq l$$

- $y_{0,j}$  is the zero-lag element in the  $j$ th column of the input.
- $i$  is the index of the lag.
- $j$  is the index of the input data column.
- $*$  denotes the complex conjugate.
- $M$  is the number of elements in each column.

- $l$  is the maximum positive lag for autocorrelation. When you choose to compute the autocorrelation with all nonnegative lags,  $l=M-1$ . Otherwise,  $l$  is the maximum nonnegative integer lag value specified.
- $u$  is an  $M$ -by- $N$  input matrix.

### Frequency-Domain Computation

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

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

## Version History

Introduced before R2006a

### Extended Capabilities

#### C/C++ Code Generation

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

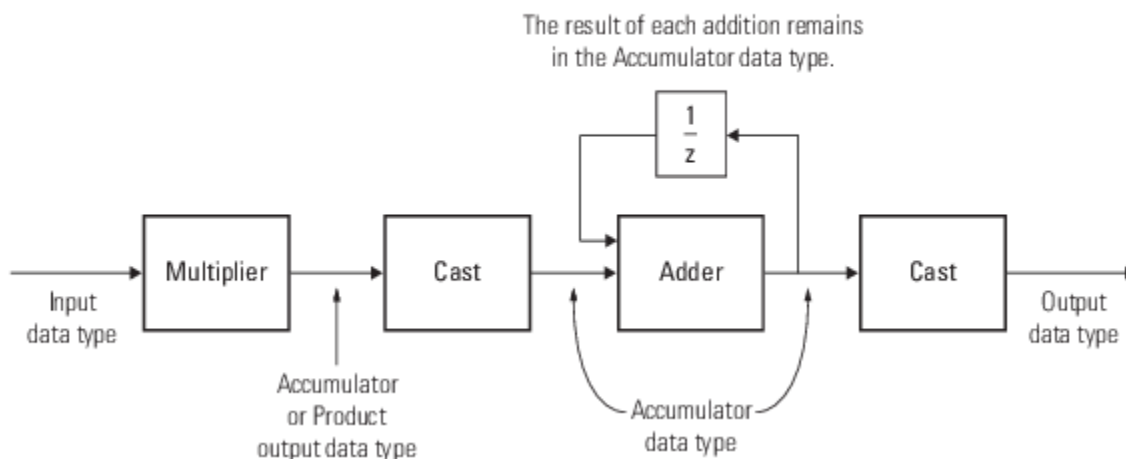
Generated code relies on the `memcpy` or `memset` function (`string.h`) under certain conditions.

#### Fixed-Point Conversion

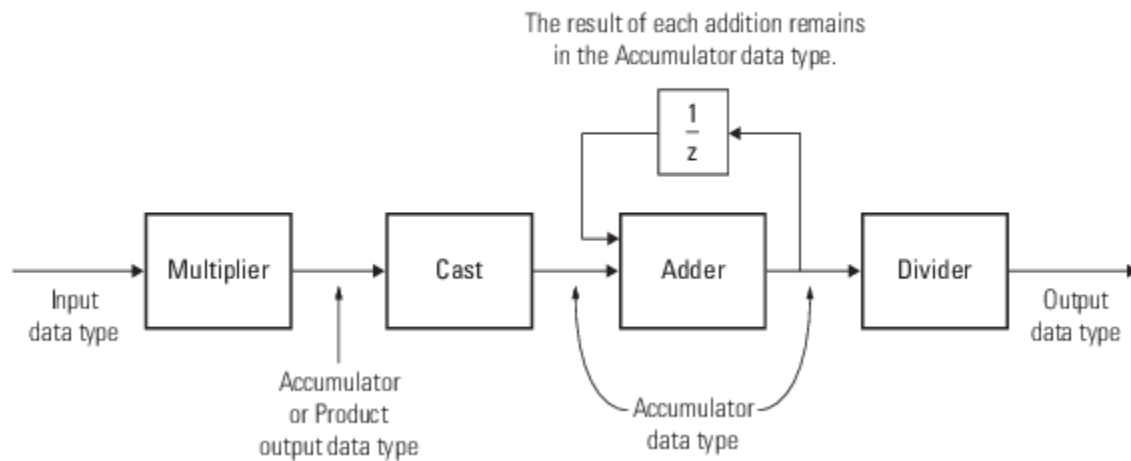
Design and simulate fixed-point systems using Fixed-Point Designer™.

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

#### Signal flow when Scaling is "None"



### Signal flow when Scaling is other than "None"



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

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

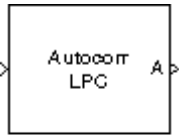
## See Also

### Blocks

Correlation

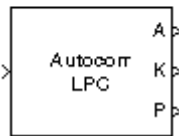
## Autocorrelation LPC

Determine coefficients of Nth-order forward linear predictors



### Libraries:

DSP System Toolbox / Estimation / Linear Prediction



## Description

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

The Autocorrelation LPC block can output the prediction error for each channel as polynomial coefficients, reflection coefficients, or both. The block can also output the prediction error power for each channel.

## Ports

### Input

#### Input 1 — Input array

unoriented vector | column vector | matrix

Specify the input *u* as an unoriented vector, column vector, or a matrix. Row vectors are not valid inputs. The block treats all *M*-by-*N* matrix inputs as *N* channels of length *M*.

Data Types: single | double

### Output

#### A — Polynomial coefficients

column vector | matrix

Polynomial coefficients generated when you set the **Output(s)** parameter to A or A and K. For each input channel, port A outputs an (*N*+1)-by-1 column vector  $a = [1 \ a_2 \ a_3 \ \dots \ a_{N+1}]^T$ , containing the coefficients of an *N*<sup>th</sup>-order moving average (MA) linear process that predicts the next value,  $\hat{u}_{M+1}$ , in the input time-series.

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

**Dependencies**

To enable port A, set **Output(s)** to A or A and K.

Data Types: single | double

**K** — Reflection coefficients

column vector | matrix

Reflection coefficients generated when **Output(s)** is set to K or A and K. For each input channel, port K outputs a length- $N$  column vector whose elements are the prediction error reflection coefficients.

**Dependencies**

To enable port K, set **Output(s)** to A or A and K.

Data Types: single | double

**P** — Prediction error power

vector

Prediction error power output at port P as a vector whose length is the number of input channels.

**Dependencies**

To enable port P, select the **Output prediction error power (P)** parameter.

Data Types: single | double

**Parameters****Output(s)** — Type of prediction coefficients

A (default) | A and K | K

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

When you set **Output(s)** to A and K, the block enables port A and K and each port outputs its respective set of prediction coefficients for each channel.

**Output prediction error power (P)** — Output prediction error power

off (default) | on

Select this parameter to enable the output port P, which outputs the output prediction error power.

**Inherit prediction order from input dimensions** — Inherit prediction order from input dimensions

off (default) | on

Select this parameter to inherit the prediction order  $N$  from the input dimensions.

**Prediction order (N)** — Prediction order

1 (default) | scalar

Specify the prediction order  $N$ . Note that  $N$  must be a scalar with a value less than the length of the input channels or the block produces an error.

### Dependencies

This parameter appears only when you do not select the **Inherit prediction order from input dimensions** parameter.

### Block Characteristics

|                                 |                 |
|---------------------------------|-----------------|
| <b>Data Types</b>               | double   single |
| <b>Direct Feedthrough</b>       | no              |
| <b>Multidimensional Signals</b> | no              |
| <b>Variable-Size Signals</b>    | no              |
| <b>Zero-Crossing Detection</b>  | no              |

### Algorithms

The Autocorrelation LPC block computes the least squares solution to

$$\min_{i \in \mathfrak{R}^n} \|U\tilde{a} - b\|$$

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

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

Solving the least squares problem via the normal equations

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

leads to the system of equations



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

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

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

## Version History

Introduced before R2006a

## References

- [1] Haykin, S. *Adaptive Filter Theory*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1996.
- [2] Ljung, L. *System Identification: Theory for the User*. Englewood Cliffs, NJ: Prentice Hall, 1987. Pgs. 278-280.
- [3] Proakis, J. and D. Manolakis. *Digital Signal Processing*. 3rd ed. Englewood Cliffs, NJ: Prentice-Hall, 1996.

## Extended Capabilities

### C/C++ Code Generation

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

Generated code relies on the memcpy or memset function (`string.h`) under certain conditions.

## See Also

### Functions

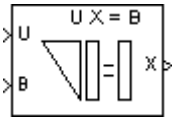
`lpc`

### Blocks

Autocorrelation | Levinson-Durbin | Yule-Walker Method

## Backward Substitution

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



### Libraries:

DSP System Toolbox / Math Functions / Matrices and Linear Algebra / Linear System Solvers

### Description

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

- $U$  is the upper triangular  $M$ -by- $M$  matrix input to the **U** port.
- $B$  is the  $M$ -by- $N$  matrix input to the **B** port.
- $X$  is the  $M$ -by- $N$  output matrix and is the solution of the system of equations.

The block does not check the rank of the inputs. The block uses only the elements in the *upper triangle* of input  $U$  and ignores the lower elements. When you select the **Input U is unit-upper triangular** check box, the block assumes the elements on the diagonal of  $U$  are 1s. This is useful when matrix  $U$  is the result of another operation, such as an LDL decomposition, that uses the diagonal elements to represent the  $D$  matrix.

### Ports

#### Input

**U** — Input upper triangular matrix  
 $M$ -by- $M$  matrix

Specify the input upper triangular  $M$ -by- $M$  matrix  $U$  through this port. Inputs  $U$  and  $B$  must have the same number of rows.

When the input  $U$  is fixed-point, it is signed only.

Data Types: single | double | int8 | int16 | int32 | fixed point

**B** —  $B$  matrix  
 $M$ -by- $N$  matrix |  $M$ -by-1 vector

Specify the  $B$  matrix in the  $UX = B$  equation as an  $M$ -by- $N$  matrix or an  $M$ -by-1 vector.

When you specify an  $M$ -by-1 vector, the block treats the length- $M$  vector input at port **B** as an  $M$ -by-1 matrix. Inputs  $U$  and  $B$  must have the same number of rows.

When the input  $B$  is fixed-point, it is signed only.

Data Types: single | double | int8 | int16 | int32 | fixed point

## Output

**X** — Solution of system of equations

*M*-by-*N* matrix | *M*-by-1 matrix

The block returns the output as an *M*-by-*N* matrix or an *M*-by-1 vector. The size of the *X* matrix is same as the size of the *B* matrix.

When the inputs are fixed-point, the output matrix *X* is signed only fixed-point.

Data Types: single | double | int8 | int16 | int32 | fixed point

## Parameters

### Main Tab

**Input U is unit-upper triangular** — Diagonal elements of input matrix have a value of 1

off (default) | on

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

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

**Diagonal of complex input U is real** — Diagonal elements of complex input are real

off (default) | on

Select to optimize simulation speed when the diagonal elements of complex input *U* are real.

---

**Note** When *U* is a complex fixed-point signal, you must select either **Input U is unit-upper triangular** or **Diagonal of complex input U is real**. When either of these options are selected, the block ignores any imaginary part of the diagonal of *U*.

---

### Dependencies

This parameter is visible only when **Input U is unit-upper triangular** is not selected.

### Data Types Tab

---

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

---

**Rounding mode** — Method of rounding operation

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

Specify the rounding mode for fixed-point operations as one of the following:

- Floor
- Ceiling
- Convergent
- Nearest
- Round
- Simplest
- Zero

For more details, see rounding mode.

**Saturate on integer overflow** — Method of overflow action

off (default) | on

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

**Product output** — Product output data type

Inherit: Inherit via internal rule (default) | Inherit: Same as first input | `fixdt(1,16,0)`

**Product output** specifies the data type of the output of a product operation in the Backward Substitution block. For more information on the product output data type, see “Fixed Point” on page 2-102 and “Multiplication Data Types”.

- `Inherit: Inherit via internal rule` — The block inherits the product output data type based on an internal rule. For more information on this rule, see “Inherit via Internal Rule”.
- `Inherit: Same as first input` — The block specifies the product output data type to be the same as the first input data type.
- `fixdt(1,16,0)` — The block specifies a signed, binary-point, scaled, fixed-point data type with a word length of 16 bits and a fraction length of 0.

Alternatively, you can set the **Product output** data type by using the **Data Type Assistant**. To use

the assistant, click the **Show data type assistant** button .

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


**Accumulator** — Accumulator data type

Inherit: Inherit via internal rule (default) | Inherit: Same as product output | Inherit: Same as first input | `fixdt(1,16,0)`

**Accumulator** specifies the data type of the output of an accumulation operation in the Backward Substitution block. For illustrations on how to use the accumulator data type in this block, see the “Fixed Point” on page 2-102 section.

- `Inherit: Inherit via internal rule` — The block inherits the accumulator data type based on an internal rule. For more information on this rule, see “Inherit via Internal Rule”.

- **Inherit: Same as product output** — The block specifies the accumulator data type to be the same as the product output data type.
- **Inherit: Same as first input** — The block specifies the accumulator data type to be the same as the first input data type.
- **fixdt(1,16,0)** — The block specifies a signed, binary-point scaled, fixed-point data type with a word length of 16 bits and a fraction length of 0.

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


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

#### **Output** — Output data type

**Inherit: Same as first input** (default) | **fixdt(1,16,0)**

**Output** specifies the data type of the output of the Backward Substitution block. For more information on the output data type, see the “Fixed Point” on page 2-102 section.

- **Inherit: Same as first input** — The block specifies the output data type to be the same as the first input data type.
- **fixdt(1,16,0)** — The block specifies a signed, binary-point scaled, fixed-point data type with a word length of 16 bits and a fraction length of 0.

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

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

#### **Output Minimum** — Minimum value the block can output

**[]** (default) | **scalar**

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

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

#### **Output Maximum** — Maximum value block can output

**[]** (default) | **scalar**

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

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

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

**off** (default) | **on**

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

## Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | double   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no                                      |
| <b>Multidimensional Signals</b> | no                                      |
| <b>Variable-Size Signals</b>    | no                                      |
| <b>Zero-Crossing Detection</b>  | no                                      |

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

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

Generated code relies on the memcpy or memset function (string.h) under certain conditions.

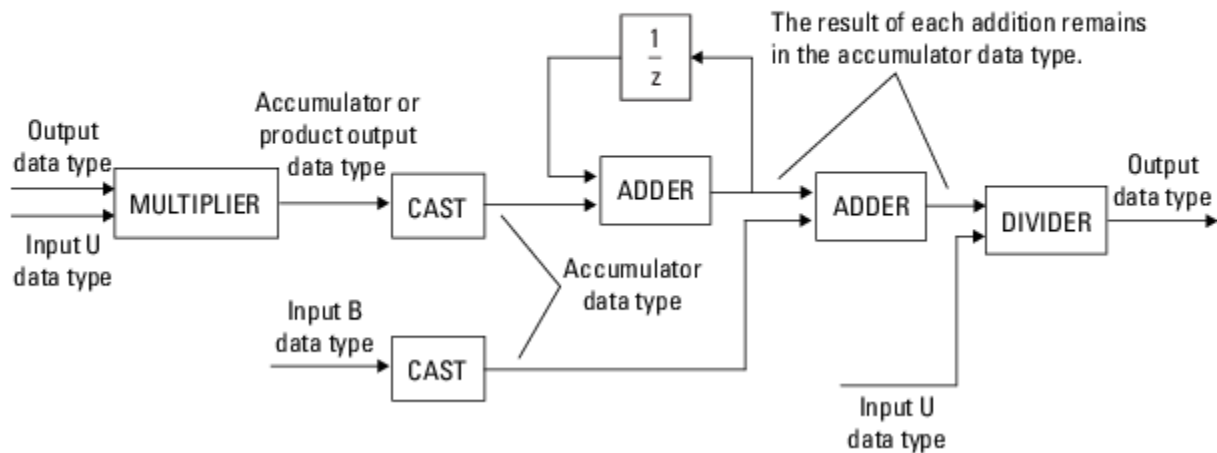
### Fixed-Point Conversion

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

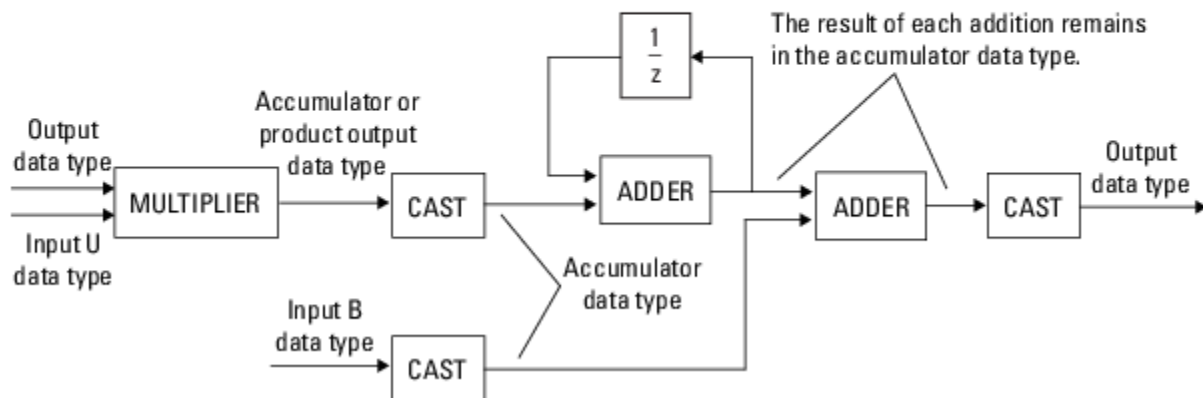
If the input is fixed point, it must be a signed integer or a signed fixed point value with a power-of-two slope and zero bias.

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

**When input  $U$  is not unit-upper triangular:**



**When input  $U$  is unit-upper triangular:**



You can set the product output, accumulator, and output data types in the block dialog box.

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

## See Also

### Blocks

Cholesky Solver | Forward Substitution | LDL Solver | Levinson-Durbin | LU Solver | QR Solver

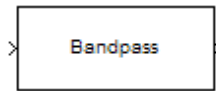
**Topics**

“Linear System Solvers”



# Bandpass Filter

Design bandpass filter



## Libraries:

DSP System Toolbox / Filtering / Filter Designs

## Description

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

This block supports SIMD code generation. For details, see “Code Generation” on page 2-113.

## Ports

### Input

**Port\_1** — Input signal

scalar | vector | matrix

Input signal to filter, specified as a scalar, vector, or matrix.

Data Types: `single` | `double`

### Output

**Port\_1** — Filtered output signal

scalar | vector | matrix

Filtered output signal, specified as a scalar, vector, or matrix.

Data Types: `single` | `double`

## Parameters

**View Filter Response** — Open Filter Visualization Tool

button

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

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

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

**Impulse response** — FIR or IIR filter

FIR (default) | IIR

Choose to implement an FIR or IIR filter.

---

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

---

**Order mode** — Mode of specifying filter order

Minimum (default) | Specify

Select Minimum to have the block implement a filter with minimum order. When you select Specify, you must enter the filter order using the **Order** parameter.

---

**Tip** When you set the **Impulse response** to IIR, you can specify different numerator and denominator orders. To specify a different denominator order, select the **Denominator order** check box.

---

**Order** — Filter order

20 (default) | positive integer

Specify the filter order as a positive integer.

**Dependencies**

To enable this parameter, set **Order mode** to Specify.

**Denominator order** — Specify denominator order

off (default) | on

Select this check box to specify a different denominator order. When you select this check box, you can specify the denominator order as a positive integer in the resulting text box.

**Dependencies**

To enable this parameter, set the **Impulse response** to IIR and the **Order mode** to Specify.

**Filter type** — Type of filter

Single-rate (default) | Decimator | Interpolator | Sample-rate converter

Select the type of filter to implement. Your choice determines the type of filter and the design methods and structures that are available to implement your filter.

**Dependencies**

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

**Decimation Factor** — Decimation factor

2 (default) | positive integer

Specify the decimation factor as a positive integer.

**Dependencies**

To enable this parameter, set the **Filter type** to Decimator or Sample-rate converter.

**Interpolation Factor** — Interpolation factor

2 (default) | positive integer

Specify the interpolation factor as a positive integer.

**Dependencies**

To enable this parameter, set the **Filter type** to Interpolator or Sample-rate converter.

**Frequency constraints** — Frequency response constraints

Passband and stopband edges (default) | Passband edges | Half power (3dB) frequencies | Half power (3dB) frequencies and passband width | Half power (3dB) frequencies and stopband width | Cutoff (6dB) frequencies

When you set the **Order mode** to Specify, this parameter allows you to choose the filter features that the block uses to define the frequency response characteristics. Depending on the **Impulse response** you choose, you can set the **Frequency constraints** to one of:

- **Passband and stopband edges** — Specify the frequencies for the edges for the stop- and passbands.
- **Passband edges** — For IIR filters, define the filter by specifying frequencies for the edges of the passband.
- **Stopband edges** — For IIR filters, define the filter by specifying frequencies for the edges of the stopbands.
- **Half power (3dB) frequencies** — For IIR filters, define the filter response by specifying the locations of the 3 dB points. The 3 dB point is the frequency for the point three decibels below the passband value.
- **Half power (3dB) frequencies and passband width** — For IIR filters, define the filter by specifying frequencies for the 3 dB points in the filter response and the width of the passband.
- **Half power (3dB) frequencies and stopband width** — For IIR filters, define the filter by specifying frequencies for the 3 dB points in the filter response and the width of the stopband.
- **Cutoff (6dB) frequencies** — For FIR filters, define the filter response by specifying the locations of the 6 dB points. The 6 dB point is the frequency for the point 6 dB below the passband value.

**Dependencies**

To enable this parameter, set the **Order mode** to Specify. The available **Frequency constraints** will depend on whether the **Impulse response** is FIR or IIR.

**Frequency units** — Frequency units

Normalized (0 to 1) (default) | Hz | kHz | MHz | GHz

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select **Normalized (0 to 1)** to enter frequencies in normalized form. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—Hz, kHz, MHz, or GHz.

**Input sample rate** — Input sample rate

2 (default) | positive scalar

$F_s$ , specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well.

**Dependencies**

To enable this parameter, set **Filter type** to Single-rate, Decimator, or Sample-rate converter and **Frequency units** to one of the unit options (Hz, kHz, MHz, or GHz).

**Output sample rate** — Output sample rate

2 (default) | positive scalar

When you design an interpolator,  $F_s$  represents the sampling frequency at the filter output.

**Dependencies**

To enable this parameter, set **Filter type** to Interpolator and **Frequency units** to one of the unit options (Hz, kHz, MHz, or GHz).

**Stopband frequency 1** — Frequency at edge of end of first stopband

0.35 (default) | positive scalar

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

**Passband frequency 1** — Frequency at edge of start of passband

0.45 (default) | positive scalar

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

**Passband frequency 2** — Frequency at edge of end of passband

.55 (default) | positive scalar

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

**Stopband frequency 2** — Frequency at edge of start of second stopband

.65 (default) | positive scalar

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

**Half power (3dB) frequency 1** — Lower frequency 3 dB point

.4 (default) | positive scalar

Specify the lower frequency 3 dB point as a positive scalar between zero and one.

**Dependencies**

To enable this parameter, set **Impulse response** to IIR, **Order mode** to Specify, and **Frequency constraints** to Half power (3dB) frequencies, Half power (3dB) frequencies and passband width, or Half power (3dB) frequencies and stopband width.

**Half power (3dB) frequency 2** — Higher frequency 3 dB point

.6 (default) | positive scalar

Specify the higher frequency 3 dB point as a positive scalar between zero and one.

**Dependencies**

To enable this parameter, set **Impulse response** to IIR, **Order mode** to Specify, and **Frequency constraints** to Half power (3dB) frequencies, Half power (3dB) frequencies and passband width, or Half power (3dB) frequencies and stopband width.

**Cutoff (6dB) frequency 1** — Lower frequency 6 dB point

.4 (default) | positive scalar

Specify the lower frequency 6 dB point as a positive scalar between zero and one.

**Dependencies**

To enable this parameter, set **Frequency constraints** to Cutoff (6dB) frequencies.

**Cutoff (6dB) frequency 2** — Higher frequency 6 dB point

.6 (default) | positive scalar

Specify the higher frequency 6 dB point as a positive scalar between zero and one.

**Dependencies**

To enable this parameter, set **Frequency constraints** to Cutoff (6dB) frequencies.

**Passband width** — Passband width

.15 (default) | positive scalar

Specify the width of the passband as a positive scalar, in units corresponding to the **Frequency units** parameter.

**Dependencies**

To enable this parameter, set **Frequency constraints** to Half power (3dB) frequencies and passband width.

**Stopband width** — Width of stopband

.25 (default) | positive scalar

Specify the width of the stopband as a positive scalar, in units corresponding to the **Frequency units** parameter.

**Dependencies**

To enable this parameter, set **Frequency constraints** to Half power (3dB) frequencies and stopband width.

**Magnitude constraints** — Magnitude constraints

Unconstrained (default) | Constrained bands | Passband ripple | Passband ripple and stopband attenuation | Stopband attenuation

Specify the magnitude constraints for the filter design.

**Dependencies**

To enable this parameter, set **Order mode** to `Specify`. The available options depend on the value of the **Frequency constraints** parameters.

**Magnitude units** — Units for magnitude specifications  
dB (default) | `Linear` | `Squared`

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

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

**Dependencies**

To enable this parameter, set **Order mode** to `Minimum`.

**Stopband attenuation 1** — Filter attenuation in first stopband  
60 (default) | real-valued positive scalar

Enter the filter attenuation in the first stopband in the units you choose for **Magnitude units**. Values must be real, positive scalars. If you are specifying values in linear units, they must be smaller than 1.

**Dependencies**

To enable this parameter, set the **Order mode** to `Minimum`.

**Passband ripple** — Allowable filter ripple in passband  
1 (default) | real-valued positive scalar

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**. Values must be real, positive scalars. If you are specifying values in linear units, they must be smaller than 1.

**Dependencies**

To enable this parameter, set the **Order mode** to `Minimum`.

**Stopband attenuation 2** — Filter attenuation in second stopband  
60 (default) | real-valued positive scalar

Enter the filter attenuation in the second stopband in the units you choose for **Magnitude units**. Values must be real, positive scalars. If you are specifying values in linear units, they must be smaller than 1.

**Dependencies**

To enable this parameter, set the **Order mode** to `Minimum`.

**Design method** — Filter design method  
`Equiripple` (default) | `Kaiser window` | `Butterworth` | `Chebyshev type I` | `Chebyshev type II` | `Elliptic`

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

**Scale SOS filter coefficients to reduce chance of overflow** — Scale filter coefficients  
on (default) | off

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

**Dependencies**

To enable this parameter, set **Impulse response** to IIR.

**Density factor** — Density factor  
16 (default) | positive scalar

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

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 16 represents a reasonable balance between the accurate approximation to the ideal filter and the time to design the filter.

**Dependencies**

To enable this parameter, set **Impulse response** to FIR and **Design method** to Equiripple.

**Phase constraint** — Phase constraint  
Linear (default) | Maximum | Minimum

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

**Dependencies**

To enable this parameter, set **Impulse response** to FIR and **Design method** to Equiripple.

**Match exactly** — Match passband, stopband, or both  
Stopband (default) | Passband | Both

Specifies that the resulting filter design matches either the passband, stopband, or both bands.

**Dependencies**

To enable this parameter, set **Impulse response** to IIR.

**Minimum order** — Minimum filter order  
Any (default) | Even | Odd

When you select this parameter, the design method determines and designs a minimum order filter to meet your specifications.

**Dependencies**

To enable this parameter, set **Impulse response** to FIR and **Order mode** to Minimum.

**Structure** — Filter structure  
Direct-form FIR (default) | Direct-form FIR transposed | Direct-form symmetric FIR | Cascade minimum-multiplier allpass | Cascade wave digital filter allpass |

Direct-form I SOS | Direct-form I transposed SOS | Direct-form II SOS | Direct-form II transposed SOS

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

**Use basic elements to enable filter customization** — Implement filter with basic Simulink blocks  
off (default) | on

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

The high-level implementation provides better compatibility across various filter structures, especially filters that would contain algebraic loops when constructed using basic elements.

#### Dependencies

When you select this check box, the block enables the following optimization parameters:

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

**Optimize for unit-scale values** — Optimize unit scale values  
off (default) | on

Select this check box to scale unit gains between sections in SOS filters.

#### Dependencies

To enable this parameter, set **Impulse response** to IIR.

**Rate options** — Enforce single-rate or allow multirate processing  
Enforce single-rate processing (default) | Allow multirate processing

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

- **Enforce single-rate processing** — When you select this option, the block maintains the sample rate of the input.
- **Allow multirate processing** — When you select this option, the block adjusts the rate at the output to accommodate an increased or reduced number of samples.

#### Dependencies

To enable this parameter, set the **Impulse response** to FIR and set **Filter type** to a multirate filter.

**Use variable names for coefficients** — Specify coefficients with MATLAB variables  
off (default) | on



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

## Block Characteristics

|                                 |                 |
|---------------------------------|-----------------|
| <b>Data Types</b>               | double   single |
| <b>Multidimensional Signals</b> | No              |
| <b>Variable-Size Signals</b>    | No              |

## Version History

Introduced in R2006b

## Extended Capabilities

### C/C++ Code Generation

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

The Bandpass Filter block supports SIMD code generation using Intel AVX2 technology under these conditions:

- **Impulse response** is set to FIR.
- **Filter type** is set to Single-rate.
- **Structure** is set to Direct-form FIR or Direct-form FIR transposed.
- **Use basic elements to enable filter customization** parameter is not selected.
- **Input processing** is set to Columns as channels (frame based).
- Input signal has a data type of single or double.

The SIMD technology significantly improves the performance of the generated code.

## See Also

### Blocks

Tunable Bandpass IIR Filter | Bandstop Filter

### Functions

filterBuilder | fdesign.bandstop

# Bandstop Filter

Design bandstop filter

**Libraries:**

DSP System Toolbox / Filtering / Filter Designs

## Description

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

This block supports SIMD code generation. For details, see “Code Generation” on page 2-122.

## Ports

### Input

**Port\_1** — Input signal

scalar | vector | matrix

Input signal to filter, specified as a scalar, vector, or matrix.

Data Types: `single` | `double`

### Output

**Port\_1** — Filtered output signal

scalar | vector | matrix

Filtered output signal, specified as a scalar, vector, or matrix.

Data Types: `single` | `double`

## Parameters

**View Filter Response** — Open Filter Visualization Tool

button

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

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

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

**Impulse response** — FIR or IIR filter

FIR (default) | IIR

Specify whether the block implements an FIR or IIR filter.

---

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

---

**Order mode** — Mode of specifying filter order

Minimum (default) | Specify

Select Minimum to have the block implement a filter with minimum order. When you select Specify, you must enter the filter order using the **Order** parameter.

---

**Tip** When you set the **Impulse response** to IIR, you can specify different numerator and denominator orders. To specify a different denominator order, select the **Denominator order** check box.

---

**Order** — Filter order

20 (default) | positive integer

Specify the filter order as a positive integer.

**Dependencies**

To enable this parameter, set **Order mode** to Specify.

**Denominator order** — Specify denominator order

off (default) | on

Select this check box to specify a different denominator order. When you select this check box, you can specify the denominator order as a positive integer in the resulting text box.

**Dependencies**

To enable this parameter, set the **Impulse response** to IIR and the **Order mode** to Specify.

**Filter type** — Type of filter

Single-rate (default) | Decimator | Interpolator | Sample-rate converter

Select the type of filter to implement. Your choice determines the type of filter and the design methods and structures that are available to implement your filter.

**Dependencies**

- This parameter applies only when you set **Impulse response** to FIR.
- Selecting Decimator or Interpolator activates the **Decimation Factor** or the **Interpolation Factor** options respectively.
- Selecting Sample-rate converter activates both factors.

**Decimation Factor** — Decimation factor

2 (default) | positive integer

Specify the decimation factor as a positive integer.

#### Dependencies

To enable this parameter, set the **Filter type** to Decimator or Sample-rate converter.

#### Interpolation Factor — Interpolation factor

2 (default) | positive integer

Specify the interpolation factor as a positive integer.

#### Dependencies

To enable this parameter, set the **Filter type** to Interpolator or Sample-rate converter.

#### Frequency constraints — Frequency response constraints

Passband and stopband edges (default) | Passband edges | Half power (3dB) frequencies | Half power (3dB) frequencies and passband width | Half power (3dB) frequencies and stopband width | Cutoff (6dB) frequencies

When you set the **Order mode** to Specify, this parameter allows you to choose the filter features that the block uses to define the frequency response characteristics. Depending on the **Impulse response** you choose, you can set the **Frequency constraints** to one of:

- **Passband and stopband edges** — Specify the frequencies for the edges for the stop- and passbands.
- **Passband edges** — For IIR filters, define the filter by specifying frequencies for the edges of the passband.
- **Stopband edges** — For IIR filters, define the filter by specifying frequencies for the edges of the stopbands.
- **Half power (3dB) frequencies** — For IIR filters, define the filter response by specifying the locations of the 3 dB points. The 3 dB point is the frequency for the point three decibels below the passband value.
- **Half power (3dB) frequencies and passband width** — For IIR filters, define the filter by specifying frequencies for the 3 dB points in the filter response and the width of the passband.
- **Half power (3dB) frequencies and stopband width** — For IIR filters, define the filter by specifying frequencies for the 3 dB points in the filter response and the width of the stopband.
- **Cutoff (6dB) frequencies** — For FIR filters, define the filter response by specifying the locations of the 6 dB points. The 6 dB point is the frequency for the point 6 dB below the passband value.

#### Dependencies

To enable this parameter, set the **Order mode** to Specify. The available **Frequency constraints** will depend on whether the **Impulse response** is FIR or IIR.

#### Frequency units — Frequency units

Normalized (0 to 1) (default) | Hz | kHz | MHz | GHz

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select **Normalized (0 to 1)** to enter frequencies in normalized form. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—Hz, kHz, MHz, or GHz.

**Input sample rate** — Input sample rate

2 (default) | positive scalar

$F_s$ , specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well.

**Dependencies**

To enable this parameter, set **Filter type** to Single-rate, Decimator, or Sample-rate converter and **Frequency units** to one of the unit options (Hz, kHz, MHz, or GHz).

**Output sample rate** — Output sample rate

2 (default) | positive scalar

When you design an interpolator,  $F_s$  represents the sampling frequency at the filter output rather than the filter input.

**Dependencies**

To enable this parameter, set **Filter type** to Interpolator and **Frequency units** to one of the unit options (Hz, kHz, MHz, or GHz).

**Passband frequency 1** — Frequency at edge of end of first passband

0.35 (default) | positive scalar

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

**Stopband frequency 1** — Frequency at edge of start of stopband

0.45 (default) | positive scalar

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

**Stopband frequency 2** — Frequency at edge of end of stopband

0.55 (default) | positive scalar

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

**Passband frequency 2** — Frequency at edge of start of second passband

0.65 (default) | positive scalar

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

**Half power (3dB) frequency 1** — Lower frequency 3 dB point

0.4 (default) | positive scalar

Specify the lower frequency 3 dB point as a positive scalar between zero and one.

**Dependencies**

To enable this parameter, set **Impulse response** to IIR, **Order mode** to Specify, and **Frequency constraints** to Half power (3dB) frequencies, Half power (3dB) frequencies and passband width, or Half power (3dB) frequencies and stopband width.

**Half power (3dB) frequency 2** — Higher frequency 3 dB point

0.6 (default) | positive scalar

Specify the higher frequency 3 dB point as a positive scalar between zero and one.

**Dependencies**

To enable this parameter, set **Impulse response** to IIR, **Order mode** to Specify, and **Frequency constraints** to Half power (3dB) frequencies, Half power (3dB) frequencies and passband width, or Half power (3dB) frequencies and stopband width.

**Cutoff (6dB) frequency 1** — Lower frequency 6 dB point

0.4 (default) | positive scalar

Specify the lower frequency 6 dB point as a positive scalar between zero and one.

**Dependencies**

To enable this parameter, set **Frequency constraints** to Cutoff (6dB) frequencies.

**Cutoff (6dB) frequency 2** — Higher frequency 6 dB point

0.6 (default) | positive scalar

Specify the higher frequency 6 dB point as a positive scalar between zero and one.

**Dependencies**

To enable this parameter, set **Frequency constraints** to Cutoff (6dB) frequencies.

**Passband width** — Passband width

0.25 (default) | positive scalar

Specify the width of the passband as a positive scalar, in units corresponding to the **Frequency units** parameter.

**Dependencies**

To enable this parameter, set **Frequency constraints** to Half power (3dB) frequencies and passband width.

**Stopband width** — Width of stopband

0.15 (default) | positive scalar

Specify the width of the stopband as a positive scalar, in units corresponding to the **Frequency units** parameter.

**Dependencies**

To enable this parameter, set **Frequency constraints** to Half power (3dB) frequencies and stopband width.

**Magnitude constraints** — Magnitude constraints

Unconstrained (default) | Constrained bands | Passband ripples and stopband attenuation

Specify the magnitude constraints for the filter design.

**Dependencies**

To enable this parameter, set **Order mode** to `Specify`. The available options depend on the value of the **Frequency constraints** parameter.

**Magnitude units** — Units for magnitude specifications  
dB (default) | `Linear` | `Squared`

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

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

**Dependencies**

To enable this parameter, set **Order mode** to `Minimum`.

**Passband ripple 1** — Allowable filter ripple in first passband  
1 (default) | real-valued positive scalar

Specify the filter ripple allowed in the first passband in the units you choose for **Magnitude units**. Values must be real, positive scalars. If you are specifying values in linear units, they must be smaller than 1.

**Dependencies**

To enable this parameter, set the **Order mode** to `Minimum`.

**Stopband attenuation** — Stopband attenuation  
60 (default) | real-valued positive scalar

Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**. Values must be real, positive scalars. If you are specifying values in linear units, they must be smaller than 1.

**Dependencies**

To enable this parameter, set the **Order mode** to `Minimum`.

**Passband ripple 2** — Allowable filter ripple in second passband  
1 (default) | real-valued positive scalar

Enter the filter ripple allowed in the second passband in the units you choose for **Magnitude units**. Values must be real, positive scalars. If you are specifying values in linear units, they must be smaller than 1.

**Dependencies**

To enable this parameter, set the **Order mode** to `Minimum`.

**Design method** — Filter design method  
`Equiripple` (default) | `Kaiser window` | `Butterworth` | `Chebyshev type I` | `Chebyshev type II` | `Elliptic`

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods

available to design filters changes as well. The default IIR design method is usually Butterworth, and the default FIR method is Equiripple.

**Scale SOS filter coefficients to reduce chance of overflow** — Scale filter coefficients

on (default) | off

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

**Dependencies**

To enable this parameter, set **Impulse response** to IIR.

**Density factor** — Density factor

16 (default) | positive scalar

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

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 16 represents a reasonable balance between the accurate approximation to the ideal filter and the time to design the filter.

**Dependencies**

To enable this parameter, set **Impulse response** to FIR and **Design method** to Equiripple.

**Phase constraint** — Phase constraint

Linear (default) | Maximum | Minimum

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

**Dependencies**

To enable this parameter, set **Impulse response** to FIR and **Design method** to Equiripple.

**Minimum order** — Minimum filter order

Any (default) | Even

When you select this parameter, the design method determines and designs a minimum order filter to meet your specifications.

**Dependencies**

To enable this parameter, set **Impulse response** to FIR, **Order mode** to Minimum, and **Design method** to Kaiser window.

**Match exactly** — Match passband, stopband, or both

Stopband (default) | Passband | Both

Specifies that the resulting filter design matches either the passband, stopband, or both bands.

**Dependencies**

To enable this parameter, set **Impulse response** to IIR.



**Structure** — Filter structure

Direct-form FIR (default) | Direct-form FIR transposed | Direct-form symmetric FIR | Cascade minimum-multiplier allpass | Cascade wave digital filter allpass | Direct-form I SOS | Direct-form I transposed SOS | Direct-form II SOS | Direct-form II transposed SOS

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

**Use basic elements to enable filter customization** — Implement filter with basic Simulink blocks  
off (default) | on

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

The high-level implementation provides better compatibility across various filter structures, especially filters that would contain algebraic loops when constructed using basic elements.

**Dependencies**

When you select this check box, the block enables the following optimization parameters:

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

**Optimize for unit scale values** — Optimize unit scale values  
off (default) | on

Select this check box to scale unit gains between sections in SOS filters.

**Dependencies**

To enable this parameter, set **Impulse response** to IIR.

**Rate options** — Enforce single-rate or allow multirate processing  
Enforce single-rate processing (default) | Allow multirate processing

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

- **Enforce single-rate processing** — When you select this option, the block maintains the sample rate of the input.
- **Allow multirate processing** — When you select this option, the block adjusts the rate at the output to accommodate an increased or reduced number of samples.

**Dependencies**

To enable this parameter, set the **Impulse response** to FIR and set **Filter type** to a multirate filter.

**Use variable names for coefficients** — Specify coefficients with MATLAB variables  
off (default) | on

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

## Block Characteristics

|                                 |                 |
|---------------------------------|-----------------|
| <b>Data Types</b>               | double   single |
| <b>Multidimensional Signals</b> | No              |
| <b>Variable-Size Signals</b>    | No              |

## Version History

Introduced in R2006b

## Extended Capabilities

### C/C++ Code Generation

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

The Bandstop Filter block supports SIMD code generation using Intel AVX2 technology under these conditions:

- **Impulse response** is set to FIR.
- **Filter type** is set to Single-rate.
- **Structure** is set to Direct-form FIR or Direct-form FIR transposed.
- **Use basic elements to enable filter customization** parameter is not selected.
- **Input processing** is set to Columns as channels (frame based).
- Input signal has a data type of single or double.

The SIMD technology significantly improves the performance of the generated code.

## See Also

### Blocks

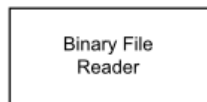
Tunable Bandstop IIR Filter | Bandpass Filter

### Functions

filterBuilder | fdesign.bandpass

# Binary File Reader

Read data from binary files



**Libraries:**  
DSP System Toolbox / Sources

## Description

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

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

## Output Ports

### Output

**data** — Binary file data  
column vector | row vector | matrix

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

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

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

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

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

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

Data Types: `Boolean`

## Parameters

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

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

**File header** — Size of the header

`struct([])` (default) | structure

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

**Export header to base workspace** — Retrieve the header

button

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

**Storage data type** — Storage class of data in file

'double' (default) | 'single' | 'int8' | 'int16' | 'int32' | 'int64' | 'uint8' | 'uint16' | 'uint32' | 'uint64'

Storage class of data in file, specified as a character vector. This parameter defines the data type of the matrix the block outputs.

**Samples per frame** — Number of samples per output frame

1024 (default) | positive integer

**Samples per frame** specifies the number of rows of the output matrix that the block outputs. The output matrix has dimensions **Samples per frame**-by-**Number of channels**. Once the end of the file is reached, the block returns zeros of the specified data type, size, and complexity.

**Data is complex** — Specify data complexity

off (default) | on

When you select this parameter, the reader treats the data as complex data. The block reads the data as interleaved real and imaginary components. Configure the block to read the data as a 2-by-2 matrix. The block reads [1 5 2 6 3 7 4 8] as [1 2; 3 4]+1j\*[5 6; 7 8]. When you do not select this parameter, the block reads the data as [1 5; 2 6].

**Number of channels** — Number of channels

1 (default) | positive integer

**Number of channels** specifies the number of columns of the output matrix that the block outputs. This parameter defines the number of consecutive interleaved data samples stored in the file for each

time instant. The size of the data is **Samples per frame-by-Number of channels**. Once the end of the file is reached, if the output matrix is not full, the block fills the matrix with zeros to make it a full-sized matrix.

**Output end-of-file indicator** — End-of-file indicator

off (default) | on

When you select this parameter, an additional output port named **EOF** appears on the block. When the block reaches the end of the file, the port outputs a 1. Otherwise, the port outputs a 0.

**Sample time (s)** — Sample time

1 (default) | nonnegative scalar

**Sample time (s)** controls the sample time at the output port of the block. This value represents  $1/F_s$ , where  $F_s$  is the sampling rate of the signal data. The Simulink sample time at the output port is **Samples per frame × Sample time (s)**.

**Simulate using** — Type of simulation to run

Code generation (default) | Interpreted execution

Specify the type of simulation to run as one of the following:

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

## Block Characteristics

|                                 |                           |
|---------------------------------|---------------------------|
| <b>Data Types</b>               | double   integer   single |
| <b>Direct Feedthrough</b>       | no                        |
| <b>Multidimensional Signals</b> | no                        |
| <b>Variable-Size Signals</b>    | no                        |
| <b>Zero-Crossing Detection</b>  | no                        |

## Version History

Introduced in R2016b

## Extended Capabilities

### C/C++ Code Generation

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

## **See Also**

### **Blocks**

Binary File Writer | Audio Device Writer

### **Objects**

`dsp.BinaryFileWriter` | `dsp.BinaryFileReader` | `dsp.MatFileWriter`

# Binary File Writer

Write data to binary files



## Libraries:

DSP System Toolbox / Sinks

## Description

The Binary File Writer block writes multichannel signal data to a binary file. The block specifies the name of the file and the structure of the header that precedes the signal data. If there is no header to write, the block specifies an empty structure, `struct([])`. The first time you write to the file, the block writes the header, followed by the data. On subsequent calls, the block writes the remaining data. If the header is empty, then no header is written.

The block writes the data in a row-major format. For example, if the input array is `[1 2 4 5; 8 7 9 2]`, the block writes the data as `[1 2 4 5 8 7 9 2]`.

## Input Ports

### Input

**Port\_1** — Data to write

column vector | row vector | matrix

The writer block writes the data to the file specified in the **File name** parameter. If the **File header** structure is not empty, then the writer writes the header before writing the data. The block can write floating-point data and integer data. The input data can be real or complex. When the data is complex, the block writes the data as interleaved real and imaginary components. The writer assumes the default endianness of the host machine.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

## Parameters

**File name** — Name of the file

'Untitled.bin' (default) | character vector

Name of the file to which the block writes the data.

**File header** — Size of the header

`struct([])` (default) | structure

The structure can have an arbitrary number of fields. Each field of the structure must be a real matrix of a built-in type. For example, if **File header** is set to `struct('field1',1:10,'field2',single(1))`, the block writes a header formed by 10 double-

precision values, (1:10), followed by 1 single precision value, `single(1)`. If there is no header to write, set this parameter to an empty structure, `struct([])`.

**Simulate using** — Type of simulation to run  
Code generation (default) | Interpreted execution

Specify the type of simulation to run as one of the following:

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

## Block Characteristics

|                                 |                           |
|---------------------------------|---------------------------|
| <b>Data Types</b>               | double   integer   single |
| <b>Direct Feedthrough</b>       | no                        |
| <b>Multidimensional Signals</b> | no                        |
| <b>Variable-Size Signals</b>    | no                        |
| <b>Zero-Crossing Detection</b>  | no                        |

## Version History

Introduced in R2016b

## Extended Capabilities

### C/C++ Code Generation

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

## See Also

### Blocks

Binary File Reader | Audio Device Writer

### Objects

`dsp.BinaryFileReader` | `dsp.BinaryFileWriter` | `dsp.MatFileReader`



# Biquad Filter

Model biquadratic IIR (SOS) filters



## Libraries:

DSP System Toolbox / Filtering / Filter Implementations  
 DSP System Toolbox HDL Support / Filtering

## Description

The Biquad Filter block independently filters each channel of the input signal with the specified biquadratic infinite impulse response (IIR) filter. When you specify the filter coefficients in the dialog box, the block implements static filters with fixed coefficients. When you provide the filter coefficients through an input port, you can tune the coefficients during simulation.

The Biquad Filter block supports the Simulink state logging feature. See “State” (Simulink) for more information.

## Ports

### Input

**In** — Data input  
 vector | matrix

Data input to the block, specified as a vector or a matrix. This block supports variable-size input signals, enabling you to change the input frame size (number of rows) during simulation. However, the number of channels (number of columns) must remain constant.

If the input is fixed-point, it must be signed fixed-point with binary point scaling.

This port is unnamed unless you set the **Coefficient source** to `Input port(s)`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `fixed point`

**Num** — Numerator coefficients  
 matrix

Numerator coefficients of the biquad filter, specified as a 3-by- $N$  matrix, where  $N$  is the number of biquad filter sections.

If **Num** is fixed-point, it must be signed fixed-point with binary point scaling.

### Dependencies

This port appears only when you set the **Coefficient source** to `Input port(s)`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `fixed point`

**Den** — Denominator coefficients  
 matrix

Denominator coefficients of the biquad filter, specified as a 2-by- $N$  matrix, where  $N$  is the number of biquad filter sections.

If **Den** is fixed-point, it must be signed fixed-point with binary point scaling.

#### Dependencies

This port appears only when you set the **Coefficient source** to `Input port(s)`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `fixed point`

#### **g** — Scale values

row vector

Scale values of the biquad filter, specified as a 1-by- $(N+1)$  vector, where  $N$  is the number of biquad filter sections.

If **g** is fixed-point, it must be signed fixed-point with binary point scaling.

#### Dependencies

This port appears only when you set the **Coefficient source** to `Input port(s)`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `fixed point`

#### Output

##### **Out** — Filtered output

vector | matrix

Filtered output, returned as a vector or a matrix.

The output dimensions always equal the dimensions of the input signal. The output of this block numerically matches the outputs of the `dsp.BiquadFilter` System object.

If **Out** is fixed-point, it must be signed fixed-point with binary point scaling.

This port is unnamed unless you set the **Coefficient source** to `Input port(s)`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `fixed point`

## Parameters

### Main Tab

#### **Coefficient source** — Mode of operation

`Dialog parameters (default)` | `Input port(s)` | `Filter object`

The Biquad Filter block can operate in three different modes:

- `Dialog parameters` — Enter information about the filter, such as structure and coefficients, in the block mask.
- `Input port(s)` — Enter information about the filter structure in the block mask using the **Filter structure** parameter. The filter coefficients come into the block through additional input ports that appear on the block icon:

- **Num** — Specify numerator coefficients.
- **Den** — Specify denominator coefficients.
- **g** — Specify scale values.

The block assumes the first denominator coefficient of each section to be 1. This configuration applies when **Coefficient source** is set to `Input port(s)` and **Scale values mode** is set to `Specify via input port (g)`.

- **Filter object** — Specify the filter using a `dsp.BiquadFilter System` object.

**Filter** — Name of filter object

BQF (default) | `dsp.BiquadFilter System` object name

Specify the name of the discrete-time filter that you want the block to implement. You must specify the filter as a `dsp.BiquadFilter System` object.

You can define the System object in the block mask or in a MATLAB workspace variable.

For information on creating System objects, see “Define Basic System Objects”.

#### Dependencies

This parameter is visible only when **Coefficient source** is set to `Filter object`.

**Filter structure** — Filter structure

Direct form II transposed (default) | Direct form I | Direct form I transposed | Direct form II

Specify the filter structure.

#### Dependencies

This parameter is visible only when **Coefficient source** is set to `Dialog parameters` or `Input port(s)`.

**SOS Matrix (Mx6)** — SOS matrix

[1 0.3 0.4 1 0.1 0.2] (default) | *M*-by-6 matrix

Specify an *M*-by-6 matrix, where *M* is the number of sections in the second-order section filter. Each row of the SOS matrix contains the numerator and denominator coefficients ( $b_{ik}$  and  $a_{ik}$ ) of the corresponding section in the filter.

$$\begin{bmatrix} b_{01} & b_{11} & b_{21} & a_{01} & a_{11} & a_{21} \\ b_{02} & b_{12} & b_{22} & a_{02} & a_{12} & a_{22} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ b_{0M} & b_{1M} & b_{2M} & a_{0M} & a_{1M} & a_{2M} \end{bmatrix}$$

The leading denominator coefficients [ $a_{01}$   $a_{02}$  ...  $a_{0N}$ ] are treated as 1s, regardless of their actual values. No scaling is applied to the SOS matrix when **a0** is not 1.

The `ss2sos` and `tf2sos` functions convert a state-space or transfer function description of your filter into the second-order section description used by this block.

### Dependencies

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

### Scale values — Scale values

1 (default) | scalar | vector

Specify scale values to be used between SOS sections. You can specify a real-valued scalar or a vector of length  $M+1$ :

- When you enter a scalar, the value specifies the gain value before the first section of the second-order filter. The rest of the gain values default to 1.
- When you enter a vector of  $M+1$  values, each value specifies a separate section of the filter. For example, the first element is the first gain value, the second element is the second gain value, and so on.



Select the **Optimize unity scale values** check box to optimize your simulation when one or more scale values equal 1. Selecting this option removes the unity gains so that the values are treated like Simulink lines or wires. In some fixed-point cases when there are unity scale values, selecting this parameter also omits certain casts. Refer to the “Fixed-Point” on page 2-141 section for more information.

### Dependencies

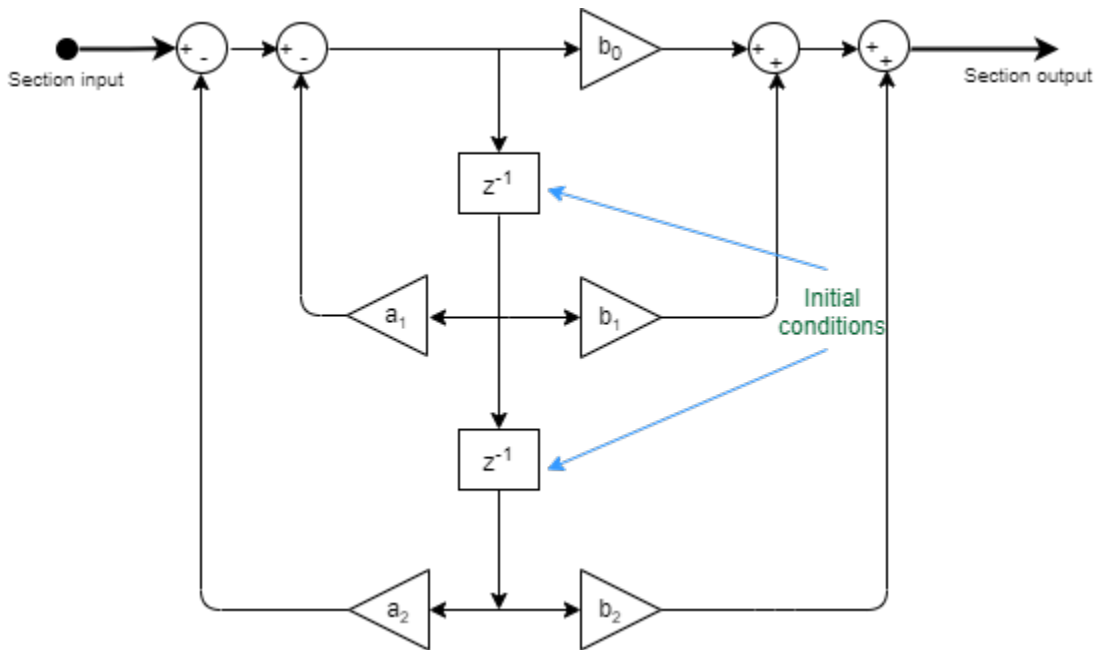
This parameter is visible only when **Coefficient source** is set to Dialog parameters.

### Initial conditions — Initial conditions

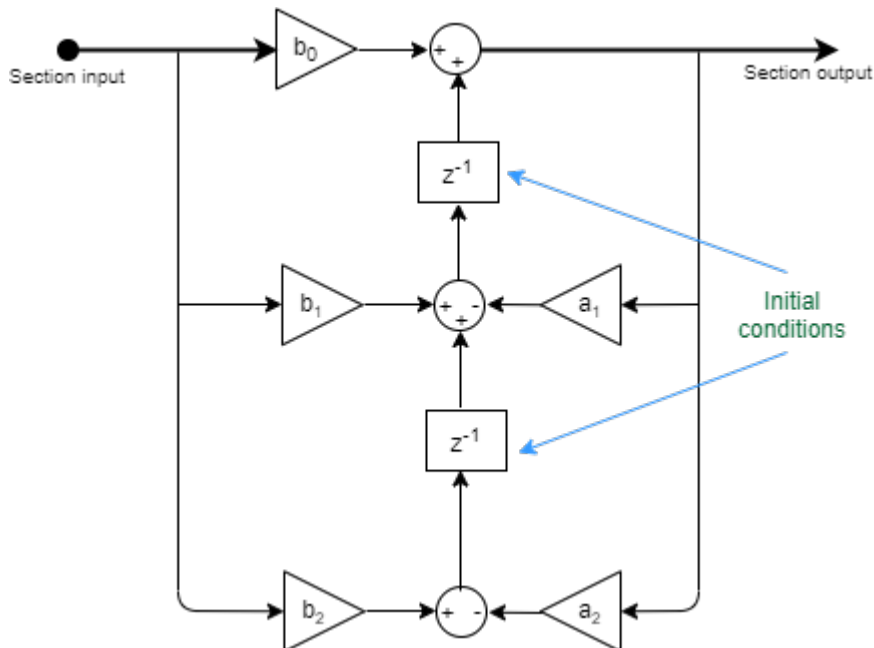
0 (default) | scalar | vector

Specify the initial conditions of the filter states when **Filter structure** is set to Direct form II or Direct form II transposed.

### Direct form II



### Direct form II transposed



The Biquad Filter block initializes the internal filter states to zero by default. To specify nonzero initial states for the filter delays, use the **Initial conditions** parameter.

To determine the number of initial conditions you must specify and how to specify them, see the following table on valid initial conditions.

**Valid Initial Conditions**

| Initial Condition   | Description  |
|---|--|
| Scalar  | The block initializes all delay elements in the filter to the scalar value.  |
| Vector or matrix<br>(for applying different delay elements to each channel) | <p>Each vector or matrix element specifies a unique initial condition for a corresponding delay element in a corresponding channel. <math>M</math> is the number of sections, and <math>N</math> is the number of input channels:</p> <ul style="list-style-type: none"> <li>The vector length must equal the number of delay elements in the filter, <math>2M</math>.</li> <li>The matrix must have the same number of rows as the number of delay elements in the filter <math>2MN</math>. The matrix must also have one column for each channel of the input signal.</li> </ul> |

**Dependencies**

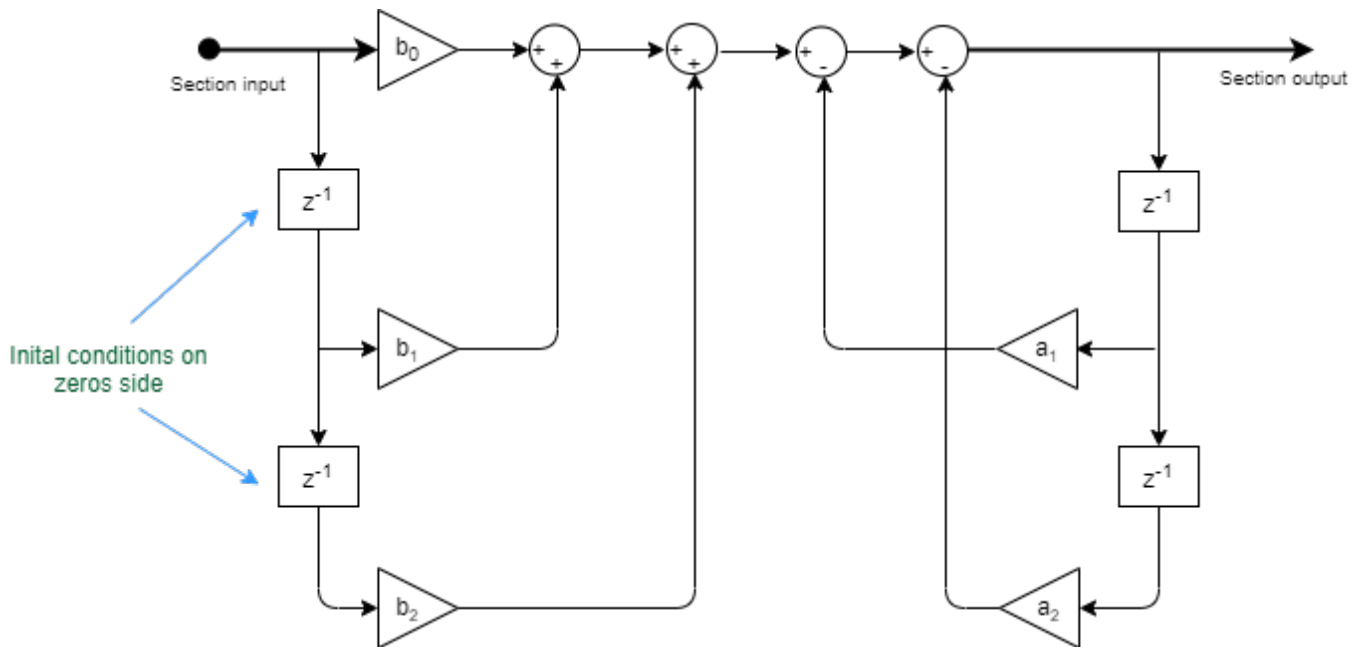
This parameter is only visible when **Coefficient source** is set to Dialog parameters or Input port(s) and the **Filter structure** is set to Direct form II or Direct form II transposed.

**Initial conditions on zeros side** — Initial conditions on zeros side

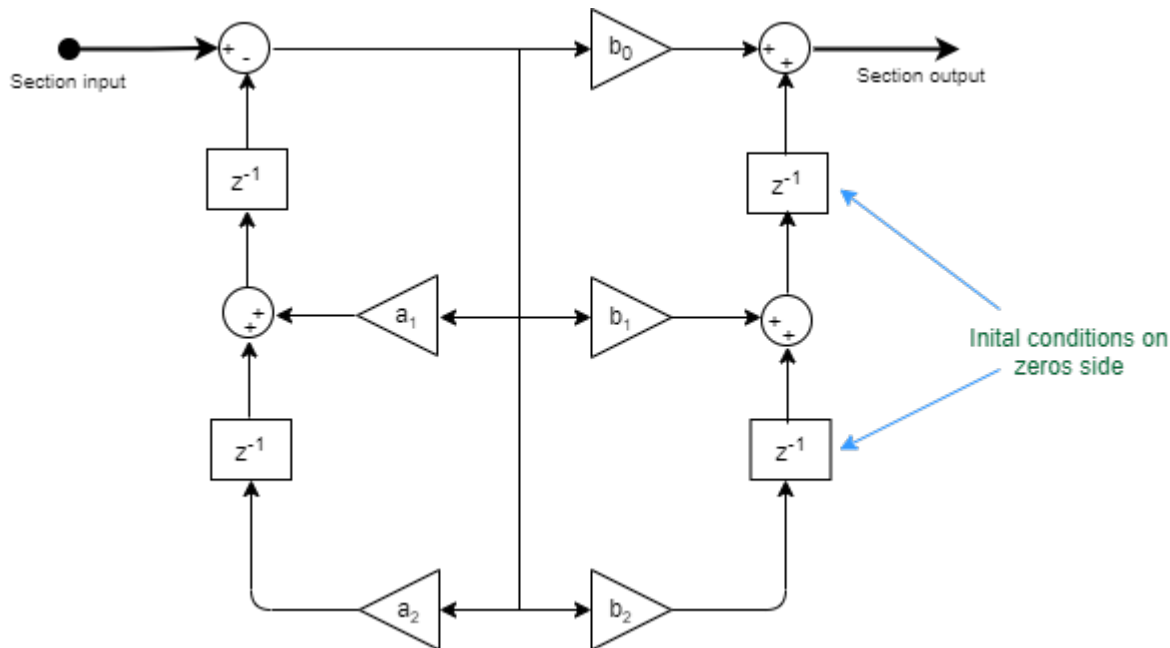
0 (default) | scalar | vector

Specify the initial conditions for the filter states on the side of the filter structure with the zeros ( $b_0, b_1, b_2, \dots$ ). This parameter applies only when **Filter structure** is set to Direct form I or Direct form I transposed.

**Direct form I**



**Direct form I transposed**



The Biquad Filter block initializes the internal filter states to zero by default. To specify nonzero initial states for the filter delays, use the **Initial conditions on zeros side** parameter.

To determine the number of initial conditions you must specify and how to specify them, see the following table on valid initial conditions.

### Valid Initial Conditions

| Initial Condition   | Description   |
|---|---|
| Scalar  | The block initializes all delay elements in the filter to the scalar value.   |
| Vector or matrix<br>(for applying different delay elements to each channel) | <p>Each vector or matrix element specifies a unique initial condition for a corresponding delay element in a corresponding channel. Where <math>M</math> is the number of sections and <math>N</math> is the number of input channels:</p> <ul style="list-style-type: none"> <li>The vector length must equal the number of delay elements in the filter, <math>2M</math>.</li> <li>The matrix must have the same number of rows as the number of delay elements in the filter <math>2MN</math>. The matrix must also have one column for each channel of the input signal.</li> </ul> |

### Dependencies

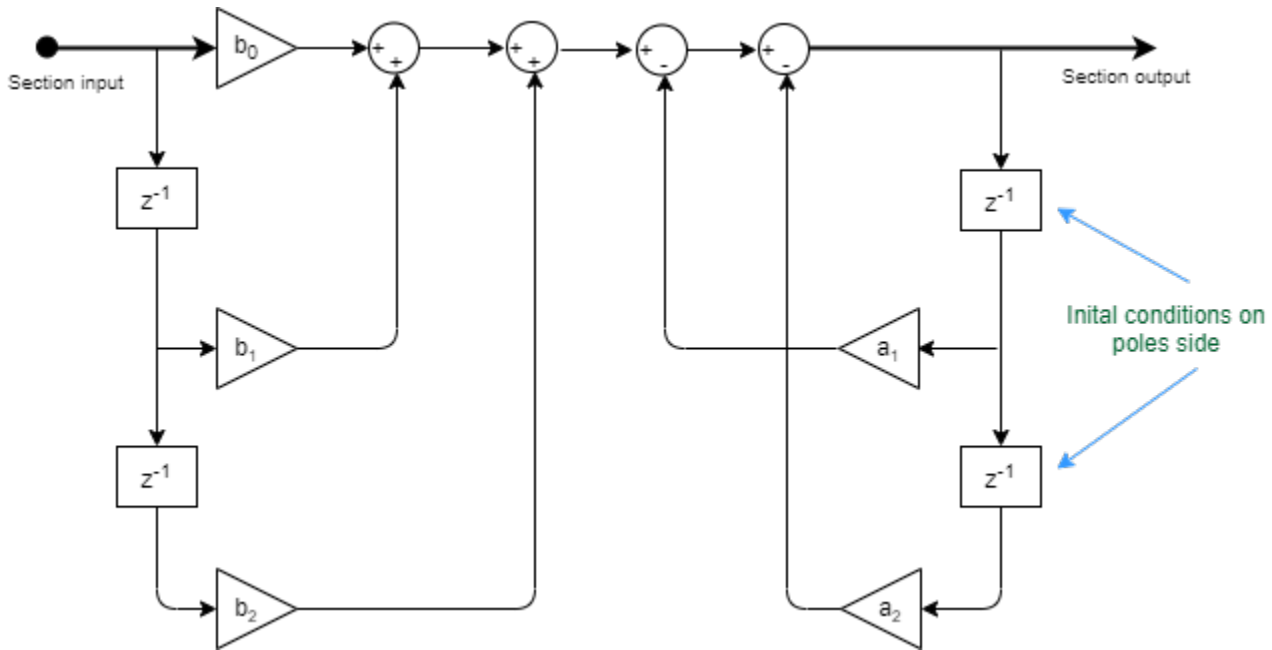
This parameter is visible only when **Coefficient source** is set to Dialog parameters or Input port(s) and the **Filter structure** is set to Direct form I or Direct form I transposed.

**Initial conditions on poles side** — Initial conditions on poles side

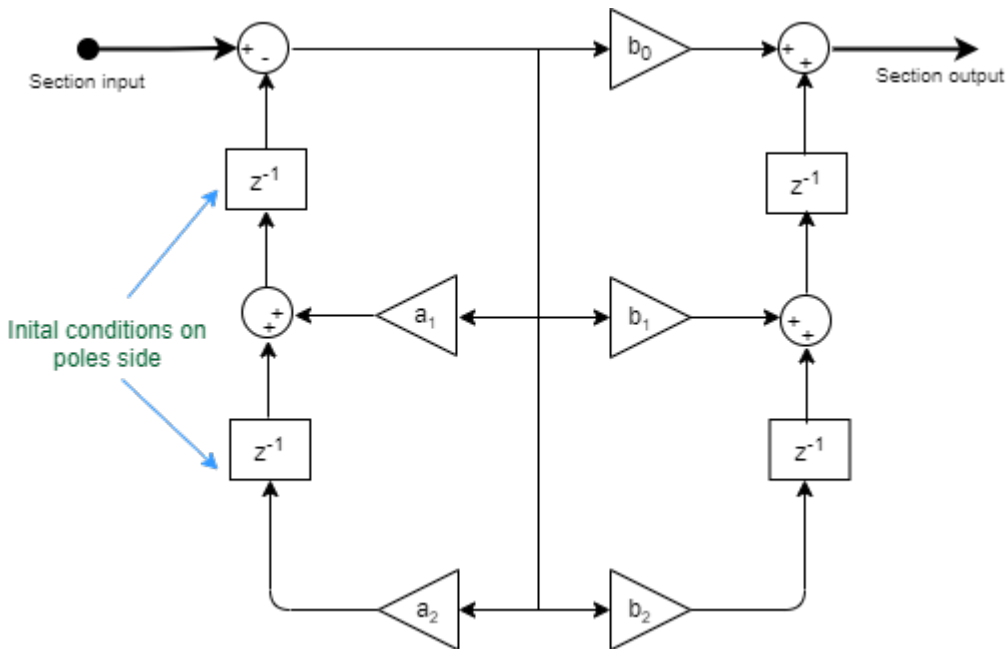
0 (default) | scalar | vector

Specify the initial conditions for the filter states on the side of the filter structure with the poles ( $a_1$ ,  $a_2$ , ...). This parameter applies only when **Filter structure** is set to Direct form I or Direct form I transposed.

**Direct form I**



**Direct form I transposed**



The Biquad Filter block initializes the internal filter states to zero by default. To specify nonzero initial states for the filter delays, use the **Initial conditions on poles side** parameter.

To determine the number of initial conditions you must specify and how to specify them, see the following table on valid initial conditions.



## Valid Initial Conditions

| Initial Condition   | Description   |
|---|---|
| Scalar  | The block initializes all delay elements in the filter to the scalar value.   |
| Vector or matrix<br>(for applying different delay elements to each channel) | <p>Each vector or matrix element specifies a unique initial condition for a corresponding delay element in a corresponding channel. Where <math>M</math> is the number of sections and <math>N</math> is the number of input channels:</p> <ul style="list-style-type: none"> <li>The vector length must equal the number of delay elements in the filter, <math>2M</math>.</li> <li>The matrix must have the same number of rows as the number of delay elements in the filter <math>2MN</math>. The matrix must also have one column for each channel of the input signal.</li> </ul> |

### Dependencies

This parameter is visible only when **Coefficient source** is set to `Dialog parameters` or `Input port(s)` and the **Filter structure** is set to `Direct form I` or `Direct form I transposed`.

### Scale values mode — Mode to specify scale values

`Specify via input port (g)` (default) | `Assume all are unity and optimize`

Choose how to specify the scale values to use between filter sections. When you select `Specify via input port (g)`, you enter the scale values as a 2-D vector at port **g**. When you select `Assume all are unity and optimize`, all scale values are removed and treated like Simulink lines or wires.

### Dependencies

This parameter is visible only when **Coefficient source** is set to `Input port(s)`.

### Action when the a0 values of the SOS matrix are not one — Action when a0 values of SOS matrix are not one

`Warning` (default) | `None` | `Error`

Specify the action the block should perform when the SOS matrix  $a_{0j}$  values do not equal one. The action can be `Warning`, `Error`, or `None`.

When you choose `None`, the leading coefficients  $a_{0j}$  are treated as 1's, regardless of their actual values. No scaling is applied on the SOS matrix when **a0** is not 1.

### Dependencies

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

### Optimize unity scale values — Optimize unity scale values

`on` (default) | `off`

Select this check box to optimize your simulation when one or more scale values equal 1. Selecting this option removes the unity gains so that the values are treated like Simulink lines or wires. In some fixed-point cases when there are unity scale values, selecting this parameter also omits certain casts. See the “Fixed-Point” on page 2-141 section for more information.

**Dependencies**

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

**Input processing** — Input processing

`Columns as channels (frame based) (default)` | `Elements as channels (sample based)`

Specify how the block should process the input. If the input is an  $M$ -by- $N$  matrix, you can set this parameter to:

- `Columns as channels (frame based) (default)` — The block treats each column as a separate channel. In this mode, the block creates  $M$  instances of the same filter, each with its own independent state buffer. Each of the  $M$  filters process  $N$  input samples at every Simulink time step.
- `Elements as channels (sample based)` — The block treats each element as a separate channel. In this mode, the block creates  $MN$  instances of the same filter, each with its own independent state buffer. Each filter processes one input sample at every Simulink time step.

**View Filter Response** — View filter response

button

This button opens the Filter Visualization Tool (**FVTool**) and displays the filter response of the filter specified in the dialog.

---

**Note** When you make changes to the filter parameters on the block dialog, you must click the **Apply** button before using the **View Filter Response** button.

---

**Data Types Tab**


---

**Note** This tab appears only when you set **Coefficient source** to either `Dialog` parameters or `Input port(s)`. When the **Coefficient source** is set to `Filter object`, the data types specified in the filter object properties are used by the block.

---

**Rounding mode** — Rounding mode

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

Specify the rounding mode for fixed-point operations.

For more details, see rounding mode. The filter coefficients do not obey this parameter; instead, they always round to `Nearest`.

**Saturate on integer overflow** — Method of overflow action

`off (default)` | `on`

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

The filter coefficients are always saturated and do not obey this parameter.

#### **Section input** — Section input data type

Same as `input` (default) | `Binary point scaling`

Choose how you specify the word and fraction lengths of the fixed-point data type going into each section of a biquadratic filter. See the “Fixed-Point” on page 2-141 section for illustrations depicting the use of the section input data type in this block. When you select:

- `Same as input` — Word length and fraction length characteristics of the **Section input** data type match those of the input to the block.
- `Binary point scaling` — Enter the word and fraction lengths of the section input, in bits.

#### **Section output** — Section output data type

Same as `section input` (default) | `Binary point scaling`

Choose how you specify the word and fraction lengths of the fixed-point data type coming out of each section of a biquadratic filter. See the “Fixed-Point” on page 2-141 section for illustrations depicting the use of the section output data type in this block. When you select:

- `Same as section input` — Word length and fraction length characteristics of the **Section output** data type match with those of the input to the block.
- `Binary point scaling` — Enter the word and fraction lengths of the section output, in bits.

#### **Multiplicand** — Multiplicand data type

Same as `output` (default) | `Binary point scaling`

Choose how you specify the word and fraction lengths of the multiplicand data type of a `Direct form I transposed` filter structure. See the “Fixed-Point” on page 2-141 section for illustrations depicting the use of the multiplicand data type in this block.

When you select:

- `Same as output` — Word length and fraction length characteristics of the **Multiplicand** data type match with those of the output of the block.
- `Binary point scaling` — Enter the word length and the fraction length of the multiplicand, in bits.

#### **Dependencies**

This parameter is visible only when the **Filter structure** parameter is set to `Direct form I transposed`.

#### **Coefficients** — Coefficients data type

Same word length as `input` (default) | `Specify word length` | `Binary point scaling`

Choose how you specify the word and fraction lengths of the filter coefficients (numerator, denominator, and scale value) when **Coefficient source** is set to `Dialog parameters`. See the “Fixed-Point” on page 2-141 section for illustrations depicting the use of the coefficient data types in this block. When you select:

- `Same word length as input` — Word length of the filter coefficients matches that of the input to the block. In this mode, the block automatically sets the fraction length of the coefficients to the binary point-only scaling that provides the best precision possible given the value and word length of the coefficients.

- **Specify word length** — Enter the word length of the coefficients, in bits. In this mode, the block automatically sets the fraction length of the coefficients to the binary point-only scaling that provides the best precision possible given the value and word length of the coefficients.
- **Binary point scaling** — Enter the word length and the fraction length of the coefficients, in bits. If applicable, enter separate fraction lengths for the numerator and denominator coefficients.

The filter coefficients do not obey the **Rounding mode** and the **Overflow mode** parameters; instead, they are always saturated and rounded to Nearest.

### Dependencies

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

### Product output — Product output data type

Same as input (default) | Inherit via internal rule | Binary point scaling

Specify how to designate the product output word and fraction lengths. See “Multiplication Data Types” and the “Fixed-Point” on page 2-141 section for illustrations depicting the use of the product output data type in this block. When you select:

- **Same as input** — Product output word length and fraction length characteristics match those of the input to the block.
- **Inherit via internal rule** — Product output word length and fraction lengths are computed based on full-precision rules. These rules prevent quantization from occurring within the block. Bits are added, as needed, so that no roundoff or overflow occurs. For more details, see “Inherit via Internal Rule”.
- **Binary point scaling** — Enter the word length and the fraction length of the product output, in bits. If applicable, enter separate fraction lengths for the numerator and denominator product output data type.

### Accumulator — Accumulator data type

Same as product output (default) | Same as input | Binary point scaling

Specify how to designate the accumulator word and fraction lengths. See “Multiplication Data Types” and the “Fixed-Point” on page 2-141 section for illustrations depicting the use of the accumulator data type in this block. When you select:

- **Same as input** — Accumulator word and fraction length characteristics match those of the input to the block.
- **Same as product output** — Accumulator word and fraction length characteristics match those of the product output.
- **Binary point scaling** — Enter the word length and the fraction length of the accumulator, in bits. If applicable, enter separate fraction lengths for the numerator and denominator accumulator data type.

### States — States data type

Same as accumulator (default) | Same as input | Binary point scaling

Specify how to designate the state word and fraction lengths when **Coefficient source** is set to **Dialog parameters**. See the “Fixed-Point” on page 2-141 section for illustrations depicting the use of the state data type in this block.

When you select:

- Same as `input` — State word and fraction length characteristics match those of the input to the block.
- Same as `accumulator` — State word and fraction length characteristics match those of the accumulator.
- `Binary point scaling` — Enter the word length and the fraction length of the state, in bits. If applicable, enter separate fraction lengths for the numerator and denominator state data type.

### Dependencies

This parameter is visible only when **Filter structure** is set to `Direct form II` or `Direct form II transposed`.

### Output — Output data type

Same as `accumulator` (default) | Same as `input` | `Binary point scaling`

Choose how you specify the output word length and fraction length. See the “Fixed-Point” on page 2-141 section for illustrations depicting the use of the output data type in this block. When you select:

- Same as `input` — Output word and fraction length characteristics match those of the input to the block.
- Same as `accumulator` — Output word and fraction length characteristics match those of the accumulator.
- `Binary point scaling` — Enter the word length and the fraction length of the output, in bits.

**Lock data type settings against changes by the fixed-point tools** — Lock data type settings `off` (default) | `on`

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

## Block Characteristics

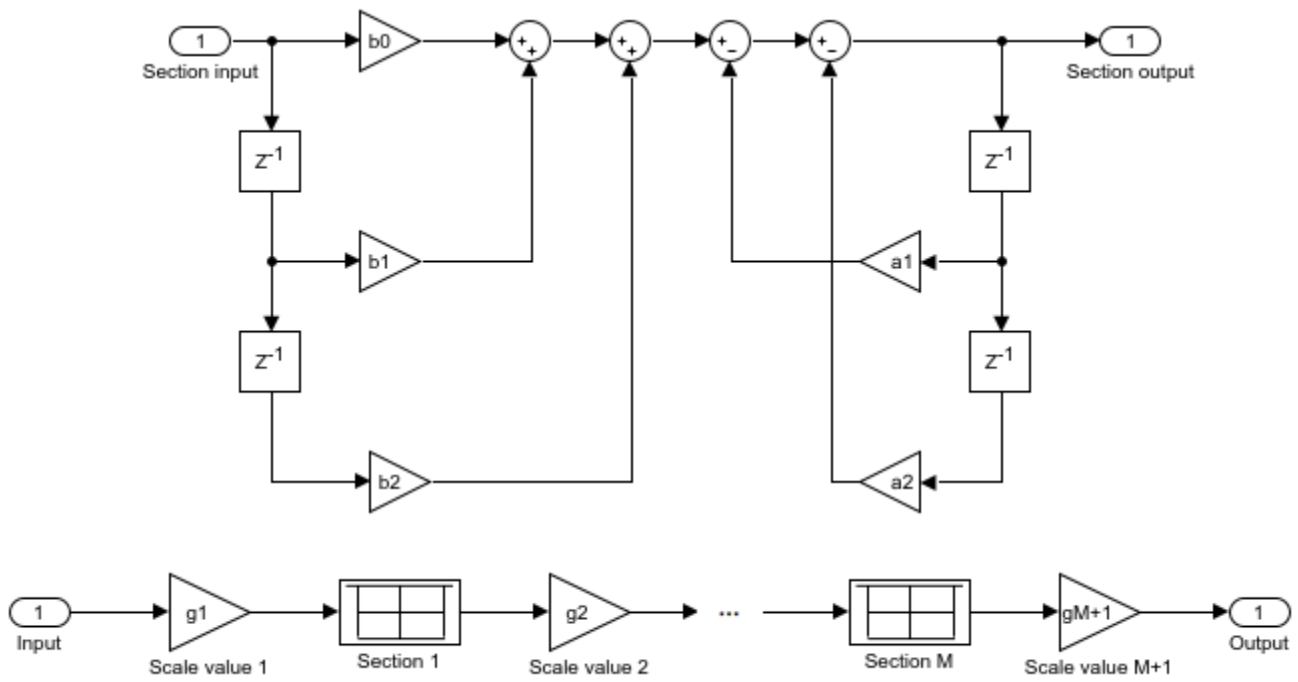
|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | <code>double</code>   <code>fixed point</code>   <code>integer</code>   <code>single</code> |
| <b>Direct Feedthrough</b>       | <code>no</code>   |
| <b>Multidimensional Signals</b> | <code>no</code>   |
| <b>Variable-Size Signals</b>    | <code>yes</code>  |
| <b>Zero-Crossing Detection</b>  | <code>no</code>   |

## More About

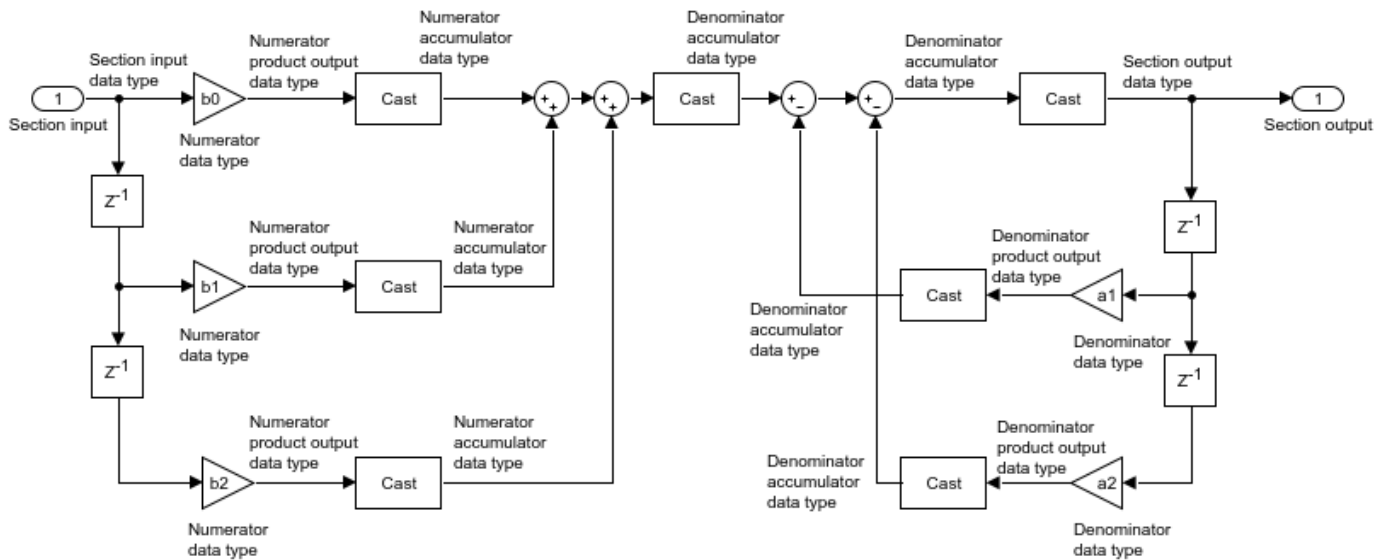
### Fixed-Point

The diagrams in these sections show the filter structures supported by the Biquad Filter block. They also show the data types used in the filter structures for fixed-point signals. You can set the data types shown in these diagrams in the block dialog box.

**Direct Form I**

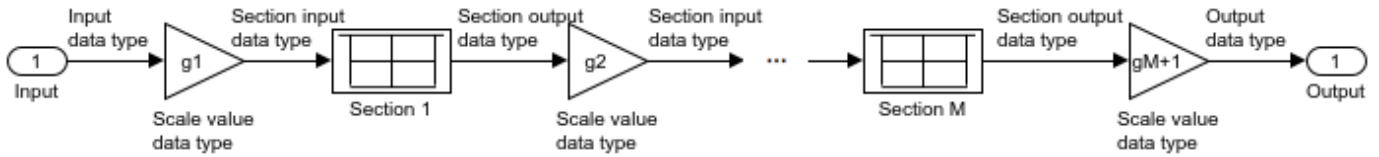


The following diagram shows the data types for one section of the filter for fixed-point signals.

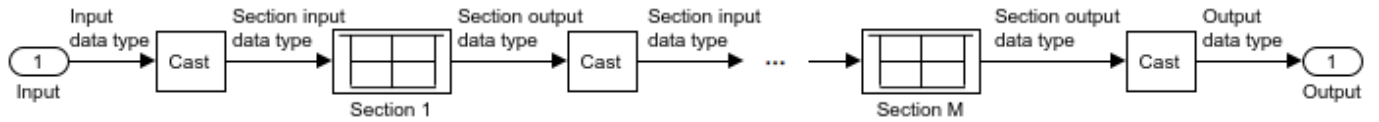


The following diagrams show the fixed-point data types between filter sections.

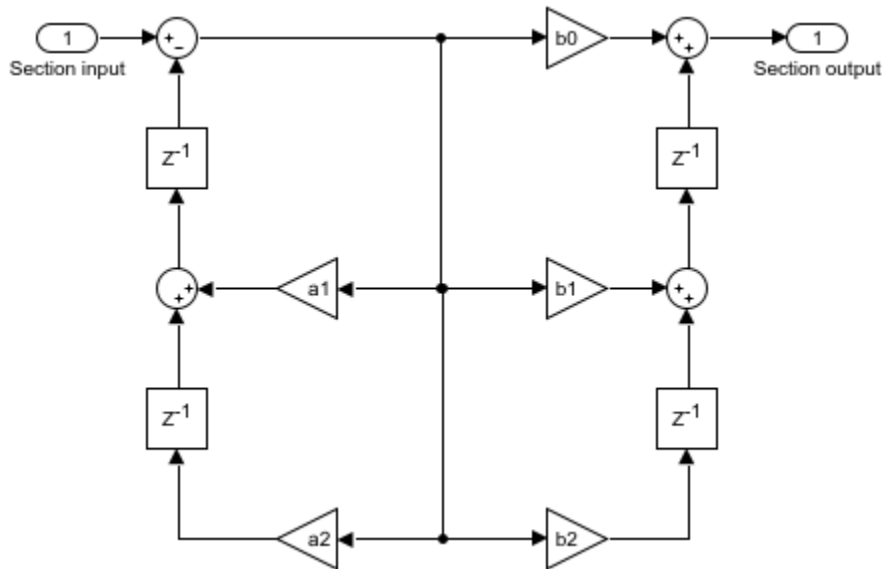
When the data is not optimized:



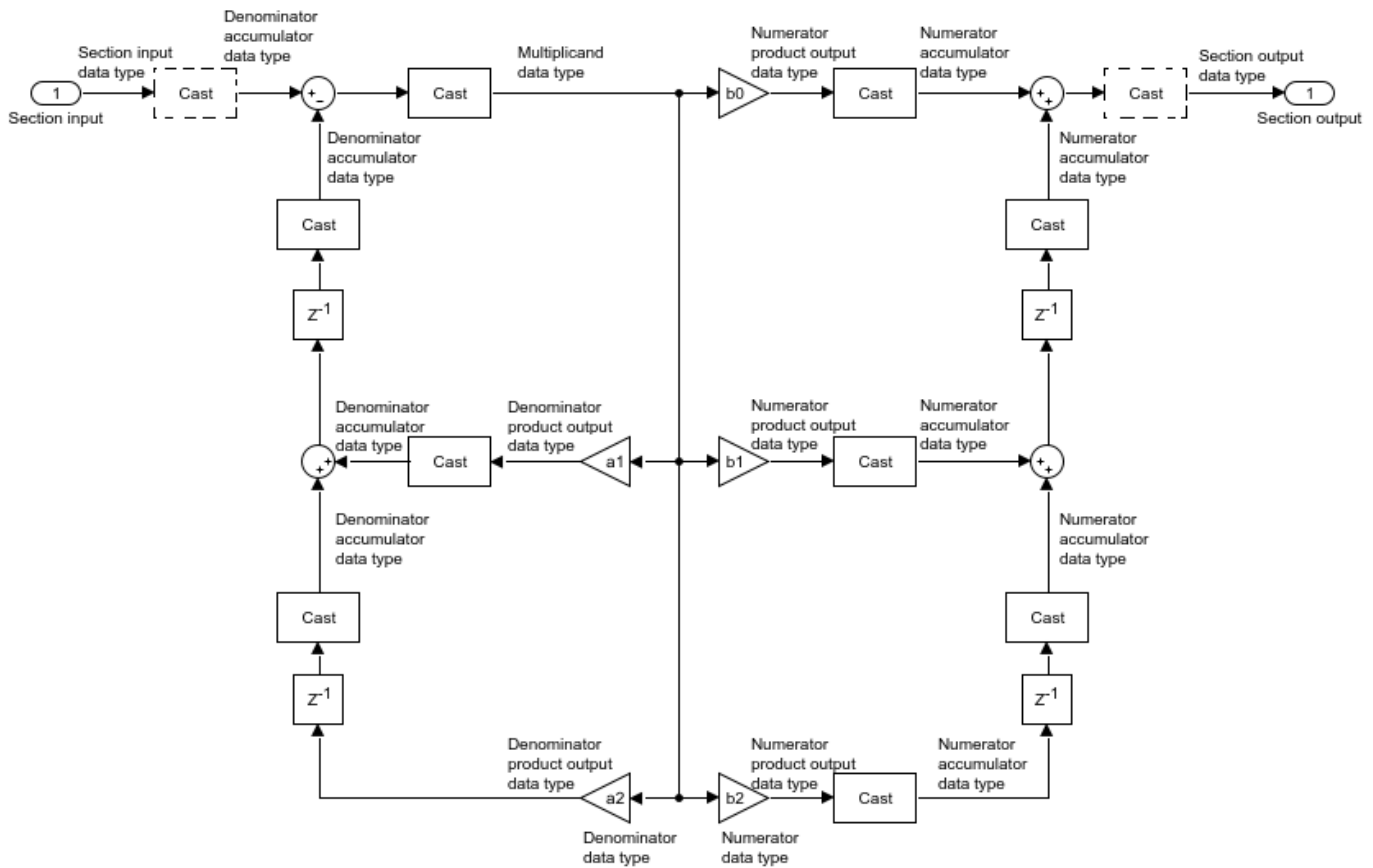
When you select **Optimize unity scale values** and scale values equal 1:



**Direct Form I Transposed**



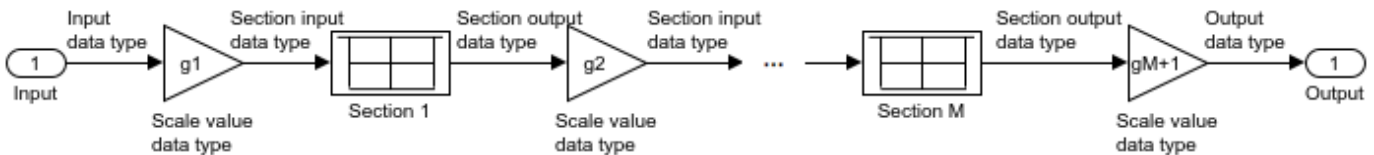
The following diagram shows the data types for one section of the filter for fixed-point signals.



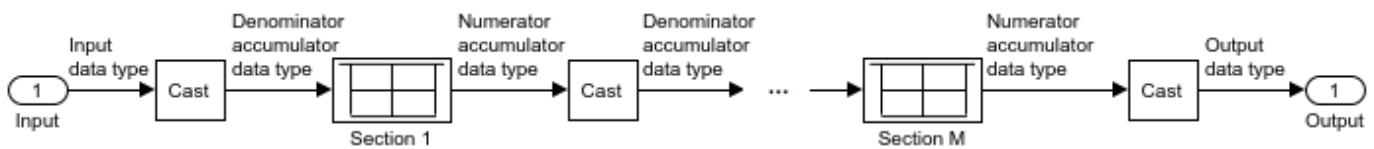
The dashed casts are omitted when **Optimize unity scale values** is selected and scale values equal one.

The following diagrams show the fixed-point data types between filter sections.

When the data is not optimized:

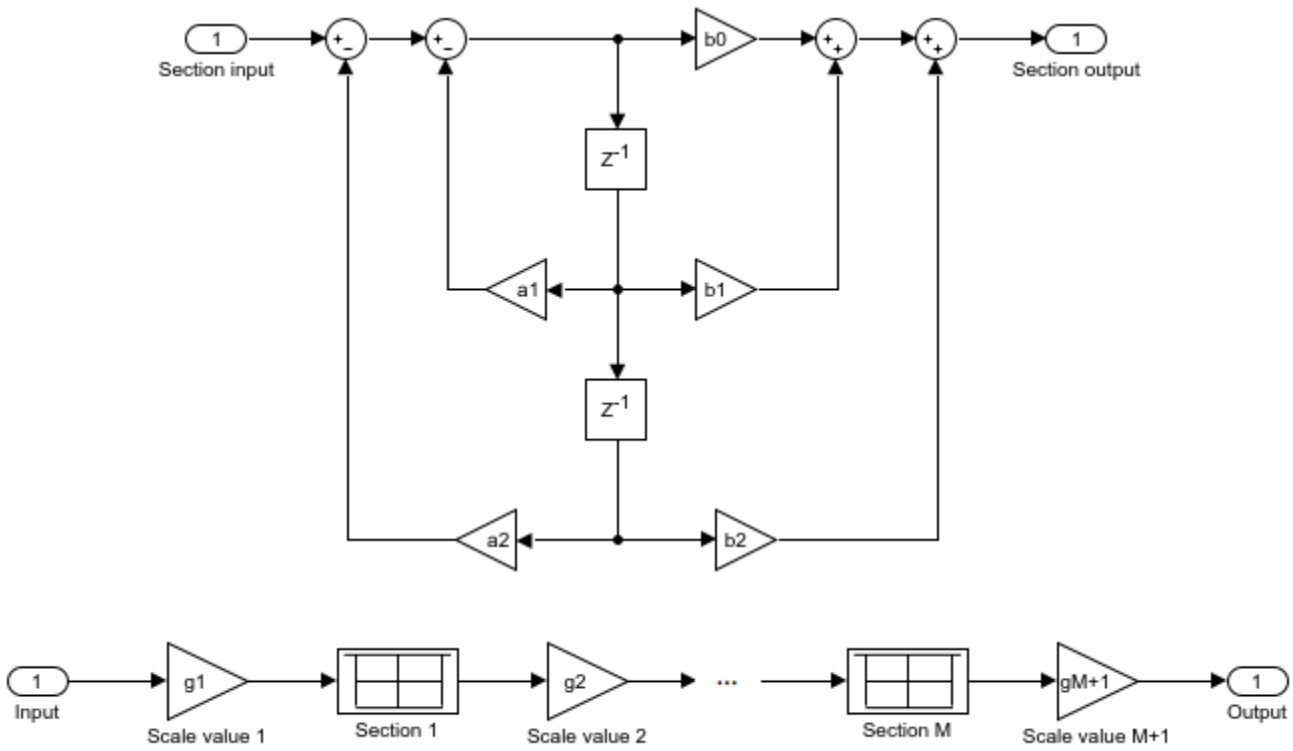


When you select **Optimize unity scale values** and scale values equal 1:

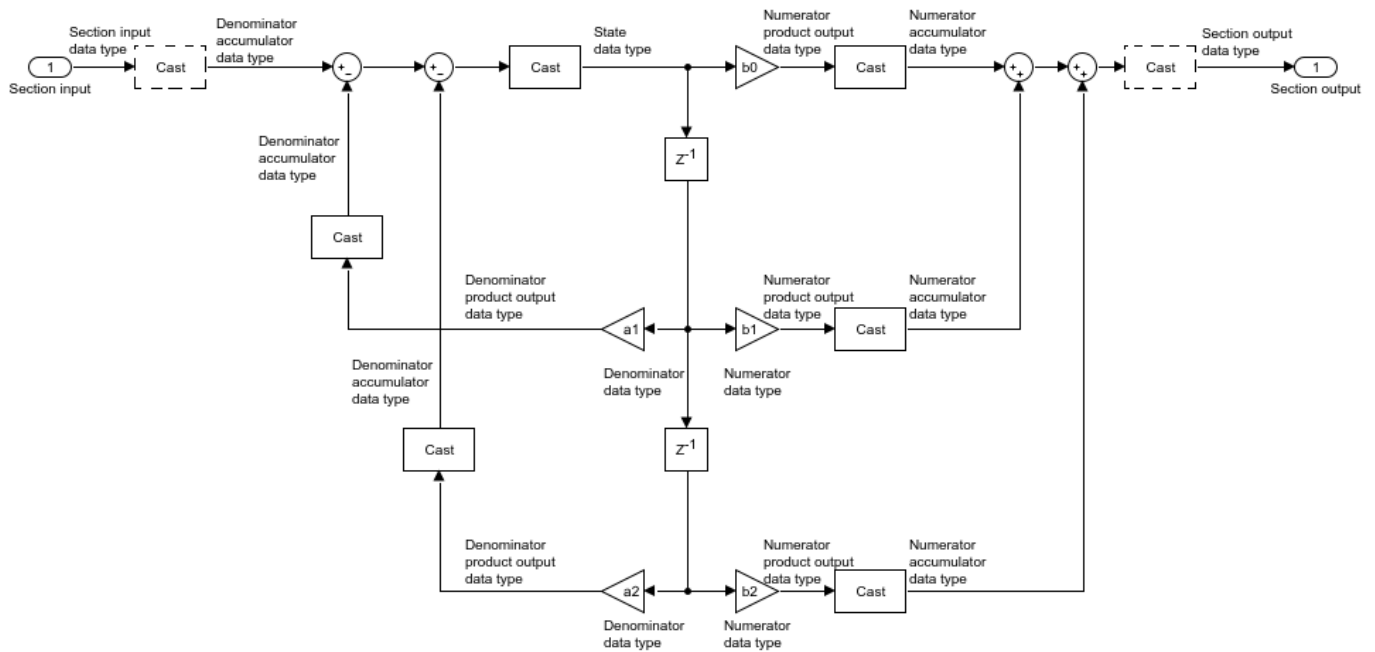




**Direct Form II**



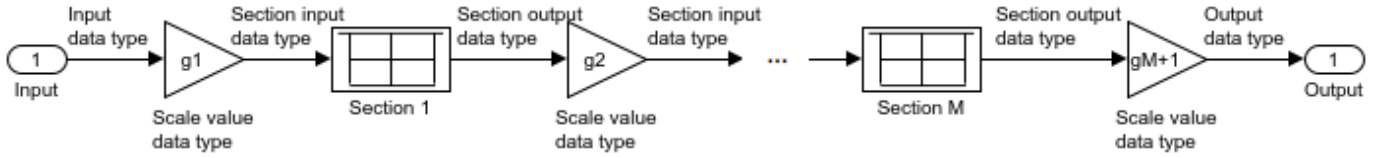
The following diagram shows the data types for one section of the filter for fixed-point signals.



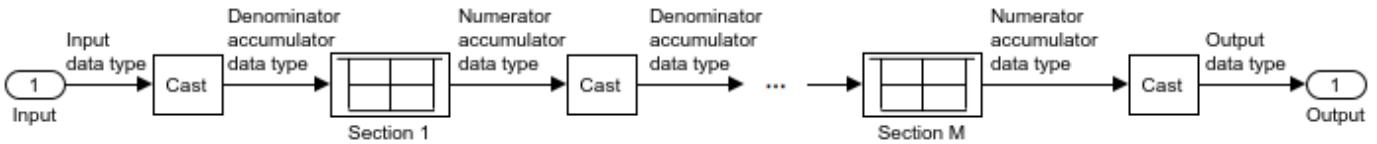
The dashed casts are omitted when **Optimize unity scale values** is selected and scale values equal one.

The following diagrams show the fixed-point data types between filter sections.

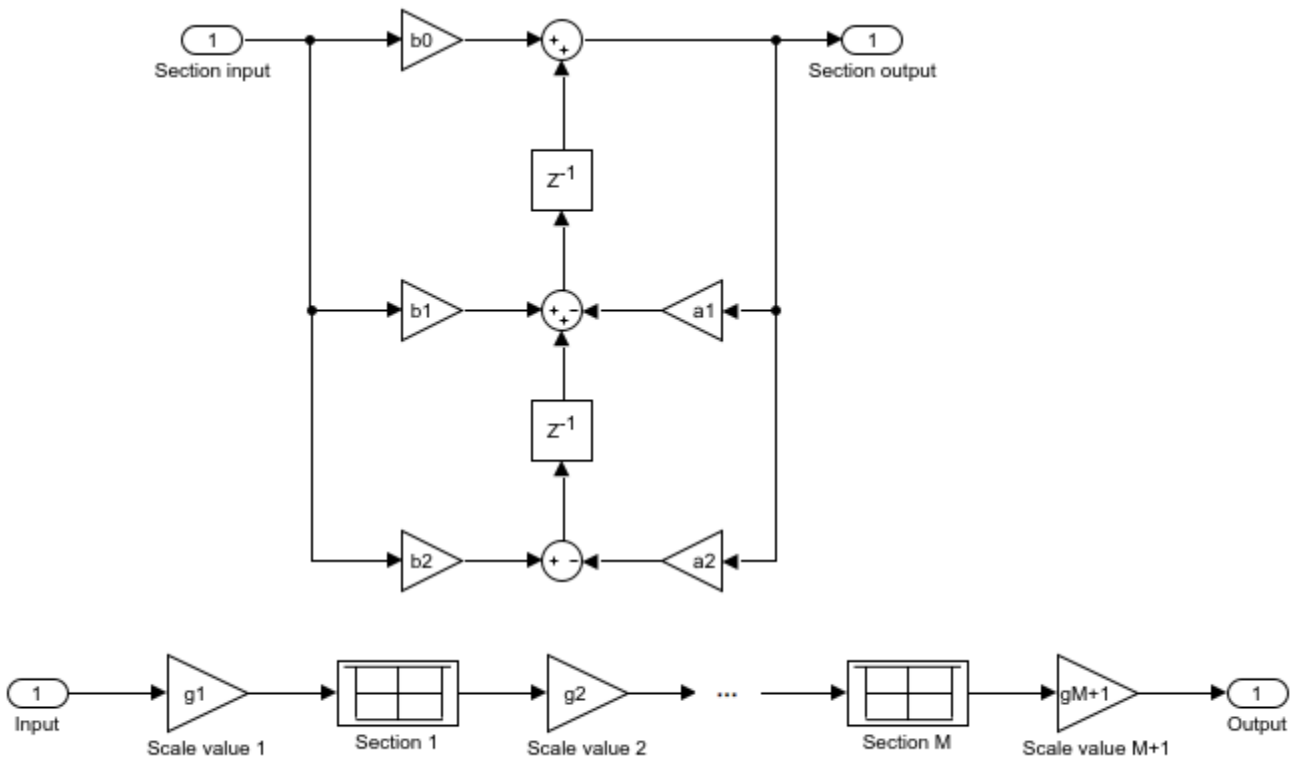
When the data is not optimized:



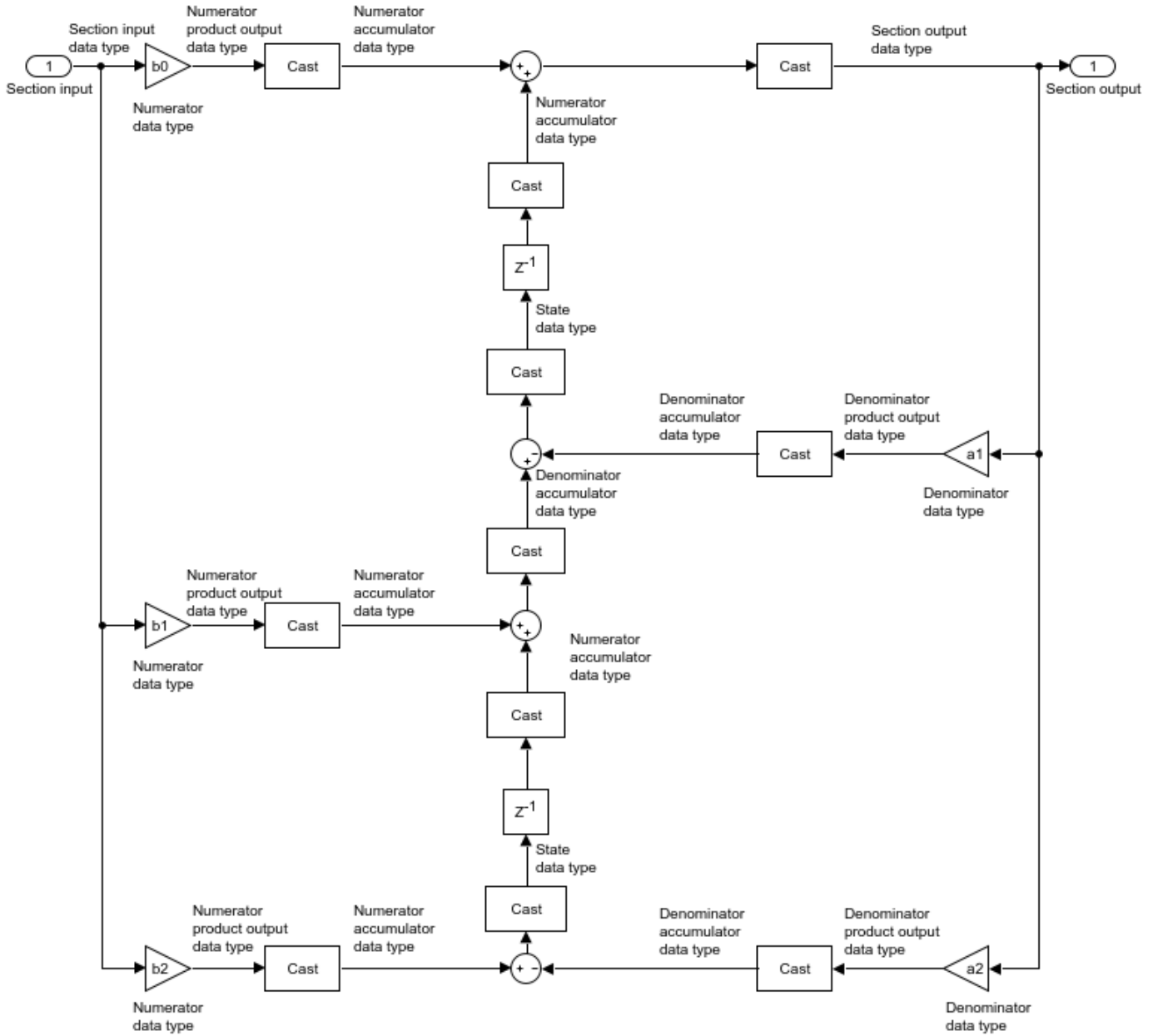
When you select **Optimize unity scale values** and scale values equal 1:



**Direct Form II Transposed**

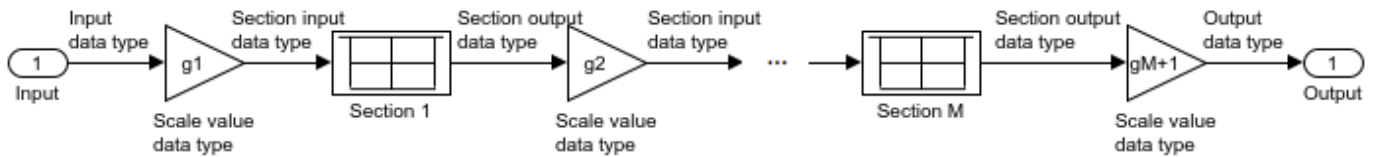


The following diagram shows the data types for one section of the filter for fixed-point signals.



The following diagrams show the fixed-point data types between filter sections.

When the data is not optimized:



When you select **Optimize unity scale values** and scale values equal 1:



## Version History

Introduced in R2008b

## Extended Capabilities

### C/C++ Code Generation

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

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

---

**Note** For an HDL-optimized biquad filter architecture with hardware-friendly control signals, use the Biquad Filter block. The DSP HDL Toolbox™ block simulates the latency of the HDL algorithm in Simulink. The DSP HDL Toolbox block generates a direct form II or direct form II transposed pipelined add-multiply operation that fits into the DSP block on an FPGA. It provides an optional pipelined feedback architecture that uses more multipliers but reduces the critical path and achieves higher clock rates. The DSP HDL Toolbox block does not support programmable coefficients. To perform multichannel filtering, put the DSP HDL Toolbox Biquad Filter block into a For Each subsystem.

---

HDL Coder™ provides additional configuration options that affect HDL implementation and synthesized logic.

### Programmable Filter Support

HDL Coder supports programmable filters for Biquad Filter blocks.

- 1 On the filter block mask, set **Coefficient source** to **Input port(s)**.
- 2 Connect vector signals to the **Num** and **Den** coefficient ports. Matrix input signals are not supported for HDL code generation. You must use one Biquad Filter block for each second-order section in your filter. For a filter with  $M$  second-order sections, use a series of  $M$  Biquad Filter blocks, each with an input **Num** that is a 3-by-1 vector and an input **Den** that is a 2-by-1 vector.

The following limitations apply to the HDL optimizations for a programmable Biquad Filter block:

- Fully serial and partly serial architectures are not supported. **Architecture** must be set to Fully parallel.
- Canonical signed digit (CSD) multiplier optimization is not supported. **CoeffMultipliers** must be set to multiplier.

### Multichannel Filter Support

HDL Coder supports the use of vector inputs to Biquad Filter blocks.

- 1 Connect a vector signal to the Biquad Filter block input port.
- 2 Specify **Input processing** as `Elements as channels (sample based)`.
- 3 To reduce area by sharing the filter kernel between channels, set the **StreamingFactor** parameter of the subsystem to the number of channels. See the Streaming section of “Subsystem Optimizations for Filters” (HDL Coder).

### Serial Architectures

To use block-level optimizations to reduce hardware resources, select a serial **Architecture**. Then set either `NumMultipliers` or `Folding Factor`. See “HDL Filter Properties” on page 2-149.

When you select a serial architecture, set **Filter structure** to `Direct form I` or `Direct form II`. The direct form transposed structures are not supported with serial architectures.

### AddPipelineRegisters Support

When you use **AddPipelineRegisters**, registers are placed based on the filter structure. The pipeline register placement determines the latency.

| Filter Structure | Pipeline Register Placement                               | Latency (Clock Cycles)                  |
|------------------|---|---|
| Any              | Pipeline registers are added between the filter sections. | NS - 1, where NS is number of sections. |

### Subsystem Optimizations

This block can participate in subsystem-level optimizations such as sharing, streaming, and pipelining. For the block to participate in subsystem-level optimizations, set **Architecture** to `Fully parallel`. See “Subsystem Optimizations for Filters” (HDL Coder).

### HDL Filter Properties

|                             |  |
|-----------------------------|--|
| <b>AddPipelineRegisters</b> | Insert a pipeline register between stages of computation in a filter. See also <code>AddPipelineRegisters</code> (HDL Coder).  |
| <b>CoeffMultipliers</b>     | Specify the use of canonical signed digit (CSD) optimization to decrease filter area by replacing coefficient multipliers with shift-and-add logic. When you choose a fully parallel filter implementation, you can set <b>CoeffMultipliers</b> to <code>csd</code> or <code>factored-csd</code> . The default is <code>multipliers</code> , which retains multipliers in the HDL. See also <code>CoeffMultipliers</code> (HDL Coder). |
| <b>FoldingFactor</b>        | Specify a serial implementation of an IIR SOS filter by the number of cycles it takes to generate the result. See also <code>FoldingFactor</code> (HDL Coder).   |
| <b>NumMultipliers</b>       | Specify a serial implementation of an IIR SOS filter by the number of hardware multipliers that are generated. See also <code>NumMultipliers</code> (HDL Coder).   |

For more details about HDL filter properties, see “HDL Filter Block Properties” (HDL Coder).

**HDL Block Properties**

|                                  |  |
|----------------------------------|--|
| <b>ConstrainedOutputPipeline</b> | Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder). |
| <b>InputPipeline</b>             | Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).                 |
| <b>OutputPipeline</b>            | Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).               |

**Restrictions**

- Frame input is not supported for HDL code generation. For frame-based input, use the Biquad Filter block.
- You must set **Initial conditions** to 0. HDL code generation is not supported for nonzero initial states.
- You must select **Optimize unity scale values**.
- You cannot generate HDL for this block inside a Resettable Synchronous Subsystem.

**Fixed-Point Conversion**

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

If the input is fixed point, it must be a signed integer or a signed fixed point value with a power-of-two slope and zero bias.

**See Also****Objects**

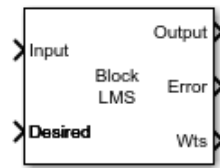
dsp.BiquadFilter

**Blocks**

Discrete FIR Filter | Variable Bandwidth FIR Filter | Variable Bandwidth IIR Filter

## Block LMS Filter

Compute output, error, and weights using LMS adaptive algorithm



## Library

Filtering / Adaptive Filters

dspadpt3

## Description

The Block LMS Filter block implements an adaptive least mean-square (LMS) filter, where the adaptation of filter weights occurs once for every block of samples. The block estimates the filter weights, or coefficients, needed to minimize the error,  $e(n)$ , between the output signal,  $y(n)$ , and the desired signal,  $d(n)$ . Connect the signal you want to filter to the **Input** port. The input signal can be a scalar or a column vector. Connect the signal you want to model to the **Desired** port. The desired signal must have the same data type, complexity, and dimensions as the input signal. The **Output** port outputs the filtered input signal. The **Error** port outputs the result of subtracting the output signal from the desired signal.

The block calculates the filter weights using the Block LMS adaptive filter algorithm. This algorithm is defined by the following equations.

$$\begin{aligned}
 n &= kN + i \\
 y(n) &= \mathbf{w}^T(k-1)\mathbf{u}(n) \\
 e(n) &= d(n) - y(n) \\
 \mathbf{w}(k) &= \mathbf{w}(k-1) + f(\mathbf{u}(n), e(n), \mu)
 \end{aligned}$$

The weight update function for the Block LMS adaptive filter algorithm is defined as

$$f(\mathbf{u}(n), e(n), \mu) = \mu \sum_{i=0}^{N-1} \mathbf{u}^*(kN + i)e(kN + i)$$

The variables are as follows.

| Variable | Description   |
|----------|---|
| $n$      | The current time index                                      |
| $i$      | The iteration variable in each block, $0 \leq i \leq N - 1$ |
| $k$      | The block number  |
| $N$      | The block size  |

| Variable        | Description                                      |
|-----------------|--|
| $\mathbf{u}(n)$ | The vector of buffered input samples at step $n$ |
| $\mathbf{w}(n)$ | The vector of filter-tap estimates at step $n$   |
| $y(n)$          | The filtered output at step $n$                  |
| $e(n)$          | The estimation error at time $n$                 |
| $d(n)$          | The desired response at time $n$                 |
| $\mu$           | The adaptation step size                         |

Use the **Filter length** parameter to specify the length of the filter weights vector.

The **Block size** parameter determines how many samples of the input signal are acquired before the filter weights are updated. The number of rows in the input must be an integer multiple of the **Block size** parameter.

The adaptation **Step-size (mu)** parameter corresponds to  $\mu$  in the equations. You can either specify a step-size using the input port, Step-size, or enter a value in the Block Parameters: Block LMS Filter dialog box.

Use the **Leakage factor (0 to 1)** parameter to specify the leakage factor,  $0 < 1 - \mu\alpha \leq 1$ , in the leaky LMS algorithm shown below.

$$\mathbf{w}(k) = (1 - \mu\alpha)\mathbf{w}(k - 1) + f(\mathbf{u}(n), e(n), \mu)$$

Enter the initial filter weights as a vector or a scalar in the **Initial value of filter weights** text box. When you enter a scalar, the block uses the scalar value to create a vector of filter weights. This vector has length equal to the filter length and all of its values are equal to the scalar value

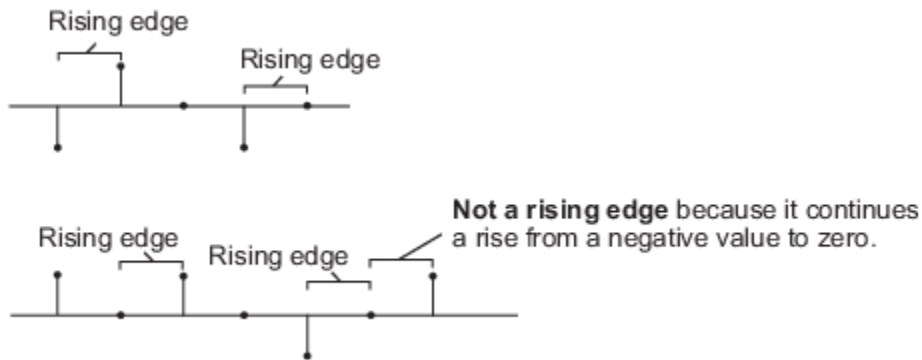
When you select the **Adapt port** check box, an **Adapt** port appears on the block. When the input to this port is greater than zero, the block continuously updates the filter weights. When the input to this port is zero, the filter weights remain at their current values.

When you want to reset the value of the filter weights to their initial values, use the **Reset input** parameter. The block resets the filter weights whenever a reset event is detected at the Reset port. The reset signal rate must be the same rate as the data signal input.

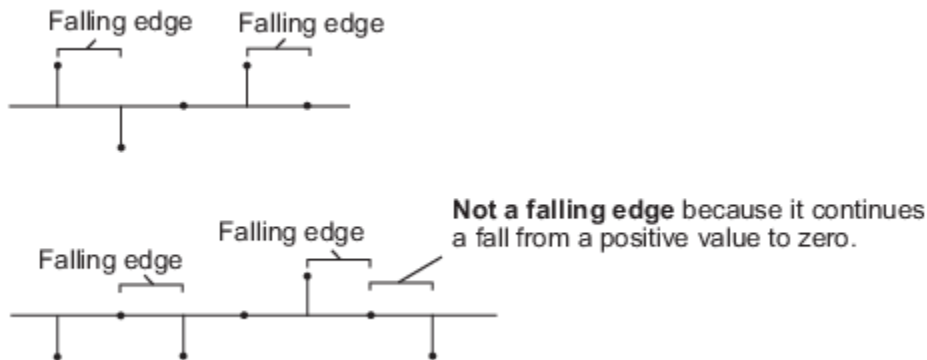
From the **Reset input** list, select **None** to disable the Reset port. To enable the Reset port, select one of the following from the **Reset input** list:

- **Rising edge** — Triggers a reset operation when the Reset input does one of the following:
  - Rises from a negative value to a positive value or zero
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure).





- **Falling edge** — Triggers a reset operation when the Reset input does one of the following:
  - Falls from a positive value to a negative value or zero
  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero (see the following figure)



- **Either edge** — Triggers a reset operation when the Reset input is a **Rising edge** or **Falling edge** (as described above)
- **Non-zero sample** — Triggers a reset operation at each sample time that the Reset input is not zero

Select the **Output filter weights** check box to create a **Wts** port on the block. For each iteration, the block outputs the current updated filter weights from this port.

## Parameters

### Filter length

Enter the length of the FIR filter weights vector.

### Block size

Enter the number of samples to acquire before the filter weights are updated. The number of rows in the input must be an integer multiple of the **Block size**.

### Specify step-size via

Select **Dialog** to enter a value for  $\mu$  in the Block parameters: LMS Filter dialog box. Select **Input port** to specify  $\mu$  using the Step-size input port.

### Step-size ( $\mu$ )

Enter the step-size. Tunable (Simulink).

**Leakage factor (0 to 1)**

Enter the leakage factor,  $0 < 1 - \mu\alpha \leq 1$ . Tunable (Simulink).

**Initial value of filter weights**

Specify the initial values of the FIR filter weights.

**Adapt port**

Select this check box to enable the Adapt input port.

**Reset port**

Select this check box to enable the Reset input port.

**Output filter weights**

Select this check box to export the filter weights from the Wts port.

**References**

Hayes, M. H. *Statistical Digital Signal Processing and Modeling*. New York: John Wiley & Sons, 1996.

**Supported Data Types**

| Port      | Supported Data Types  |
|-----------|---|
| Input     | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> </ul>  |
| Desired   | <ul style="list-style-type: none"> <li>• Must be the same as Input</li> </ul>   |
| Step-size | <ul style="list-style-type: none"> <li>• Must be the same as Input</li> </ul>   |
| Adapt     | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul> |
| Reset     | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul> |
| Output    | <ul style="list-style-type: none"> <li>• Same as Input</li> </ul>   |
| Error     | <ul style="list-style-type: none"> <li>• Same as Input</li> </ul>   |
| Wts       | <ul style="list-style-type: none"> <li>• Same as Input</li> </ul>   |

**See Also**

Fast Block LMS Filter

Kalman Adaptive Filter (Obsolete)

DSP System Toolbox

DSP System Toolbox

LMS Filter

DSP System Toolbox

RLS Filter

DSP System Toolbox

See “Noise Cancellation in Simulink Using Normalized LMS Adaptive Filter” for related information.

## **Version History**

**Introduced before R2006a**

## **Extended Capabilities**

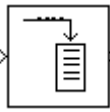
### **C/C++ Code Generation**

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

Generated code relies on the `memcpy` or `memset` function (`string.h`) under certain conditions.

## Buffer

Buffer input sequence to smaller or larger frame size

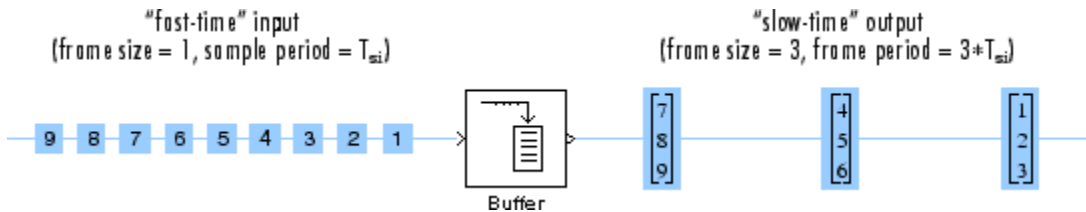


### Libraries:

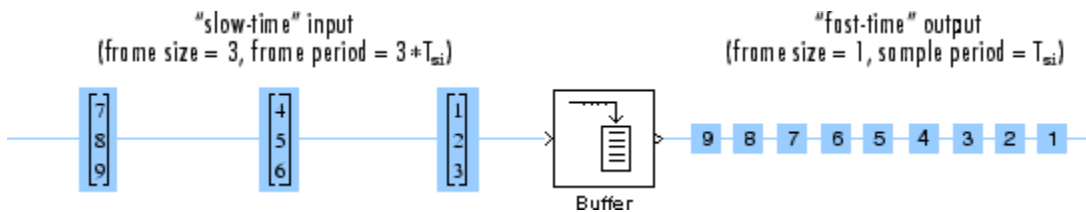
DSP System Toolbox / Signal Management / Buffers

## Description

The Buffer block always performs frame-based processing. The block redistributes the data in each column of the input to produce an output with a different frame size. Buffering a signal to a larger frame size yields an output with a *slower* frame rate than the input. For example, consider this illustration for a scalar input.



Buffering a signal to a smaller frame size yields an output with a *faster* frame rate than the input. For example, consider this illustration of a scalar output.



The block coordinates the output *frame size* and *frame rate* of nonoverlapping buffers such that the sample period of the signal is the same at both the input and output:  $T_{so} = T_{si}$ .

This block supports triggered subsystems when the block input and output rates are the same.

## Ports

### Input

#### Port\_1 — Input signal

scalar | vector | matrix

To buffer single-channel signals, input a scalar or column vector. To buffer multichannel signals, input a row vector or matrix. To buffer single-channel signals, input a scalar or column vector of the size  $M_i$ -by-1, where  $M_i$  is the input frame size.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

Complex Number Support: Yes

## Output

**Port\_1** — Output signal  
vector | matrix

The buffered input signal returned as a vector or a matrix.

When the input is a single-channel signal:

- 1-by-1 (scalar) — Output is an  $M_o$ -by-1 vector.
- $M_i$ -by-1 (column vector) — Output is an  $M_o$ -by-1 vector.

where

- $M_i$  — Input frame size
- $M_o$  — Value of the **Output buffer size** parameter

The input frame period is  $M_i \cdot T_{si}$ , where  $T_{si}$  is the input sample period. The output frame period is  $(M_o - L)T_{si}$ , where  $L$  is the value of the **Buffer overlap** parameter and  $T_{si}$  is the input sample period. When you set the **Buffer overlap** parameter to  $M_o - 1$ , the output frame period equals the input sample period.

When the input is a multichannel signal:

- 1-by- $N$  — Output is an  $M_o$ -by- $N$  matrix.
- $M_i$ -by- $N$  — Output is an  $M_o$ -by- $N$  matrix.

where

- $N$  — Number of channels in the signal
- $M_i$  — Input frame period
- $M_o$  is the value of the **Output buffer size** parameter and can be greater or less than the input frame size,  $M_i$ . The block buffers each of the  $N$  input channels independently.

The input frame period is  $M_i \cdot T_{si}$ . The output frame period is  $(M_o - L)T_{si}$ , which equals the sequence sample period when the **Buffer overlap** is  $M_o - 1$ . Thus, the output sample period  $T_{so}$  is related to the input sample period  $T_{si}$  by

$$T_{so} = \frac{(M_o - L)T_{si}}{M_i}$$

The output has the same data type and complexity as the input

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

Complex Number Support: Yes

## Parameters

**Output buffer size (per channel)** — Buffer size  
64 (default) | positive integer

Specify the number of consecutive samples,  $M_o$ , from each channel to buffer into the output frame.

Data Types: double | int8 | int16 | int32 | uint8 | uint16 | uint32

**Buffer overlap** — Buffer overlap

0 (default) | integer

Specify the number of samples,  $L$ , which specifies the amount of overlap or underlap in each successive output frame.

To overlap the data in the buffer, specify a value of  $L$  in the range  $0 \leq L < M_o$ , where  $M_o$  is the value of the **Output buffer size** parameter. The block takes  $L$  samples (rows) from the current output and repeats them in the next output. In cases of overlap, the block acquires  $M_o - L_{new}$  input samples before propagating the buffered data to the output.

When  $L < 0$ , you are buffering the signal with underlap. The block discards  $L$  input samples after the buffer fills and outputs the buffer with period  $(M_o - L)T_{si}$ , which is longer than in the zero-overlap case.

The output frame period is  $(M_o - L)T_{si}$ , which equals the input sequence sample period,  $T_{si}$ , when the **Buffer overlap** is  $M_o - 1$ .

Data Types: double | int8 | int16 | int32 | uint8 | uint16 | uint32

**Initial conditions** — Initial output

0 (default) | scalar | vector | matrix

Specify the value of the block's initial output, in cases of nonzero latency, as a scalar, vector, or matrix.

For all cases of single-tasking operation (Simulink Coder) other than those listed in “Zero-Tasking Latency” on page 2-159, the Buffer block's buffer is initialized to the value(s) specified by the **Initial conditions** parameter. The block reads from this buffer to generate the first  $D$  output samples, where

$$D = \begin{cases} M_o + L & (L \geq 0) \\ M_o & (L < 0) \end{cases}$$

The dimensions of the **Initial conditions** parameter depend on the **Buffer overlap**,  $L$ , and whether the input is single-channel or multichannel:

- When  $L \neq 0$ , the **Initial conditions** parameter must be a scalar.
- When  $L = 0$ , the **Initial conditions** parameter can be a scalar, or it can be a vector with either of these constraints:
  - For single-channel inputs, the **Initial conditions** parameter can be a vector of length  $M_o$  if  $M_i$  is 1, or a vector of length  $M_i$  if  $M_o$  is 1.
  - For multichannel inputs, the **Initial conditions** parameter can be a vector of length  $M_o * N$  if  $M_i$  is 1, or a vector of length  $M_i * N$  if  $M_o$  is 1.

For all multitasking operations (Simulink Coder), use the `rebuffer_delay` function to compute the exact delay in samples that the Buffer block introduces for a given combination of buffer size and buffer overlap.

For general buffering between arbitrary frame sizes, the **Initial conditions** parameter must be a scalar, which is then repeated across all elements of the initial output(s). However, in the special case where the input is a 1-by- $N$  row vector, and the output of the block is an  $M_o$ -by- $N$  matrix, **Initial conditions** can be:

- An  $M_o$ -by- $N$  matrix
- A length- $M_o$  vector to be repeated across all columns of the initial output(s)
- A scalar to be repeated across all elements of the initial output(s)

In the special case where the output is a 1-by- $N$  row vector, which is the result of unbuffering an  $M_i$ -by- $N$  matrix, the **Initial conditions** can be:

- A vector containing  $M_i$  samples to output sequentially for each channel during the first  $M_i$  sample times
- A scalar to be repeated across all elements of the initial output(s)

For more information on latency and the Simulink tasking modes, see “Excess Algorithmic Delay (Tasking Latency)” and “Time-Based Scheduling and Code Generation” (Simulink Coder).

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixed point`

Complex Number Support: Yes

## Block Characteristics

|                                 |  |
|---------------------------------|--|
| <b>Data Types</b>               | <code>Boolean</code>   <code>double</code>   <code>fixed point</code>   <code>integer</code>   <code>single</code> |
| <b>Direct Feedthrough</b>       | no   |
| <b>Multidimensional Signals</b> | no   |
| <b>Variable-Size Signals</b>    | no   |
| <b>Zero-Crossing Detection</b>  | no   |

## More About

### Zero-Tasking Latency

*Zero-tasking latency* means that the first input sample, received at  $t = 0$ , appears as the first output sample. In the Simulink single-tasking mode (Simulink Coder), the Buffer block has zero-tasking latency for these special cases:

- Scalar input and output ( $M_o = M_i = 1$ ) with zero or negative **Buffer overlap** ( $L \leq 0$ )
- Input frame size is an integer multiple of the output frame size

$$M_i = kM_o$$

where  $k$  is an integer with zero **Buffer overlap** ( $L = 0$ ); notable cases of this include:

- Any input frame size  $M_i$  with scalar output ( $M_o = 1$ ) and zero **Buffer overlap** ( $L = 0$ )
- Equal input and output frame sizes ( $M_o = M_i$ ) with zero **Buffer overlap** ( $L = 0$ )

### Overflow and Underflow

The Buffer block cannot be used in an enabled subsystem under the following conditions:

- In a multirate, multitasking environment
- When the **Buffer overlap** parameter is set to a negative value

The Buffer block has an internal reservoir that temporarily stores data. When the Buffer block is used in an enabled subsystem, there is the possibility that the reservoir can overflow or underflow. The block implements safeguards against these occurrences.

Overflow occurs when more data enters the buffer than what it can hold. For example, consider buffering a scalar input to a frame of size three with a buffer that accepts an input every second and outputs every three seconds. If you place this buffer inside an enabled subsystem that is disabled every three seconds at  $t = 3s$ ,  $t = 6s$ , and so on, the buffer accumulates data in its internal reservoir without being able to empty it. This condition results in an overflow.

Underflow occurs when the buffer runs out of data to output. For example, again consider buffering a scalar input to a frame size of three with a buffer that accepts an input every second and outputs every three seconds. If you place this buffer inside an enabled subsystem that is disabled at  $t = 10s$ ,  $t = 11s$ ,  $t = 13s$ ,  $t = 14s$ ,  $t = 16s$ , and  $t = 17s$ , its internal reservoir becomes drained, and there is no data to output at  $t = 18s$ . This condition results in an underflow.

To protect from an overflow or an underflow, the Buffer block keeps a record of the amount of data in its internal reservoir. When the Buffer block reads data, the amount of data in its reservoir goes up. When the Buffer block outputs the data, the amount of data in its reservoir goes down. To protect from overflow, the oldest samples in the reservoir are discarded whenever the amount of data in the reservoir is larger than the actual buffer size. To protect from underflow, the most recent samples are repeated whenever an output is due and there is no data in the reservoir.

## Version History

Introduced before R2006a

### Extended Capabilities

#### C/C++ Code Generation

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

The Buffer block supports code generation for Simulink Real-Time™. The code generated for this target is executed concurrently. To generate this code, set the system target file of the model to `slrealtime.tlc`. The Buffer block also needs to satisfy certain conditions.

To set the system target file:

- 1 In the Simulink model window, in the **Modeling** tab, click **Model Settings**.
- 2 In the Model Settings > **Code Generation** pane, set the **System target file** to `slrealtime.tlc`.



In addition, the Buffer block must satisfy the following conditions:

- Ratio between output and input buffer size must be a pure integer, that is  $1/N$  or  $N/1$ . Ratios such as  $3/12$ ,  $1/7$ ,  $7/1$ , or  $2/4$  are all valid ratios. In the Buffer block, you can specify the appropriate output buffer size using the **Output buffer size (per channel)** parameter.
- There should be no overlap between frames. In the Buffer block, set the **Buffer overlap** parameter to 0.

### Fixed-Point Conversion

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

## See Also

### Functions

rebuffer\_delay

### Blocks

Delay Line | Unbuffer

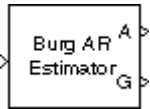
### Topics

“Convert Sample and Frame Rates in Simulink Using Frame Rebuffering Blocks”

“Buffering and Frame-Based Processing”

## Burg AR Estimator

Compute estimate of autoregressive (AR) model parameters using Burg method



### Libraries:

DSP System Toolbox / Estimation / Parametric Estimation

## Description

The Burg AR Estimator block uses the Burg method to fit an autoregressive (AR) model to the input data by minimizing (least squares) the forward and backward prediction errors while constraining the AR parameters to satisfy the Levinson-Durbin recursion.

## Ports

### Input

#### Input — Input

column vector | unoriented vector

Specify the input data as a column vector or an unoriented vector. The block assumes that the input data is the output of an AR system driven by white noise and represents a frame of consecutive time samples from a single-channel signal.

Data Types: `single` | `double`

### Output

#### A — Normalized estimate of the AR model polynomial coefficients

column vector

Normalized estimate of the AR model polynomial coefficients  $A(z)$ , returned as a column vector of length  $p+1$  in descending powers of  $z$ .

The block computes the estimate of these coefficients independently for each successive input frame.

$$H(z) = \frac{G}{A(z)} = \frac{G}{1 + a(2)z^{-1} + \dots + a(p+1)z^{-p}}$$

where,

- $H(z)$  -- Transfer function of the estimated AR model
- $G$  -- Scalar gain
- $A(z)$  -- Polynomial coefficients of the AR model

### Dependency

To enable this port, set the **Output(s)** parameter to A or A and K.

Data Types: `single` | `double`

**K** — Reflection coefficients  
column vector

Reflection coefficients (which are a secondary result of the Levinson recursion), returned as a column vector of length  $p$ .

#### Dependency

To enable this port, set the **Output(s)** parameter to K or A and K.

Data Types: single | double

**G** — Model gain  
scalar

Gain of the estimated AR model, returned as a scalar.

$$H(z) = \frac{G}{A(z)} = \frac{G}{1 + a(2)z^{-1} + \dots + a(p+1)z^{-p}}$$

where,

- $H(z)$  -- Transfer function of the estimated AR model
- $G$  -- Scalar gain
- $A(z)$  -- Polynomial coefficients of the AR model

Data Types: single | double

## Parameters

**Output(s)** — Output

A (default) | A and K | K

Specify whether the block outputs model coefficients (A), reflection coefficients (K), or both (A and K).

**Inherit estimation order from input dimensions** — Inherit estimation order from input dimensions

on (default) | off

When you select the **Inherit estimation order from input dimensions** parameter, the block sets the order  $p$  of the all-pole model to a value that is one less than the length of the input vector. Otherwise, the order is the value you specify in the **Estimation order** parameter.

**Estimation order** — Order of AR model

4 (default) | nonnegative integer

Specify the order of the AR model  $p$  as a nonnegative integer.

#### Dependency

To enable this parameter, clear the **Inherit estimation order from input dimensions** parameter.

## Block Characteristics

|                                 |                 |
|---------------------------------|-----------------|
| <b>Data Types</b>               | double   single |
| <b>Multidimensional Signals</b> | No              |
| <b>Variable-Size Signals</b>    | No              |

## More About

### AR(p) Model

In an AR model of order  $p$ , the current output is a linear combination of the past  $p$  outputs plus a white noise input.

The weights on the  $p$  past outputs minimize the mean squared prediction error of the autoregression. If  $y(n)$  is the current value of the output and  $x(n)$  is a zero-mean white noise input, the AR( $p$ ) model is:

$$y(n) + \sum_{k=1}^p a(k)y(n-k) = x(n).$$

### Reflection Coefficients

The reflection coefficients are the partial autocorrelation coefficients scaled by -1. The reflection coefficients indicate the time dependence between  $y(n)$  and  $y(n-k)$  after subtracting the prediction based on the intervening  $k-1$  time steps.

### Compare AR Model Parameter Estimation Methods

This table compares the features of the Burg AR Estimator block to the Covariance AR Estimator, Modified Covariance AR Estimator, and the Yule-Walker AR Estimator blocks.

The Yule-Walker AR Estimator and Burg AR Estimator blocks return similar results for large frame sizes.

|                        | <b>Burg AR Estimator</b>   | <b>Covariance AR Estimator</b>                                    | <b>Modified Covariance AR Estimator</b>   | <b>Yule-Walker AR Estimator</b>  |
|------------------------|--|---|---|--|
| <b>Characteristics</b> | Does not apply window to data  | Does not apply window to data                                     | Does not apply window to data   | Applies window to data   |
|                        | Minimizes the forward and backward prediction errors in the least squares sense, with the AR coefficients constrained to satisfy the L-D recursion | Minimizes the forward prediction error in the least squares sense | Minimizes the forward and backward prediction errors in the least squares sense | Minimizes the forward prediction error in the least squares sense (also called "autocorrelation method") |
| <b>Advantages</b>      | Always produces a stable model   |   |   | Always produces a stable model   |

|                                      | <b>Burg AR Estimator</b> | <b>Covariance AR Estimator</b>                               | <b>Modified Covariance AR Estimator</b>                      | <b>Yule-Walker AR Estimator</b>  |
|--------------------------------------|--------------------------|--|--|--|
| <b>Disadvantages</b>                 |                          | Can produce unstable models                                  | Can produce unstable models                                  | Performs relatively poorly for short data records  |
| <b>Conditions for Nonsingularity</b> |                          | Order must be less than or equal to 1/2 the input frame size | Order must be less than or equal to 2/3 the input frame size | Because of the biased estimate, the autocorrelation matrix is guaranteed to positive-definite, hence nonsingular |

## Algorithms

Burg's method estimates the reflection coefficients and uses the reflection coefficients to estimate the AR parameters recursively. You can find the recursion and lattice filter relations describing the update of the forward and backward prediction errors in [1].

## Version History

Introduced before R2006a

## References

- [1] Kay, S. M. *Modern Spectral Estimation: Theory and Application*. Englewood Cliffs, NJ: Prentice-Hall, 1988.
- [2] Marple, S. L., Jr., *Digital Spectral Analysis with Applications*. Englewood Cliffs, NJ: Prentice-Hall, 1987.

## Extended Capabilities

### C/C++ Code Generation

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

## See Also

### Functions

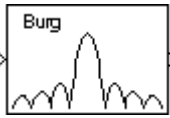
arburg

### Blocks

Burg Method | Covariance AR Estimator | Modified Covariance AR Estimator | Yule-Walker AR Estimator

## Burg Method

Power spectral density estimate using Burg method



### Libraries:

DSP System Toolbox / Estimation / Power Spectrum Estimation

## Description

The Burg Method block estimates the power spectral density (PSD) of the input frame using the Burg method. This method fits an autoregressive (AR) model to the signal by minimizing (least squares) the forward and backward prediction errors. The block minimizes the errors by constraining the AR parameters to satisfy the Levinson-Durbin recursion.

The block computes the spectrum from the FFT of the estimated AR model parameters.

## Ports

### Input

#### Input — Input

column vector | unoriented vector

Specify the input as a column vector or an unoriented vector. This input represents a frame of consecutive time samples from a single-channel signal.

Data Types: `single` | `double`

### Output

#### Output — Power spectral density estimate

column vector

Power spectral density estimate of the signal at  $N_{fft}$  equally spaced frequency points, returned as a column vector. The frequency points are in the range  $[0, F_s)$ , where  $F_s$  is the sampling rate of the signal.

Data Types: `single` | `double`

## Parameters

**Inherit estimation order from input dimensions** — Inherit estimation order from input dimensions

`off` (default) | `on`

When you select the **Inherit estimation order from input dimensions** parameter, the order of the all-pole model (estimation order) is one less than the input frame size. Otherwise, the **Estimation order** parameter determines the model order. The block computes the spectrum from the FFT of the estimated AR model parameters.

**Estimation order** — Order of AR model

6 (default) | nonnegative integer

Specify the estimation order of the AR model as a nonnegative integer.

**Dependencies**

To enable this parameter, clear the **Inherit estimation order from input dimensions** parameter.

**Inherit FFT length from estimation order** — Inherit FFT length from estimation order

off (default) | on

When you select this parameter, the FFT length  $N_{fft}$  is one greater than the estimation order. To specify the number of points on which to perform the FFT, clear the **Inherit FFT length from estimation order** parameter. You can then specify the FFT length as a power of 2 using the **FFT length** parameter. The block zero-pads or wraps the input to  $N_{fft}$  before computing the FFT.

**FFT length** — FFT length

256 (default) | positive integer greater than or equal to 2

Enter the number of data points  $N_{fft}$  on which to perform the FFT as a positive integer greater than or equal to 2. When  $N_{fft}$  is larger than the input frame size, the block zero-pads each frame as needed. When  $N_{fft}$  is smaller than the input frame size, the block wraps each frame as needed.

**Dependencies**

To enable this parameter, clear the **Inherit FFT length from input dimensions** parameter.

**Inherit sample time from input** — Inherit sample time from input

on (default) | off

When you select the **Inherit sample time from input** parameter, the block computes the frequency data from the sample period of the input signal. For the block to produce a valid output, the following conditions must hold:

- The input to the block is an original signal with no samples added or deleted (by insertion of zeros, for example).
- The sample period of the time-domain signal in the simulation equals the sample period of the original time series.

If these conditions do not hold, clear the **Inherit sample time from input** parameter. You can then specify a sample time using the **Sample time of original time series** parameter.

**Sample time of original time series** — Sample time of original time-domain signal

1 (default) | positive scalar

Specify the sample time of the original time-domain signal as a positive scalar.

**Dependencies**

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

## Block Characteristics

|                                 |                 |
|---------------------------------|-----------------|
| <b>Data Types</b>               | double   single |
| <b>Multidimensional Signals</b> | No              |
| <b>Variable-Size Signals</b>    | No              |

## More About

### Compare Power Spectral Density Estimation Methods

The Burg Method and Yule-Walker Method blocks return similar results for large frame sizes.

This table compares the features of the Burg Method block to the Covariance Method, Modified Covariance Method, and the Yule-Walker Method blocks.

|                        | <b>Burg</b>  | <b>Covariance</b>   | <b>Modified Covariance</b>  | <b>Yule-Walker</b>   |
|------------------------|--|---|---|--|
| <b>Characteristics</b> | Does not apply window to data  | Does not apply window to data   | Does not apply window to data   | Applies window to data   |
|                        | Minimizes the forward and backward prediction errors in the least squares sense, with the AR coefficients constrained to satisfy the L-D recursion | Minimizes the forward prediction error in the least squares sense                   | Minimizes the forward and backward prediction errors in the least squares sense   | Minimizes the forward prediction error in the least squares sense (also called autocorrelation method) |
| <b>Advantages</b>      | High resolution for short data records   | Better resolution than Yule-Walker for short data records (more accurate estimates) | High resolution for short data records  | Performs as well as other methods for large data records   |
|                        | Always produces a stable model   | Able to extract frequencies from data consisting of $p$ or more pure sinusoids      | Able to extract frequencies from data consisting of $p$ or more pure sinusoids<br>Does not suffer spectral line-splitting | Always produces a stable model   |
| <b>Disadvantages</b>   | Peak locations highly dependent on initial phase   | Can produce unstable models   | Can produce unstable models   | Performs relatively poorly for short data records  |



|                                      | <b>Burg</b>  | <b>Covariance</b>  | <b>Modified Covariance</b>                                   | <b>Yule-Walker</b>  |
|--------------------------------------|--|--|--|---|
|                                      | Can suffer spectral line-splitting for sinusoids in noise, or when order is very large | Frequency bias for estimates of sinusoids in noise           | Peak locations slightly dependent on initial phase           | Frequency bias for estimates of sinusoids in noise  |
|                                      | Frequency bias for estimates of sinusoids in noise                                     |  | Minor frequency bias for estimates of sinusoids in noise     |   |
| <b>Conditions for Nonsingularity</b> |  | Order must be less than or equal to 1/2 the input frame size | Order must be less than or equal to 2/3 the input frame size | Because of the biased estimate, the autocorrelation matrix is guaranteed to be positive-definite, hence nonsingular |

## Version History

Introduced before R2006a

## References

- [1] Kay, S. M. *Modern Spectral Estimation: Theory and Application*. Englewood Cliffs, NJ: Prentice-Hall, 1988.
- [2] Orfanidis, S. J. *Introduction to Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1995.
- [3] Orfanidis, S. J. *Optimum Signal Processing: An Introduction*. 2nd ed. New York, NY: Macmillan, 1985.

## Extended Capabilities

### C/C++ Code Generation

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

Usage notes and limitations:

- When the FFT length is not a power of 2, the executable generated from this block relies on prebuilt dynamic library files (.dll files) included with MATLAB. Use the `packNGo` function to package the code generated from this block and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see “How To Run a Generated Executable Outside MATLAB”.
- When the FFT length is a power of 2, you can generate standalone C and C++ code from this block.

## **See Also**

### **Blocks**

Burg AR Estimator | Covariance Method | Modified Covariance Method | Short-Time FFT | Yule-Walker Method

### **Topics**

“Spectral Analysis”

# Channelizer

Polyphase FFT analysis filter bank



## Libraries:

DSP System Toolbox / Filtering / Multirate Filters

## Description

The Channelizer block separates a broadband input signal into multiple narrow subbands using an FFT-based analysis filter bank. The filter bank uses a prototype lowpass filter and is implemented using a polyphase structure. You can specify the filter coefficients directly or through design parameters. When you specify the design parameters, the filter is designed using the `designMultirateFIR` function.

This block accepts variable-size inputs, so you can change the size of each input channel during simulation. However, the number of channels cannot change.

## Input/Output Ports

### Input

**x** — Broadband signal

*L*-by-1 column vector | *L*-by-*N* matrix

Specify the input broadband signal that the channelizer splits into multiple narrow bands.

- When you input a variable-size signal (frame length changes during simulation), the frame length of the signal can be arbitrary, that is, the input frame length does not have to be a multiple of the decimation factor.
- When you input a fixed-size signal (frame length does not change during simulation), the frame length can be arbitrary only when you select the **Allow arbitrary frame length for fixed-size input signals** parameter.

This port is unnamed until you set **Polyphase filter specification** to **Coefficients** and select the **Specify coefficients from input port** parameter.

Data Types: `single` | `double`

**coeffs** — Prototype lowpass filter coefficients

row vector

Coefficients of the prototype lowpass filter. There must be at least one coefficient per frequency band. If the length of the lowpass filter is less than the number of frequency bands, the block zero-pads the coefficients.

If you specify complex coefficients, the block designs a prototype filter that is centered at a nonzero frequency, also known as a bandpass filter. The modulated versions of the prototype bandpass filter appear with respect to the prototype filter and are wrapped around the frequency range  $[-F_s, F_s]$ .

### Dependencies

This port appears when you set **Polyphase filter specification** to `Coefficients` and select the **Specify coefficients from input port** parameter.

Data Types: `single` | `double`

### Output

**Port 1** — Multiple narrowband signals

$L/M$ -by- $M$  |  $L/M$ -by- $M$ -by- $N$  array

Multiple narrow subbands of the input broadband signal. Each narrow band signal forms a column in the output.

The dimensions of the output signal depend on the dimensions of the input signal and whether you select the **Allow arbitrary frame length for fixed-size input signals** parameter.

This table provides more details on the dimensions of the output signal when you input a fixed-size input signal.

#### Fixed-Size Input Signal

| Input Signal  | Output Signal Dimensions   |
|---|--|
| $L$ -by-1 column vector, where $L$ is a multiple of $M$     | $(L/M)$ -by- $M$   |
| $L$ -by-1 column vector, where $L$ is not a multiple of $M$ | $\text{ceil}(L/M)$ -by- $M$ when you select <b>Allow arbitrary frame length for fixed-size input signals</b> .<br><br>If you do not select <b>Allow arbitrary frame length for fixed-size input signals</b> , the block errors.          |
| $L$ -by- $N$ matrix, where $L$ is a multiple of $M$         | $(L/M)$ -by- $M$ -by- $N$  |
| $L$ -by- $N$ matrix, where $L$ is not a multiple of $M$     | $\text{ceil}(L/M)$ -by- $M$ -by- $N$ when you select <b>Allow arbitrary frame length for fixed-size input signals</b> .<br><br>If you do not select <b>Allow arbitrary frame length for fixed-size input signals</b> , the block errors. |

This table provides more details on the dimensions of the output signal when you input a variable-size input signal.

## Variable-Size Input Signal

| Input Signal  | Output Signal Dimensions   |
|---|--|
| $L$ -by-1 column vector, where $L$ is a multiple of $M$     | $(L/M)$ -by- $M$   |
| $L$ -by-1 column vector, where $L$ is not a multiple of $M$ | $\text{ceil}(L/M)$ -by- $M$<br><b>Allow arbitrary frame length for fixed-size input signals</b> parameter setting is ignored.          |
| $L$ -by- $N$ matrix, where $L$ is a multiple of $M$         | $(L/M)$ -by- $M$ -by- $N$  |
| $L$ -by- $N$ matrix, where $L$ is not a multiple of $M$     | $\text{ceil}(L/M)$ -by- $M$ -by- $N$<br><b>Allow arbitrary frame length for fixed-size input signals</b> parameter setting is ignored. |

Data Types: single | double

## Parameters

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

**Number of frequency bands** — Number of frequency bands

8 (default) | positive integer greater than 1

Number of frequency bands  $M$  into which the block separates the input broadband signal. This parameter indicates the FFT length and the decimation factor used by the algorithm.

**Polyphase filter specification** — Filter design parameters or coefficients

Number of taps per band and stopband attenuation (default) | Coefficients

- **Number of taps per band and stopband attenuation** — Specify the filter design parameters through the **Number of filter taps per frequency band** and **Stopband attenuation (dB)** parameters. When you specify the design parameters, the filter is designed using the `designMultirateFIR` function.
- **Coefficients** — Specify the filter coefficients directly using the **Prototype lowpass filter coefficients** parameter or input them through the **coeffs** port.

**DecimationFactor** — Decimation factor

8 (default) | positive integer

Decimation factor  $D$  specified as a positive integer less than or equal to the number of frequency bands  $M$ .

If the decimation factor  $D$  equals the number of frequency bands  $M$ , then the  $M/D$  ratio equals 1, and the channelizer is known as the maximally decimated channelizer.

If the  $M/D$  ratio is greater than 1, the output sample rate is different from the channel spacing, and the channelizer is known as the non-maximally decimated channelizer. If the ratio is an integer, the channelizer is known as the integer-oversampled channelizer. If the ratio is not an integer, say  $4/3$ ,

the channelizer is known as the rationally oversampled channelizer. For more details, see “Algorithm” on page 4-200.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**Number of filter taps per frequency band** — Number of filter coefficients per frequency band  
12 (default) | positive integer

Number of filter coefficients that each polyphase branch uses. The number of polyphase branches matches the number of frequency bands. The total number of filter coefficients for the prototype lowpass filter is given by **Number of frequency bands** × **Number of filter taps per frequency band**. For a given stopband attenuation, increasing the number of taps per band narrows the transition width of the filter. As a result, there is more usable bandwidth for each frequency band, at the expense of increased computation.

#### Dependencies

To enable this parameter, set **Polyphase filter specification** to Number of taps per band and stopband attenuation.

**Stopband attenuation (dB)** — Stopband attenuation  
80 (default) | positive real scalar

Stopband attenuation of the lowpass filter, in dB. This value controls the maximum amount of aliasing from one frequency band to the next. As the stopband attenuation increases, the passband ripple decreases.

#### Dependencies

To enable this parameter, set **Polyphase filter specification** to Number of taps per band and stopband attenuation.

**Specify coefficients from input port** — Flag to specify lowpass filter coefficients  
off (default) | on

When you select this parameter, the lowpass filter coefficients are input through the **coeffs** port. When you clear this parameter, the coefficients are specified on the block dialog through the **Prototype lowpass filter coefficients** parameter.

#### Dependencies

To enable this parameter, set **Polyphase filter specification** to Coefficients.

**Prototype lowpass filter coefficients** — Coefficients of prototype lowpass filter  
`rcosdesign(0.25,6,8,'sqrt')` (default) | row vector

Coefficients of the prototype lowpass filter. The default value is the coefficients vector that `rcosdesign(0.25,6,8,'sqrt')` returns. There must be at least one coefficient per frequency band. If the length of the lowpass filter is less than the number of frequency bands, the block zero-pads the coefficients.

If you specify complex coefficients, the block designs a prototype filter that is centered at a nonzero frequency, also known as a bandpass filter. The modulated versions of the prototype bandpass filter appear with respect to the prototype filter and are wrapped around the frequency range  $[-F_s, F_s]$ .

**Tunable:** Yes

### Dependencies

To enable this parameter, set **Polyphase filter specification** to `Coefficients` and clear the **Specify coefficients from input port** parameter.

**Allow arbitrary frame length for fixed-size input signals** — Allow arbitrary frame length for fixed-size input signals

off (default) | on

Specify whether fixed-size input signals (whose size does not change during simulation) can have an arbitrary frame length, where the frame length does not have to be a multiple of the decimation factor. The block uses this parameter setting only for fixed-size input signals and ignores this parameter if the input has a variable-size.

When the input signal is a variable-size signal, the signal can have arbitrary frame length, that is, the frame length does not have to be a multiple of the decimation factor.

For fixed-size input signals, if you:

- Select the **Allow arbitrary frame length for fixed-size input signals** parameter, the frame length of the signal does not have to be a multiple of the decimation factor. If the input is not a multiple of the decimation factor, then the output is generally a variable-size signal. Therefore, to support arbitrary input size, the block must also support variable-size operations, which you can enable by selecting the **Allow arbitrary frame length for fixed-size input signals** parameter.
- Clear the **Allow arbitrary frame length for fixed-size input signals** parameter, the input frame length must be a multiple of the decimation factor.

**Simulate using** — Type of simulation to run

Interpreted execution (default) | Code generation

Type of simulation to run. You can set this parameter to:

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

## Block Characteristics

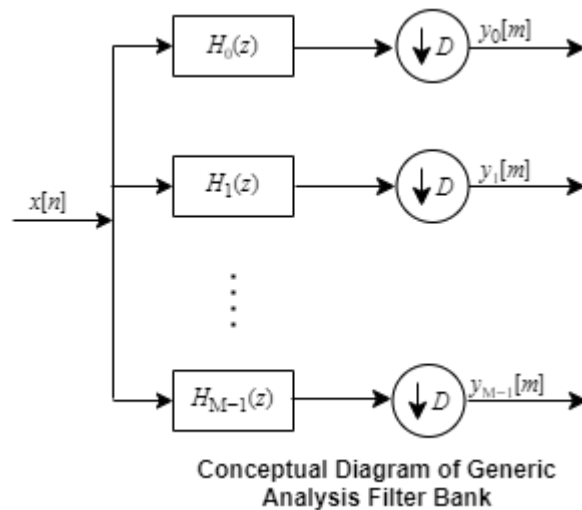
|                                 |                 |
|---------------------------------|-----------------|
| <b>Data Types</b>               | double   single |
| <b>Multidimensional Signals</b> | No              |

|                              |     |
|------------------------------|-----|
| <b>Variable-Size Signals</b> | Yes |
|------------------------------|-----|

## More About

### Analysis Filter Bank

The generic analysis filter bank consists of a series of parallel bandpass filters that split an input broadband signal,  $x[n]$ , into a series of narrow subbands. Each bandpass filter retains a different portion of the input signal. After the bandwidth is reduced by one of the bandpass filters, the signal is downsampled to a lower sampling rate commensurate with the new bandwidth.



### Prototype Lowpass Filter

To implement the analysis filter bank efficiently, the channelizer uses a prototype lowpass filter.

The prototype lowpass filter has an impulse response of  $h[n]$ , a normalized two-sided bandwidth of  $2\pi/M$ , and a cutoff frequency of  $\pi/M$ .  $M$  is the number of frequency bands, that is, the branches of the analysis filter bank. This value corresponds to the FFT length that the filter bank uses.  $M$  can be high on the order of 2048 or more. The stopband attenuation determines the minimum level of interference (aliasing) from one frequency band to another. The passband ripple must be small so that the input signal is not distorted in the passband.

The prototype lowpass filter corresponds to  $H_0(z)$  in the filter bank. The first branch of the filter bank contains  $H_0(z)$  followed by the decimator. The other  $M - 1$  branches contain filters that are modulated versions of the prototype filter. The modulation factor is given by the following equation:

$$e^{-jw_k n}, \quad w_k = 2\pi k/M, \quad k = 0, 1, \dots, M - 1$$

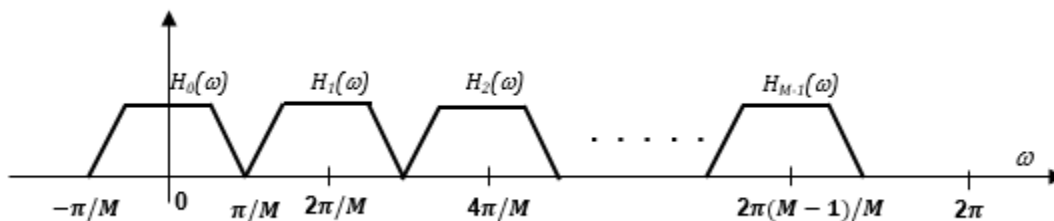
### Using the Prototype Lowpass Filter

The transfer function of the modulated  $k$ th bandpass filter is given by:

$$H_k(z) = H_0(z e^{-jw_k}), \quad w_k = 2\pi k/M, \quad k = 1, 2, \dots, M - 1$$

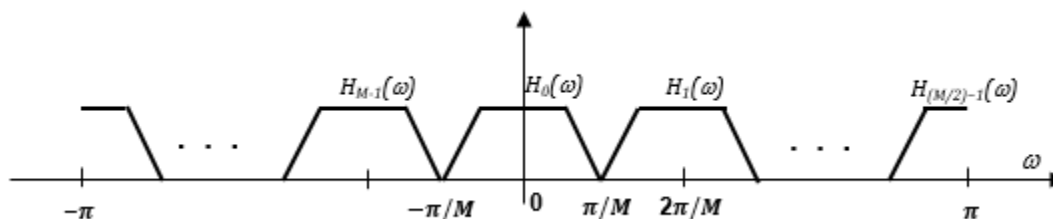
This figure shows the frequency response of  $M$  filters.





To obtain the frequency response characteristics of the filter  $H_k(z)$ , where  $k = 1, \dots, M-1$ , uniformly shift the frequency response of the prototype filter,  $H_0(z)$ , by multiples of  $2\pi/M$ . Each subband filter,  $H_k(z)$ ,  $\{k = 1, \dots, M-1\}$ , is derived from the prototype filter.

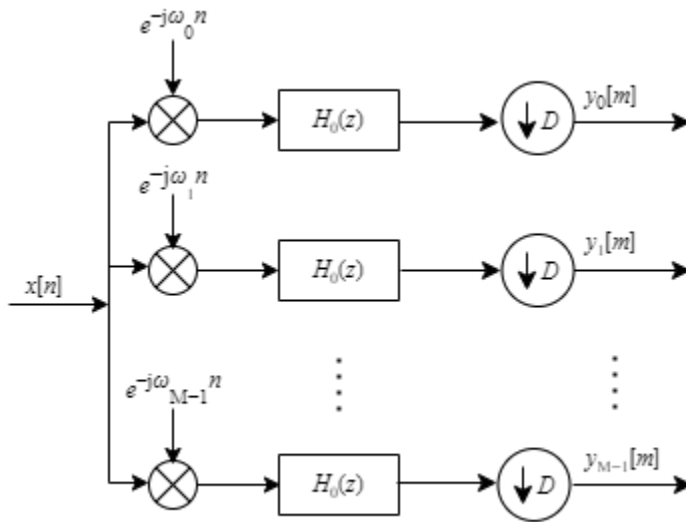
Following is an equivalent representation of the frequency response diagram with  $\omega$  ranging from  $[-\pi, \pi]$ .



### Shift Narrow Subbands to Baseband

The frequency components in the input signal,  $x[n]$ , are translated in frequency to baseband by multiplying  $x[n]$  with the complex exponentials,  $e^{-jw_k n}$ ,  $w_k = 2\pi k/M$ ,  $k = 1, 2, \dots, M-1$ , where  $w_k = 2\pi k/M$ , and  $k = 1, 2, \dots, M-1$ . The resulting product signals are passed through the lowpass filters,  $H_0(z)$ . The output of the lowpass filter is relatively narrow in bandwidth. Downsample the signal commensurate with the new bandwidth. Choose a decimation factor,  $D \leq M$ , where  $M$  is the number of branches of the analysis filter bank. When  $D < M$ , the channelizer is known as oversampled or non-maximally decimated channelizer.

The figure shows an analysis filter bank that uses the prototype lowpass filter.



Conceptual Diagram

$y_1[m], y_2[m], \dots, y_{M-1}[m]$  are narrow subband signals translated into baseband.

## Algorithms

### Polyphase Implementation

The analysis filter bank can be implemented efficiently using the polyphase structure. For more details on the analysis filter bank, see Analysis Filter Bank on page 2-176.

To derive the polyphase structure, start with the transfer function of the prototype lowpass filter:

$$H_0(z) = b_0 + b_1z^{-1} + \dots + b_Nz^{-N}$$

$N + 1$  is the length of the prototype filter.

You can rearrange this equation as follows:

$$\begin{aligned}
 H_0(z) = & (b_0 + b_Mz^{-M} + b_{2M}z^{-2M} + \dots + b_{N-M+1}z^{-(N-M+1)}) + \\
 & z^{-1}(b_1 + b_{M+1}z^{-M} + b_{2M+1}z^{-2M} + \dots + b_{N-M+2}z^{-(N-M+1)}) + \\
 & \vdots \\
 & z^{-(M-1)}(b_{M-1} + b_{2M-1}z^{-M} + b_{3M-1}z^{-2M} + \dots + b_{N-M+1}z^{-(N-M+1)})
 \end{aligned}$$

$M$  is the number of polyphase components.

You can write this equation as:

$$H_0(z) = E_0(z^M) + z^{-1}E_1(z^M) + \dots + z^{-(M-1)}E_{M-1}(z^M)$$

$E_0(z^M), E_1(z^M), \dots, E_{M-1}(z^M)$  are polyphase components of the prototype lowpass filter  $H_0(z)$ .

The other filters in the filter bank,  $H_k(z)$ , where  $k = 1, \dots, M-1$ , are modulated versions of this prototype filter.

You can write the transfer function of the  $k^{\text{th}}$  modulated bandpass filter as  $H_k(z) = H_0(ze^{-jw_k})$ .

Replacing  $z$  with  $ze^{jw_k}$ ,

$$H_k(z) = h_0 + h_1 e^{-jw_k} z^{-1} + h_2 e^{-j2w_k} z^{-2} \dots + h_N e^{-jNw_k} z^{-N}$$

$N + 1$  is the length of the  $k^{\text{th}}$  filter.

In polyphase form, the equation is as follows:

$$H_k(z) = \begin{bmatrix} 1 & e^{-jw_k} & e^{-j2w_k} & \dots & e^{-j(M-1)w_k} \end{bmatrix} \begin{bmatrix} E_0(z^M) \\ z^{-1}E_1(z^M) \\ \vdots \\ z^{-(M-1)}E_{M-1}(z^M) \end{bmatrix}$$

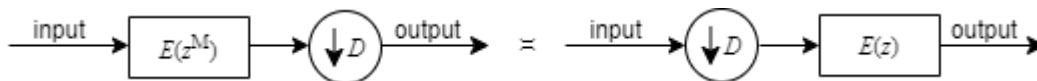
For all  $M$  channels in the filter bank, the transfer function  $H(z)$  is given by:

$$H(z) = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & e^{-jw_1} & e^{-j2w_1} & \dots & e^{-j(M-1)w_1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & e^{-jw_{M-1}} & e^{-j2w_{M-1}} & \dots & e^{-j(M-1)w_{M-1}} \end{bmatrix} \begin{bmatrix} E_0(z^M) \\ z^{-1}E_1(z^M) \\ \vdots \\ z^{-(M-1)}E_{M-1}(z^M) \end{bmatrix}$$

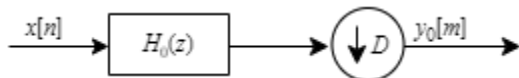
**Maximally decimated channelizer**

When  $D = M$ , the channelizer is known as the maximally decimated channelizer or critically sampled channelizer.

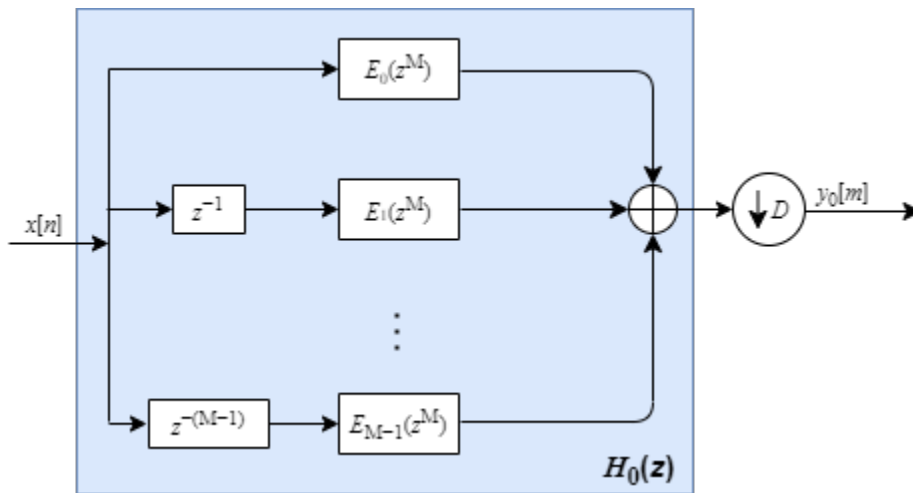
Here is the multirate noble identity for decimation, assuming that  $D = M$ .



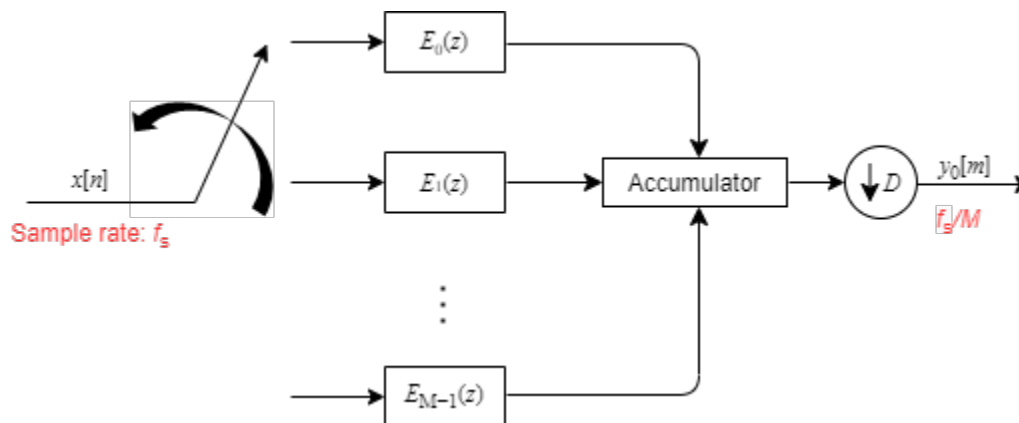
For example, consider the first branch of the filter bank that contains the lowpass filter.



Replace  $H_0(z)$  with its polyphase representation on page 2-178.



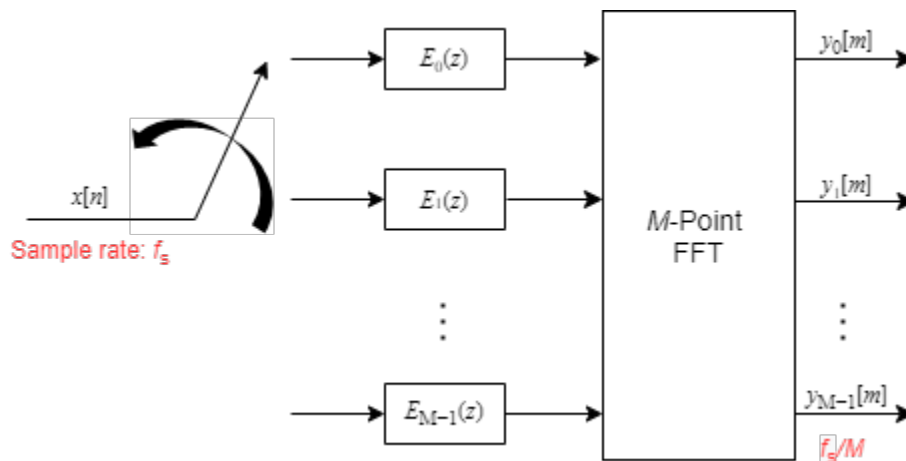
After applying the noble identity for decimation, you can replace the delays and the decimation factor with a commutator switch. The switch starts on the first branch 0 and moves in the counterclockwise direction as shown in the following diagram. The accumulator at the output receives the processed input samples from each branch of the polyphase structure and accumulates these processed samples until the switch goes to branch 0. When the switch goes to branch 0, the accumulator outputs the accumulated value.



For all  $M$  channels in the filter bank, the transfer function  $H(z)$  is given by:

$$H(z) = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & e^{-jw_1} & e^{-j2w_1} & \dots & e^{-j(M-1)w_1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & e^{-jw_{M-1}} & e^{-j2w_{M-1}} & \dots & e^{-j(M-1)w_{M-1}} \end{bmatrix} \begin{bmatrix} E_0(z) \\ E_1(z) \\ \vdots \\ E_{M-1}(z) \end{bmatrix}$$

The matrix on the left is a discrete Fourier transform (DFT) matrix. With the DFT matrix, the efficient implementation of the lowpass prototype-based filter bank looks like this.



When the first input sample is delivered, the switch feeds this input to the branch 0 and the channelizer computes the first set of output values. As more input samples come in, the switch moves in the counterclockwise direction through branches  $M-1, M-2$ , all the way up to branch 0, delivering one sample at a time to each branch. When the switch comes to branch 0, the channelizer outputs the next set of output values. This process continues as the data keeps coming in. Every time the switch comes to the first branch 0, the channelizer outputs  $y_0[m], y_1[m], \dots, y_{M-1}[m]$ . Each branch in the channelizer effectively outputs one sample for every  $M$  samples it receives. Hence, the sample rate at the output of the channelizer is  $f_s/M$ .

### Non-maximally decimated or oversampled channelizer

When  $D < M$ , the channelizer is known as the non-maximally decimated channelizer or oversampled channelizer. In this configuration, the output sample rate is different from the channel spacing. The non-maximally decimated channelizers offer increased design freedom, but at the expense of increasing computational cost.

If the ratio  $M/D$  equals an integer that is greater than 1 and is less than or equal to  $M-1$ , the channelizer is known as integer-oversampled channelizer. If the ratio  $M/D$  is not an integer, then the channelizer is known as rationally-oversampled channelizer.

In this configuration, when the first input sample is delivered, the switch feeds this input to branch 0 and the channelizer computes the first set of output values. As more input samples come in, the switch moves in the counterclockwise direction through branches  $D-1, D-2$ , all the way up to branch 0, delivering one sample at a time to each branch. When the switch comes to branch 0, the channelizer outputs the next set of output values. This process continues as the data keeps coming in. Every time the switch comes to the first branch 0, the channelizer outputs  $y_0[m], y_1[m], \dots, y_{M-1}[m]$ .

As more data keeps coming in and the switch feeds these samples to the first  $D$  addresses, the formal contents of these addresses are shifted to the next set of  $D$  addresses, and this process of data shift continues every time there is a new set of  $D$  input samples.

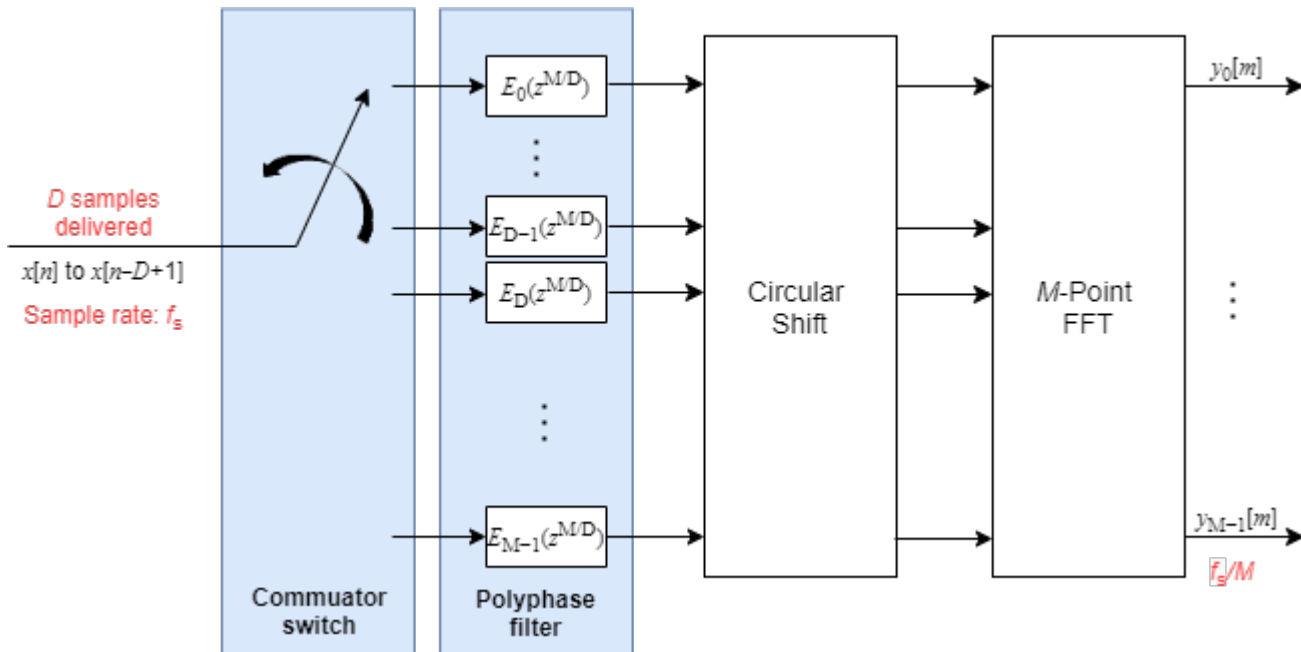
For every  $D$  input samples that are fed to the polyphase structure, the channelizer outputs  $M$  samples,  $y_0[m], y_1[m], \dots, y_{M-1}[m]$ . This process increases the output sample rate from  $f_s/M$  in the case of a maximally decimated channelizer, to  $f_s/D$  in the case of a non-maximally decimated channelizer.

For more details, see [2].

After each  $D$ -point data sequence is delivered to the partitioned  $M$ -stage polyphase filter, the outputs of the  $M$  stages are computed and conditioned for delivery to the  $M$ -point FFT. The data shifting

through the filter introduces frequency-dependent phase shift. To correct for this phase shift and alias all bands to DC, a circular shift buffer is inserted after the polyphase filters and before the  $M$ -point FFT.

With the commutator switch followed by  $M$ -stage polyphase filter, circular shift buffer, and a DFT matrix, the efficient implementation of the lowpass prototype-based filter bank looks like this.



## Version History

Introduced in R2017a

### R2022b: Support for arbitrary input frame length

Channelizer block supports input signals with arbitrary frame lengths when the:

- Input signal is a fixed-size signal (frame length does not change during simulation) and you select the **Allow arbitrary frame length for fixed-size input signals** parameter
- Input signal is a variable-size signal (frame length changes during simulation)

The frame length is arbitrary when it is not a multiple of the decimation factor.

## References

- [1] Harris, Fredric J, *Multirate Signal Processing for Communication Systems*, Prentice Hall PTR, 2004.
- [2] Harris, F.J., Chris Dick, and Michael Rice. "Digital Receivers and Transmitters Using Polyphase Filter Banks for Wireless Communications." *IEEE Transactions on Microwave Theory and Techniques*. 51, no. 4 (2003).

## Extended Capabilities

### C/C++ Code Generation

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

## See Also

### Blocks

Channel Synthesizer | Dyadic Analysis Filter Bank | Two-Channel Analysis Subband Filter

### Objects

`dsp.Channelizer` | `dsp.ChannelSynthesizer` | `dsp.SubbandAnalysisFilter` |  
`dsp.DyadicAnalysisFilterBank`

# Channel Synthesizer

Polyphase FFT synthesis filter bank



## Libraries:

DSP System Toolbox / Filtering / Multirate Filters

## Description

The Channel Synthesizer block merges multiple narrowband signals into a broadband signal by using an FFT-based synthesis filter bank. The filter bank uses a prototype lowpass filter and is implemented using a polyphase structure. You can specify the filter coefficients directly or through design parameters. When you specify the design parameters, the filter is designed using the `designMultirateFIR` function.

This block also accepts variable-size inputs. That is, during the simulation, you can change the size of each input channel. The number of channels cannot change.

## Input/Output Ports

### Input

**x** — Input narrowband signals  
*L*-by-*M* matrix | *L*-by-*M*-by-*N* array

Input narrowband signals, which the channel synthesizer merges to form the broadband signal. Each narrowband signal forms a column in the input signal. The number of columns in the input correspond to the number of frequency bands of the filter bank. If the input is three-dimensional, each matrix corresponds to a separate channel.

This port is unnamed until you set **Polyphase filter specification** to `Coefficients` and select the **Specify coefficients from input port** parameter.

Data Types: `single` | `double`

**coeffs** — Prototype lowpass filter coefficients  
 row vector

Coefficients of the prototype lowpass filter. There must be at least one coefficient per frequency band. If the length of the lowpass filter is less than the number of frequency bands, the block zero-pads the coefficients.

If you specify complex coefficients, the block designs a prototype filter that is centered at a nonzero frequency, also known as a bandpass filter. The modulated versions of the prototype bandpass filter appear with respect to the prototype filter and are wrapped around the frequency range  $[-F_s, F_s]$ .



### Dependencies

This port appears when you set **Polyphase filter specification** to `Coefficients` and select the **Specify coefficients from input port** parameter.

Data Types: `single` | `double`

### Output

**Port\_1** — Broadband signal

$L \times M$ -by-1 vector |  $L \times M$ -by- $N$  matrix

Broadband signal the channel synthesizer forms from the multiple input narrow subbands.

If the input is one of the following:

- $L$ -by- $M$  matrix — The output is a  $L \times M$ -by-1 vector.  $M$  is the number of frequency bands.
- $L$ -by- $M$ -by- $N$  matrix — The output is a  $L \times M$ -by- $N$  matrix.

Data Types: `single` | `double`

### Parameters

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

**Polyphase filter specification** — Filter design parameters or coefficients

Number of taps per band and stopband attenuation (default) | `Coefficients`

- Number of taps per band and stopband attenuation — Specify the filter design parameters through the **Number of filter taps per frequency band** and **Stopband attenuation (dB)** parameters. When you specify the design parameters, the filter is designed using the `designMultirateFIR` function.
- `Coefficients` — Specify the filter coefficients directly using the **Prototype lowpass filter coefficients** parameter or input them through the **coeffs** port.

**Number of filter taps per frequency band** — Number of filter coefficients per frequency band

12 (default) | positive integer

Number of filter coefficients that each polyphase branch uses. The number of polyphase branches matches the number of frequency bands. The total number of filter coefficients for the prototype lowpass filter is given by the product of the number of frequency bands and the number of filter taps per frequency band. The number of frequency bands equals the number of columns in the input. For a given stopband attenuation, increasing the number of taps per band narrows the transition width of the filter. As a result, there is more usable bandwidth for each frequency band, at the expense of increased computation.

### Dependencies

To enable this parameter, set **Polyphase filter specification** to `Number of taps per band and stopband attenuation`.

**Stopband attenuation (dB)** — Stopband attenuation

80 (default) | positive real scalar

Stopband attenuation of the lowpass filter, in dB. This value controls the maximum amount of aliasing from one frequency band to the next. As the stopband attenuation increases, the passband ripple decreases.

**Dependencies**

To enable this parameter, set **Polyphase filter specification** to Number of taps per band and stopband attenuation.

**Specify coefficients from input port** — Flag to specify lowpass filter coefficients  
off (default) | on

When you select this check box, the lowpass filter coefficients are input through the **coeffs** port. When you clear this check box, the coefficients are specified on the block dialog through the **Prototype lowpass filter coefficients** parameter.

**Dependencies**

To enable this parameter, set **Polyspace filter specification** to Coefficients.

**Prototype lowpass filter coefficients** — Coefficients of prototype lowpass filter  
`rcosdesign(0.25,6,8,'sqrt')` (default) | row vector

This parameter specifies the coefficients of the prototype lowpass filter. The default value is the coefficients vector that `rcosdesign(0.25,6,8,'sqrt')` returns. There must be at least one coefficient per frequency band. If the length of the lowpass filter is less than the number of frequency bands, the block zero-pads the coefficients.

If you specify complex coefficients, the block designs a prototype filter that is centered at a nonzero frequency, also known as a bandpass filter. The modulated versions of the prototype bandpass filter appear with respect to the prototype filter and are wrapped around the frequency range  $[-F_s, F_s]$ .

**Tunable:** Yes

**Dependencies**

To enable this parameter, set **Polyphase filter specification** to Coefficients and clear the **Specify coefficients from input port** parameter.

**Simulate using** — Type of simulation to run

Interpreted execution (default) | Code generation

Type of simulation to run. You can set this parameter to:

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

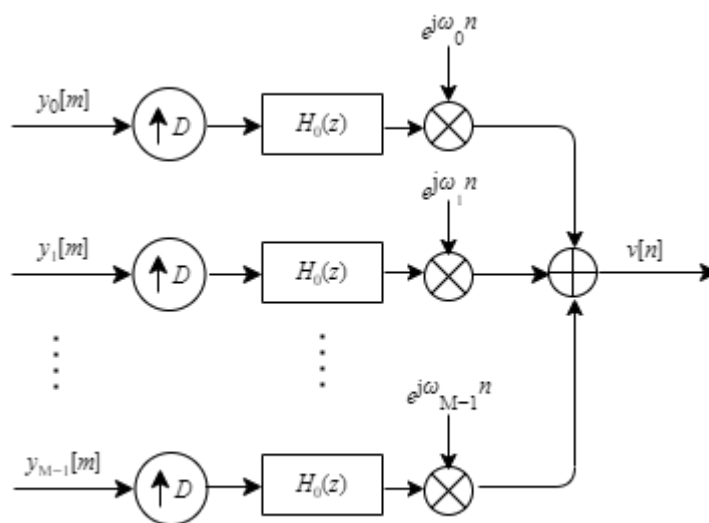
## Block Characteristics

|                                 |                 |
|---------------------------------|-----------------|
| <b>Data Types</b>               | double   single |
| <b>Multidimensional Signals</b> | No              |
| <b>Variable-Size Signals</b>    | Yes             |

## More About

### Synthesis Filter Bank

The synthesis filter bank consists of a set of parallel bandpass filters that merge multiple input narrowband signals,  $y_0[m]$ ,  $y_1[m]$ , ...,  $y_{M-1}[m]$  into a single broadband signal,  $v[n]$ . The input narrowband signals are in the baseband. Each narrowband signal is interpolated to a higher sampling rate by using the upsampler, and then filtered by the lowpass filter. A complex exponential that follows the lowpass filter centers the baseband signal around  $w_k$ .



Conceptual Diagram

### Prototype Lowpass Filter

To implement the synthesis filter bank efficiently, the synthesizer uses a prototype lowpass filter. This filter has an impulse response of  $h[n]$ , a normalized two-sided bandwidth of  $2\pi/M$ , and a cutoff frequency of  $\pi/M$ .  $M$  is the number of frequency bands, that is, the branches of the synthesis filter bank. This value corresponds to the FFT length that the filter bank uses.  $M$  can be high, in the order of 2048 or more. The stopband attenuation determines the minimum level of interference (aliasing) from one frequency band to another. The passband ripple must be small so that the input signal is not distorted in the passband.

The prototype lowpass filter models the first branch of the filter bank. The other  $M - 1$  branches are modeled by filters that are modulated versions of the prototype filter. The modulation factor is given by  $e^{jw_k n}$ ,  $w_k = 2\pi k/M$ ,  $k = 0, 1, \dots, M - 1$ .

The output of each bandpass filter forms a specific portion of the broadband signal. The output of all the branches are added to form the broadband signal,  $v[n]$ .

## Algorithms

### Polyphase Implementation

The synthesis filter bank can be implemented efficiently using the polyphase structure.

To derive the polyphase structure, start with the transfer function of the prototype lowpass filter.

$$H_0(z) = b_0 + b_1z^{-1} + \dots + b_Nz^{-N}$$

$N + 1$  is the length of the prototype filter.

You can rearrange this equation as follows:

$$H_0(z) = \begin{aligned} & (b_0 + b_Mz^{-M} + b_{2M}z^{-2M} + \dots + b_{N-M+1}z^{-(N-M+1)}) + \\ & z^{-1}(b_1 + b_{M+1}z^{-M} + b_{2M+1}z^{-2M} + \dots + b_{N-M+2}z^{-(N-M+1)}) + \\ & \quad \vdots \\ & z^{-(M-1)}(b_{M-1} + b_{2M-1}z^{-M} + b_{3M-1}z^{-2M} + \dots + b_Nz^{-(N-M+1)}) \end{aligned}$$

$M$  is the number of polyphase components.

You can write this equation as:

$$H_0(z) = E_0(z^M) + z^{-1}E_1(z^M) + \dots + z^{-(M-1)}E_{M-1}(z^M)$$

$E_0(z^M), E_1(z^M), \dots, E_{M-1}(z^M)$  are polyphase components of the prototype lowpass filter,  $H_0(z)$ .

The other filters in the filter bank,  $H_k(z)$ , where  $k = 1, \dots, M - 1$ , are modulated versions of this prototype filter.

You can write the transfer function of the  $k$ th modulated bandpass filter as  $H_k(z) = H_0(ze^{jw_k})$ .

Replacing  $z$  with  $ze^{jw_k}$ ,

$$H_k(z) = h_0 + h_1e^{jw_k}z^{-1} + h_2e^{j2w_k}z^{-2} \dots + h_Ne^{jNw_k}z^{-N}$$

$N + 1$  is the length of the  $k$ th filter.

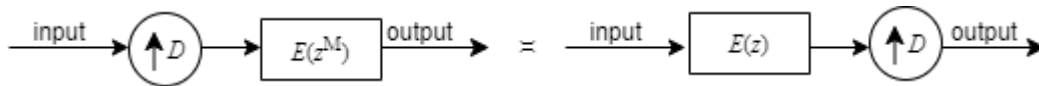
In polyphase form, the equation is as follows:

$$H_k(z) = \begin{bmatrix} 1 & e^{jw_k} & e^{j2w_k} & \dots & e^{j(M-1)w_k} \end{bmatrix} \begin{bmatrix} E_0(z^M) \\ z^{-1}E_1(z^M) \\ \vdots \\ z^{-(M-1)}E_{M-1}(z^M) \end{bmatrix}$$

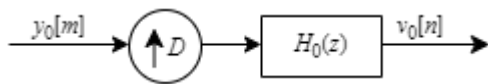
For all  $M$  channels in the filter bank, the MIMO transfer function,  $H(z)$ , is given by:

$$H(z) = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & e^{jw_1} & e^{j2w_1} & \dots & e^{j(M-1)w_1} \\ \vdots & & & & \\ 1 & e^{jw_M-1} & e^{j2w_M-1} & \dots & e^{j(M-1)w_M-1} \end{bmatrix} \begin{bmatrix} E_0(z^M) \\ z^{-1}E_1(z^M) \\ \vdots \\ z^{-(M-1)}E_{M-1}(z^M) \end{bmatrix}$$

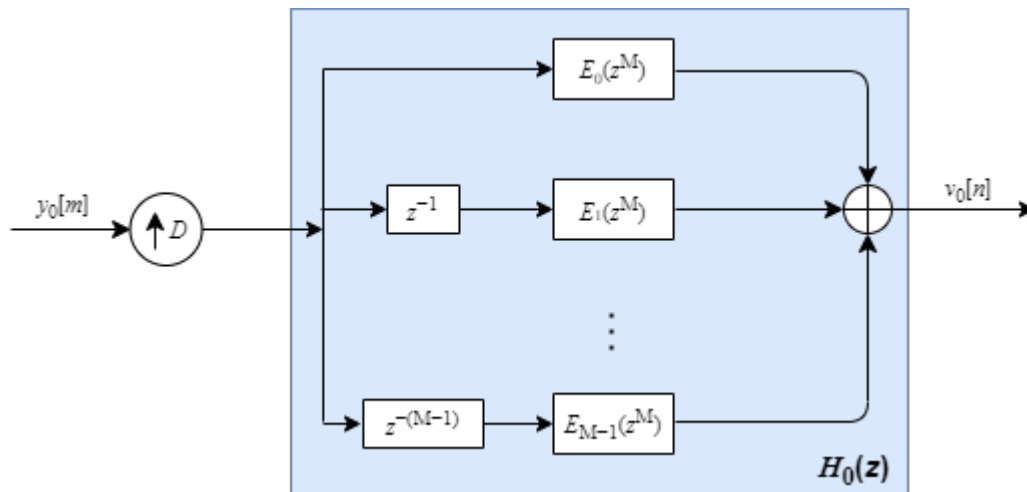
Here is the multirate noble identity for interpolation, assuming that  $D = M$ :



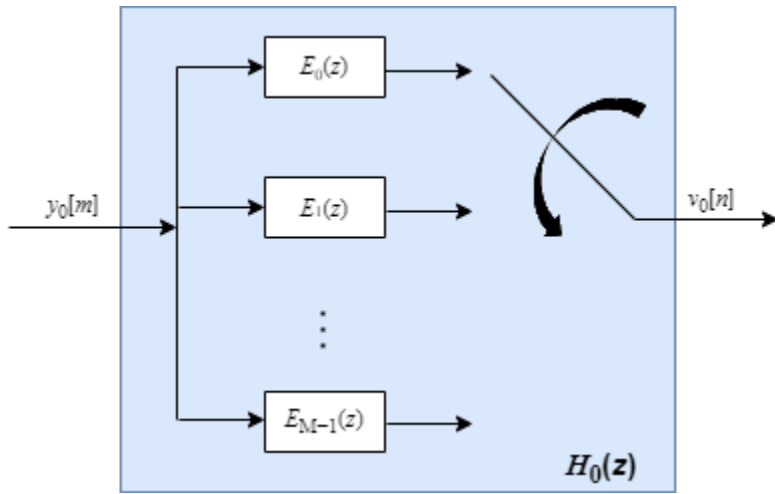
For illustration, consider the first branch of the filter bank that contains the lowpass filter.



Replace  $H_0(z)$  with its polyphase representation.



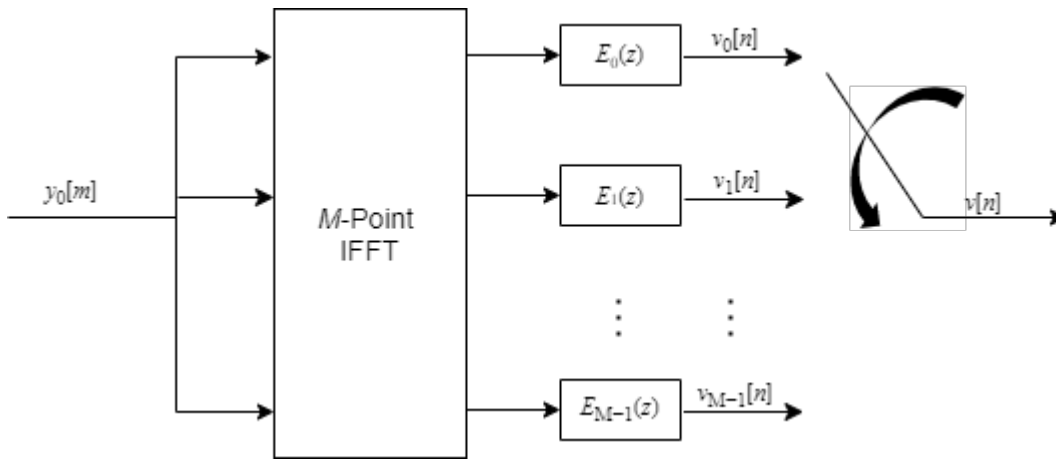
After applying the noble identity for interpolation, you can replace the delays, interpolation factor, and the adder with a commutator switch.



For all the  $M$  channels in the filter bank, the MIMO transfer function,  $H(z)$ , is given by:

$$H(z) = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & e^{jw_1} & e^{j2w_1} & \dots & e^{j(M-1)w_1} \\ & & & \vdots & \\ 1 & e^{jw_{M-1}} & e^{j2w_{M-1}} & \dots & e^{j(M-1)w_{M-1}} \end{bmatrix} \begin{bmatrix} E_0(z) \\ E_1(z) \\ \vdots \\ E_{M-1}(z) \end{bmatrix}$$

The matrix on the left is an IDFT matrix. With the IDFT matrix, the efficient implementation of the lowpass prototype based filter bank looks like the following.



## Version History

Introduced in R2017a

## References

- [1] Harris, Fredric J, *Multirate Signal Processing for Communication Systems*, Prentice Hall PTR, 2004.

[2] Harris, F.J., Chris Dick, and Michael Rice. "Digital Receivers and Transmitters Using Polyphase Filter Banks for Wireless Communications." *IEEE Transactions on Microwave Theory and Techniques*. Vol. 51, Number 4, April 2003.

## Extended Capabilities

### C/C++ Code Generation

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

## See Also

### Blocks

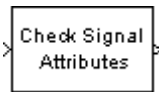
Channelizer | Dyadic Synthesis Filter Bank | Two-Channel Synthesis Subband Filter

### Objects

`dsp.ChannelSynthesizer` | `dsp.Channelizer` | `dsp.SubbandSynthesisFilter` | `dsp.DyadicSynthesisFilterBank`

## Check Signal Attributes

Error when input signal does or does not match selected attributes exactly



### Libraries:

DSP System Toolbox / Signal Management / Signal Attributes

### Description

The Check Signal Attributes block terminates the simulation with an error when the input characteristics differ from the characteristics you specify in the block parameters.

The Check Signal Attributes block can test for up to five different signal attributes. When you select **Ignore** for any parameter, the block does not check the signal for the corresponding attribute. For example, when you set **Complexity** to **Ignore**, neither real nor complex inputs cause the block to generate an error.

### Ports

#### Input

**Input** — Input signal

scalar | vector | matrix | N-D array

Input signal to be checked, specified as a scalar, vector, matrix, or N-D array. The input can be any real- or complex-valued signal.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated

#### Output

**Output** — Output signal

scalar | vector | matrix | N-D array

Output signal, returned as a scalar, vector, matrix, or N-D array, with the same values as the input signal.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated

### Parameters

**Error when input** — Generate error when input

Does not match attributes exactly (default) | Matches attributes exactly

Specify whether the block generates an error when the input *does* or *does not* possess *all* of the required attributes.



When you set this parameter to **Does not match attributes exactly**, the block generates an error when the input fails to match *any* of the specified attributes. Only signals that possess *all* of the specified attributes propagate to the output unaltered and do not cause the block to generate an error.

When you set this parameter to **Matches attributes exactly**, the block generates an error only when the input possesses *all* specified attributes. Signals that do not possess *all* of the specified attributes propagate to the output unaltered, and do not cause the block to generate an error.

### **Complexity** — Specify complexity

Ignore (default) | Real | Complex

Specify the complexity for which you want to check the input as **Real** or **Complex**. You can display this information in a model by attaching a Probe block with **Probe complex signal** selected. Alternatively, in the **Debug** tab, select **Information Overlays > Port Data Type**.

When you set this parameter to **Ignore**, the block does not check the complexity of the input.

### **Dimensionality** — Dimensionality check

Ignore (default) | Is... | Is not...

Specify whether you want to check the input for compliance or noncompliance with the attributes in the subordinate **Dimensions** parameter. When you set this parameter to **Ignore**, the block does not check the dimensionality of the input.

### **Dimensions** — Specify dimensions

1-D (default) | 2-D | Scalar (1-D or 2-D) | Vector (1-D or 2-D) | Vector (1-D or 2-D) or scalar | Row vector (2-D) | Row vector (2-D) or scalar | Column vector (2-D) | Column vector (2-D) or scalar | Full matrix (2-D) | Square matrix (2-D)

Specify the dimensions for which you want to check the input.

Check the dimensionality of the input for compliance or noncompliance with the attributes in this parameter. See the following table.  $M$  and  $N$  are positive integers unless otherwise indicated.

| <b>Dimensions</b>      | <b>Is...</b>  | <b>Is not...</b>  |
|------------------------|---|---|
| 1-D                    | 1-D vector,<br>1-D scalar   | $M$ -by- $N$ matrix,<br>1-by- $N$ matrix (row vector),<br>$M$ -by-1 matrix (column vector),<br>1-by-1 matrix (2-D scalar) |
| 2-D                    | $M$ -by- $N$ matrix,<br>1-by- $N$ matrix (row vector),<br>$M$ -by-1 matrix (column vector),<br>1-by-1 matrix (2-D scalar) | 1-D vector,<br>1-D scalar   |
| Scalar<br>(1-D or 2-D) | 1-D scalar,<br>1-by-1 matrix (2-D scalar)   | 1-D vector with length > 1,<br>$M$ -by- $N$ matrix with $M > 1$ and/or $N > 1$  |

| Dimensions             | Is...  | Is not...   |
|------------------------|--|---|
| Vector<br>(1-D or 2-D) | 1-D vector,<br>1-D scalar,<br>1-by- $N$ matrix (row vector),<br>$M$ -by-1 matrix (column vector),<br>1-by-1 matrix (2-D scalar)<br>Vector (1-D or 2-D) or scalar | $M$ -by- $N$ matrix with $M > 1$ and $N > 1$  |
| Row Vector<br>(2-D)    | 1-by- $N$ matrix (row vector),<br>1-by-1 matrix (2-D scalar)<br>Row vector (2-D) or scalar   | 1-D vector,<br>1-D scalar,<br>$M$ -by- $N$ matrix with $M > 1$  |
| Column Vector<br>(2-D) | $M$ -by-1 matrix (column vector), 1-by-1<br>matrix (2-D scalar)<br>Column vector (2-D) or scalar   | 1-D vector,<br>1-D scalar,<br>$M$ -by- $N$ matrix with $N > 1$  |
| Full matrix            | $M$ -by- $N$ matrix with $M > 1$ and $N > 1$   | 1-D vector,<br>1-D scalar,<br>1-by- $N$ matrix (row vector),<br>$M$ -by-1 matrix (column vector),<br>1-by-1 matrix (2-D scalar) |
| Square matrix          | $M$ -by- $N$ matrix with $M = N$ ,<br>1-D scalar,<br>1-by-1 matrix (2-D scalar)  | $M$ -by- $N$ matrix with $M \neq N$ ,<br>1-D vector,<br>1-by- $N$ matrix (row vector),<br>$M$ -by-1 matrix (column vector)      |

In the **Debug** tab, when you select **Information Overlays > Signal Dimensions**, Simulink displays the size of a 1-D vector signal as an unbracketed integer, and displays the dimension of a 2-D signal as a pair of bracketed integers, [MxN]. Simulink *does not display* any size information for a 1-D or 2-D scalar signal. You can also display dimension information for a signal in a model by attaching a Probe block with **Probe signal dimensions** selected.

### Dependencies

To enable this parameter, set **Dimensionality** to **Is...** or **Is not...**

### Data type — Data type check

Ignore (default) | **Is...** | **Is not...**

Specify whether you want to check the input for compliance **Is...** or noncompliance **Is not...** with the attributes in the subordinate **General data type** parameter. When you set this parameter to **Ignore**, the block does not check the input data type.

See the following table. You can individually select any of the specific data types listed in the (**Is...**) column from the subordinate **Specific data type** parameter.

| General Data Type | Is...   | Is not...  |
|-------------------|---------|--|
| Boolean           | boolean | single, double, uint8, int8, uint16, int16, uint32, int32, fixed point, enumerated |

| General Data Type         | Is...   | Is not...   |
|---------------------------|---|---|
| Enumerated                | A user-defined enumerated data type. See "Data Types" (Simulink) .                | boolean, single, double, uint8, int8, uint16, int16, uint32, int32, fixed point |
| Floating point            | single, double  | boolean, uint8, int8, uint16, int16, uint32, int32, fixed point, enumerated     |
| Floating point or Boolean | single, double, boolean   | uint8, int8, uint16, int16, uint32, int32, fixed point, enumerated              |
| Fixed point               | fixed point, uint8, int8, uint16, int16, uint32, int32                            | boolean, single, double, enumerated   |
| Integer                   | Signed integer<br>int8, int16, int32<br>Unsigned integer<br>uint8, uint16, uint32 | boolean, single, double, fixed point, enumerated                                |

To display data type information, in the **Debug** tab, select **Information Overlays > Port Data Type**.

#### General data type — General data type check

Boolean (default) | Enumerated | Floating-point | Floating-point or boolean | Fixed-point | Integer

Specify the general data type for which you want to check the input.

#### Dependencies

To enable this parameter, set **Data type** to **Is...** or **Is not...**

#### Specific floating-point — Specific floating-point check

Any floating-point (default) | Double precision | Single precision

Specify the floating-point data type for which you want to check the input.

#### Dependencies

To enable this parameter, set **General data type** to **Floating-point** or **Floating-point or boolean**.

#### Specific fixed-point — Specific fixed-point check

Any fixed-point (default) | Signed fixed-point | Unsigned fixed-point

Specify the fixed-point data type for which you want to check the input.

#### Dependencies

To enable this parameter, set **General data type** to **Fixed-point**.

**Specific integer** — Specific integer check

Any integer (default) | Signed integer | Unsigned integer | int8 | int16 | int32 | uint8 | uint16 | uint32

Specify the integer data type for which you want to check the input.

**Dependencies**

To enable this parameter, set **General data type** to Integer.

**Sample time** — Sample time check

Ignore (default) | Discrete | Continuous

Specify the sample time for which you want to check the input as **Discrete** or **Continuous**. When you set this parameter to **Ignore**, the block does not check the sample time of the input.

The block checks whether the signal is discrete time or continuous time. In the **Debug** tab, when you select **Information Overlays > Colors**, Simulink displays continuous-time signal lines in black or grey and discrete-time signal lines in colors corresponding to the relative rate.

When you attach a Probe block with **Probe sample time** enabled to a continuous-time signal, the block icon displays  $T_s: [0 \ T_o]$ , where  $T_o$  is the sample time offset. Valid values of  $T_o$  for continuous-time signals are 0 and 1. When  $T_o$  is 0, updates occur at every major and minor time step. When  $T_o$  is 1, updates occur only at major time steps and the sample time is *fixed in minor time step*.

When you attach a Probe block with **Probe sample time** enabled to a discrete-time signal, the block icon displays  $T_s: [T_s \ T_o]$  for sample-based signals, and  $T_f: [T_f \ T_o]$  for frame-based signals.  $T_s$  and  $T_f$  are the positive sample period and frame period, respectively.  $T_o$  is the offset, such that  $0 \leq \text{offset} < \text{period}$ . Frame-based signals are almost always discrete time.

**Block Characteristics**

|                                 |  |
|---------------------------------|--|
| <b>Data Types</b>               | Boolean   double   enumerated   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no   |
| <b>Multidimensional Signals</b> | no   |
| <b>Variable-Size Signals</b>    | yes  |
| <b>Zero-Crossing Detection</b>  | no   |

**Version History**

Introduced before R2006a

**Extended Capabilities****C/C++ Code Generation**

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

**Fixed-Point Conversion**

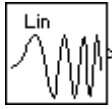
Design and simulate fixed-point systems using Fixed-Point Designer™.

**See Also**

[Buffer](#) | [Convert 1-D to 2-D](#) | [Convert 2-D to 1-D](#) | [Data Type Conversion](#) | [Inherit Complexity](#) | [Probe](#) | [Reshape](#) | [Submatrix](#)

# Chirp

Generate swept-frequency cosine (chirp) signal



**Libraries:**  
DSP System Toolbox / Sources

## Description

The Chirp block outputs a swept-frequency cosine (chirp) signal with unity amplitude and continuous phase. To specify the desired output chirp signal, you must define its instantaneous frequency function, also known as the output frequency sweep. The frequency sweep can be linear, quadratic, or logarithmic, and repeats once every **Sweep time** by default. For a description of the algorithms used by the Chirp block, see “Algorithms” on page 2-202.

## Ports

### Output

**Port\_1** — Swept-frequency cosine (chirp) signal  
scalar | vector

Swept-frequency cosine (chirp) signal. In **Linear**, **Logarithmic**, and **Quadratic** modes (set by the **Frequency sweep** parameter), the block outputs a swept-frequency cosine with instantaneous frequency values specified by the frequency and time parameters. In **Swept cosine** mode, the block outputs a swept-frequency cosine with a linear instantaneous output frequency that may differ from the one specified by the frequency and time parameters.

For more information about how the block computes the output, see “Algorithms” on page 2-202.

Data Types: `single` | `double`

## Parameters

**Frequency sweep** — Type of frequency sweep

`Linear` (default) | `Swept cosine` | `Logarithmic` | `Quadratic`

The type of output instantaneous frequency sweep,  $f_i(t)$ : **Linear**, **Logarithmic**, **Quadratic**, or **Swept cosine**. For more information, see “Shaping the Frequency Sweep” on page 2-201 and “Algorithms” on page 2-202.

### Limitations

When you want a linearly swept chirp signal, we recommend that you use a **Linear** frequency sweep. Though a **Swept cosine** frequency sweep also yields a linearly swept chirp signal, the output might have unexpected frequency content.

- The swept cosine sweep value at the **Target time** is not necessarily the **Target frequency**. This is because the user-specified sweep is not the actual frequency sweep of the swept cosine output, as

noted in “Output Computation Method for Swept Cosine Frequency Sweep” on page 2-204. See the table Instantaneous Frequency Sweep Values for the actual value of the swept cosine sweep at the **Target time**.

- In **Swept cosine** mode, do not set the parameters so that  $1/T_{sw}$  is much greater than the values of the **Initial frequency** and **Target frequency** parameters. In such cases, the actual frequency content of the swept cosine sweep might be closer to  $1/T_{sw}$ , far exceeding the **Initial frequency** and **Target frequency** parameter values.

**Sweep mode** — Sweep mode

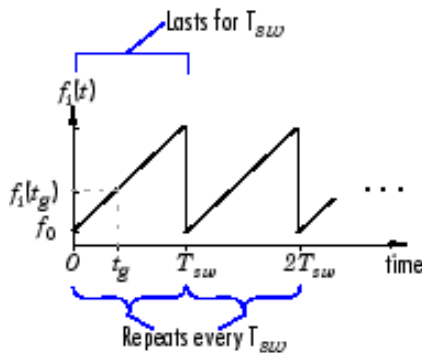
Unidirectional (default) | Bidirectional

The **Sweep mode** parameter determines whether your sweep is unidirectional or bidirectional, which affects the shape of your output frequency sweep (see “Shaping the Frequency Sweep” on page 2-201). The following table describes the characteristics of unidirectional and bidirectional sweeps.

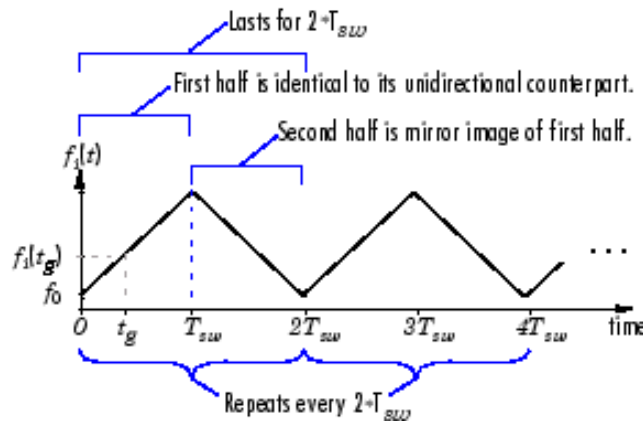
| Sweep Mode Parameter Settings | Sweep Characteristics  |
|-------------------------------|--|
| Unidirectional                | <ul style="list-style-type: none"> <li>• Lasts for one <b>Sweep time</b>, <math>T_{sw}</math></li> <li>• Repeats once every <math>T_{sw}</math></li> </ul>   |
| Bidirectional                 | <ul style="list-style-type: none"> <li>• Lasts for twice the <b>Sweep time</b>, <math>2T_{sw}</math></li> <li>• Repeats once every <math>2T_{sw}</math></li> <li>• First half is identical to its unidirectional counterpart.</li> <li>• Second half is a mirror image of the first half.</li> </ul> |

The following diagram illustrates a linear sweep in both sweep modes. For information on setting the frequency values in your sweep, see “Setting Instantaneous Frequency Sweep Values” on page 2-202.

**Unidirectional Linear Sweep**



**Bidirectional Linear Sweep**



**Initial frequency (Hz)** — Initial frequency

1000 (default) | scalar

For Linear, Quadratic, and Swept cosine sweeps, the initial frequency,  $f_0$ , of the output chirp signal. You can specify the **Initial frequency (Hz)** as a scalar, greater than or equal to zero. For Logarithmic sweeps, **Initial frequency** is one less than the actual initial frequency of the sweep.

Also, when the sweep is **Logarithmic**, you must set the **Initial frequency** to be less than the **Target frequency**.

For more information, see “Setting Instantaneous Frequency Sweep Values” on page 2-202.

**Tunable:** Yes

**Target frequency (Hz)** — Target frequency

4000 (default) | scalar

For **Linear**, **Quadratic**, and **Logarithmic** sweeps, the instantaneous frequency,  $f_i(t_g)$ , of the output at the **Target time**,  $t_g$ . You can specify the **Target frequency (Hz)** as a scalar, greater than or equal to zero. For a **Swept cosine** sweep, **Target frequency** is the instantaneous frequency of the output at half the **Target time**,  $t_g/2$ . When **Frequency sweep** is **Logarithmic**, you must set the **Target frequency** to be greater than the **Initial frequency**.

For more information, see “Setting Instantaneous Frequency Sweep Values” on page 2-202.

**Tunable:** Yes

**Target time (s)** — Target time of sweep

1 (default) | scalar

For **Linear**, **Quadratic**, and **Logarithmic** sweeps, the time,  $t_g$ , at which the sweep reaches the **Target frequency**,  $f_i(t_g)$ . For a **Swept cosine** sweep, **Target time** is the time at which the sweep reaches  $2f_i(t_g) - f_0$ . **Target time** must be a scalar that is greater than or equal to zero, and less than or equal to **Sweep time**,  $T_{sw} \geq t_g$ .

For more information, see “Setting Instantaneous Frequency Sweep Values” on page 2-202.

**Tunable:** Yes

**Sweep time (s)** — Sweep time

1 (default) | scalar

In **Unidirectional Sweep mode**, the **Sweep time**,  $T_{sw}$ , is the period of the output frequency sweep. In **Bidirectional Sweep mode**, the **Sweep time** is half the period of the output frequency sweep. **Sweep time** must be a scalar that is greater than or equal to **Target time**,  $T_{sw} \geq t_g$ .

**Tunable:** Yes

**Initial phase (rad)** — Initial phase of cosine output

0 (default) | scalar

The phase,  $\phi_0$ , of the cosine output at  $t=0$ ;  $y_{chirp}(t) = \cos(\phi_0)$ . You can specify the **Initial phase (rad)** as a scalar that is greater than or equal to zero.

**Tunable:** Yes

**Sample time** — Output sample period



1/8000 (default) | positive scalar

The sample period,  $T_s$ , of the output. The output frame period is  $M_o T_s$ , where  $M_o$  is the number of samples per frame.

**Samples per frame** — Samples per frame

1 (default) | positive integer

The number of samples,  $M_o$ , to buffer into each output frame, specified as a positive integer scalar.

**Output data type** — Output data type

Double (default) | Single

The data type of the output, specified as single precision or double precision.

## Block Characteristics

|                                 |                 |
|---------------------------------|-----------------|
| <b>Data Types</b>               | double   single |
| <b>Direct Feedthrough</b>       | no              |
| <b>Multidimensional Signals</b> | no              |
| <b>Variable-Size Signals</b>    | no              |
| <b>Zero-Crossing Detection</b>  | no              |

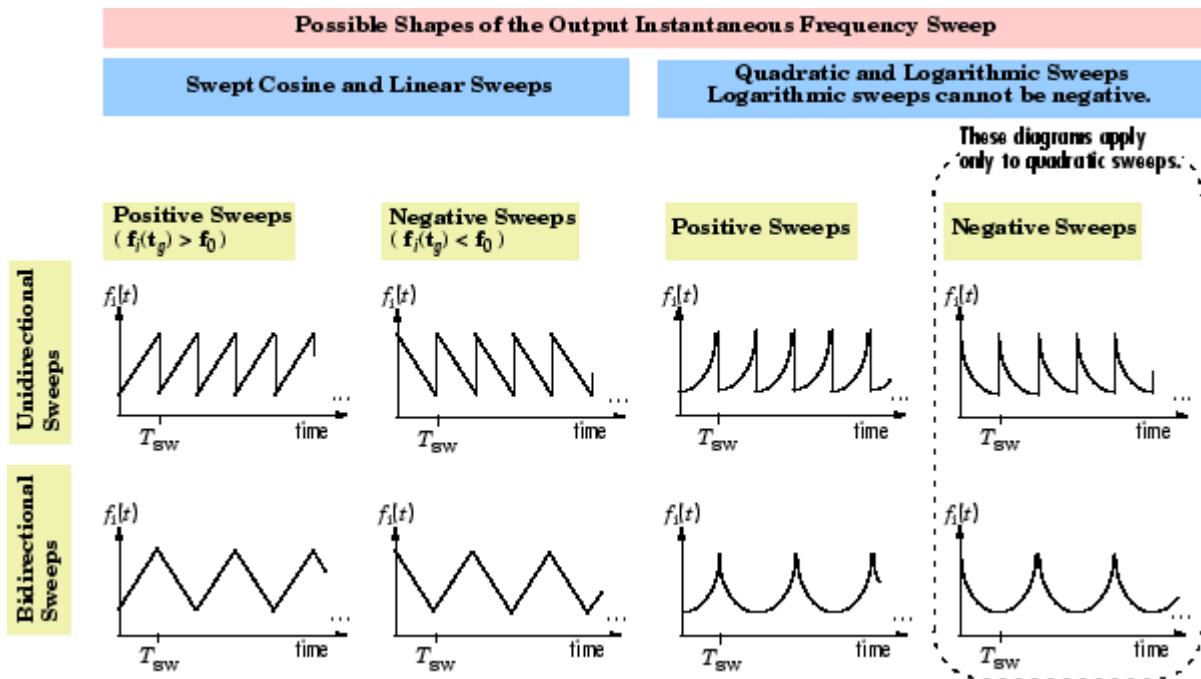
## More About

### Shaping the Frequency Sweep

You control the basic shape of the output instantaneous frequency sweep,  $f_i(t)$ , using the **Frequency sweep** and **Sweep mode** parameters.

| Parameters for Setting Sweep Shape | Possible Setting                                   | Parameter Description   |
|------------------------------------|--|---|
| <b>Frequency sweep</b>             | Linear<br>Quadratic<br>Logarithmic<br>Swept cosine | Determines whether the sweep frequencies vary linearly, quadratically, or logarithmically. Linear and swept cosine sweeps both vary linearly. |
| <b>Sweep mode</b>                  | Unidirectional<br>Bidirectional                    | Determines whether the sweep is unidirectional or bidirectional. For details, see "Sweep mode" on page 2-0                                    |

The following diagram illustrates the possible shapes of the frequency sweep that you can obtain by setting the **Frequency sweep** and **Sweep mode** parameters.



For information on how to set the frequency values in your sweep, see “Setting Instantaneous Frequency Sweep Values” on page 2-202.

### Setting Instantaneous Frequency Sweep Values

Set the following parameters to tune the frequency values of your output frequency sweep.

- **Initial frequency** (Hz),  $f_0$
- **Target frequency** (Hz),  $f_i(t_g)$
- **Target time** (seconds),  $t_g$

The following table summarizes the sweep values at specific times for all **Frequency sweep** settings. For information on the formulas used to compute sweep values at other times, see “Algorithms” on page 2-202.

### Instantaneous Frequency Sweep Values

| Frequency Sweep     | Sweep Value at $t = 0$ | Sweep Value at $t = t_g$ | Time When Sweep Value Is Target Frequency, $f_i(t_g)$ |
|---------------------|------------------------|--------------------------|---|
| <b>Linear</b>       | $f_0$                  | $f_i(t_g)$               | $t_g$   |
| <b>Quadratic</b>    | $f_0$                  | $f_i(t_g)$               | $t_g$   |
| <b>Logarithmic</b>  | $f_0$                  | $f_i(t_g)$               | $t_g$   |
| <b>Swept cosine</b> | $f_0$                  | $2f_i(t_g) - f_0$        | $t_g/2$   |

### Algorithms

The Chirp block uses one of two formulas to compute the block output, depending on the **Frequency Sweep** parameter setting. For details, see the following sections.

## Equations for Output Computation

The following table shows the equations used by the block to compute the user-specified output frequency sweep,  $f_i(t)$ , the block output,  $y_{chirp}(t)$ , and the actual output frequency sweep,  $f_{i(actual)}(t)$ . The only time the user-specified sweep is not the actual output sweep is when the **Frequency sweep** parameter is set to Swept cosine.

**Note** The following equations apply only to unidirectional sweeps in which  $f_i(0) < f_i(t_g)$ . To derive equations for other cases, examine the following table and the diagram in “Shaping the Frequency Sweep” on page 2-201.

The table of equations used by the block contains the following variables:

- $f_i(t)$  — the user-specified frequency sweep
- $f_{i(actual)}(t)$  — the actual output frequency sweep, usually equal to  $f_i(t)$
- $y(t)$  — the Chirp block output
- $\psi(t)$  — the phase of the chirp signal, where  $\psi(0) = 0$ , and  $2\pi f_i(t)$  is the derivative of the phase

$$f_i(t) = \frac{1}{2\pi} \cdot \frac{d\psi(t)}{dt}$$

- $\phi_0$  — the **Initial phase** parameter value, where  $y_{chirp}(0) = \cos(\phi_0)$

## Equations for Unidirectional Positive Sweeps

| Frequency Sweep | Block Output Chirp Signal            | User-Specified Frequency Sweep, $f_i(t)$   | $\beta$                                | Actual Frequency Sweep, $f_{i(actual)}(t)$ |
|-----------------|--------------------------------------|--|--|--|
| Linear          | $y(t) = \cos(\psi(t) + \phi_0)$      | $f_i(t) = f_0 + \beta t$   | $\beta = \frac{f_i(t_g) - f_0}{t_g}$   | $f_{i(actual)}(t) = f_i(t)$                |
| Quadratic       | Same as Linear                       | $f_i(t) = f_0 + \beta t^2$   | $\beta = \frac{f_i(t_g) - f_0}{t_g^2}$ | $f_{i(actual)}(t) = f_i(t)$                |
| Logarithmic     | Same as Linear                       | $F_i(t) = f_0 \left( \frac{f_i(t_g)}{f_0} \right)^{\frac{t}{t_g}}$<br>Where $f_i(t_g) > f_0 > 0$ | N/A                                    | $f_{i(actual)}(t) = f_i(t)$                |
| Swept cosine    | $y(t) = \cos(2\pi f_i(t)t + \phi_0)$ | Same as Linear   | Same as Linear                         | $f_{i(actual)}(t) = f_i(t) + \beta t$      |

## Output Computation Method for Linear, Quadratic, and Logarithmic Frequency Sweeps

The derivative of the phase of a chirp function gives the instantaneous frequency of the chirp function. The Chirp block uses this principle to calculate the chirp output when the **Frequency Sweep** parameter is set to Linear, Quadratic, or Logarithmic.

$$y_{chirp}(t) = \cos(\psi(t) + \phi_0)$$

Linear, quadratic, or logarithmic chirp signal with phase  $\psi(t)$

$$f_i(t) = \frac{1}{2\pi} \cdot \frac{d\psi(t)}{dt}$$

Phase derivative is instantaneous frequency

For instance, if you want a chirp signal with a linear instantaneous frequency sweep, set the **Frequency Sweep** parameter to **Linear**, and tune the linear sweep values by setting other parameters appropriately. The block outputs a chirp signal, the phase derivative of which is the specified linear sweep. This ensures that the instantaneous frequency of the output is the linear sweep you desired. For equations describing the linear, quadratic, and logarithmic sweeps, see “Equations for Output Computation” on page 2-203.

### Output Computation Method for Swept Cosine Frequency Sweep

To generate the swept cosine chirp signal, the block sets the swept cosine chirp output as follows.

$$y_{chirp}(t) = \cos(\psi(t) + \phi_0) = \cos(2\pi f_i(t)t + \phi_0)$$

Swept cosine chirp output  
(Instantaneous frequency equation, does not hold.)

The instantaneous frequency equation, shown in “Output Computation Method for Linear, Quadratic, and Logarithmic Frequency Sweeps” on page 2-203, does not hold for the swept cosine chirp, so the user-defined frequency sweep,  $f_i(t)$ , is not the actual output frequency sweep,  $f_{i(actual)}(t)$ , of the swept cosine chirp. Thus, the swept cosine output might not behave as you expect. To learn more about swept cosine chirp behavior, see “Frequency sweep” on page 2-0 described for the **Frequency sweep** parameter and “Equations for Output Computation” on page 2-203.

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

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

## See Also

### Blocks

Signal From Workspace | Signal Generator | Sine Wave

### Functions

chirp | spectrogram

### Objects

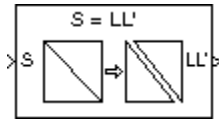
dsp.Chirp

### Topics

“Sample- and Frame-Based Concepts”

# Cholesky Factorization

Factor square Hermitian positive definite matrix into triangular components



## Library

Math Functions / Matrices and Linear Algebra / Matrix Factorizations

dspfacto

## Description

The Cholesky Factorization block uniquely factors the square Hermitian positive definite input matrix  $S$  as

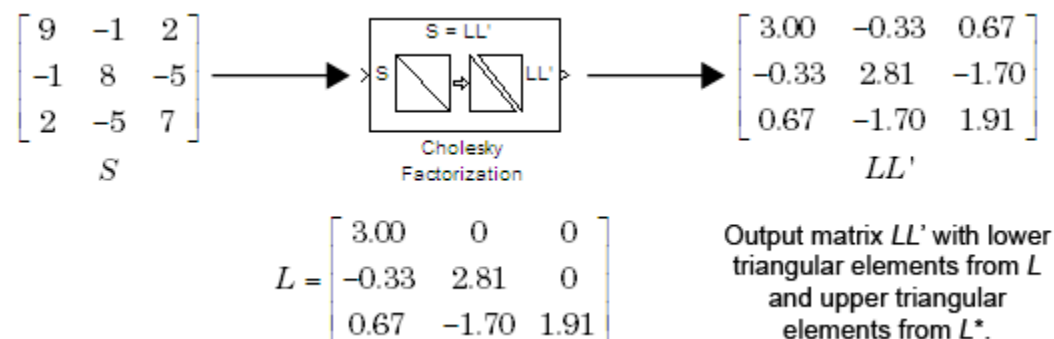
$$S = LL^*$$

where  $L$  is a lower triangular square matrix with positive diagonal elements and  $L^*$  is the Hermitian (complex conjugate) transpose of  $L$ . The block outputs a matrix with lower triangle elements from  $L$  and upper triangle elements from  $L^*$ . The output is not in the same form as the output of the MATLAB `chol` function. In order to convert the output of the Cholesky Factorization block to the MATLAB form, use the following equation:

$$R = \text{triu}(LL');$$

In order to extract the  $L$  matrix exclusively, pass the output of the Cholesky Factorization block,  $LL'$ , to the Extract Triangular Matrix block. Setting the **Extract** parameter of the Extract Triangular Matrix to **Lower** extracts the  $L$  matrix. Setting the **Extract** parameter to **Upper** extracts the  $L'$  matrix.

Here,  $LL'$  is the output of the Cholesky Factorization block. Due to roundoff error, these equations do not produce a result that is exactly the same as the MATLAB result.



**Block Output Composed of  $L$  and  $L^*$**

### Input Requirements for Valid Output

The block output is valid only when its input has the following characteristics:

- Hermitian — The block does *not* check whether the input is Hermitian; it uses only the diagonal and upper triangle of the input to compute the output.
- Real-valued diagonal entries — The block disregards any imaginary component of the input's diagonal entries.
- Positive definite — Set the block to notify you when the input is not positive definite as described in “Response to Nonpositive Definite Input” on page 2-206.

### Response to Nonpositive Definite Input

To generate a valid output, the block algorithm requires a positive definite input (see “Input Requirements for Valid Output” on page 2-206). Set the **Non-positive definite input** parameter to determine how the block responds to a nonpositive definite input:

- Ignore — Proceed with the computation and *do not* issue an alert. The output is *not* a valid factorization. A partial factorization will be present in the upper left corner of the output.
- Warning — Display a warning message in the MATLAB Command Window, and continue the simulation. The output is *not* a valid factorization. A partial factorization will be present in the upper left corner of the output.
- Error — Display an error dialog and terminate the simulation.

---

**Note** The **Non-positive definite input** parameter is a diagnostic parameter. Like all diagnostic parameters on the Configuration Parameters dialog box, it is set to **Ignore** in the code generated for this block by Simulink Coder™ code generation software.

---

### Performance Comparisons with Other Blocks

Note that  $L$  and  $L^*$  share the same diagonal in the output matrix. Cholesky factorization requires half the computation of Gaussian elimination (LU decomposition), and is always stable.

## Parameters

### Non-positive definite input

Response to nonpositive definite matrix inputs: **Ignore**, **Warning**, or **Error**. See “Response to Nonpositive Definite Input” on page 2-206.

## References

Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

## Supported Data Types

| Port | Supported Data Types   |
|------|--|
| S    | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> </ul> |
| LL'  | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> </ul> |

## See Also

|                     |                    |
|---------------------|--------------------|
| Autocorrelation LPC | DSP System Toolbox |
| Cholesky Inverse    | DSP System Toolbox |
| Cholesky Solver     | DSP System Toolbox |
| LDL Factorization   | DSP System Toolbox |
| LU Factorization    | DSP System Toolbox |
| QR Factorization    | DSP System Toolbox |
| chol                | MATLAB             |

See “Matrix Factorizations” for related information.

## Version History

Introduced before R2006a

## Extended Capabilities

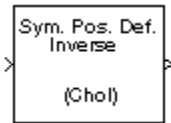
### C/C++ Code Generation

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

Generated code relies on the memcpy or memset function (string.h) under certain conditions.

## Cholesky Inverse

Compute inverse of Hermitian positive definite matrix using Cholesky factorization



### Library

Math Functions / Matrices and Linear Algebra / Matrix Inverses

dspinverses

### Description

The Cholesky Inverse block computes the inverse of the Hermitian positive definite input matrix  $S$  by performing Cholesky factorization.

$$S^{-1} = (LL^*)^{-1}$$

$L$  is a lower triangular square matrix with positive diagonal elements and  $L^*$  is the Hermitian (complex conjugate) transpose of  $L$ . Only the diagonal and upper triangle of the input matrix are used, and any imaginary component of the diagonal entries is disregarded. Cholesky factorization requires half the computation of Gaussian elimination (LU decomposition), and is always stable.

### Response to Nonpositive Definite Input

The algorithm requires that the input be Hermitian positive definite. When the input is not positive definite, the block reacts with the behavior specified by the **Non-positive definite input** parameter. The following options are available:

- **Ignore** — Proceed with the computation and *do not* issue an alert. The output is *not* a valid inverse.
- **Warning** — Display a warning message in the MATLAB Command Window, and continue the simulation. The output is *not* a valid inverse.
- **Error** — Display an error dialog box and terminate the simulation.

---

**Note** The **Non-positive definite input** parameter is a diagnostic parameter. Like all diagnostic parameters on the Configuration Parameters dialog box, it is set to **Ignore** in the code generated for this block by Simulink Coder code generation software.

---

### Parameters

#### Non-positive definite input

Response to nonpositive definite matrix inputs: Ignore, Warning, or Error. See “Response to Nonpositive Definite Input” on page 2-208.



## References

Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

## Supported Data Types

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

## See Also

|                        |                    |
|------------------------|--------------------|
| Cholesky Factorization | DSP System Toolbox |
| Cholesky Solver        | DSP System Toolbox |
| LDL Inverse            | DSP System Toolbox |
| LU Inverse             | DSP System Toolbox |
| Pseudoinverse          | DSP System Toolbox |
| inv                    | MATLAB             |

See “Matrix Inverses” for related information.

## Version History

Introduced before R2006a

## Extended Capabilities

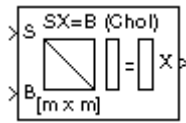
### C/C++ Code Generation

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

Generated code relies on the memcopy or memset function (string.h) under certain conditions.

## Cholesky Solver

Solve  $SX = B$  for  $X$  when  $S$  is a square Hermitian positive definite matrix



### Libraries:

DSP System Toolbox / Math Functions / Matrices and Linear Algebra / Linear System Solvers

### Description

The Cholesky Solver block solves the linear system  $SX = B$  by applying the Cholesky factorization to the input matrix, where:

- $S$  is an  $M$ -by- $M$  square matrix input through the **S** port. The matrix must be Hermitian positive definite.
- $B$  is an  $M$ -by- $N$  matrix input through the **B** port.
- $X$  is the  $M$ -by- $N$  output matrix and is the unique solution to the equations.

### Ports

#### Input

**S** — Input  $S$  matrix  
 $M$ -by- $M$  matrix

Specify the input  $M$ -by- $M$  square matrix  $S$  through this port. The matrix must be Hermitian positive definite. The block uses only the diagonal and upper triangle of the matrix and disregards any imaginary component in the diagonal entries. Inputs  $S$  and  $B$  must have the same number of rows.

Data Types: `single` | `double`

**B** —  $B$  matrix  
 $M$ -by- $N$  matrix |  $M$ -by-1 vector

Specify the  $B$  matrix in the  $SX = B$  equation as an  $M$ -by- $N$  matrix or an  $M$ -by-1 vector.

When you specify an  $M$ -by-1 vector, the block treats the length- $M$  unoriented vector input at port **B** as an  $M$ -by-1 matrix. Inputs  $S$  and  $B$  must have the same number of rows.

Data Types: `single` | `double`

#### Output

**X** — Solution to system of equations  
 $M$ -by- $N$  matrix |  $M$ -by-1 matrix

The block returns the output as an  $M$ -by- $N$  matrix or an  $M$ -by-1 matrix. The size of the matrix  $X$  is same as the size of the matrix  $B$ .

Data Types: `single` | `double`

## Parameters

**Non-positive definite input** — Response to nonpositive definite matrix input

Ignore (default) | Warning | Error

The input matrix  $S$  must be Hermitian positive definite.

When the input is not positive definite, the block reacts as per the behavior you specify in the **Non-positive definite input** parameter.

- **Ignore** — Proceed with the computation and do not issue an alert. The output is not a valid solution.
- **Warning** — Proceed with the computation and display a warning message in the MATLAB command window. The output is not a valid solution.
- **Error** — Display an error dialog box and terminate the simulation.

---

**Note** The **Non-positive definite input** parameter is a diagnostic parameter. Like all diagnostic parameters in the Configuration Parameters dialog box, it is set to **Ignore** in the code generated for this block by Simulink Coder code generation software.

---

## Block Characteristics

|                                 |                 |
|---------------------------------|-----------------|
| <b>Data Types</b>               | double   single |
| <b>Direct Feedthrough</b>       | no              |
| <b>Multidimensional Signals</b> | no              |
| <b>Variable-Size Signals</b>    | no              |
| <b>Zero-Crossing Detection</b>  | no              |

## Algorithms

Cholesky factorization uniquely factors the Hermitian positive definite input matrix  $S$  as

$$S = LL^*$$

where  $L$  is a lower triangular square matrix with positive diagonal elements.

The equation  $SX = B$  then becomes

$$LL^*X = B,$$

which is solved for  $X$  by substituting  $Y = L^*X$  and solving the following two triangular systems by forward and backward substitution, respectively.

$$LY = B$$

$$L^*X = Y$$

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

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

Generated code relies on the memcpy or memset function (string.h) under certain conditions.

## See Also

### Blocks

Autocorrelation LPC | Cholesky Factorization | Cholesky Inverse | LDL Solver | LU Solver | QR Solver

### Functions

chol

# CIC Compensator (Obsolete)

(Removed) Design CIC compensator



## Compatibility

---

**Note** The CIC Compensator block has been removed from the DSP System Toolbox block library. Existing instances of the CIC Compensator block will continue to operate. For new models, use the CIC Compensation Decimator and CIC Compensation Interpolator blocks. These blocks replace the functionality of the CIC Compensator block, when **FilterType** is set to **Decimator** and **Interpolator**, respectively.

---

## Library

Filtering / Filter Designs

dspfdesign

## Description

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

## Dialog Box

See “CIC Compensator Design — Main Pane” on page 5-608 for more information about the parameters of this block. The **Data Types** and **Code Generation** panes are not available for blocks in the DSP System Toolbox Filter Designs library.

Function Block Parameters: CIC Compensator

CIC Compensator

Design a CIC compensating filter.

View Filter Response

Filter specifications

Order mode: Minimum Order:

Filter Type: Single-rate

Number of CIC sections: 2 Differential delay: 1

Frequency specifications

Frequency units: Normalized (0 to 1) Input Fs: 2

Fpass: .45 Fstop: .55

Magnitude specifications

Magnitude units: dB

Apass: 1 Astop: 60

Algorithm

Design method: Equiripple

Design options

Filter Implementation

Structure: Direct-form FIR

Use basic elements to enable filter customization

Input processing: Columns as channels (frame based)

Use symbolic names for coefficients

OK Cancel Help Apply

### View filter response

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

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

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

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

In its normal mode of operation, the CIC Compensator block allows the adder's numbers to wrap around. The Fixed-Point infrastructure then causes warnings to appear on the command line.

## Filter Specifications

In this group, you specify your filter format, such as the filter order mode and the filter type.

### Filter order mode

Select either `Minimum` (the default) or `Specify` from the drop-down list. Selecting `Specify` enables the **Order** option (see the following sections) so you can enter the filter order.

### Filter type

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

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

### Order

Enter the filter order. This option is enabled only if `Specify` was selected for **Filter order mode**.

### Decimation Factor

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

### Interpolation Factor

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

### Number of CIC sections

Specify the number of sections in the CIC filter for which you are designing this compensator. Select the number of sections from the drop-down list or enter the number.

### Differential Delay

Specify the differential delay of your target CIC filter. The default value is 1. Most CIC filters use 1 or 2.

## Frequency Specifications

The parameters in this group allow you to specify your filter response curve.

### Frequency Specifications

#### Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0–1)` to enter frequencies in normalized form. This behavior is

the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—Hz, kHz, MHz, or GHz. Selecting one of the unit options enables the **Input Fs** parameter.

### **Input Fs**

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

### **Output Fs**

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter output. When you provide an output sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available only when you design interpolators.

### **Fpass**

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

### **Fstop**

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

## **Magnitude Specifications**

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

### **Magnitude units**

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

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

### **Apass**

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

### **Algorithm**

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

### **Design Method**

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default method is **Equi ripple**.

### **Design Options**

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



**Density factor**

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

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

**Phase constraint**

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

**Minimum order**

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

**Match Exactly**

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

**Stopband Shape**

Stopband shape lets you specify how the stopband changes with increasing frequency. Choose one of the following options:

- **Flat** — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.
- **Linear** — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.
- **1/f** — Specifies that the stopband attenuation changes exponentially as the frequency increases, where  $f$  is the frequency. Set the power (exponent) for the decay in **Stopband decay**.

**Stopband Decay**

When you set Stopband shape, Stopband decay specifies the amount of decay applied to the stopband. the following conditions apply to Stopband decay based on the value of Stopband Shape:

- When you set **Stopband shape** to **Flat**, **Stopband decay** has no affect on the stopband.
- When you set **Stopband shape** to **Linear**, enter the slope of the stopband in units of dB/rad/s. The block applies that slope to the stopband.
- When you set **Stopband shape** to **1/f**, enter a value for the exponent  $n$  in the relation  $(1/f)^n$  to define the stopband decay. The block applies the  $(1/f)^n$  relation to the stopband to result in an exponentially decreasing stopband attenuation.

**Filter Implementation****Structure**

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, the filter uses direct-form structure.

### Use basic elements to enable filter customization

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

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

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

### Input processing

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

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

---

**Note** The Inherited (this choice will be removed – see release notes) option will be removed in a future release. See “Frame-Based Processing” in the *DSP System Toolbox Release Notes* for more information.

---

### Rate options

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

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

### Use symbolic names for coefficients

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

## Supported Data Types

| Port   | Supported Data Types  |
|--------|---|
| Input  | <ul style="list-style-type: none"><li>• Double-precision floating point</li><li>• Single-precision floating point</li></ul> |
| Output | <ul style="list-style-type: none"><li>• Double-precision floating point</li><li>• Single-precision floating point</li></ul> |

## Version History

Introduced in R2006b

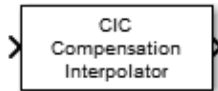
## See Also

### Blocks

CIC Compensation Decimator | CIC Compensation Interpolator

# CIC Compensation Interpolator

Compensate for CIC filter using FIR interpolator



## Libraries:

DSP System Toolbox / Filtering / Multirate Filters

## Description

The CIC Compensation Interpolator block uses an FIR polyphase interpolator as the compensation filter. CIC compensation interpolators are multirate FIR filters that you cascade with CIC interpolators to mitigate the drawbacks of the CIC filters.

CIC interpolation filters are used in areas that require high interpolation. These filters are popular in ASICs and FPGAs, since they do not have any multipliers. CIC filters have two drawbacks:

- CIC filters have a magnitude response that causes a droop in the passband region. This magnitude response is:

$$\text{abs}\left(\frac{\sin\left(M\frac{\omega}{2}\right)}{\sin\left(\frac{\omega}{2}\right)}\right)^n$$

- $M$  — Differential delay
- $n$  — Number of stages
- $\omega$  — Normalized angular frequency
- CIC filters have a wide transition region.

The compensation interpolator filters have an inverse sinc passband response to correct for the CIC droop, and they have a narrow transition width.

This block brings the capabilities of the `dsp.CICCompensationInterpolator` System object to the Simulink environment.

## Ports

### Input

**Input 1** — Data input

vector | matrix

Specify the data input as a vector or a matrix.

The block treats a  $P$ -by- $Q$  input matrix as  $Q$  independent channels, interpolating each channel over the first dimension.

Data Types: single | double | int8 | int16 | int32 | int64 | fixed point

## Output

**Output 1** — Upsampled and filtered signal  
vector | matrix

Upsampled and filtered signal, returned as a vector or matrix. For a  $P$ -by- $Q$  input matrix, the output is a  $LP$ -by- $Q$  matrix, where  $L$  is the interpolation factor.

When the output is fixed-point, it is signed only.

Data Types: single | double | int8 | int16 | int32 | int64 | fixed point

## Parameters

### Main Tab

**Rate change factor** — Rate-change factor of the CIC filter

2 (default) | positive integer

Specify the rate-change factor of the CIC filter as a positive integer. The block compensates the CIC filter using an FIR polyphase interpolator as the compensation filter.

**Number of sections** — Number of sections in the CIC filter being compensated

2 (default) | positive integer

Specify the number of sections in the CIC filter as a positive integer. The block compensates the CIC filter using an FIR polyphase interpolator as the compensation filter.

**Differential delay** — Differential delay of the CIC filter

1 (default) | positive integer

Specify the differential delay value used in each comb section of the CIC filter as a positive integer. The block compensates the CIC filter using an FIR polyphase interpolator as the compensation filter.

**Interpolation factor** — Interpolation factor of compensation filter

2 (default) | positive integer

Specify the interpolation factor  $L$  of the compensation filter as a positive integer.

**Minimum order filter design** — Design compensation filter of minimum order or of specified order

on (default) | off

When you select this check box, the block designs filter with the minimum order that meets the passband frequency, stopband frequency, passband ripple, and stopband attenuation specifications. When you clear this check box, the block designs filter with the order that you specify in the **Filter order** parameter.

**Filter order** — Order of compensation filter

12 (default) | positive integer

Specify the order of the compensator filter as a positive integer.

**Dependencies**

To enable this parameter, clear the **Minimum order filter design** parameter.

**Passband edge frequency (Hz)** — Passband edge frequency of compensation filter

1000000 (default) | positive real scalar

Specify the passband edge frequency of the compensation filter as a positive real scalar in Hz.

**Passband edge frequency (Hz)** must be less than  $F_s/2$ , where  $F_s$  is the output sample rate.

**Stopband edge frequency (Hz)** — Stopband edge frequency of compensation filter

4000000 (default) | positive scalar

Specify the stopband edge frequency of the compensation filter as a positive scalar in Hz. **Stopband edge frequency (Hz)** must be less than  $F_s/2$ , where  $F_s$  is the sample rate of the output signal.

**Passband ripple (dB)** — Passband ripple of compensation filter

0.1 (default) | positive scalar

Specify the filter passband ripple of the compensation filter as a positive scalar in dB.

**Stopband attenuation (dB)** — Stopband attenuation of compensation filter

60 (default) | positive scalar

Specify the stopband attenuation of the compensation filter as a positive scalar in dB.

**Inherit sample rate from input** — Inherit sample rate from input signal

off (default) | on

When you select this check box, the block inherits its sample rate from the input signal. When you clear this check box, you must specify the sample rate in **Input sample rate (Hz)**.

**Input sample rate (Hz)** — Sample rate of input signal

6000000 (default) | positive scalar

Specify the sample rate of input signal as a positive scalar in Hz.

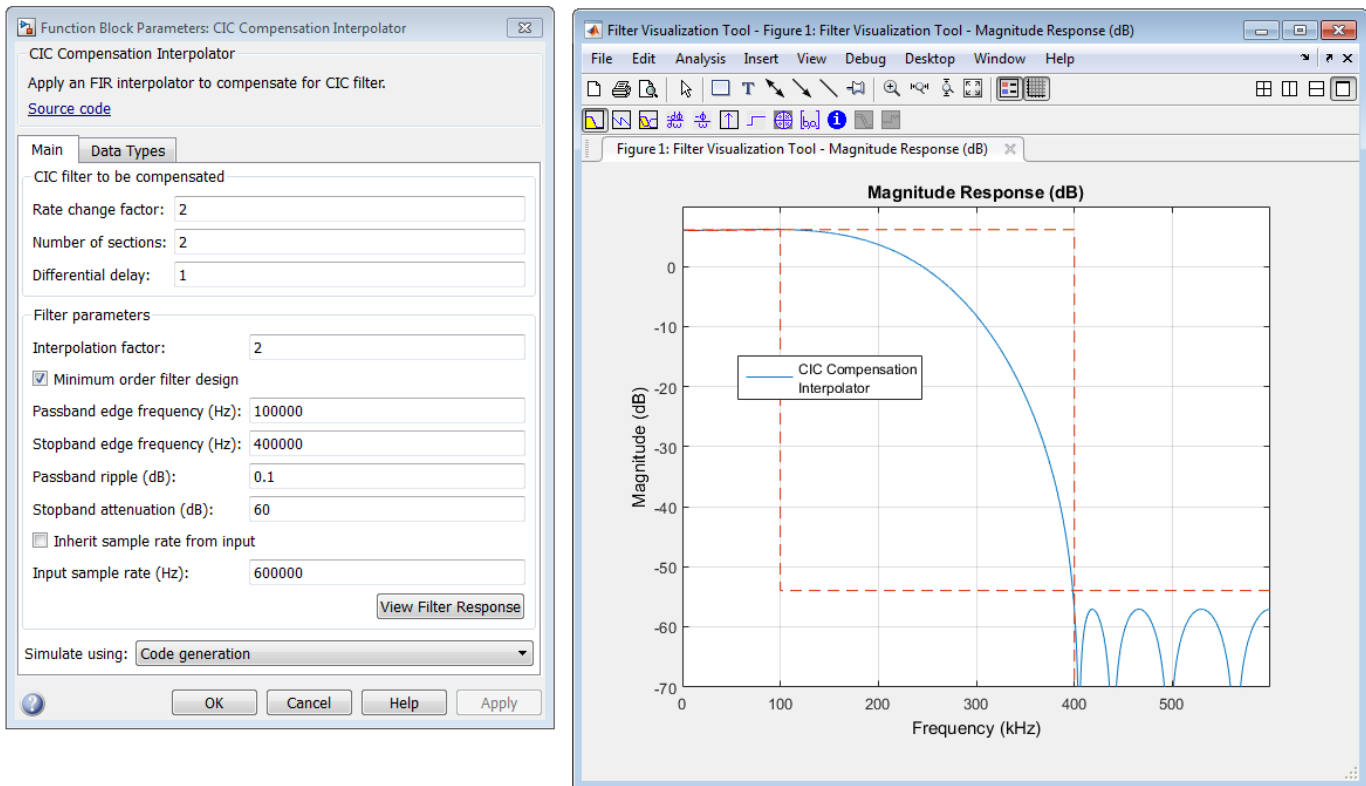
**Dependencies**

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

**View Filter Response** — View Filter Response

button

Click this button to open the Filter Visualization Tool (FVTool) and display the magnitude and phase response of the CIC Compensation Interpolator. The response is based on the values you specify in the block parameters dialog box. Changes made to these parameters update FVTool.



To update the magnitude response while FVTool is running, modify the dialog box parameters and click **Apply**.

### **Simulate using** — Simulate using

Code generation (default) | Interpreted execution

Type of simulation to run. You can set this parameter to:

- Code generation (default)

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

- Interpreted execution

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

### **Data Types Tab**

#### **Rounding mode** — Rounding mode for output fixed-point operations

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


Select the rounding mode for fixed-point operations. The default is Floor.

**Coefficients** — Word and fraction lengths of coefficients

`fixdt(1,16)` (default) | `fixdt(1,16,0)`

Specify the fixed-point data type of the coefficients as one of the following:

- `fixdt(1,16)` — Signed fixed-point data type of word length 16 with binary point scaling. The block determines the fraction length automatically from the coefficient values in such a way that the coefficients occupy maximum representable range without overflowing.
- `fixdt(1,16,0)` — Signed fixed-point data type of word length 16 with fraction length 0. You can change the fraction length to any other integer value.
- `<data type expression>` — Specify the coefficients data type by using an expression that evaluates to a data type object. For example, `numerictype(fixdt([ ],16,15))`. Specify the sign mode of this data type as `[ ]` or `true`.
- Refresh Data Type — Refresh to the default data type.

Click the **Show data type assistant** button  to display the data type assistant, which helps you set the coefficients data type.

**Block Characteristics**

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | double   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no                                      |
| <b>Multidimensional Signals</b> | no                                      |
| <b>Variable-Size Signals</b>    | no                                      |
| <b>Zero-Crossing Detection</b>  | no                                      |

**Algorithms**

The response of a CIC filter is given by:

$$H_{cic}(\omega) = \left[ \frac{\sin\left(\frac{RD\omega}{2}\right)}{\sin\left(\frac{\omega}{2}\right)} \right]^N$$

$R$ ,  $D$ , and  $N$  are the rate change factor, the differential delay, and the number of sections in the CIC filter, respectively.

After decimation, the CIC response has the form:

$$H_{cic}(\omega) = \left[ \frac{\sin\left(\frac{D\omega}{2}\right)}{\sin\left(\frac{\omega}{2R}\right)} \right]^N$$

The normalized version of this last response is the one that the CIC compensator needs to compensate. Hence, the passband response of the CIC compensator should take the following form:



$$H_{ciccomp}(\omega) = \left[ RD \frac{\sin\left(\frac{\omega}{2R}\right)}{\sin\left(\frac{D\omega}{2}\right)} \right]^N \text{ for } \omega \leq \omega_p < \pi$$

where  $\omega_p$  is the passband frequency of the CIC compensation filter.

Notice that when  $\omega/2R \ll \pi$ , the previous equation for  $H_{ciccomp}(\omega)$  can be simplified using the fact that  $\sin(x) \cong x$ :

$$H_{ciccomp}(\omega) \approx \left[ \frac{\left(\frac{D\omega}{2}\right)}{\sin\left(\frac{D\omega}{2}\right)} \right]^N = \left[ \text{sinc}\left(\frac{D\omega}{2}\right) \right]^{-N} \text{ for } \omega \leq \omega_p < \pi$$

This previous equation is the inverse sinc approximation to the true inverse passband response of the CIC filter.

## Version History

Introduced in R2015b

## Extended Capabilities

### C/C++ Code Generation

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

### Fixed-Point Conversion

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

## See Also

### Objects

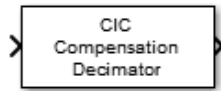
`dsp.CICCompensationInterpolator`

### Blocks

CIC Compensation Decimator

## CIC Compensation Decimator

Compensate for CIC filter using FIR decimator



### Libraries:

DSP System Toolbox / Filtering / Multirate Filters

### Description

The CIC Compensation Decimator block uses an FIR polyphase decimator as the compensation filter. CIC compensation decimators are multirate FIR filters that you cascade with CIC decimators to mitigate the drawbacks of the CIC filters.

CIC decimation filters are used in areas that require high decimation. These filters are popular in ASICs and FPGAs, since they do not have any multipliers. CIC filters have two drawbacks:

- CIC filters have a magnitude response that causes a droop in the passband region. This magnitude response is:

$$\text{abs}\left(\frac{\sin\left(M\frac{\omega}{2}\right)}{\sin\left(\frac{\omega}{2}\right)}\right)^n$$

- $M$  — Differential delay
- $n$  — Number of stages
- $\omega$  — Normalized angular frequency
- CIC filters have a wide transition region.

The compensation decimator filters have an inverse passband response to correct for the CIC droop, and they have a narrow transition width.

This block brings the capabilities of the `dsp.CICCompensationDecimator` System object to the Simulink environment.

### Ports

#### Input

**Input 1** — Data input

vector | matrix

Specify the data input as a vector or a matrix.

The block treats a  $P$ -by- $Q$  input matrix as  $Q$  independent channels, decimating each channel over the first dimension.

When you input a variable-size signal (frame length changes during simulation), the frame length of the signal  $P$  can be arbitrary, that is, the input frame length does not have to be a multiple of the decimation factor. When you input a fixed-size signal (frame length does not change during

simulation), the frame length can be arbitrary only when you select the **Allow arbitrary frame length for fixed-size input signals** parameter.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `fixed point`

## Output

**Output 1** — Filtered and downsampled signal  
vector | matrix

Filtered and downsampled signal, returned as a vector or matrix.

The output signal dimensions depend on the input signal dimensions, the decimation factor of the compensator, and whether you select the **Allow arbitrary frame length for fixed-size input signals** parameter.

This table provides more details on the dimensions of the output signal when you input a fixed-size input signal and a variable-size input signal.

| Input Signal Size | Input Signal Dimensions   | Output Signal Dimensions  |
|-------------------|---|---|
| Fixed             | $P$ -by-1 column vector, where $P$ is a multiple of the decimation factor $M$     | $(P/M)$ -by-1   |
| Fixed             | $P$ -by-1 column vector, where $P$ is not a multiple of the decimation factor $M$ | $\text{ceil}(P/M)$ -by-1 when you select <b>Allow arbitrary frame length for fixed-size input signals</b> .<br><br>If you do not select <b>Allow arbitrary frame length for fixed-size input signals</b> , the block errors.    |
| Fixed             | $P$ -by- $Q$ matrix, where $P$ is a multiple of the decimation factor $M$         | $(P/M)$ -by- $Q$  |
| Fixed             | $P$ -by- $Q$ matrix, where $P$ is not a multiple of the decimation factor $M$     | $\text{ceil}(P/M)$ -by- $Q$ when you select <b>Allow arbitrary frame length for fixed-size input signals</b> .<br><br>If you do not select <b>Allow arbitrary frame length for fixed-size input signals</b> , the block errors. |

| Input Signal Size | Input Signal Dimensions | Output Signal Dimensions   |
|-------------------|-------------------------|--|
| Variable          | $P$ -by- $Q$            | $\text{ceil}(P/M)$ -by- $Q$<br><br><b>Allow arbitrary frame length for fixed-size input signals</b> parameter appears on the block dialog box but does not have any impact on the input frame length. You can input a variable-size signal of any frame length even if you do not select the <b>Allow arbitrary frame length for fixed-size input signals</b> parameter. |

When the output is fixed-point, it is signed only.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `fixed point`

## Parameters

### Main Tab

**Rate change factor** — Rate-change factor of the CIC filter

2 (default) | positive integer

Specify the rate-change factor of the CIC filter as a positive integer. The block compensates the CIC filter using an FIR polyphase decimator as the compensation filter.

**Number of sections** — Number of sections in the CIC filter

2 (default) | positive integer

Specify the number of decimator and comb sections in the CIC filter as a positive integer. The block compensates the CIC filter using an FIR polyphase decimator as the compensation filter.

**Differential delay** — Differential delay of the CIC filter

1 (default) | positive integer

Specify the delay value used in each comb section of the CIC filter as a positive integer. The block compensates the CIC filter using an FIR polyphase decimator as the compensation filter.

**Decimation factor** — Decimation factor of compensation filter

2 (default) | positive integer scalar

Specify the decimation factor  $M$  of the compensation filter as a positive integer. The number of input rows must be a multiple of the decimation factor.

**Minimum order filter design** — Design compensation filter of minimum order or of specified order

on (default) | off

When you select this check box, the block designs filter with the minimum order that meets the passband frequency, stopband frequency, passband ripple, and stopband attenuation specifications. When you clear this check box, the block designs filter with the order that you specify in the **Filter order** parameter.

By default, this check box is selected.

**Filter order** — Order of compensation filter

12 (default) | positive integer

Specify the order of the compensation filter as a positive integer.

#### Dependency

To enable this parameter, clear the **Minimum order filter design** parameter.

**Passband edge frequency (Hz)** — Passband edge frequency of compensation filter

100000 (default) | positive real scalar

Specify the passband edge frequency of the compensation filter as a positive real scalar in Hz. The value you specify in **PassbandFrequency** must be less than  $F_s/2$ , where  $F_s$  is the input sample rate.

**Stopband edge frequency (Hz)** — Stopband edge frequency of compensation filter

400000 (default) | positive real scalar

Specify the stopband edge frequency of the compensation filter as a positive real scalar in Hz. The value you specify in **StopbandFrequency** must be less than  $F_s/2$ , where  $F_s$  is the sample rate of the input signal.

#### Dependency

To enable this parameter, select the **Minimum order filter design** parameter.

**Passband ripple (dB)** — Passband ripple of compensation filter

0.1 (default) | positive real scalar

Specify the passband ripple of the compensation filter as a positive real scalar in dB.

**Stopband attenuation (dB)** — Stopband attenuation of compensation filter

60 (default) | positive real scalar

Specify the stopband attenuation of the compensation filter as a positive real scalar in dB.

**Inherit sample rate from input** — Inherit sample rate from input signal

off (default) | on

When you select this check box, the block inherits its sample rate from the input signal. When you clear this check box, you must specify the sample rate in **Input sample rate (Hz)**.

**Input sample rate (Hz)** — Sample rate of input signal

1200000 (default) | positive real scalar

Specify the sample rate of the input signal as a positive real scalar in Hz.

**Allow arbitrary frame length for fixed-size input signals** — Allow arbitrary frame length for fixed-size input signals

off (default) | on

Specify whether fixed-size input signals (whose size does not change during simulation) can have an arbitrary frame length, where the frame length does not have to be a multiple of the decimation factor. The block uses this parameter setting only for fixed-size input signals and ignores this parameter if the input has a variable-size.

When the input signal is a variable-size signal, the signal can have arbitrary frame length, that is, the frame length does not have to be a multiple of the decimation factor.

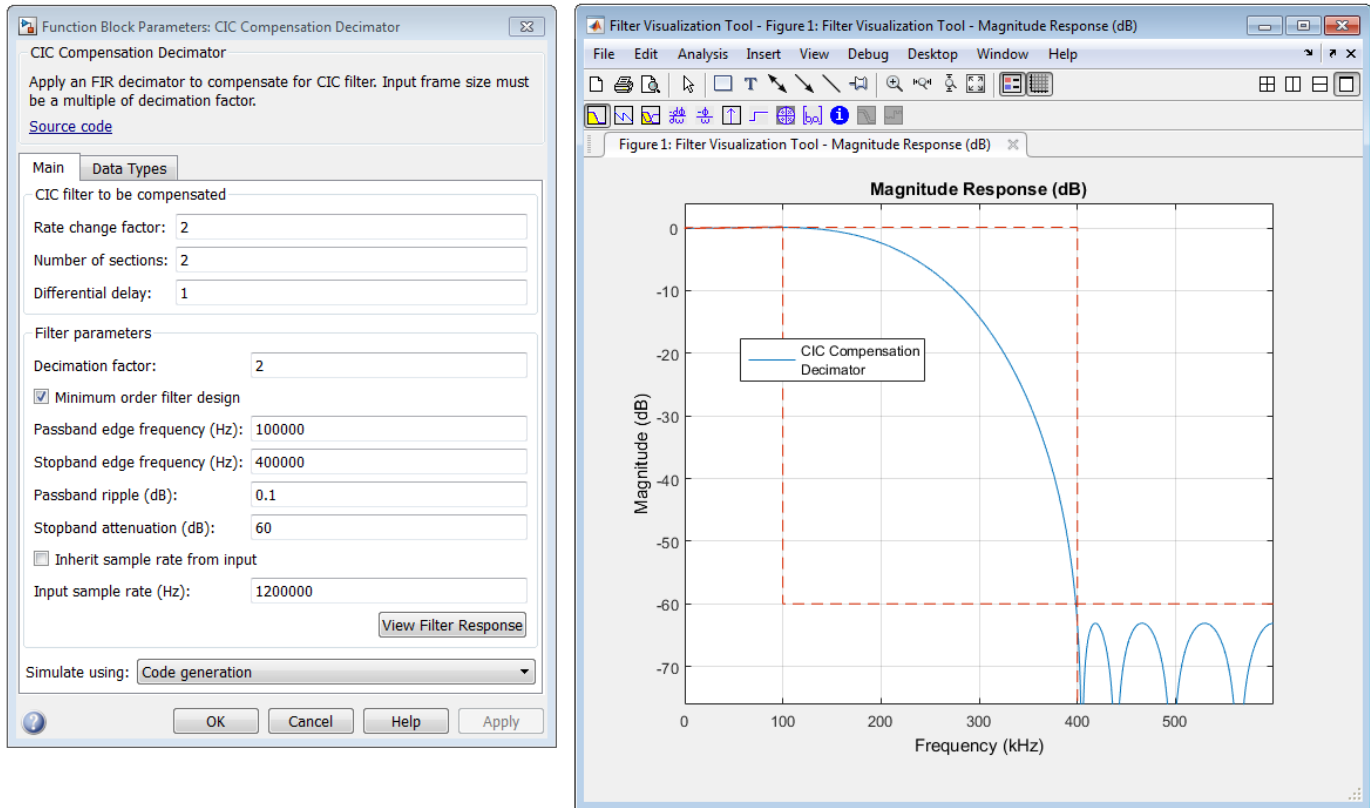
For fixed-size input signals, if you:

- Select the **Allow arbitrary frame length for fixed-size input signals** parameter, the frame length of the signal does not have to be a multiple of the decimation factor. If the input is not a multiple of the decimation factor, then the output is generally a variable-size signal. Therefore, to support arbitrary input size, the block must also support variable-size operations, which you can enable by selecting the **Allow arbitrary frame length for fixed-size input signals** parameter.
- Clear the **Allow arbitrary frame length for fixed-size input signals** parameter, the input frame length must be a multiple of the decimation factor.

**View Filter Response** — View Filter Response

button (default)

Click this button to open the Filter Visualization Tool FVTool and display the magnitude and phase response of the CIC Compensation Decimator. The response is based on the values you specify in the block parameters dialog box. Changes made to these parameters update FVTool.



**Simulate using** — Simulate using

Code generation (default) | Interpreted execution

Specify the type of simulation to run. You can set this parameter to:

- Code generation (default)

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

- Interpreted execution

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

### Data Types Tab

**Rounding mode** — Rounding mode for output fixed-point operations

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


Select the rounding mode for fixed-point operations. The default is Floor.

**Coefficients** — Word and fraction lengths of coefficients

`fixdt(1,16)` (default) | `fixdt(1,16,0)`

Specify the fixed-point data type of the coefficients as one of the following:

- `fixdt(1,16)` (default) — Signed fixed-point data type of word length 16 with binary point scaling. The block determines the fraction length automatically from the coefficient values in such a way that the coefficients occupy maximum representable range without overflowing.
- `fixdt(1,16,0)` — Signed fixed-point data type of word length 16 and fraction length 0. You can change the fraction length to any other integer value.
- `<data type expression>` — Specify the coefficients data type by using an expression that evaluates to a data type object. For example, `numerictype(fixdt([ ],16,15))`. Specify the sign mode of this data type as `[ ]` or `true`.
- Refresh Data Type — Refresh to the default data type.

Click the **Show data type assistant** button  to display the data type assistant, which helps you set the coefficients data type.

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

## Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | double   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no                                      |
| <b>Multidimensional Signals</b> | no                                      |
| <b>Variable-Size Signals</b>    | yes                                     |
| <b>Zero-Crossing Detection</b>  | no                                      |

## Algorithms

The response of a CIC filter is given by:

$$H_{cic}(\omega) = \left[ \frac{\sin\left(\frac{RD\omega}{2}\right)}{\sin\left(\frac{\omega}{2}\right)} \right]^N$$

$R$ ,  $D$ , and  $N$  are the rate change factor, the differential delay, and the number of sections in the CIC filter, respectively.

After decimation, the CIC response has the form:

$$H_{cic}(\omega) = \left[ \frac{\sin\left(\frac{D\omega}{2}\right)}{\sin\left(\frac{\omega}{2R}\right)} \right]^N$$

The normalized version of this last response is the one that the CIC compensator needs to compensate. Hence, the passband response of the CIC compensator should take the following form:



$$H_{ciccomp}(\omega) = \left[ RD \frac{\sin\left(\frac{\omega}{2R}\right)}{\sin\left(\frac{D\omega}{2}\right)} \right]^N \text{ for } \omega \leq \omega_p < \pi$$

where  $\omega_p$  is the passband frequency of the CIC compensation filter.

Notice that when  $\omega/2R \ll \pi$ , the previous equation for  $H_{ciccomp}(\omega)$  can be simplified using the fact that  $\sin(x) \cong x$ :

$$H_{ciccomp}(\omega) \approx \left[ \frac{\left(\frac{D\omega}{2}\right)}{\sin\left(\frac{D\omega}{2}\right)} \right]^N = \left[ \text{sinc}\left(\frac{D\omega}{2}\right) \right]^{-N} \text{ for } \omega \leq \omega_p < \pi$$

This previous equation is the inverse sinc approximation to the true inverse passband response of the CIC filter.

## Version History

Introduced in R2015b

### R2023a: Support for arbitrary input frame length

This block supports input signals with arbitrary frame lengths when the:

- Input signal is a fixed-size signal (frame length does not change during simulation) and you select the **Allow arbitrary frame length for fixed-size input signals** parameter
- Input signal is a variable-size signal (frame length changes during simulation)

When this block supports an input signal with an arbitrary frame length, the input frame length does not have to be a multiple of the decimation factor.

## Extended Capabilities

### C/C++ Code Generation

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

### Fixed-Point Conversion

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

## See Also

### Objects

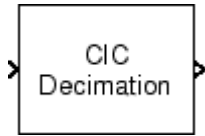
dsp.CICCompensationDecimator

### Blocks

CIC Compensation Interpolator

## CIC Decimation

Decimate signal using cascaded integrator-comb filter



### Libraries:

DSP System Toolbox / Filtering / Multirate Filters  
 DSP System Toolbox HDL Support / Filtering

## Description

The CIC Decimation block performs a sample rate decrease (decimation) on an input signal by an integer factor. Cascaded Integrator-Comb (CIC) filters are a class of linear phase FIR filters comprised of a comb part and an integrator part.

The block supports real and complex fixed-point inputs. In its normal mode of operation, the CIC Decimation block allows the adder's numeric values to overflow and wrap around [1] [3]. The Fixed-Point infrastructure then causes overflow warnings to appear on the command line. This overflow is of no consequence.

The CIC Decimation block requires a Fixed-Point Designer™ license.

## Ports

### Input

**Port\_1** — Input signal  
 vector | matrix

Data input, specified as a vector or matrix. The number of input rows must be a multiple of the decimation factor.

If the input is fixed-point, it must be signed integer or signed fixed point with power-of-two slope and zero bias.

Data Types: int8 | int16 | int32 | int64 | fixed point

### Output

**Port\_1** — CIC decimated output  
 vector | matrix

CIC decimated output, returned as a vector or a matrix. The data type of the output is determined by the settings in the block dialog. The complexity of the output matches that of the input. The number of output rows is  $(1/R) \times Num$ , where  $R$  is the decimation factor and  $Num$  is the number of input rows.

Data Types: int8 | int16 | int32 | int64 | fixed point

## Parameters

**Coefficient source** — Source of the filter information  
Dialog parameters (default) | Filter object

Source of the filter information, specified as one of the following:

- Dialog parameters — Enter information about the filter, such as **Decimation factor (R)**, **Differential delay (M)** and **Number of sections (N)**, in the block dialog.
- Filter object — Specify the filter using a `dsp.CICDecimator` System object.

Different items appear on the CIC Decimation block dialog depending on whether you select Dialog parameters or Filter object in the **Coefficient source** parameter.

**Decimation factor (R)** — Decimation factor  
2 (default) | integer

Decimation factor of the filter, specified as an integer greater than 1.

### Dependencies

This parameter appears when you set **Coefficient source** to Dialog parameters.

**Differential Delay (M)** — Differential delay  
1 (default) | positive integer

Specify the differential delay of the comb part of the filter,  $M$ , as a positive integer. For more details, see “CIC Decimation Filter” on page 4-253.

### Dependencies

This parameter appears when you set **Coefficient source** to Dialog parameters.

**Number of sections (N)** — Number of filter sections  
2 (default) | positive integer

Specify the number of filter sections. The number you specify determines the number of sections in either the comb part of the filter or the integrator part of the filter. This value does not represent the total number of sections in the comb and integrator parts combined.

### Dependencies

This parameter appears when you set **Coefficient source** to Dialog parameters.

**Data type specification mode** — Specify word length and fraction length of filter sections and output  
Full precision (default) | Minimum section word lengths | Specify word lengths | Binary point scaling

Choose how you specify the fixed-point word length and fraction length of the filter sections and/or output:

- **Full precision** — The word and fraction lengths of the filter sections and outputs are automatically selected for you. All word lengths (WL) are set to:

$$WL = \text{ceil}(N \times \log_2(M \times R)) + I$$

where,

- $I$  -- Input word length
- $M$  -- Differential delay
- $N$  -- Number of sections
- $R$  -- Decimation factor

All fraction lengths are set to the input fraction length.

- **Minimum section word lengths** — Specify the word length of the filter output in the **Output word length** parameter. The block automatically selects the word lengths of the filter sections and all fraction lengths such that each of the section word lengths is as small as possible. The precision of each filter section is less than in **Full precision** mode, but the range of each section is preserved.
- **Specify word lengths** — Specify the word lengths of the filter sections and output in the **Section word lengths** and **Output word length** parameters. The block automatically selects fraction lengths for the filter sections and output such that the range of each section is preserved when the least significant bits are discarded.
- **Binary point scaling** — Specify the word and fraction lengths of the filter sections and output in the **Section word lengths**, **Section fraction lengths**, **Output word length**, and **Output fraction length** parameters.

#### Dependencies

This parameter appears when you set **Coefficient source** to **Dialog** parameters.

**Section word lengths** — Word length of filter sections

[16 16 16 16] (default) | scalar | row vector

Word lengths of filter sections, specified as a scalar or a vector of length equal to  $2N$ , where  $N$  is the number of filter sections. The section word length must be in the range [2, 128].

#### Dependencies

This parameter appears when you set **Coefficient source** to **Dialog** parameters and **Data type specification mode** to either **Specify word lengths** or **Binary point scaling**.

**Section fraction lengths** — Fraction lengths of filter sections

0 (default) | integer

Fraction lengths of filter sections, specified as an integer.

#### Dependencies

This parameter appears when you set **Coefficient source** to **Dialog** parameters and **Data type specification mode** to **Binary point scaling**.

**Output word length** — Word length of filter output  
32 (default) | integer

Word length of the filter output, specified as an integer in the range [2, 128].

**Dependencies**

This parameter appears when you set **Coefficient source** to Dialog parameters and **Data type specification mode** to any option other than Full precision.

**Output fraction length** — Fraction length of filter output  
0 (default) | integer

Fraction length of the filter output, specified as an integer.

**Dependencies**

This parameter appears when you set **Coefficient source** to Dialog parameters and **Data type specification mode** to Binary point scaling.

**Rate options** — Rate processing rule  
Enforce single-rate processing (default) | Allow multirate processing

Specify the rate processing rule for the block:

- **Enforce single-rate processing** — The block performs frame-based processing and produces an output that has the same sample rate as the input. To decimate the signal while maintaining the input sample rate, the block decreases the output frame size. In this mode, the input column size must be a multiple of **Decimation Factor (R)**.
- **Allow multirate processing** — In this mode, the block produces an output with a sample rate that is R times slower than the input sample rate.

**Filter object** — Multirate filter object  
dsp.CICDecimator

Specify the name of the multirate filter object that you want the block to implement. You must specify the filter as a dsp.CICDecimator System object.

You can define the System object in the block dialog or in a MATLAB workspace variable.

For information on creating System objects, see “Define Basic System Objects”.

**Dependencies**

This parameter appears when you set **Coefficient source** to Filter object.

**View Filter Response** — View filter response  
gui button

This button opens the Filter Visualization Tool (**FVTool**) from the Signal Processing Toolbox product and displays the filter response of the filter defined in the block. For more information on FVTool, see the Signal Processing Toolbox documentation.

---

**Note** If you specify a filter in the **Filter object** parameter, you must apply the filter by clicking the **Apply** button before using the **View Filter Response** button.

---

## Block Characteristics

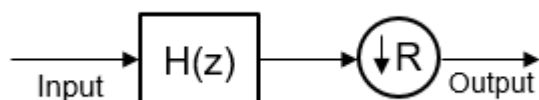
|                                 |                       |
|---------------------------------|-----------------------|
| <b>Data Types</b>               | fixed point   integer |
| <b>Direct Feedthrough</b>       | no                    |
| <b>Multidimensional Signals</b> | no                    |
| <b>Variable-Size Signals</b>    | no                    |
| <b>Zero-Crossing Detection</b>  | no                    |

## More About

### CIC Filter

CIC filters are an optimized class of linear phase FIR filters composed of a comb part and an integrator part.

The CIC decimation filter is conceptually given by a single rate CIC filter,  $H(z)$  which is a lowpass anti-imaging filter, followed by a downsampler. The CIC decimation filter decreases the sample rate of an input signal by an integer factor using a cascaded integrator-comb (CIC) filter.



In a more efficient implementation, the single rate CIC filter  $H(z)$  is factorized this way:

$$H(z) = \left[ \sum_{k=0}^{RM-1} z^{-k} \right]^N = \frac{(1 - z^{-RM})^N}{(1 - z^{-1})^N} = \frac{1}{(1 - z^{-1})^N} \cdot \frac{(1 - z^{-RM})^N}{1} = H_I^N(z) \cdot H_C^N(z)$$

where,

- $H_I$  is the transfer function of the integrator part of the filter containing  $N$  stages of integrators.
- $H_C$  is the transfer function of the  $N$  sections of the cascaded comb filters, each with a width of  $RM$ .
- $N$  is the number of sections. The number of sections in a CIC filter is defined as the number of sections in either the comb part or the integrator part of the filter. This value does not represent the total number of sections throughout the entire filter.

- $R$  is the decimation factor.
- $M$  is the differential delay.

In the overall multirate realization, the algorithm applies the noble identity for decimation and moves the rate change factor,  $R$ , to follow after the  $N$  sections of the cascaded integrators. The transfer function of the resulting filter is given by the following equation:

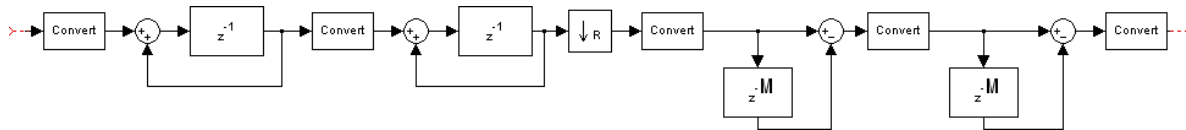
$$H(z) = \frac{(1 - z^{-M})^N}{(1 - z^{-1})^N}.$$

For a block diagram that shows the multirate implementation, see Algorithms on page 2-239.

## Algorithms

### CIC Decimation Filter

The CIC decimation filter in More About on page 2-238 is realized as a cascade of  $N$  sections of the integrators followed by a rate change factor of  $R$ , followed by  $N$  sections of comb filters.



This diagram shows two sections of cascaded integrators and two sections of cascaded comb filters. The unit delay in the integrator portion of the CIC filter can be located in either the feedforward or the feedback path. These two configurations yield identical filter frequency response. However, the numerical outputs from these two configurations are different due to the latency. This block puts the unit delay in the feedforward path of the integrator because it is a preferred configuration for HDL implementation.

## Version History

Introduced before R2006a

## References

- [1] Hogenauer, E.B. "An Economical Class of Digital Filters for Decimation and Interpolation." *IEEE Transactions on Acoustics, Speech and Signal Processing*. Vol. 29, Number 2, 1981, pp. 155-162.
- [2] Meyer-Baese, U. *Digital Signal Processing with Field Programmable Gate Arrays*. New York: Springer Verlag, 2001.
- [3] Harris, Fredric J., *Multirate Signal Processing for Communication Systems*. Upper Saddle River, NJ: Prentice Hall PTR, 2004.

## Extended Capabilities

### C/C++ Code Generation

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

Generated code relies on the memcopy or memset function (`string.h`) under certain conditions.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

---

**Note** For an HDL-optimized filter architecture with hardware-friendly control signals, use the CIC Decimator (DSP HDL Toolbox) block. The DSP HDL Toolbox block simulates the latency of the HDL algorithm in Simulink.

---

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Coder supports **Coefficient source** options **Dialog parameters** and **Filter object**.

### AddPipelineRegisters Support

When you use **AddPipelineRegisters**, registers are placed based on the filter structure. The pipeline register placement determines the latency.

| Pipeline Register Placement  | Latency (clock cycles)                                       |
|--|--|
| A pipeline register is added between the comb stages of the differentiators. | NS - 1, where NS is number of sections (at the output side). |

### HDL Filter Properties

|                             |  |
|-----------------------------|--|
| <b>AddPipelineRegisters</b> | Insert a pipeline register between stages of computation in a filter. See also AddPipelineRegisters (HDL Coder). |
|-----------------------------|--|

### HDL Block Properties

|                                  |  |
|----------------------------------|--|
| <b>ConstrainedOutputPipeline</b> | Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder). |
| <b>InputPipeline</b>             | Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).                 |
| <b>OutputPipeline</b>            | Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).               |

### Restrictions

- Vector and frame inputs are not supported for HDL code generation.
- When you select **Dialog parameters**, the **Filter Structure** option Zero-latency decimator is not supported for HDL code generation. From the **Filter Structure** drop-down list, select Decimator.



**Fixed-Point Conversion**

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

If the input is fixed point, it must be a signed integer or a signed fixed point value with a power-of-two slope and zero bias.

**See Also****Functions**

`filter`

**Objects**

`dsp.CICDecimator` | `dsp.CICInterpolator` | `dsp.FIRDecimator` | `dsp.FIRInterpolator` |  
`dsp.CICCompensationDecimator` | `dsp.CICCompensationInterpolator` |  
`dsp.FIRHalfbandDecimator` | `dsp.FIRHalfbandInterpolator`

**Blocks**

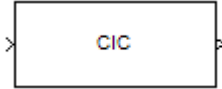
CIC Interpolation | FIR Decimation | FIR Interpolation | Digital Down-Converter | Digital Up-Converter

**Topics**

“Sigma-Delta A/D Conversion”

## CIC Filter

Design Cascaded Integrator-Comb (CIC) Filter



## Library

Filtering / Filter Designs

dspfdesign

## Description

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

## Dialog Box

### Main Pane

See “CIC Filter Design — Main Pane” on page 5-607 for more information about the parameters of this block. The **Data Types** and **Code Generation** panes are not available for blocks in the DSP System Toolbox Filter Designs library.

### View filter response

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

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

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

In its normal mode of operation, the CIC Filter block allows the adder’s numbers to wrap around. The Fixed-Point infrastructure then causes warnings to appear on the command line.

### Filter Specifications

In this group, you specify your CIC filter format, such as the filter type and the differential delay.

### Filter type

Select whether your filter will be a **decimator** or an **interpolator**. Your choice determines the type of filter and the design methods and structures that are available to implement your filter.

Selecting **decimator** or **interpolator** activates the **Factor** option. When you design an interpolator, you enable the **Output sample rate** parameter.

When you design either a decimator or interpolator, the resulting filter is a CIC filter that decimates or interpolates your input signal.

### Differential delay

Specify the differential delay of your CIC filter as an integer value greater than or equal to 1. The default value is 1. The differential delay changes the shape, number, and location of nulls in the filter response. Increasing the differential delay increases the sharpness of the nulls and the response between the nulls. In practice, differential delay values of 1 or 2 are the most common.

### Factor

Specify the decimation or interpolation factor for your filter as an integer value greater than or equal to 1. The default value is 2.

### Frequency units

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

### Input sample rate

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

### Output sample rate

F<sub>s</sub>, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter output. When you provide an output sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available only when you design interpolators.

### Passband frequency

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

### Magnitude units

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

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

### Stopband attenuation

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

## Filter Implementation

### Use basic elements to enable filter customization

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

The high-level implementation provides better compatibility across various filter structures, especially filters that would contain algebraic loops when constructed using basic elements.

### Rate options

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

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

## Data Types Pane

See the Data Types Pane subsection of the `filterBuilder` function reference page for more information about specifying data type parameters.

## Supported Data Types

| Port   | Supported Data Types  |
|--------|---|
| Input  | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul> |
| Output | <ul style="list-style-type: none"> <li>• Fixed point</li> </ul>   |

## Version History

Introduced in R2007b

## Extended Capabilities

### C/C++ Code Generation

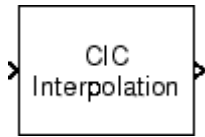
Generate C and C++ code using Simulink® Coder™.

### Fixed-Point Conversion

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

# CIC Interpolation

Interpolate signal using cascaded integrator-comb filter



## Libraries:

DSP System Toolbox / Filtering / Multirate Filters  
 DSP System Toolbox HDL Support / Filtering

## Description

The CIC Interpolation block performs a sample rate increase (interpolation) on an input signal by an integer factor. Cascaded integrator-comb (CIC) filters are a class of linear phase FIR filters that consist of a comb part and an integrator part.

The CIC Interpolation block requires a Fixed-Point Designer license.

## Ports

### Input

**Port 1** — Input signal  
 vector | matrix

Data input, specified as a vector or matrix. If the input is fixed point, it must be signed integer or signed fixed point with power-of-two slope and zero bias.

Data Types: int8 | int16 | int32 | int64 | fixed point

### Output

**Port 1** — CIC interpolated output  
 vector | matrix

CIC interpolated output, returned as a vector or a matrix. The data type of the output is determined by the settings in the block dialog. The complexity of the output matches that of the input. The number of output rows is  $R \times Num$ , where  $R$  is the interpolation factor and  $Num$  is the number of input rows.

Data Types: int8 | int16 | int32 | int64 | fixed point

## Parameters

**Coefficient source** — Source of filter information  
 Dialog parameters (default) | Filter object

Source of the filter information, specified as one of the following:

- Dialog parameters — Enter information about the filter, such as **Interpolation factor (R)**, **Differential delay (M)** and **Number of sections (N)**, in the block dialog.

- **Filter object** — Specify the filter using a `dsp.CICInterpolator System` object.

**Interpolation factor (R)** — Interpolation factor  
2 (default) | integer

Interpolation factor of the filter, specified as an integer greater than 1.

**Dependencies**

This parameter appears when you set **Coefficient source** to `Dialog` parameters.

**Differential Delay (M)** — Differential delay  
1 (default) | positive integer

Specify the differential delay of the comb part of the filter,  $M$ , as a positive integer. For more details, see “CIC Interpolation Filter” on page 4-268.

**Dependencies**

This parameter appears when you set **Coefficient source** to `Dialog` parameters.

**Number of sections (N)** — Number of filter sections  
2 (default) | positive integer

Specify the number of filter sections. The number you specify determines the number of sections in either the comb part of the filter or the integrator part of the filter. This value does not represent the total number of sections in the comb and integrator parts combined.

**Dependencies**

This parameter appears when you set **Coefficient source** to `Dialog` parameters.

**Data type specification mode** — Specify word length and fraction length of filter sections and output  
`Full precision (default) | Minimum section word lengths | Specify word lengths | Binary point scaling`

Choose how you specify the fixed-point word length and fraction length of the filter sections and/or output:

- **Full precision** — The word and fraction lengths of the filter sections and outputs are automatically selected for you. The output and last section word lengths (WL) are set to:

$$WL = \text{ceil}\left(\log_2\left(\frac{(RM)^N}{R}\right)\right) + I$$

where,

- $I$  -- Input word length
- $M$  -- Differential delay
- $N$  -- Number of sections

- $R$  -- Interpolation factor

The other section word lengths are set to accommodate the bit growth, as described in Hogenauer's paper [1]. All fraction lengths are set to the input fraction length.

- **Minimum section word lengths** — Specify the word length of the filter output in the **Output word length** parameter. The word lengths of the filter sections are set in the same way as in Full precision mode.

The section fraction lengths are set to the input fraction length. The output fraction length is set to the input fraction length minus the difference between the last section word length and the output word length.

- **Specify word lengths** — Specify the word lengths of the filter sections and output in the **Section word lengths** and **Output word length** parameters. The fraction lengths of the filter sections are set such that the spread between word length and fraction length is the same as in full-precision mode. The output fraction length is set to the input fraction length minus the difference between the last section word length and the output word length.
- **Binary point scaling** — Specify the word and fraction lengths of the filter sections and output in the **Section word lengths**, **Section fraction lengths**, **Output word length**, and **Output fraction length** parameters.

#### Dependencies

This parameter appears when you set **Coefficient source** to Dialog parameters.

**Section word lengths** — Word length of filter sections

[16 16 16 16] (default) | scalar | row vector

Word lengths of filter sections, specified as a scalar or a vector of length equal to  $2N$ , where  $N$  is the number of filter sections. The section word length must be in the range [2, 128].

#### Dependencies

This parameter appears when you set **Coefficient source** to Dialog parameters and **Data type specification mode** to either Specify word lengths or Binary point scaling.

**Section fraction lengths** — Fraction length of filter sections

0 (default) | integer

Fraction lengths of filter sections, specified as an integer.

#### Dependencies

This parameter appears when you set **Coefficient source** to Dialog parameters and **Data type specification mode** to Binary point scaling.

**Output word length** — Word length of filter output

32 (default) | integer

Word length of the filter output, specified as an integer in the range [2, 128].

**Dependencies**

This parameter appears when you set **Coefficient source** to Dialog parameters and **Data type specification mode** to any option other than Full precision.

**Output fraction length** — Fraction length of filter output  
0 (default) | integer

Fraction length of the filter output, specified as an integer.

**Dependencies**

This parameter appears when you set **Coefficient source** to Dialog parameters and **Data type specification mode** to Binary point scaling.

**Input processing** — Method of processing input  
Columns as channels (frame based) (default) | Elements as channels (sample based)

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

- **Columns as channels (frame based)** — The block treats each column of the input as a separate channel. In this mode, the block always performs single-rate processing.
- **Elements as channels (sample based)** — The block treats each element of the input as a separate channel. In this mode, the input to the block must be a scalar or a vector. You can use the **Rate options** parameter to specify whether the block performs single-rate or multirate processing.

**Rate options** — Rate processing rule  
Enforce single-rate processing (default) | Allow multirate processing

Specify the rate processing rule for the block. You can select one of the following options:

- **Enforce single-rate processing** — The block maintains the sample rate of the input.
- **Allow multirate processing** — The block produces an output with a sample rate that is  $R$  times faster than the input sample rate. To select this option, you must set the **Input processing** parameter to **Elements as channels (sample based)**.

**Filter object** — Multirate filter object  
dsp.CICInterpolator System object

Specify the name of the multirate filter object that you want the block to implement. You must specify the filter as a dsp.CICInterpolator System object.

You can define the System object in the block dialog or in a MATLAB workspace variable.

For information on creating System objects, see “Define Basic System Objects”.

**Dependencies**

This parameter appears when you set **Coefficient source** to Filter object.



**View Filter Response** — View filter response button

This button opens the Filter Visualization Tool (**FVTool**) from the Signal Processing Toolbox product and displays the filter response of the filter defined in the block. For more information on FVTool, see the Signal Processing Toolbox documentation.

**Note** If you specify a filter in the **Filter object** parameter, you must apply the filter by clicking the **Apply** button before using the **View Filter Response** button.

## Block Characteristics

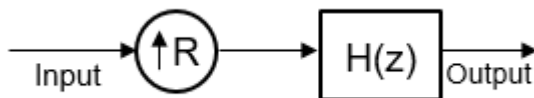
|                                 |                       |
|---------------------------------|-----------------------|
| <b>Data Types</b>               | fixed point   integer |
| <b>Direct Feedthrough</b>       | no                    |
| <b>Multidimensional Signals</b> | no                    |
| <b>Variable-Size Signals</b>    | no                    |
| <b>Zero-Crossing Detection</b>  | no                    |

## More About

### CIC Filter

CIC filters are an optimized class of linear phase FIR filters composed of a comb part and an integrator part.

The CIC interpolation filter is conceptually given by an upsampler followed by a single rate CIC filter,  $H(z)$ , which is a lowpass anti-imaging filter. The CIC interpolation filter increases the sample rate of an input signal by an integer factor using a cascaded integrator-comb (CIC) filter.



In a more efficient implementation, the single rate CIC filter  $H(z)$  is factorized this way:

$$H(z) = \left[ \sum_{k=0}^{RM-1} z^{-k} \right]^N = \frac{(1 - z^{-RM})^N}{(1 - z^{-1})^N} = \frac{(1 - z^{-RM})^N}{1} \cdot \frac{1}{(1 - z^{-1})^N} = H_C^N(z) \cdot H_I^N(z)$$

where,

- $H_C$  is the transfer function of the  $N$  sections of the cascaded comb filters, each with a width of  $RM$ .
- $H_I$  is the transfer function of the integrator part of the filter containing  $N$  stages of integrators.

- $N$  is the number of sections. The number of sections in a CIC filter is defined as the number of sections in either the comb part *or* the integrator part of the filter. This value does not represent the total number of sections throughout the entire filter.
- $R$  is the interpolation factor.
- $M$  is the differential delay.

In the overall multirate realization, the algorithm applies the noble identity for interpolation and moves the rate change factor,  $R$ , to follow after the  $N$  sections of the cascaded comb filters.

The transfer function of the resulting filter is given by the following equation:

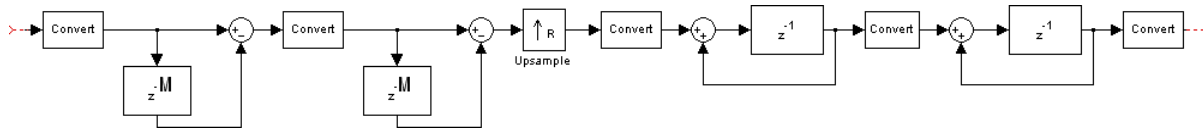
$$H(z) = \frac{(1 - z^{-M})^N}{(1 - z^{-1})^N}.$$

For a block diagram that shows the multirate implementation, see Algorithms on page 2-250.

## Algorithms

### CIC Interpolation Filter

The CIC interpolation filter in More About on page 2-249 is realized as a cascade of  $N$  sections of comb filters followed by a rate change by a factor  $R$ , followed by  $N$  sections of cascaded integrators.



This diagram shows two sections of cascaded comb filters and two sections of cascaded integrators. The unit delay in the integrator portion of the CIC filter can be located in either the feedforward or the feedback path. These two configurations yield identical filter frequency response. However, the numerical outputs from these two configurations are different due to the latency. This algorithm puts the unit delay in the feedforward path of the integrator since it is a preferred configuration for HDL implementation.

## Version History

Introduced before R2006a

## References

- [1] Hogenauer, E.B. "An Economical Class of Digital Filters for Decimation and Interpolation" *IEEE Transactions on Acoustics, Speech and Signal Processing*. Vol. 29, Number 2, 1981, pp. 155-162, 1981.
- [2] Meyer-Baese, U. *Digital Signal Processing with Field Programmable Gate Arrays*. New York: Springer Verlag, 2001.
- [3] Harris, Fredric J., *Multirate Signal Processing for Communication Systems*. Upper Saddle River, NJ: Prentice Hall PTR, 2004.

## Extended Capabilities

### C/C++ Code Generation

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

Generated code relies on the memcpy or memset function (`string.h`) under certain conditions.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

---

**Note** For an HDL-optimized filter architecture with hardware-friendly control signals, use the CIC Interpolator block. The DSP HDL Toolbox block simulates the latency of the HDL algorithm in Simulink.

---

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Coder supports **Coefficient source** options **Dialog parameters** and **Filter object**.

### AddPipelineRegisters Support

When you use **AddPipelineRegisters**, registers are placed based on the filter structure. The pipeline register placement determines the latency.

| Pipeline Register Placement  | Latency (clock cycles)                          |
|--|---|
| A pipeline register is added between the comb stages of the differentiators. | NS, the number of sections (at the input side). |

### HDL Filter Properties

|                             |  |
|-----------------------------|--|
| <b>AddPipelineRegisters</b> | Insert a pipeline register between stages of computation in a filter. See also AddPipelineRegisters (HDL Coder). |
|-----------------------------|--|

### HDL Block Properties

|                                  |  |
|----------------------------------|--|
| <b>ConstrainedOutputPipeline</b> | Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder). |
| <b>InputPipeline</b>             | Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).                 |
| <b>OutputPipeline</b>            | Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).               |

**Restrictions**

- Vector and frame inputs are not supported for HDL code generation.
- When you select **Dialog parameters**, the **Filter Structure** option **Zero-latency interpolator** is not supported for HDL code generation. From the **Filter Structure** drop-down list, select **Interpolator**.
- When you use **AddPipelineRegisters**, delays in parallel paths are not automatically balanced. Manually add delays where required by your design.

**Fixed-Point Conversion**

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

If the input is fixed point, it must be a signed integer or a signed fixed point value with a power-of-two slope and zero bias.

**See Also****Functions**

`filter`

**Objects**

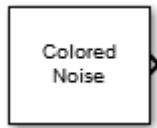
`dsp.CICDecimator` | `dsp.CICInterpolator` | `dsp.FIRDecimator` | `dsp.FIRInterpolator` | `dsp.CICCompensationDecimator` | `dsp.CICCompensationInterpolator` | `dsp.FIRHalfbandDecimator` | `dsp.FIRHalfbandInterpolator`

**Blocks**

`CIC Decimation` | `FIR Decimation` | `FIR Interpolation` | `Digital Down-Converter` | `Digital Up-Converter`

# Colored Noise

Generate colored noise signal



**Libraries:**  
DSP System Toolbox / Sources

## Description

The Colored Noise block generates a colored noise signal with a power spectral density of  $1/|f|^\alpha$  over its entire frequency range. The inverse power spectral density component,  $\alpha$ , can be any value in the interval  $[-2 \ 2]$ . The type of colored noise the block generates depends on the **Noise color** option you choose in the block dialog box. When you set **Noise color** to **custom**, you can specify the power density of the noise through the **Power of inverse frequency** parameter.

## Ports

### Output

**Port\_1** — Colored noise signal  
scalar | vector | matrix

Colored noise output signal. The size and data type of the signal depend on the values of the **Number of output channels**, **Number of samples per output channel**, and **Output data type** parameters.

Data Types: single | double

## Parameters

**Noise color** — Color of the generated noise

pink (default) | white | brown | blue | purple | custom

Color of the noise the block generates. You can set this parameter to:

- **pink** — Generates pink noise. This option is equivalent to setting **Power of inverse frequency** to 1.
- **white** — Generates white noise (flat power spectral density). This option is equivalent to setting **Power of inverse frequency** to 0.
- **brown** — Generates brown noise. Also known as red or Brownian noise. This option is equivalent to setting **Power of inverse frequency** to 2.
- **blue** — Generates blue noise. Also known as azure noise. This option is equivalent to setting **Power of inverse frequency** to -1.
- **purple** — Generates violet (purple) noise. This option is equivalent to setting **Power of inverse frequency** to -2.

- **custom** — Specify the power density of the noise using the **Power of inverse frequency** parameter.

**Power of inverse frequency** — Inverse power spectral density component

1 (default) | scalar in the range [-2 2]

Inverse power spectral density component,  $\alpha$ , specified as a real-valued scalar in the interval [-2 2]. The inverse exponent defines the power spectral density of the random process by  $1/|f|^\alpha$ . The default value of this property is 1. When **Power of inverse frequency** is greater than 0, the block generates lowpass noise, with a singularity (pole) at  $f=0$ . These processes exhibit long memory. When **Power of inverse frequency** is less than 0, the block generates highpass noise with negatively correlated increments. These processes are referred to as antipersistent. In a log-log plot of power as a function of frequency, processes generated by the Colored Noise block exhibit an approximate linear relationship, with the slope equal to  $-\alpha$ .

#### Dependencies

To enable this parameter, set **Noise color** to **custom**.

**Guarantee the output is bounded (+/-1)** — Set output bounds to +1 and -1

off (default) | on

Select the parameter to make the output bounded between +1 and -1.

When you select the parameter, the internal random source that generates the noise is uniform. If **Noise color** is set to **white**, there is no color filter applied to the output of the random source. The output is uniform noise of amplitude between +1 and -1. If **Noise color** is set to any other option, then a coloring filter is applied to the output of the random source, followed by a gain that ensures the absolute maximum output never exceeds 1.

When you do not select the parameter, the internal random source is Gaussian. The output is not bounded.

**Number of output channels** — Number of output channels

1 (default) | positive integer

Number of output channels, specified as a positive integer scalar. This parameter defines the number of columns in the generated signal.

**Output data type** — Output data type

double (default) | single

Data type of the output specified as **double** or **single**.

**Number of samples per output channel** — Samples per frame of output

1024 (default) | positive integer

Number of samples in each frame of the output signal, specified as a positive integer scalar. This parameter defines the number of rows in the generated signal.

**Output sample time (s)** — Sample time of the output

1 (default) | positive scalar

Sample time of the output signal, specified as a positive scalar in seconds.

**Initial seed** — Initial seed of random number generator

67 (default) | positive integer

Initial seed of the random number generator algorithm, specified as a real-valued positive integer scalar.

**Simulate using** — Type of simulation to run

Interpreted execution (default) | Code generation

Type of simulation to run. You can set this parameter to:

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

## Block Characteristics

|                                 |                 |
|---------------------------------|-----------------|
| <b>Data Types</b>               | double   single |
| <b>Multidimensional Signals</b> | No              |
| <b>Variable-Size Signals</b>    | No              |

## More About

### Colored Noise Processes

Many phenomena in diverse fields, such as hydrology and finance, produce time series with PSD functions that follow a power law of the form

$$S(f) = \frac{L(f)}{|f|^\alpha}$$

where  $\alpha$  is a real number in the interval  $[-2,2]$  and  $L(f)$  is a positive, slowly-varying or constant function. Plotting the PSD of such processes on a log-log plot displays an approximate linear relationship between the log frequency and log PSD with slope equal to  $-\alpha$

$$\ln S(f) = -\alpha \ln |f| + \ln L(f).$$

It is often convenient to plot the PSD in dB as a function of the frequency on a base-2 logarithmic scale. The slope of the plot is then dB/octave. Rewriting the preceding equation, you obtain

$$10\log S(f) = -10\alpha \frac{\ln(2)\log_2(f)}{\ln(10)} + 10 \frac{\ln(L(f))}{\ln(10)}$$

with the slope in dB/octave given by

$$-10\alpha \frac{\ln(2)\log_2(f)}{\ln(10)}$$

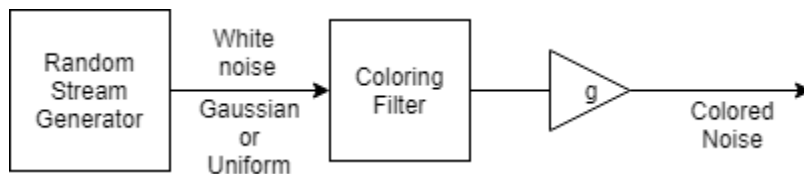
If  $\alpha > 0$ ,  $S(f)$  goes to infinity as the frequency,  $f$ , approaches 0. Stochastic processes with PSDs of this form exhibit long memory. Long-memory processes have autocorrelations that persist for a long time as opposed to decaying exponentially like many common time-series models. If  $\alpha < 0$ , the process is antipersistent and exhibits negative correlation between increments [1].

Special examples of  $\frac{1}{|f|^\alpha}$  processes include:

- $\alpha = 0$  — White noise, where  $L(f)$  is a constant proportional to the process variance.
- $\alpha = 1$  — Pink, or flicker noise. Pink noise has equal energy per octave. See “Measure Pink Noise Power in Octave Bands” on page 4-274 for a demonstration. The power spectral density of pink noise decreases 3 dB per octave.
- $\alpha = 2$  — brown noise, or Brownian motion. Brownian motion is a nonstationary process with stationary increments. You can think of Brownian motion as the integral of a white noise process. Even though Brownian motion is nonstationary, you can still define a generalized power spectrum, which behaves like  $\frac{1}{|f|^2}$ . Accordingly, power in a brown noise decreases 6 dB per octave.
- $\alpha = -1$  — blue noise. The power spectral density of blue noise increases 3 dB per octave.
- $\alpha = -2$  — violet, or purple noise. The power spectral density of violet noise increases 6 dB per octave. You can think of violet noise as the derivative of white noise process.

## Algorithms

The figure shows the overall process of generating the colored noise.



The random stream generator produces a stream of white noise that is either Gaussian or uniform in distribution. A coloring filter applied to the white noise generates colored noise with a power spectral density (PSD) function given by:

$$S(f) = \frac{L(f)}{|f|^\alpha}$$

When  $\alpha$ , the inverse frequency power, equals 0, no coloring filter is applied to the output of the random stream generator. If the bounded option is enabled, the output is uniform white noise with amplitude between +1 and -1. If the bounded output is not enabled, the output is a Gaussian white noise and the values are not bounded between +1 and -1. If  $\alpha$  is set to any other value, then a coloring filter is applied to the output of the random stream generator. If the bounded output option is



enabled, a gain  $g$  is applied to the output of the coloring filter to ensure that the absolute maximum output never exceeds 1.

For details on colored noise processes and how the value of  $\alpha$  affects the PSD of the colored noise, see “Colored Noise Processes” on page 2-255.

When the inverse frequency power  $\alpha$  is positive, the colored noise is generated using an autoregressive (AR) model of order 63. The AR coefficients are:

$$a_0 = 1,$$

$$a_k = (k - 1 - \frac{\alpha}{2}) \frac{a_{k-1}}{k}, \quad k = 1, 2, \dots, 63$$

Pink and brown noises are special cases, which are generated from specially tuned SOS filters of orders 12 and 10, respectively. These filters are optimized for better performance.

When the inverse frequency power  $\alpha$  is negative, the colored noise is generated using a moving average (MA) model of order 255. The MA coefficients are:

$$b_0 = 1,$$

$$b_k = (k - 1 + \frac{\alpha}{2}) \frac{b_{k-1}}{k}, \quad k = 1, 2, \dots, 255$$

Purple noise is generated from a first order filter,  $B = [1 \ -1]$ .

The coloring filters applied (except pink, brown, and purple) are detailed on pp. 820-822 in [2].

## Version History

Introduced in R2015a

## References

- [1] Beran, J., Feng, Y., Ghosh, S., and Kulik, R. *Long-Memory Processes: Probabilistic Properties and Statistical Methods*. Springer, 2013.
- [2] Kasdin, N.J. "Discrete Simulation of Colored Noise and Stochastic Processes and  $1/f^\alpha$  Power Law Noise Generation". *Proceedings of the IEEE*. Vol. 83, No. 5, 1995, pp. 802-827.

## Extended Capabilities

### C/C++ Code Generation

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

## See Also

### Functions

randn

### Objects

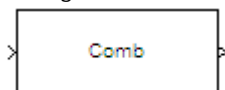
dsp.ColoredNoise

**Topics**

“Sample- and Frame-Based Concepts”

# Comb Filter

Design comb Filter



## Library

Filtering / Filter Designs

dspfdesign

## Description

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

## Dialog Box

See “Comb Filter Design —Main Pane” on page 5-611 for more information about the parameters of this block. The **Data Types** and **Code Generation** panes are not available for blocks in the DSP System Toolbox Filter Designs library.

### View filter response

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

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

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

## Filter Specifications

In this group, you specify the type of comb filter and the number of peaks or notches.

### Comb Type

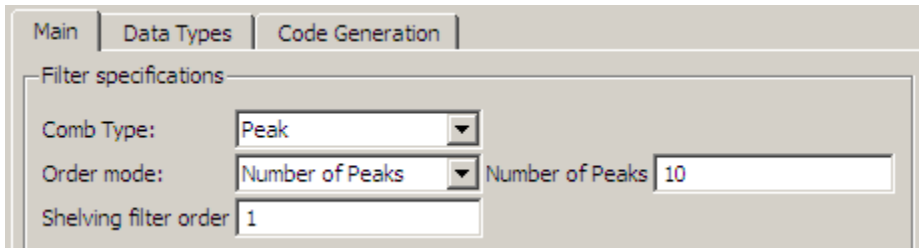
Select either **Notch** or **Peak** from the drop-down list. **Notch** creates a comb filter that attenuates a set of harmonically related frequencies. **Peak** creates a comb filter that amplifies a set of harmonically related frequencies.

### Order mode

Select either **Order** or **Number of Peaks/Notches** from the drop-down menu.

Select **Order** to enter the desired filter order in the dialog box. The comb filter has notches or peaks at increments of  $2/\text{Order}$  in normalized frequency units.

Select **Number of Peaks** or **Number of Notches** to specify the number of peaks or notches and the **Shelving filter order**



### Shelving filter order

The **Shelving filter order** is a positive integer that determines the sharpness of the peaks or notches. Larger values result in sharper peaks or notches.

### Frequency specifications

In this group, you specify the frequency constraints and frequency units.

#### Frequency specifications

Select either **Quality factor** or **Bandwidth**.

**Quality factor** is the ratio of the center frequency of the peak or notch to the bandwidth calculated at the -3 dB point.

**Bandwidth** specifies the bandwidth of the peak or notch. By default the bandwidth is measured at the -3 dB point. For example, setting the bandwidth equal to 0.1 results in 3 dB frequencies at normalized frequencies 0.05 above and below the center frequency of the peak or notch.

#### Frequency Units

Specify the frequency units. The default is normalized frequency. Choosing an option in Hz enables the **Input sample rate** dialog box.

#### Magnitude specifications

Specify the units for the magnitude specification and the gain at which the bandwidth is measured. This menu is disabled if you specify a filter order. Select one of the following magnitude units from the drop down list:

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

**Bandwidth gain** — Specify the gain at which the bandwidth is measured. The default is -3 dB.

### Algorithm

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

#### Design Method

The IIR Butterworth design is the only option for peaking or notching comb filters.

## Filter Implementation

### Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter.

### Use basic elements to enable filter customization

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

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

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

### Optimize for unit-scale values

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

### Input processing

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

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

### Rate options

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

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

### Use variable names for coefficients

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

## Supported Data Types

| Port   | Supported Data Types  |
|--------|---|
| Input  | <ul style="list-style-type: none"><li>• Double-precision floating point</li><li>• Single-precision floating point</li></ul> |
| Output | <ul style="list-style-type: none"><li>• Double-precision floating point</li><li>• Single-precision floating point</li></ul> |

## Version History

Introduced in R2010a

## Extended Capabilities

### C/C++ Code Generation

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

# Complex Bandpass Decimator

Extract a frequency subband using a one-sided (complex) bandpass decimator



## Libraries:

DSP System Toolbox / Filtering / Multirate Filters

## Description

The Complex Bandpass Decimator block extracts a specific subband of frequencies using a one-sided, multistage, complex bandpass decimator. The block determines the bandwidth of interest using the specified center frequency, decimator factor, and bandwidth values.

This block supports SIMD code generation. For details, see “Code Generation” on page 2-270.

## Ports

### Input

**x** — Input data  
vector | matrix

Specify the input data as a vector or a matrix.

The block treats each column of the input signal as a separate channel. If the input is a two-dimensional signal, the first dimension represents the channel length (or frame size) and the second dimension represents the number of channels. If the input is a one-dimensional signal, then the block interprets it as a single-channel signal.

When you input a variable-size signal (frame length changes during simulation), the frame length of the signal can be arbitrary, that is, the input frame length does not have to be a multiple of the decimation factor. When you input a fixed-size signal (frame length does not change during simulation), the frame length can be arbitrary only when you select the **Allow arbitrary frame length for fixed-size input signals** parameter.

To determine the decimation factor, click the **View Info** button.

This port is unnamed until you select the **Specify center frequency from input port** parameter.

Data Types: `single` | `double`

**Fc** — Center frequency  
real scalar

Center frequency of the desired band in Hz, specified as a real, finite numeric scalar in the range  $[-F_s/2, F_s/2]$ . The value of  $F_s$  depends on the **Inherit sample rate from input** parameter. When you select this parameter,  $F_s$  is the value the block inherits from the input signal. When you clear this parameter,  $F_s$  is the value you specify in the **Input sample rate (Hz)** parameter.

To enable this port, select the **Specify center frequency from input port** parameter.

Data Types: `single` | `double`

## Output

**Port\_1** — Filtered output  
vector | matrix

Output of the complex bandpass decimator, returned as a vector or a matrix. The output contains the subband of frequencies specified by the parameters on the block dialog.

The output signal dimensions depend on the input signal dimensions, the overall decimation factor of the Complex Bandpass Decimator block, and whether you select the **Allow arbitrary frame length for fixed-size input signals** parameter. You can determine the overall decimator factor  $D$  by clicking the **View Info** button in the block dialog box.

This table provides more details on the dimensions of the output signal when you input a fixed-size input signal.

### Fixed-Size Input Signal

| Input Signal  | Output Signal Dimensions  |
|---|---|
| $P$ -by-1 column vector, where $P$ is a multiple of the decimation factor $D$     | $(P/D)$ -by-1   |
| $P$ -by-1 column vector, where $P$ is not a multiple of the decimation factor $D$ | $\text{ceil}(P/D)$ -by-1 when you select <b>Allow arbitrary frame length for fixed-size input signals</b> .<br><br>If you do not select <b>Allow arbitrary frame length for fixed-size input signals</b> , the block errors.    |
| $P$ -by- $Q$ matrix, where $P$ is a multiple of the decimation factor $D$         | $(P/D)$ -by- $Q$  |
| $P$ -by- $Q$ matrix, where $P$ is not a multiple of the decimation factor $D$     | $\text{ceil}(P/D)$ -by- $Q$ when you select <b>Allow arbitrary frame length for fixed-size input signals</b> .<br><br>If you do not select <b>Allow arbitrary frame length for fixed-size input signals</b> , the block errors. |

This table gives more details on the dimensions of the output signal when you input a variable-size input signal. When you input a variable-size signal (frame length changes during simulation), the **Allow arbitrary frame length for fixed-size input signals** parameter is visible in the block dialog box but does not have any impact on the input frame length. You can input a variable-size signal of any frame length even if you do not select the **Allow arbitrary frame length for fixed-size input signals** parameter.

### Variable-Size Input Signal

| Input Signal | Output Signal Dimensions    |
|--------------|-----------------------------|
| $P$ -by- $Q$ | $\text{ceil}(P/D)$ -by- $Q$ |

The data type of the output is same as the data type of the input. The output signal is always complex.



Data Types: single | double

## Parameters

**Filter specification** — Filter design parameters

Decimation factor (default) | Bandwidth | Decimation factor and bandwidth

Filter design parameters, specified as one of the following:

- **Decimation factor** -- The block specifies the decimation factor through the **Decimation factor** parameter. The bandwidth of interest ( $BW$ ) is computed using the following equation:

$$BW = Fs/D$$

where

- $Fs$  -- Sample rate specified through the **Input sample rate (Hz)** parameter.
- $D$  -- Decimation factor.
- **Bandwidth** -- The block specifies the bandwidth through the **Bandwidth (Hz)** parameter. The decimation factor ( $D$ ) is computed using the following equation:

$$D = \text{floor}\left(\frac{Fs}{BW + TW}\right)$$

where

- $Fs$  -- Sample rate specified through the **Input sample rate (Hz)** parameter.
- $BW$  -- Bandwidth of interest.
- $TW$  -- Transition width specified through the **Transition width (Hz)** parameter.
- **Decimation factor and bandwidth** -- The decimation factor and the bandwidth of interest are specified through the **Decimation factor** and **Bandwidth (Hz)** parameters.

**Decimation factor** — Decimation factor

2 (default) | positive integer

Factor by which to reduce the bandwidth of the input signal, specified as a positive integer. The frame size (number of rows) of the input signal must be a multiple of the decimation factor.

### Dependencies

This parameter applies when you set **Filter specification** to either **Decimation factor** or **Decimation factor and bandwidth**.

**Bandwidth (Hz)** — Bandwidth in Hz

5000 (default) | real positive scalar

Width of the frequency band of interest, specified as a real positive scalar in Hz.

**Dependencies**

This parameter applies when you set **Filter specification** to either Bandwidth or Decimation factor and bandwidth.

Data Types: single | double

**Specify center frequency from input port** — Flag to specify center frequency  
off (default) | on

When you select this check box, the center frequency is input through the **Fc** port. When you clear this check box, the center frequency is specified on the block dialog through the **Center frequency (Hz)** parameter.

When you select this check box, the block does not compute the filter response. To view the filter response, clear this check box, specify the center frequency on the block dialog, and click **View Filter Response** button.

**Center frequency (Hz)** — Center frequency in Hz  
0 (default) | real scalar

Center frequency of the desired band in Hz, specified as a real, finite numeric scalar in the range  $[-Fs/2, Fs/2]$ .

**Tunable:** Yes

Data Types: single | double

**Stopband attenuation (dB)** — Stopband attenuation in dB  
80 (default) | positive scalar

Stopband attenuation of the filter in dB, specified as a finite positive scalar.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**Passband ripple (dB)** — Passband ripple in dB  
1 (default) | positive scalar

Passband ripple of the filter, specified as a positive scalar in dB.

**Dependencies**

This parameter applies when you set **Filter specification** to either Bandwidth or Decimation factor and bandwidth.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**Transition width (Hz)** — Transition width in Hz  
100 (default) | positive scalar

Transition width of the filter in Hz, specified as a positive scalar.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**Reduce number of complex coefficients** — Minimize number of complex coefficients  
on (default) | off

Minimize the number of complex coefficients. When you select this parameter, the first stage of the multistage filter is bandpass (with complex coefficients) centered at the specified center frequency. The first stage is followed by a mixing stage that heterodynes the signal to DC. The remaining filter stages, all with real coefficients, follow.

When you clear the parameter, the input signal is first passed through the different stages of the multistage filter. All stages are bandpass (complex coefficients). The signal is then heterodyned to DC if **Mix signal to baseband** parameter is selected and the frequency offset resulting from the decimation is nonzero.

**Mix signal to baseband** — Mix signal to baseband  
on (default) | off

Mix the signal to baseband. When you select this parameter, the block heterodynes the filtered, decimated signal to DC. This mixing stage runs at the output sample rate of the filter. When you clear this parameter, the block skips the mixing stage.

#### Dependencies

This parameter applies when you clear the **Reduce number of complex coefficients** parameter.

**Inherit sample rate from input** — Flag to specify input sample rate  
off (default) | on

When you select this parameter, the block inherits its sample rate from the input signal. The block calculates the sample rate based on the sample time of the input port. When you clear this parameter, specify the sample rate in **Input sample rate (Hz)**.

**Input sample rate (Hz)** — Input sample rate in Hz  
44100 (default) | real positive scalar

Sampling rate of the input signal in Hz, specified as a real positive scalar.

#### Dependencies

This parameter applies when you clear the **Inherit sample rate from input** parameter.

Data Types: single | double

**Allow arbitrary frame length for fixed-size input signals** — Allow arbitrary frame length for fixed-size input signals

off (default) | on

Specify whether fixed-size input signals (whose size does not change during simulation) can have an arbitrary frame length, where the frame length does not have to be a multiple of the decimation factor. The block uses this parameter only for fixed-size input signals and ignores it if the input data varies in size during simulation.

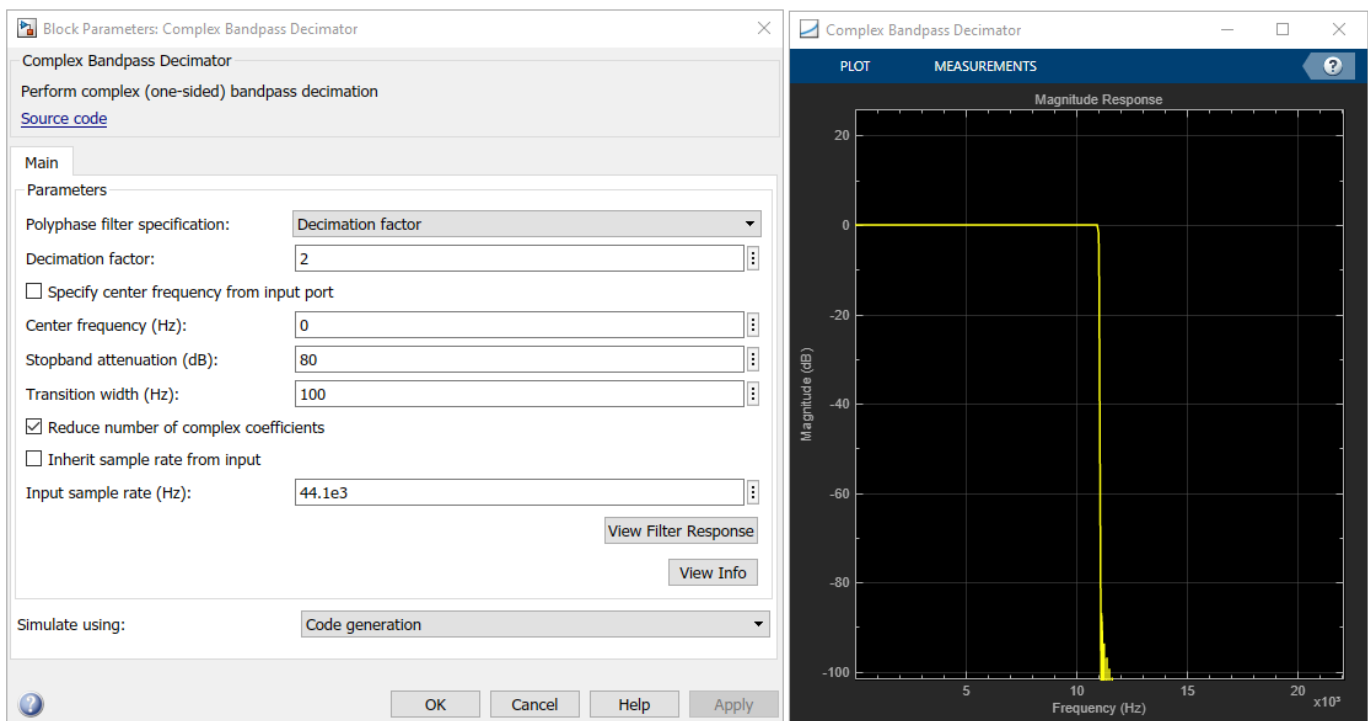
When the input signal is a variable-size signal, the signal can have an arbitrary frame length, that is, the frame length does not have to be a multiple of the decimation factor.

For fixed-size input signals, if you:

- Select the **Allow arbitrary frame length for fixed-size input signals** parameter, the frame length of the signal does not have to be a multiple of the decimation factor. If the input is not a multiple of the decimation factor, then the output is generally a variable-size signal. Therefore, to support arbitrary input size, the block must also support variable-size operations, which you can enable by selecting the **Allow arbitrary frame length for fixed-size input signals** parameter.
- Clear the **Allow arbitrary frame length for fixed-size input signals** parameter, the input frame length must be a multiple of the decimation factor.

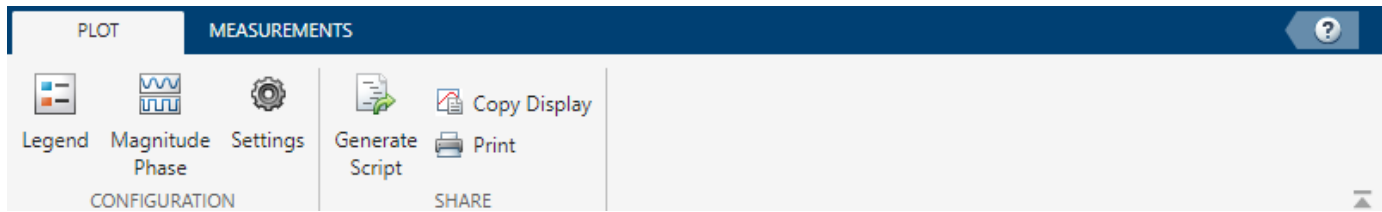
**View Filter Response** — Visualize frequency response button

Click this button to open the Dynamic Filter Visualizer window and display the magnitude and phase response of the complex bandpass decimator. The response is based on the parameters you select in the block dialog box. To update the response while the visualizer is running, modify the parameters in the dialog box and click **Apply**.

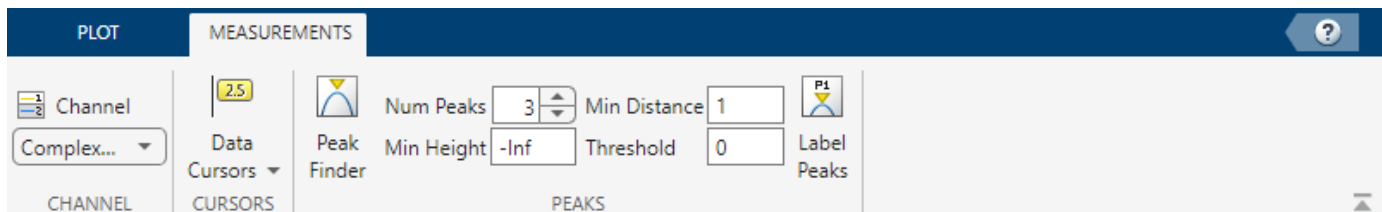


You can configure the plot settings and the frequency response measurements from the interface of the visualizer.

On the **Plot** tab, you can enable the legend, specify to plot both magnitude and phase responses of the filter, modify the plot settings, generate a script to re-create the plot, and even save or share the settings.



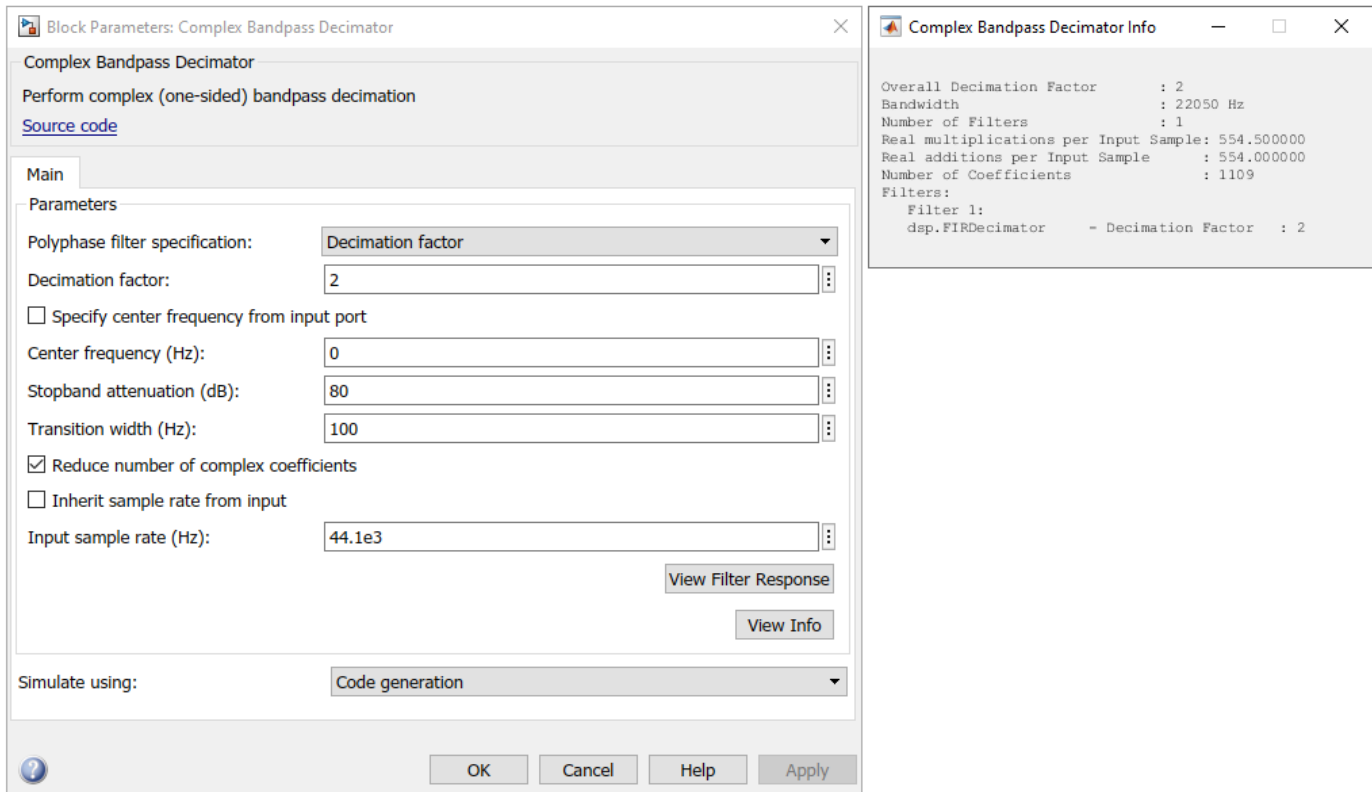
On the **Measurements** tab, you can enable data cursors, and display the peak values of the filter response.



For more details on the Dynamic Filter Visualizer interface and its tools, see “Configure Filter Visualizer”.

**View Info** — View filter information button

Display filter information of the Complex Bandpass Decimator block:



This button brings the functionality of the `info` analysis method into the Simulink environment.

**Simulate using** — Type of simulation to run  
Code generation (default) | Interpreted execution

Specify the type of simulation to run as one of the following:

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

## Block Characteristics

|                                 |                 |
|---------------------------------|-----------------|
| <b>Data Types</b>               | double   single |
| <b>Multidimensional Signals</b> | No              |
| <b>Variable-Size Signals</b>    | Yes             |

## Algorithms

The complex bandpass decimator is designed by applying a complex frequency shift transformation on a lowpass prototype filter. The lowpass prototype in this case is a multirate, multistage finite impulse response (FIR) filter. The desired frequency shift applies only to the first stage. Subsequent stages scale the desired frequency shift by their respective cumulative decimation factors. For details, see “Complex Bandpass Filter Design” and “Zoom FFT”.

## Version History

**Introduced in R2018a**

### **R2023a: Support for arbitrary input frame length**

This block supports input signals with arbitrary frame lengths when the:

- Input signal is a fixed-size signal (frame length does not change during simulation) and you select the **Allow arbitrary frame length for fixed-size input signals** parameter.
- Input signal is a variable-size signal (frame length changes during simulation).

When this block supports an input signal with an arbitrary frame length, the input frame length does not have to be a multiple of the decimation factor and the output signal is a variable-size signal.

## Extended Capabilities

### **C/C++ Code Generation**

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

The Complex Bandpass Decimator block supports SIMD code generation using Intel AVX2 technology under these conditions:

- Input signal is complex-valued.
- Input signal has a data type of `single` or `double`.

The SIMD technology significantly improves the performance of the generated code.

## **See Also**

### **Objects**

`dsp.ComplexBandpassDecimator`

### **Topics**

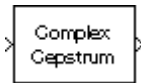
“IF Subsampling with Complex Multirate Filters”

“Complex Bandpass Filter Design”

“Zoom FFT”

## Complex Cepstrum

Compute complex cepstrum of input



**Libraries:**  
DSP System Toolbox / Transforms

### Description

The Complex Cepstrum block computes the complex cepstrum of the input.

### Ports

#### Input

**Input** — Input  
vector | matrix

Specify the input  $u$  as a vector or a matrix of size  $M$ -by- $N$ . The block treats each column of the input as an independent channel containing  $M$  consecutive samples. The block always processes unoriented vector inputs as a single channel, and returns the result as a length- $M$  column vector. The block does not accept complex-valued inputs.

The input is altered by the application of a linear phase term so that there is no phase discontinuity at  $\pm\pi$  radians. That is, each input channel is independently zero padded and circularly shifted to have zero phase at  $\pi$  radians.

Data Types: single | double

#### Output

**Output** — Complex cepstrum output  
vector | matrix

Complex cepstrum output, returned as an  $M_o$ -by- $N$  matrix.

When you clear the **Inherit FFT length from input port dimensions** parameter,  $M_o$  is the value you specify in the **FFT length** parameter. When you select the **Inherit FFT length from input port dimensions** parameter,  $M_o = M$ , where  $M$  is the frame size (number of rows) of the input signal.

Each output column contains the length- $M_o$  complex cepstrum of the corresponding input column.

Here is the equivalent MATLAB code for computing the complex cepstrum.

```
y = cceps(u,Mo)
```

where,

- $u$  -- Input
- $y$  -- Complex cepstrum output



- $M_o$  -- Output frame size

The output port rate is the same as the input port rate.

Data Types: `single` | `double`

## Parameters

**Inherit FFT length from input port dimensions** — Match the FFT length to the dimensions of the input port

`on` (default) | `off`

When you select this check box, the output frame size  $M_o$  matches the input frame size  $M$ .

**FFT length** — Compute FFT length

64 (default) | positive integer

The number of frequency points at which to compute the FFT, which is also the output frame size  $M_o$ .

### Dependency

To enable this parameter, clear the **Inherit FFT length from input port dimensions** parameter.

## Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | <code>double</code>   <code>single</code> |
| <b>Multidimensional Signals</b> | No  |
| <b>Variable-Size Signals</b>    | No  |

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

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

Generated code relies on the `memcpy` or `memset` function (`string.h`) under certain conditions.

## See Also

### Functions

`cceps`

### Blocks

DCT | FFT | Real Cepstrum

## Constant

Generate constant value



### Libraries:

Simulink / Commonly Used Blocks  
 Simulink / Sources  
 DSP System Toolbox / Sources  
 HDL Coder / Commonly Used Blocks  
 HDL Coder / Sources

## Description

The Constant block generates a real or complex constant value signal. Use this block to provide a constant signal input. The block generates scalar, vector, or matrix output, depending on:

- The dimensionality of the **Constant value** parameter
- The setting of the **Interpret vector parameters as 1-D** parameter

The output of the block has the same dimensions and elements as the **Constant value** parameter. If you specify for this parameter a vector that you want the block to interpret as a vector, select the **Interpret vector parameters as 1-D** check box. Otherwise, if you specify a vector for the **Constant value** parameter, the block treats that vector as a matrix.

---

**Tip** To output a constant enumerated value, consider using the Enumerated Constant block instead. The Constant block provides block parameters that do not apply to enumerated types, such as **Output minimum** and **Output maximum**.

---

### Using Bus Objects as the Output Data Type

The Constant block supports nonvirtual buses as the output data type. Using a bus object as the output data type can help simplify your model. If you use a bus object as the output data type, set the **Constant value** to  $\emptyset$  or to a MATLAB structure that matches the bus object.

### Using Structures for the Constant Value of a Bus

The structure you specify must contain a value for every element of the bus represented by the bus object. The block output is a nonvirtual bus signal.

You can use the `Simulink.Bus.createMATLABStruct` to create a full structure that corresponds to a bus.

You can use `Simulink.Bus.createObject` to create a bus object from a MATLAB structure.

If the signal elements in the output bus use numeric data types other than `double`, you can specify the structure fields by using typed expressions such as `uint16(37)` or untyped expressions such as `37`. To control the field data types, you can use the bus object as the data type of a `Simulink.Parameter` object. To decide whether to use typed or untyped expressions, see “Control Data Types of Initial Condition Structure Fields” (Simulink).

## Setting Configuration Parameters to Support Using a Bus Object Data Type

To enable the use of a bus object as an output data type, before you start a simulation, set **Configuration Parameters > Diagnostics > Data Validity > Advanced parameters > Underspecified initialization detection** to Simplified. For more information, see “Underspecified initialization detection” (Simulink).

## Ports

### Output

#### Port\_1 — Constant value

scalar | vector | matrix | N-D array

Constant value, specified as a real or complex valued scalar, vector, matrix, or N-D array. By default, the Constant block outputs a signal whose dimensions, data type, and complexity are the same as those of the **Constant value** parameter. However, you can specify the output to be any data type that Simulink supports, including fixed-point and enumerated data types.

---

**Note** If you specify a bus object as the data type for this block, do not set the maximum value for bus data on the block. Simulink ignores this setting. Instead, set the maximum values for bus elements of the bus object specified as the data type. For more information, see `Simulink.BusElement`.

---

Data Types: single | double | half | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | Boolean | fixed point | enumerated | bus

## Parameters

### Main

#### Constant value — Constant output value

1 (default) | scalar | vector | matrix | N-D array

Specify the constant value output of the block.

- You can enter any expression that MATLAB evaluates as a matrix, including the Boolean keywords `true` and `false`.
- If you set the **Output data type** to be a bus object, you can specify one of these options:
  - A full MATLAB structure corresponding to the bus object
  - `0` to indicate a structure corresponding to the ground value of the bus object

For details, see “Using Bus Objects as the Output Data Type” on page 2-274.

- For nonbus data types, Simulink converts this parameter from its value data type to the specified output data type offline, using a rounding method of nearest and overflow action of saturate.

### Programmatic Use

**Block Parameter:** Value

**Type:** character vector  
**Value:** scalar | vector | matrix | N-D array  
**Default:** '1'

### Interpret vector parameters as 1-D — Treat vectors as 1-D

on (default) | off

Select this check box to output a vector of length N if the **Constant value** parameter evaluates to an N-element row or column vector.

- When you select this check box, the block outputs a vector of length N, provided the **Constant value** parameter evaluates to an N-element row or column vector.
- When you clear this check box, the block outputs a matrix of dimension 1-by-N or N-by-1, provided the **Constant value** parameter evaluates to an N-element row or column vector. For example, the block outputs a matrix of dimension 1-by-N or N-by-1.

#### Programmatic Use

**Block Parameter:** VectorParams1D

**Type:** character vector  
**Values:** 'on' | 'off'  
**Default:** 'on'

### Sample time — Sampling interval

inf (default) | scalar | vector

Specify the interval between times that the Constant block output can change during simulation (for example, due to tuning the **Constant value** parameter).

The default value of `inf` indicates that the block output can never change. This setting speeds simulation and generated code by avoiding the need to recompute the block output.

See “Specify Sample Time” (Simulink) for more information.

#### Programmatic Use

**Block Parameter:** SampleTime

**Type:** character vector  
**Values:** scalar | vector  
**Default:** 'inf'

### Signal Attributes

#### Output minimum — Minimum output value for range checking

[] (default) | scalar

Specify the lower value of the output range that Simulink checks as a finite, real, double, scalar value.

---

**Note** If you specify a bus object as the data type for this block, do not set the minimum value for bus data on the block. Simulink ignores this setting. Instead, set the minimum values for bus elements of the bus object specified as the data type. For information on the Minimum parameter for a bus element, see `Simulink.BusElement`.

---

Simulink uses the minimum to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters” (Simulink)) for some blocks.
- Simulation range checking (see “Specify Signal Ranges” (Simulink) and “Enable Simulation Range Checking” (Simulink)).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see Optimize using the specified minimum and maximum values (Embedded Coder).

---

**Note Output minimum** does not saturate or clip the actual output signal. Use the Saturation block instead.

---

#### Programmatic Use

**Block Parameter:** OutMin

**Type:** character vector

**Values:** scalar

**Default:** ' [ ] '

**Output maximum** — Maximum output value for range checking

[ ] (default) | scalar

Specify the upper value of the output range that Simulink checks as a finite, real, double, scalar value.

---

**Note** If you specify a bus object as the data type for this block, do not set the maximum value for bus data on the block. Simulink ignores this setting. Instead, set the maximum values for bus elements of the bus object specified as the data type. For information on the Maximum parameter for a bus element, see `Simulink.BusElement`.

---

Simulink uses the maximum value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters” (Simulink)) for some blocks.
- Simulation range checking (see “Specify Signal Ranges” (Simulink) and “Enable Simulation Range Checking” (Simulink)).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see Optimize using the specified minimum and maximum values (Embedded Coder).

---


**Note Output maximum** does not saturate or clip the actual output signal. Use the Saturation block instead.

---

**Programmatic Use****Block Parameter:** OutMax**Type:** character vector**Values:** scalar**Default:** '[ ]'**Output data type** — Output data type

Inherit: Inherit from 'Constant value' (default) | Inherit: Inherit via back propagation | double | single | half | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | boolean | fixdt(1,16) | fixdt(1,16,0) | fixdt(1,16,2^0,0) | Enum: <class name> | Bus: <object name> | <data type expression>

Specify the output data type. The type can be inherited, specified directly, or expressed as a data type object such as `Simulink.NumericType`.

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant” (Simulink).

**Programmatic Use****Block Parameter:** OutDataTypeStr**Type:** character vector

**Values:** 'Inherit: Inherit from 'Constant value'' | 'Inherit: Inherit via back propagation' | 'double' | 'single' | 'half' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'int64' | 'uint64' | 'boolean' | 'fixdt(1,16)' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)' | 'Enum: <class name>' | 'Bus: <object name>'

**Default:** 'Inherit: Inherit from 'Constant value''**Lock output data type setting against changes by the fixed-point tools** — Prevent fixed-point tools from overriding Output data type

off (default) | on

Select this parameter to prevent the fixed-point tools from overriding the **Output** data type you specify on the block. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

**Programmatic Use****Block Parameter:** LockScale**Type:** character vector**Values:** 'off' | 'on'**Default:** 'off'**Mode** — Category of data to specify

Inherit (default) | Built in | Fixed point | Enumerated | Bus object | Expression

Select the category of data to specify.

**Inherit**

Inheritance rules for data types. Selecting **Inherit** enables a second menu/text box to the right. Select one of the following choices:

- Inherit from 'Constant value' (default)
- Inherit via back propagation

#### Built in

Built-in data types. Selecting **Built in** enables a second menu/text box to the right. Select one of the following choices:

- double (default)
- single
- int8
- uint8
- int16
- uint16
- int32
- uint32
- boolean

#### Fixed point

Fixed-point data types.

#### Enumerated

Enumerated data types. Selecting **Enumerated** enables a second menu/text box to the right, where you can enter the class name.

#### Bus object

Bus object. Selecting **Bus** enables a **Bus object** parameter to the right, where you enter the name of a bus object that you want to use to define the structure of the bus. If you need to create or change a bus object, click **Edit** to the right of the **Bus object** field to open the Simulink Type Editor. For details, see “Create and Specify Simulink.Bus Objects” (Simulink).

#### Expression

Expressions that evaluate to data types. Selecting **Expression** enables a second menu/text box to the right, where you can enter the expression.

Do not specify a bus object as the expression.

**Data type override** — Specify data type override mode for this signal

Inherit | Off

Select the data type override mode for this signal.

- When you select **Inherit**, Simulink inherits the data type override setting from its context, that is, from the block, Simulink.Signal object or Stateflow® chart in Simulink that is using the signal.
- When you select **Off**, Simulink ignores the data type override setting of its context and uses the fixed-point data type specified for the signal.

For more information, see “Specify Data Types Using Data Type Assistant” (Simulink) in the Simulink documentation.

**Dependencies**

To enable this parameter, click the **Show data type assistant** button, and set the **Mode** to **Built in** or **Fixed point**.

**Tips**

The ability to turn off data type override for an individual data type provides greater control over the data types in your model when you apply data type override. For example, you can use this option to ensure that data types meet the requirements of downstream blocks regardless of the data type override setting.

**Signedness** — Specify signed or unsigned

Signed (default) | Unsigned

Specify whether the fixed-point data is signed or unsigned. Signed data can represent positive and negative values, but unsigned data represents positive values only.

- **Signed**, specifies the fixed-point data as signed.
- **Unsigned**, specifies the fixed-point data as unsigned.

For more information, see “Specify Data Types Using Data Type Assistant” (Simulink).

**Dependencies**

To enable this parameter, set the **Mode** to **Fixed point**.

**Scaling** — Method for scaling fixed-point data

Best precision (default) | Binary point | Slope and bias

Specify the method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors. For more information, see “Specifying a Fixed-Point Data Type” (Simulink).

**Dependencies**

To enable this parameter, set **Mode** to **Fixed point**.

Setting **Scaling** to **Binary point** enables:

- **Fraction length**
- **Calculate Best-Precision Scaling**

Setting **Scaling** to **Slope and bias** enables:

- **Slope**
- **Bias**
- **Calculate Best-Precision Scaling**

**Word length** — Bit size of the word that holds the quantized integer

16 (default) | integer from 0 to 32

Specify the bit size of the word that holds the quantized integer. For more information, see “Specifying a Fixed-Point Data Type” (Simulink).



**Dependencies**

To enable this parameter, set **Mode** to `Fixed point`.

**Fraction length** — Specify fraction length for fixed-point data type

0 (default) | scalar integer

Specify fraction length for fixed-point data type as a positive or negative integer. For more information, see “Specifying a Fixed-Point Data Type” (Simulink).

**Dependencies**

To enable this parameter, set **Scaling** to `Binary point`.

**Slope** — Specify slope for the fixed-point data type

$2^0$  (default) | positive, real-valued scalar

Specify slope for the fixed-point data type. For more information, see “Specifying a Fixed-Point Data Type” (Simulink).

**Dependencies**

To enable this parameter, set **Scaling** to `Slope and bias`.

**Bias** — Specify bias for the fixed-point data type

0 (default) | real-valued scalar

Specify bias for the fixed-point data type as any real number. For more information, see “Specifying a Fixed-Point Data Type” (Simulink).

**Dependencies**

To enable this parameter, set **Scaling** to `Slope and bias`.

**Block Characteristics**

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | Boolean   bus   double   enumerated   fixed point   half   integer   single |
| <b>Direct Feedthrough</b>       | no  |
| <b>Multidimensional Signals</b> | yes   |
| <b>Variable-Size Signals</b>    | no  |
| <b>Zero-Crossing Detection</b>  | no  |

**Version History**

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

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

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

### Tunable Parameters

You can use a tunable parameter in a Constant block intended for HDL code generation. For details, see “Generate DUT Ports for Tunable Parameters” (HDL Coder).

### HDL Architecture

| Architecture     | Parameters     | Description  |
|------------------|----------------|--|
| default Constant | None           | This implementation emits the value of the Constant block.   |
| Logic Value      | None           | By default, this implementation emits the character 'Z' for each bit in the signal. For example, for a 4-bit signal, the implementation would emit 'ZZZZ'.   |
|                  | {'Value', 'Z'} | If the signal is in a high-impedance state, use this parameter value. This implementation emits the character 'Z' for each bit in the signal. For example, for a 4-bit signal, the implementation would emit 'ZZZZ'. |
|                  | {'Value', 'X'} | If the signal is in an unknown state, use this parameter value. This implementation emits the character 'X' for each bit in the signal. For example, for a 4-bit signal, the implementation would emit 'XXXX'.       |

### HDL Block Properties

|                                  |  |
|----------------------------------|--|
| <b>ConstrainedOutputPipeline</b> | Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder). |
| <b>InputPipeline</b>             | Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).                 |
| <b>OutputPipeline</b>            | Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).               |

### Complex Data Support

This block supports code generation for complex signals.

### Restrictions

- The `Logic Value` implementation does not support the `double` data type. If you specify this implementation for a constant value of type `double`, a code generation error occurs.
- Delay balancing does not support a Constant block that has **Sample time** set to `inf` when the infinite sample time propagates to the device under test (DUT) output. If there is an infinite sample rate error during HDL code generation, set **Sample time** to `-1`.

### PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

### Fixed-Point Conversion

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

### See Also

Enumerated Constant | `Simulink.Parameter` | `Simulink.BusElement`

### Topics

“Set Block Parameter Values” (Simulink)

“Specify Bus Properties with `Simulink.Bus` Object Data Types” (Simulink)

“Specify Initial Conditions for Bus Elements” (Simulink)

“Create Array of Buses from MATLAB Structures” (Simulink)

## Constant Ramp

Generate ramp signal with length based on input dimensions



**Libraries:**  
DSP System Toolbox / Signal Operations

### Description

The Constant Ramp block generates the constant ramp signal

$$y = (0:L-1)*m + b$$

where  $m$  is the slope specified by the scalar **Slope** parameter, and  $b$  is the y-intercept specified by the scalar **Offset** parameter.

For an unoriented vector input,  $L$  is equal to the length of the input vector. For an N-D input array, the length  $L$  of the output ramp is equal to the length of the input in the dimension specified by the **Ramp length equals number of** or **Dimension** parameter. The output,  $y$ , is always an unoriented vector.

### Ports

#### Input

**Port\_1** — Input signal

scalar | vector | matrix | N-D array

Input signal used to generate ramp signal, specified as a scalar, vector, matrix, or N-D array.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | Boolean | fixed point

#### Output

**Port\_1** — Output signal

vector

Output signal, specified as an unoriented vector. The block determines the length of the output based on the length of the input signal, and the **Ramp length equals number of** or **Dimension** parameter.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | Boolean | fixed point

### Parameters

#### Main

**Ramp length equals number of** — Dimension used to determine ramp length

Rows (default) | Columns | Elements in specified dimension

Specify whether the length of the output ramp is the number of rows, number of columns, or the length of the specified dimension of the input.

**Dimension** — Dimension that determines length of output ramp

1 (default) | positive integer

Specify the one-based dimension of the input array that determines the length of the output ramp as a positive integer scalar.

#### Dependencies

To enable this parameter, set **Ramp length equals number of** to `Elements in specified dimension`.

**Slope** — Slope of the ramp

1 (default) | scalar

Specify the slope of the ramp as a real-valued double-precision scalar.

**Offset** — y-intercept of the ramp

0 (default) | scalar

Specify the scalar y-intercept of the ramp as a real-valued double-precision scalar.


#### Data Types

**Output data type** — Output data type

Inherit: Same as input (default) | Inherit: Inherit via back propagation | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | boolean | fixdt(1,16) | fixdt(1,16,0) | <data type conversion>

Specify the output data type for this block. You can select one of the following:

- A rule that inherits a data type, for example, `Inherit: Same as input`.
- A built-in data type, such as `double`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16)`

For help setting data type parameters, display the **Data Type Assistant** by clicking the **Show data type assistant** button .

See “Control Data Types of Signals” (Simulink) for more information.

#### Data Type Assistant Parameters

**Mode** — Data type mode

Inherit (default) | Built in | Fixed point | Expression

Select how you would like to specify the data type properties of the **Output data type**. You can choose:

- **Inherit** — Lets you specify a rule for inheriting a data type, for example, `Inherit: Same as input`
- **Built in**— Lets you specify a built-in data type, for example, `double`
- **Fixed point** — Lets you specify the fixed-point attributes of the data type.
- **Expression** — Lets you specify an expression that evaluates to a valid data type, for example, `fixdt(1,16)`

For more information, see “Specify Data Types Using Data Type Assistant” (Simulink).

**Signedness** — Signedness of fixed-point data

`Signed (default) | Unsigned`

Specify whether the fixed-point data is signed or unsigned. **Signed** data can represent positive and negative values, but **Unsigned** data represents positive values only. For more information, see “Specify Data Types Using Data Type Assistant” (Simulink).

**Word length** — Bit size of the word that holds the quantized integer

`16 (default) | integer from 2 to 128`

Specify the bit size of the word that holds the quantized integer as an integer scalar from 2 to 128. For more information, see “Specify Data Types Using Data Type Assistant” (Simulink).

**Dependencies**

To enable this parameter, set **Mode** to `Fixed point`

**Scaling** — Method for scaling fixed-point data

`Best precision (default) | Binary point`

Specify the method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors. For more information, see “Specify Data Types Using Data Type Assistant” (Simulink).

**Dependencies**

To enable this parameter, set **Mode** to `Fixed point`

**Data type override** — Data type override mode

`Inherit (default) | Off`

Select the data type override mode for this signal. You can select:

- **Inherit** — Inherits the data type override setting specified for the model.
- **Off** — Ignores the data type override setting specified for the model and uses the fixed-point data type you specify.

**Dependencies**

To enable this parameter, set **Mode** to **Built in** or **Fixed point**. For more information, see “Specify Data Types Using Data Type Assistant” (Simulink).

**Fraction length** — Fraction length for fixed-point data type

0 (default) | integer scalar

Specify the fraction length of the fixed-point data type as a positive or negative integer scalar.

**Dependencies**

To enable this parameter, set **Mode** to **Fixed point** and **Scaling** to **Binary point**.

**Block Characteristics**

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | double   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no                                      |
| <b>Multidimensional Signals</b> | no                                      |
| <b>Variable-Size Signals</b>    | no                                      |
| <b>Zero-Crossing Detection</b>  | no                                      |

**Version History**

Introduced before R2006a

**Extended Capabilities****C/C++ Code Generation**

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

Generated code relies on the memcpy or memset function (string.h) under certain conditions.

**Fixed-Point Conversion**

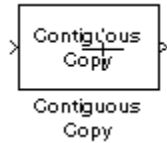
Design and simulate fixed-point systems using Fixed-Point Designer™.

**See Also****Blocks**

Create Diagonal Matrix | Constant | Ramp | Identity Matrix

## Contiguous Copy (Obsolete)

Create discontinuous input in contiguous block of memory



## Library

dspobslib

## Description

---

**Note** The Contiguous Copy block is still supported but is likely to be obsoleted in a future release.

---

The Contiguous Copy block copies the input to a contiguous block of memory, and passes this new copy to the output. The output is identical to the input, but is guaranteed to reside in a contiguous section of memory.

Because Simulink software employs an efficient copy-by-reference method for propagating data in a model, some operations produce outputs with discontinuous memory locations.

Although this does not present a problem during simulation, blocks linked to versions of DSP Blockset prior to 4.0 may require contiguous inputs for code generation with the Simulink Coder product. When such blocks are used in a model intended for code generation, they should be preceded by the Contiguous Copy block to ensure that their inputs are contiguous.

## Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point
- Boolean
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

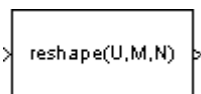
## Version History

**Introduced in R2008b**



## Convert 1-D to 2-D

Reshape 1-D or 2-D input to 2-D matrix with specified dimensions



### Libraries:

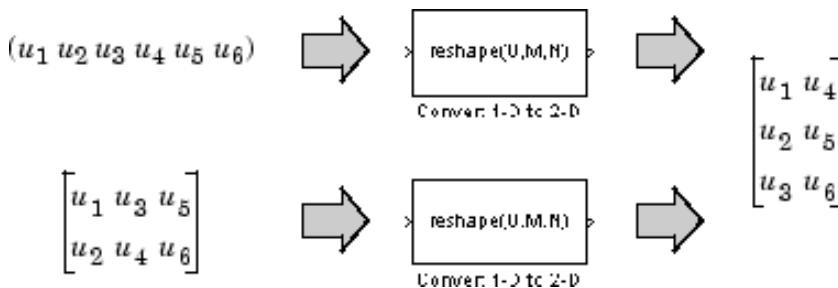
DSP System Toolbox / Signal Management / Signal Attributes  
 DSP System Toolbox HDL Support / Signal Management

### Description

The Convert 1-D to 2-D block reshapes a length- $M_i$  1-D vector or an  $M_i$ -by- $N_i$  matrix to an  $M_o$ -by- $N_o$  matrix, where  $M_o$  is specified by the **Number of output rows** parameter and  $N_o$  is specified by the **Number of output columns** parameter.

```
y = reshape(u,Mo,No) % Equivalent MATLAB code
```

The input is reshaped *column-wise*, as shown in the two cases below. The length-6 vector and the 2-by-3 matrix are both reshaped to the same 3-by-2 output matrix.



An error is generated when  $(M_o * N_o) \neq (M_i * N_i)$ . In other words, the total number of input elements must be conserved in the output.

The output is frame based when you select the **Frame-based output** check box; otherwise, the output is sample based.

### Ports

#### Input

**Port\_1** — Input signal

vector | matrix

Input signal, specified as a 1-D vector or 2-D matrix.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | Boolean | fixed point

#### Output

**Port\_1** — Output signal

matrix

Output signal, returned as a 2-D matrix.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point`

## Parameters

**Number of output rows** — Number of rows in output matrix

1 (default) | positive integer

Number of rows  $M_o$  in the output matrix, specified as a positive integer.

**Number of output columns** — Number of columns in output matrix

1 (default) | positive integer

Number of columns  $N_o$  in the output matrix, specified as a positive integer.

**Frame-based output** — Option to enable frame-based output

`off` (default) | `on`

Select this parameter to create a frame-based output.

## Block Characteristics

|                                 |  |
|---------------------------------|--|
| <b>Data Types</b>               | <code>Boolean</code>   <code>double</code>   <code>fixed point</code>   <code>integer</code>   <code>single</code> |
| <b>Direct Feedthrough</b>       | <code>no</code>  |
| <b>Multidimensional Signals</b> | <code>no</code>  |
| <b>Variable-Size Signals</b>    | <code>no</code>  |
| <b>Zero-Crossing Detection</b>  | <code>no</code>  |

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

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

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

**HDL Architecture**

This block has a pass-through implementation.

**HDL Block Properties**

|                                  |  |
|----------------------------------|--|
| <b>ConstrainedOutputPipeline</b> | Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see "ConstrainedOutputPipeline" (HDL Coder). |
| <b>InputPipeline</b>             | Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "InputPipeline" (HDL Coder).                 |
| <b>OutputPipeline</b>            | Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "OutputPipeline" (HDL Coder).               |

**Complex Data Support**

This block supports code generation for complex signals.

**Fixed-Point Conversion**

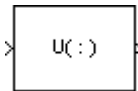
Design and simulate fixed-point systems using Fixed-Point Designer™.

**See Also**

Buffer | Convert 2-D to 1-D | Reshape | Submatrix

## Convert 2-D to 1-D

Convert 2-D matrix input to 1-D vector



### Libraries:

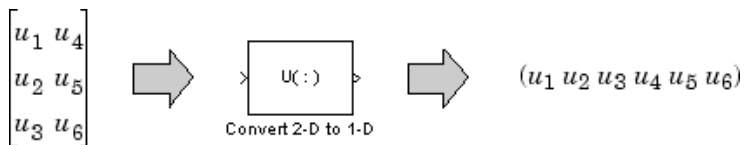
DSP System Toolbox / Signal Management / Signal Attributes

### Description

The Convert 2-D to 1-D block reshapes an  $M$ -by- $N$  matrix input to a 1-D vector that has a length of  $M*N$ .

```
y = u(:) % Equivalent MATLAB code
```

The input is reshaped column-wise for a 3-by-2 matrix.



### Ports

#### Input

**Port\_1** — Input signal

matrix

Input signal, specified as a 2-D matrix.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | Boolean | fixed point | enumerated

#### Output

**Port\_1** — Output signal

vector

Output signal, returned as a 1-D vector.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | Boolean | fixed point | enumerated

### Block Characteristics

|                           |  |
|---------------------------|--|
| <b>Data Types</b>         | Boolean   double   enumerated   fixed point   integer   single |
| <b>Direct Feedthrough</b> | no   |

|                                 |    |
|---------------------------------|----|
| <b>Multidimensional Signals</b> | no |
| <b>Variable-Size Signals</b>    | no |
| <b>Zero-Crossing Detection</b>  | no |

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

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

### Fixed-Point Conversion

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

## See Also

### Blocks

Buffer | Convert 1-D to 2-D | Reshape | Submatrix

## Convolution

Convolution of two inputs



**Libraries:**  
DSP System Toolbox / Signal Operations

### Description

The Convolution block convolves the first dimension of an  $N$ -D input array  $u$  with the first dimension of an  $N$ -D input array  $v$ . The block can also convolve a column vector with the first-dimension of an  $N$ -D input array.

The general equation for convolution is:

$$y(k) = \sum_n u(n - k)v(k)$$

Two DSP System Toolbox blocks can be used for convolving two input signals:

- Convolution
- Discrete FIR Filter

The Convolution block assumes that all elements of  $u$  and  $v$  are available at each Simulink time step and computes the entire convolution at every step.

The Discrete FIR Filter block can be used for convolving signals in situations where all elements of  $v$  is available at each time step, but  $u$  is a sequence that comes in over the life of the simulation. When you use the Discrete FIR Filter block, the convolution is computed only once.

To determine which block best fits your needs, see “Selecting the Appropriate Convolution Block” on page 2-298.

### Ports

#### Input

**Port\_1** — First input signal

scalar | vector | matrix |  $N$ -D array

First input  $u$  specified as a scalar, vector, matrix, or  $N$ -D array. When both inputs are real, the output is real. When one or both inputs are complex, the output is complex. All the input port dimensions for both the inputs, except the first dimension, must have the same value.

Inputs  $u$  and  $v$  are zero when indexed outside their valid ranges.

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

**Port\_2** — Second input signal

scalar | vector | matrix |  $N$ -D array

Second input  $v$  specified as a scalar, vector, matrix, or  $N$ -D array. When both inputs are real, the output is real. When one or both inputs are complex, the output is complex. All the input port dimensions for both the inputs, except the first dimension, must have the same value.

Inputs  $u$  and  $v$  are zero when indexed outside their valid ranges.

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

## Output

**Port\_1** — Output signal

scalar | vector | matrix |  $N$ -D array

Convolved signal, returned as a scalar, vector, matrix, or  $N$ -D array depending on the input signals. When both inputs are real, the output is real. When one or both inputs are complex, the output is complex. Inputs  $u$  and  $v$  are zero when indexed outside of their valid ranges. For more details on how the convolved signal changes based on the inputs, see “More About” on page 2-298.

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

## Parameters

### Main Tab

**Computation domain** — Computation domain

Time (default) | Frequency | Fastest

Set the domain in which the block computes convolutions:

- **Time** — The block computes in the time domain, which minimizes memory use.
- **Frequency** — The block computes in the frequency domain, which might require fewer computations than computing in the time domain, depending on the input length.
- **Fastest** — The block computes in the domain that minimizes the number of computations.

Fixed-point signals are only supported in the time domain. When inputting fixed-point signals, make sure you set the **Computation domain** parameter to **Time**.

### Data Types Tab

**Rounding mode** — Method of rounding operation

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

Specify the rounding mode for fixed-point operations as one of the following:

- Floor
- Ceiling
- Convergent
- Nearest
- Round

- Simplest
- Zero

For more details, see Rounding Modes.

---

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

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

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

---

**Saturate on integer overflow** — Method of overflow action  
off (default) | on

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

---

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

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


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

---

**Product output** — Product output data type  
Inherit: Inherit via internal rule (default) | Inherit: Same as first input |  
fixdt([],16,0)

**Product output** specifies the data type of the output of a product operation in the Convolution block.

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

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



For more information on the data type assistant, see “Specify Data Types Using Data Type Assistant” (Simulink).

For more information on the product output data type, see “Multiplication Data Types” and Fixed-Point Conversion in “Extended Capabilities” on page 2-0 .


**Accumulator** — Data type of accumulator

Inherit: `Inherit via internal rule (default)` | Inherit: Same as first input |

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

**Accumulator** specifies the data type of the output of an accumulation operation in the Convolution block.

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

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

For more information on the data type assistant, see “Specify Data Types Using Data Type Assistant” (Simulink).

For a diagrammatic representation on how to use the accumulator data type in this block, see Fixed-Point Conversion in “Extended Capabilities” on page 2-0 .


**Output** — Data type of output

Inherit: Same as accumulator (default) | Inherit: Same as input | Inherit: Same as

product output | `fixdt([],16,0)`

**Output** specifies the data type of the output of the Convolution block.

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

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

For more information, see “Control Data Types of Signals” (Simulink).

For more information on the output data type, see Fixed-Point Conversion in “Extended Capabilities” on page 2-0 .

**Output Minimum** — Minimum value that block can output  
[] (default) | scalar

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

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

**Output Maximum** — Maximum value that block can output  
[] (default) | scalar

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

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

**Lock data type settings against changes by the fixed-point tools** — Prevent fixed-point tools from overriding data types  
off (default) | on

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

## Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | double   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no                                      |
| <b>Multidimensional Signals</b> | no                                      |
| <b>Variable-Size Signals</b>    | no                                      |
| <b>Zero-Crossing Detection</b>  | no                                      |

## More About

### Selecting the Appropriate Convolution Block

| Question  | Answer  | Recommended Block(s)   |
|---|---|--|
| How many convolutions do you intend to perform? | Many convolutions, one at each time step        | <ul style="list-style-type: none"> <li>• Convolution block</li> </ul>                                      |
|   | One convolution over the life of the simulation | <ul style="list-style-type: none"> <li>• Convolution block</li> <li>• Discrete FIR Filter block</li> </ul> |

| Question                                   | Answer  | Recommended Block(s)  |
|--|---|---|
| How long are your input sequences?         | Both sequences have a finite length                     | <ul style="list-style-type: none"> <li>Convolution block</li> <li>Discrete FIR Filter block</li> </ul>                              |
|  | One sequence has an infinite (not predetermined) length | <ul style="list-style-type: none"> <li>Discrete FIR Filter block</li> </ul>   |
| How many of the inputs are scalar streams? | None  | <ul style="list-style-type: none"> <li>Convolution block</li> <li>Discrete FIR Filter block</li> </ul>                              |
|  | One or both   | <ul style="list-style-type: none"> <li>Buffer block followed by the Convolution block</li> <li>Discrete FIR Filter block</li> </ul> |

### Convoluting Two N-D Arrays

The block always computes the convolution of two N-D input arrays along the first dimension. When both inputs are N-D arrays, the size of their first dimension can differ, but the size of all other dimensions must be equal. For example, when  $u$  is an  $M_u$ -by- $N$ -by- $P$  array and  $v$  is an  $M_v$ -by- $N$ -by- $P$  array, the output is an  $(M_u+M_v-1)$ -by- $N$ -by- $P$  array.

When  $u$  is a  $M_u$ -by- $N$  matrix and  $v$  is an  $M_v$ -by- $N$  matrix, the output  $y$  is a  $(M_u+M_v-1)$ -by- $N$  matrix whose  $j$ th column has these elements

$$y_{i,j} = \sum_{k=0}^{\max(M_u, M_v) - 1} u_{k,j} v_{(i-k),j} \quad 0 \leq i \leq (M_u + M_v - 2)$$

Inputs  $u$  and  $v$  are zero when indexed outside their valid ranges. When both inputs are real, the output is real. When one or both inputs are complex, the output is complex.

### Convoluting a Column Vector with an N-D Array

When one input is a column vector and the other is an N-D array, the block independently convolves the vector with the first dimension of the N-D input array. For example, when  $u$  is a  $M_u$ -by-1 column vector and  $v$  is an  $M_v$ -by- $N$  matrix, the output is an  $(M_u+M_v-1)$ -by- $N$  matrix whose  $j$ th column has these elements:

$$y_{i,j} = \sum_{k=0}^{\max(M_u, M_v) - 1} u_k v_{(i-k),j} \quad 0 \leq i \leq (M_u + M_v - 2)$$

### Convoluting Two Column Vectors

The Convolution block also accepts two column vector inputs. When  $u$  and  $v$  are column vectors with lengths  $M_u$  and  $M_v$ , the Convolution block performs the vector convolution with lengths  $M_u$  and  $M_v$  such that:

$$y_i = \sum_{k=0}^{\max(M_u, M_v) - 1} u_k v_{(i-k)} \quad 0 \leq i \leq (M_u + M_v - 2)$$

The output is a  $(M_u+M_v-1)$ -by-1 column vector.

## Version History

Introduced before R2006a

## Extended Capabilities

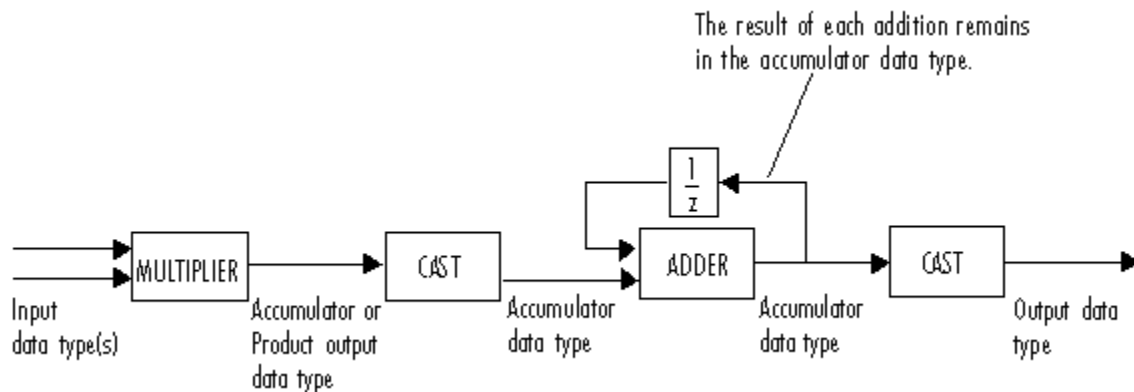
### C/C++ Code Generation

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

### Fixed-Point Conversion

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

This diagram shows the data types used within the Convolution block for fixed-point signals (time domain only).



You can set the product output, accumulator, and output data types in the block dialog as discussed in "Parameters" on page 2-295.

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

---

**Note** When one or both of the inputs are signed fixed-point signals, all internal block data types are signed fixed point. The internal block data types are unsigned fixed point only when *both* inputs are unsigned fixed-point signals.

---

## See Also

### Functions

conv

### Blocks

Correlation | Discrete FIR Filter

# Correlation

Cross-correlation of two inputs



**Libraries:**  
DSP System Toolbox / Statistics

## Description

The Correlation block computes the cross-correlation of two  $N$ -D input arrays along the first-dimension. The computation can be done in the time domain or frequency domain. You can specify the domain through the **Computation domain** parameter. In the time domain, the block convolves the first input signal,  $u$ , with the time-reversed complex conjugate of the second input signal,  $v$ . In the frequency domain, to compute the cross-correlation, the block:

- 1 Takes the Fourier transform of both input signals,  $U$  and  $V$ .
- 2 Multiplies  $U$  and  $V^*$ , where  $*$  denotes the complex conjugate.
- 3 Computes the inverse Fourier transform of the product.

If you set **Computation domain** to **Fastest**, the block chooses the domain that minimizes the number of computations. For information on these computation methods, see “Algorithms” on page 2-306.

## Ports

### Input

**Port\_1** — First data input signal  
vector | matrix |  $N$ -D array

The block accepts real-valued or complex-valued multichannel and multidimensional inputs. The input can be a fixed-point signal when you set the **Computation domain** to **Time**. When one or both of the input signals are complex, the output signal is also complex.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fixed point

**Port\_2** — Second data input signal  
vector | matrix |  $N$ -D array

The block accepts real-valued or complex-valued multichannel and multidimensional inputs. The input can be a fixed-point signal when you set the **Computation domain** to **Time**. When one or both of the input signals are complex, the output signal is also complex.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fixed point

## Output

**Port\_1** — Cross-correlated output

vector | matrix |  $N$ -D array

Cross-correlated output of the two input signals.

When the inputs are  $N$ -D arrays, the object outputs an  $N$ -D array, where all the dimensions, except for the first dimension, match with the input array. For example,

- When the inputs  $u$  and  $v$  have dimensions  $M_u$ -by- $N$ -by- $P$  and  $M_v$ -by- $N$ -by- $P$ , respectively, the Correlation block outputs an  $(M_u + M_v - 1)$ -by- $N$ -by- $P$  array.
- When the inputs  $u$  and  $v$  have the dimensions  $M_u$ -by- $N$  and  $M_v$ -by- $N$ , the block outputs an  $(M_u + M_v - 1)$ -by- $N$  matrix.

If one input is a column vector and the other input is an  $N$ -D array, the Correlation block computes the cross-correlation of the vector with each column in the  $N$ -D array. For example,

- When the input  $u$  is an  $M_u$ -by-1 column vector and  $v$  is an  $M_v$ -by- $N$  matrix, the block outputs an  $(M_u + M_v - 1)$ -by- $N$  matrix.
- Similarly, when  $u$  and  $v$  are column vectors with lengths  $M_u$  and  $M_v$ , respectively, the block performs the vector cross-correlation.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fixed point

## Parameters

### Main Tab

**Computation domain** — Domain in which the block computes the cross-correlation

Time (default) | Frequency | Fastest

- **Time** — Computes the cross-correlation in the time domain, which minimizes the memory usage.
- **Frequency** — Computes the cross-correlation in the frequency domain. For more information, see “Algorithms” on page 2-306.
- **Fastest** — Computes the cross-correlation in the domain that minimizes the number of computations.

To cross-correlate fixed-point signals, set this parameter to **Time**.

### Data Types Tab

---

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

---

**Rounding mode** — Method of rounding operation

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

Specify the rounding mode for fixed-point operations as one of the following:

- Floor
- Ceiling
- Convergent
- Nearest
- Round
- Simplest
- Zero

For more details, see rounding mode.

---

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

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

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

---

**Saturate on integer overflow** — Method of overflow action  
off (default) | on

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

---

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

- **Product output** data type is Inherit: Inherit via internal rule.
- **Accumulator** data type is Inherit: Inherit via internal rule.

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


---

**Product output** — Product output data type  
Inherit: Inherit via internal rule (default) | Inherit: Same as input |  
fixdt([],16,0)

**Product output** specifies the data type of the output of a product operation in the Correlation block. For more information on the product output data type, see “Multiplication Data Types” and the ‘Fixed-Point Conversion’ section in “Extended Capabilities” on page 2-0 .

- **Inherit: Inherit via internal rule** — The block inherits the product output data type based on an internal rule. For more information on this rule, see “Inherit via Internal Rule”.
- **Inherit: Same as input** — The block specifies the product output data type to be the same as the input data type.

- `fixdt([],16,0)` — The block specifies an autosigned, binary-point, scaled, fixed-point data type with a word length of 16 bits and a fraction length of 0.

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


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

#### **Accumulator** — Accumulator data type

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

**Accumulator** specifies the data type of output of an accumulation operation in the Correlation block. For illustrations on how to use the accumulator data type in this block, see the 'Fixed-Point Conversion' section in “Extended Capabilities” on page 2-0 .

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

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

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


#### **Output** — Output data type

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

**Output** specifies the data type of the output of the Correlation block. For more information on the output data type, see the 'Fixed-Point Conversion' section in “Extended Capabilities” on page 2-0 .

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



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

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

**Output Minimum** — Minimum value block can output  
[] (default) | scalar

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

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

**Output Maximum** — Maximum value the block can output  
[] (default) | scalar

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

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

**Lock data type settings against changes by the fixed-point tools** — Prevent fixed-point tools from overriding data types  
off (default) | on

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

## Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | double   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no                                      |
| <b>Multidimensional Signals</b> | no                                      |
| <b>Variable-Size Signals</b>    | no                                      |
| <b>Zero-Crossing Detection</b>  | no                                      |

## More About

### Cross-Correlation

Cross-correlation is the measure of similarity of two discrete-time sequences as a function of the lag of one relative to the other.

For two length- $N$  deterministic inputs or realizations of jointly wide-sense stationary (WSS) random processes,  $x$  and  $y$ , the cross-correlation is computed using the following relationship:

$$r_{xy}(h) = \begin{cases} \sum_{n=0}^{N-h-1} x(n+h)y^*(n) & 0 \leq h \leq N-1 \\ r_{yx}^*(h) & -(N-1) \leq h \leq 0 \end{cases}$$

where  $h$  is the lag and  $*$  denotes the complex conjugate. If the inputs are realizations of jointly WSS stationary random processes,  $r_{xy}(h)$  is an unnormalized estimate of the theoretical cross-correlation:

$$\rho_{xy}(h) = E\{x(n+h)y^*(n)\}$$

where  $E\{ \}$  is the expectation operator.

## Algorithms

### Time-Domain Computation

When you set the computation domain to time, the algorithm computes the cross-correlation of two signals in the time domain. The input signals can be fixed-point signals in this domain.

### Correlate Two 2-D Arrays

When the inputs are two 2-D arrays, the  $j$ th column of the output,  $y_{uv}$ , has these elements:

$$y_{uv}(i, j) = \sum_{k=0}^{\max(M_u, M_v) - 1} u_{k, j}^* v_{(k+i), j} \quad 0 \leq i < M_v$$

$$y_{uv}(i, j) = y_{vu}^*(-i, j) \quad -M_u < i < 0$$

where:

- $*$  denotes the complex conjugate.
- $u$  is an  $M_u$ -by- $N$  input matrix.
- $v$  is an  $M_v$ -by- $N$  input matrix.
- $y_{u,v}$  is an  $(M_u + M_v - 1)$ -by- $N$  matrix.

Inputs  $u$  and  $v$  are zero when indexed outside their valid ranges.

### Correlate a Column Vector with a 2-D Array

When one input is a column vector and the other input is a 2-D array, the algorithm independently cross-correlates the input vector with each column of the 2-D array. The  $j$ th column of the output,  $y_{u,v}$ , has these elements:

$$y_{uv}(i, j) = \sum_{k=0}^{\max(M_u, M_v) - 1} u_k^* v_{(k+i), j} \quad 0 \leq i < M_v$$

$$y_{uv}(i, j) = y_{vu}^*(-i, j) \quad -M_u < i < 0$$

where:

- $*$  denotes the complex conjugate.
- $u$  is an  $M_u$ -by-1 column vector.
- $v$  is an  $M_v$ -by- $N$  matrix.
- $y_{uv}$  is an  $(M_u + M_v - 1)$ -by- $N$  matrix.

Inputs  $u$  and  $v$  are zero when indexed outside their valid ranges.

### Correlate Two Column Vectors

When the inputs are two column vectors, the  $j$ th column of the output,  $y_{uv}$ , has these elements:

$$y_{uv}(i) = \sum_{k=0}^{\max(M_u, M_v) - 1} u_k^* v_{(k+i)} \quad 0 \leq i < M_v$$

$$y_{uv}(i) = y_{vu}^*(-i) \quad -M_u < i < 0$$

where:

- $*$  denotes the complex conjugate.
- $u$  is an  $M_u$ -by-1 column vector.
- $v$  is an  $M_v$ -by-1 column vector.
- $y_{uv}$  is an  $(M_u + M_v - 1)$ -by-1 column vector.

Inputs  $u$  and  $v$  are zero when indexed outside their valid ranges.

### Frequency-Domain Computation

When you set the computation domain to frequency, the algorithm computes the cross-correlation in the frequency domain.

To compute the cross-correlation, the algorithm:

- 1 Takes the Fourier transform of both input signals,  $U$  and  $V$ .
- 2 Multiplies  $U$  and  $V^*$ , where  $*$  denotes the complex conjugate.
- 3 Computes the inverse Fourier transform of the product.

In this domain, depending on the input length, the algorithm can require fewer computations.

## Version History

Introduced before R2006a

### Extended Capabilities

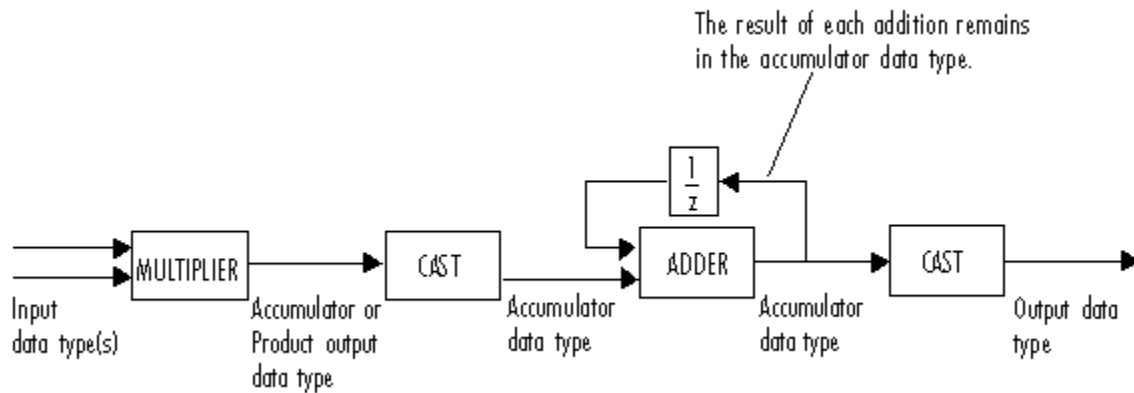
#### C/C++ Code Generation

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

#### Fixed-Point Conversion

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

The following diagram shows the data types the Correlation block uses for fixed-point signals (time domain only).



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

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

---

**Note** When one or both of the inputs are signed fixed-point signals, all internal block data types are signed fixed point. The internal block data types are unsigned fixed point only when both inputs are unsigned fixed-point signals.

---

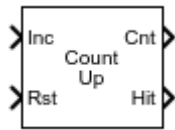
## See Also

### Blocks

Autocorrelation

# Counter

Count up or down through specified range of numbers



## Libraries:

DSP System Toolbox / Signal Management / Switches and Counters

## Description

The Counter block counts up or down through a specified range of numbers. The block enables the **Inc** (increment) port when you set the **Count direction** parameter to Up. When you set the **Count direction** parameter to Down, the block enables the **Dec** (decrement) port. If you set the **Count event** parameter to Free running, the block disables the **Inc** or **Dec** port and counts at a constant time interval. For all other settings of the **Count event** parameter, the block increments or decrements the counter each time a trigger event occurs at the **Inc** or **Dec** input port. When a trigger event occurs at the optional **Rst** port, the block resets the counter to its initial state.

The Counter block accepts single-channel inputs. For more information about scalar input operation, vector input operation, and free-running operation, see “Algorithms” on page 2-315.

## Ports

### Input

**Inc/Dec** — Input signal to trigger count event

scalar | vector

Input signal used to determine when the block increments or decrements the counter, specified as a real-valued scalar or vector. If the input to the **Inc** or **Dec** port is a vector, the block treats the vector as a frame. Each time a triggering event occurs at the **Inc** or **Dec** input port, the block increments or decrements the counter, respectively. You control the type of triggering event using the **Count event** parameter.

### Dependencies

The block enables the **Inc** (increment) port when you set the **Count direction** parameter to Up.

The block enables the **Dec** (decrement) port when you set the **Count direction** parameter to Down.

The block disables the **Inc/Dec** input port when you set the **Count event** parameter to Free running. In free running mode, the block counts at a constant time interval.

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

**Rst** — Reset signal

scalar

Input signal used to determine when the block resets the counter, specified as a real-valued scalar. The **Rst** port must have the same port sample time as the **Inc** or **Dec** input port. Each time a

triggering event occurs at the **Rst** port, the block resets the counter to its initial value. For more information about triggering events, see “Count event” on page 2-0 .

**Dependencies**

To enable this port, select the **Reset input** check box.

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

**Max** — Maximum counter size  
scalar

Specify the maximum counter size as any unsigned integer that the **Count data type** can represent. The counter values range from 0 to the value you specify as an input to the **Max** port.

**Dependencies**

To enable this port, set the **Counter size** parameter to `Specify via input port`.

Data Types: `uint8` | `uint16` | `uint32`

**Output**

**Cnt** — Current value of counter  
scalar | vector

Current value of the counter, specified as a scalar or vector. When you set the **Count event** parameter to `Free running`, the **Cnt** output is a  $M$ -by-1 vector containing the count value at each of  $M$  consecutive sample times, where  $M$  is the value you specify for the **Samples per output frame** parameter.

**Dependencies**

To enable this port, set the **Output** parameter to `Count` or `Count and Hit`.

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

**Hit** — Hit status  
scalar | vector

Hit status of the integer values you specified in the **Hit values** parameter. When a value you specify occurs in the count, the block outputs a 1 at the **Hit** port.

---

**Note** The block might output Boolean values from the **Hit** output port depending on the setting of the **Hit data type** parameter.

---

**Dependencies**

To enable this port, set the **Output** parameter to `Hit` or `Count and Hit`.

Data Types: `Boolean` | `Logical`

**Parameters**

**Count direction** — Count up or down

Up (default) | Down

Specify whether to count **Up** or **Down**. The port label on the block icon changes to **Inc** (increment) or **Dec** (decrement) based on the value of this parameter.

- When you set the **Count direction** parameter to **Up** and the counter reaches the upper limit of the counter range, the block restarts the counter at zero the next time a trigger event occurs at the **Inc** port.
- When you set the **Count direction** parameter to **Down** and the counter reaches zero, the block restarts the counter at the upper limit of the counter range the next time a trigger event occurs at the **Dec** port.

This parameter is tunable (Simulink) in Simulink normal mode.

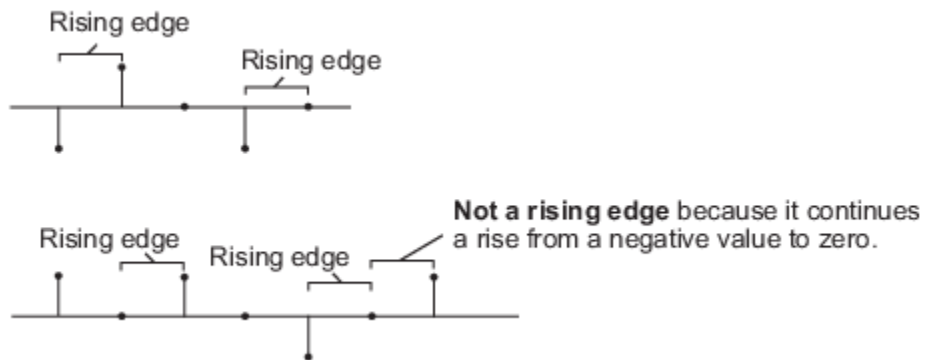
**Tunable:** Yes

**Count event** — Type of trigger event

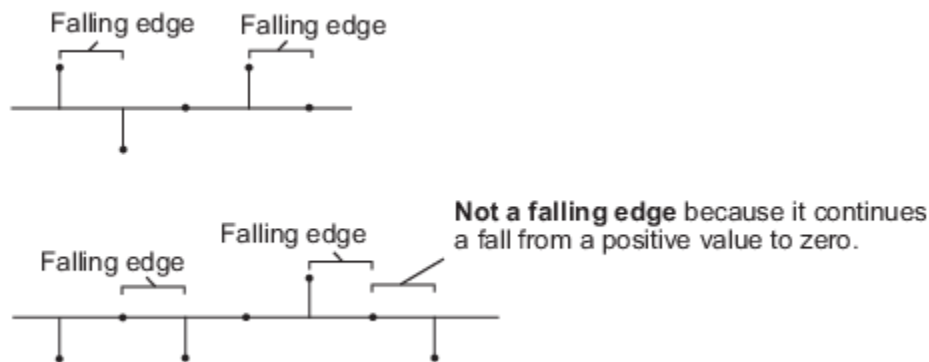
Rising edge (default) | Falling edge | Either edge | Non-zero sample | Free running

Specify the type of event that triggers the block to increment, decrement, or reset the counter when received at the **Inc/Dec** or **Rst** ports. You can select:

- **Rising edge** — Triggers a count or reset operation when the input to the **Inc/Dec** or **Rst** port behaves in one of the following ways:
  - Rises from a negative value to a positive value or zero.
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure).



- **Falling edge** — Triggers a count or reset operation when the input to the **Inc/Dec** or **Rst** port behaves in one of the following ways:
  - Falls from a positive value to a negative value or zero.
  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero (see the following figure).



- **Either edge** — Triggers a count or reset operation when the input to the **Inc/Dec** or **Rst** port is a Rising edge or Falling edge.
- **Non-zero sample** — Triggers a count or reset operation at each sample time when the input to the **Inc/Dec** or **Rst** port is not zero.
- **Free running** — Disables the **Inc/Dec** port and enables the **Samples per output frame** and **Sample time** block parameters. The block increments or decrements the counter at a constant interval,  $T_s$ , which you specify using the **Sample time** parameter. For more information, see “Free-Running Operation” on page 2-316. In this mode, the block resets the counter whenever it receives a non-zero sample at the **Rst** port.

#### Dependencies

When you set this parameter to **Free running**, the block disables the **Inc/Dec** port and counts at the constant interval specified by the **Sample time** parameter.

**Counter size** — Range of integer values to count through

8 bits (default) | 16 bits | 32 bits | User defined | Specify via input port

Specify the range of integer values the block counts through. When the block counts through the entire counter range, the next time a trigger event occurs at the **Inc/Dec** port, the block resets the counter as follows:

- When you set the **Count direction** parameter to **Up** and the counter reaches the upper limit of the counter range, the block restarts the counter at zero.
- When you set the **Count direction** parameter to **Down** and the counter reaches zero, the block restarts the counter at the upper limit of the counter range.

You can set the **Counter size** parameter to one of the following options:

- **8 bits** — Specifies a counter with a range of 0 to 255.
- **16 bits** — Specifies a counter with a range of 0 to 65535.
- **32 bits** — Specifies a counter with a range of 0 to  $2^{32}-1$ .
- **User defined** — Enables the **Maximum count** parameter, which allows you to specify the upper-count limit as any arbitrary unsigned integer that the **Count data type** can represent. The counter values range from 0 to the value of the **Maximum count** parameter.
- **Specify via input port** — Enables the **Max** input port, which allows you to specify the upper-count limit as any arbitrary unsigned integer that the **Count data type** can represent. The counter values range from 0 to the value you specify as an input to the **Max** port.



**Maximum count** — Maximum value of counter

255 (default) | positive integer

Specify the maximum value of the counter as any unsigned integer representable by the data type you specify for the **Counter data type** parameter. Tunable (Simulink) in Simulink normal mode.

**Tunable:** Yes

**Dependencies**

To enable this parameter, set the **Counter size** to `User defined`.

**Initial count** — Initial value of counter

0 (default) | integer  $\geq 0$

Specify the initial value of the counter as any unsigned integer in the range defined by the **Counter size** parameter. The block uses the initial value of the counter at the start of simulation and resets the counter back to that initial value each time a trigger event occurs at the **Rst** port.

**Tunable:** Yes

**Output** — Output count value, hit value, or both

Count (default) | Hit | Count and Hit

Select the output ports to enable. You can choose to enable the `Count`, `Hit`, or `Count and Hit` ports.

**Hit values** — Count values to flag

32 (default) | scalar | vector

Specify an integer or vector of integers whose occurrence in the count should be flagged by a 1 at the (optional) **Hit** output port. This parameter appears only when you set the **Output** parameter to `Hit` or `Count and Hit`.

**Tunable:** Yes

**Reset input** — Enable Rst input port

on (default) | off

Select this check box to enable the **Rst** input port. When you enable the **Rst** port, the block resets the counter to its initial value each time a trigger event occurs at the **Rst** port. To specify the type of event that triggers a reset of the counter, set the **Count event** parameter. When you clear the **Reset input** check box, you cannot reset the counter during simulation.

**Samples per output frame** — Number of samples in each output vector

1 (default) | positive integer

Specify the number of samples,  $M$ , in each output vector as a positive integer.

**Dependencies**

To enable this parameter, set the **Count event** to Free running.

**Sample time** — Sample time in Free Running mode

1 (default) | -1 | scalar  $\geq 0$

Specify the constant interval,  $T_s$ , at which the block increments or decrements the counter when in free-running mode. You can specify a scalar that is greater than or equal to zero, or specify a value of -1 to inherit the sample time.

For example, to have the block increment the counter every 5 seconds, set the **Count direction** parameter to Up, the **Count event** parameter to Free running, and specify a value of 5 for the **Sample time** parameter. In free running mode, the sample time of the output ports is always  $MT_s$ .

**Dependencies**

To enable this parameter, set the **Count event** to Free running.

**Count data type** — Data type of Cnt port

double (default) | single | int8 | uint8 | int16 | uint16 | int32 | uint32

Specify the data type of the output at the **Cnt** port.

**Dependencies**

To enable this parameter, set the **Output** parameter to Count or Count and Hit.

**Hit data type** — Data type of Hit port

Logical (default) | Boolean

Specify the data type of the output at the **Hit** port.

**Dependencies**

To enable this parameter, set the **Output** parameter to Hit or set it to Count and Hit with the **Count data type** parameter set to Double.

**Block Characteristics**

|                                 |                                     |
|---------------------------------|-------------------------------------|
| <b>Data Types</b>               | Boolean   double   integer   single |
| <b>Direct Feedthrough</b>       | no                                  |
| <b>Multidimensional Signals</b> | no                                  |
| <b>Variable-Size Signals</b>    | no                                  |
| <b>Zero-Crossing Detection</b>  | no                                  |

## Algorithms

### Scalar Input Operation

When you set the **Count direction** parameter to Up, a trigger event at the **Inc** (increment) input port causes the block to increase the counter by one. Assuming no reset events occur, the block continues increasing the counter value when triggered, until the counter value reaches the upper-count limit. The next time a trigger event occurs at the **Inc** port, the block restarts the counter at 0 and resumes increasing the counter by one for each subsequent trigger event at the **Inc** port.

When you set the **Count direction** parameter to Down, a trigger event at the **Dec** (decrement) input port causes the block to decrease the counter by one. Assuming no reset events occur, the block continues decreasing the counter value when triggered until the counter value reaches zero. The next time a trigger event occurs at the **Dec** port, the block restarts the counter at the upper-count limit and resumes decreasing the counter by one for each subsequent trigger event at the **Dec** port.

Between triggering events, the block holds the output at its most recent value. The block resets the counter to its initial state when the trigger event specified by the **Count event** parameter occurs at the optional **Rst** input port. When the **Inc/Dec** and **Rst** ports receive trigger events simultaneously, the block first resets the counter and then increments or decrements the counter appropriately. If you do not need to reset the counter during simulation, you can disable the **Rst** port by clearing the **Reset input** check box.

The **Output** parameter allows you to specify which values the block outputs:

- **Count** enables a **Cnt** output port on the block. The **Cnt** port provides the current value of the counter as a scalar value. The **Cnt** output port has the same port sample time as the **Inc/Dec** input port.
- **Hit** enables a **Hit** output port on the block. The **Hit** port produces zeros while the value of the counter does not equal any of the integers you specify for the **Hit values** parameter. You can specify an integer or a vector of integers for the **Hit values** parameter. When the counter value does equal one or more of the values you specify for the **Hit values** parameter, the block outputs a value of 1 at the **Hit** output port. The **Hit** output port has the same port sample time as the **Inc/Dec** input port.
- **Count** and **Hit** enables both the **Cnt** and **Hit** output ports.

### Vector Input Operation

The block treats vector inputs to the **Inc/Dec** port as a frame. Vector operation is the same as scalar operation, except that the block increments or decrements the counter by the total number of trigger events contained in the **Inc/Dec** input vector. Thus, the counter may change multiple times during the processing of a single **Inc/Dec** input vector.

When the block has a **Hit** port, the block outputs a value of 1 if any of the **Hit values** match any of the counter values during the processing of the **Inc/Dec** input vector.

When a trigger event splits across two consecutive vectors, that event is counted in the vector that contains the conclusion of the event. When the **Rst** port receives a trigger event at the same time as the **Inc/Dec** port, the block first resets the counter. The block then increments or decrements the counter by the number of trigger events contained in the **Inc/Dec** input vector.

When the input to the **Inc/Dec** port is a length  $N$  vector, the port sample time of the **Inc/Dec** input port is equal to the frame period of the input, or  $N$  times the sample time of the input signal. The port sample time of the **Cnt** and **Hit** output ports equals that of the **Inc/Dec** input port.

### Free-Running Operation

The block operates in free-running mode when you select **Free running** for the **Count event** parameter.

The **Inc/Dec** input port is disabled in this mode, and the block simply increments or decrements the counter at the constant interval,  $T_s$ , which you specify using the **Sample time** parameter.

In this mode, the **Rst** port always behaves as if the **Count event** parameter were set to **Non-zero sample**. Thus, the block triggers a reset event at each sample time that the **Rst** input is not zero.

In this mode, the **Cnt** output is an  $M$ -by-1 vector containing the count value at each of  $M$  consecutive sample times, where  $M$  is the value you specify for the **Samples per output frame** parameter. The **Hit** output is an  $M$ -by-1 vector containing the hit status (0 or 1) at each of those  $M$  consecutive sample times. Both the **Cnt** and **Hit** output ports have a port sample time of  $MT_s$ .

## Version History

Introduced before R2006a

### Extended Capabilities

#### C/C++ Code Generation

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

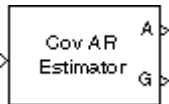
### See Also

#### Blocks

Edge Detector | N-Sample Enable | N-Sample Switch

# Covariance AR Estimator

Compute estimate of autoregressive (AR) model parameters using covariance method



## Libraries:

DSP System Toolbox / Estimation / Parametric Estimation

## Description

The Covariance AR Estimator block uses the covariance method to fit an autoregressive (AR) model to the input data. This method minimizes the forward prediction error in the least squares sense.

## Ports

### Input

#### Input — Input

column vector | unoriented vector

Specify the input data as a column vector or an unoriented vector. The block assumes that the input data is the output of an AR system driven by white noise and represents a frame of consecutive time samples from a single-channel signal.

Data Types: `single` | `double`

### Output

#### **A** — Normalized estimate of the AR model polynomial coefficients

column vector

Normalized estimate of the AR model polynomial coefficients  $A(z)$ , returned as a column vector of length  $p+1$  in descending powers of  $z$ .

The block computes the estimate of these coefficients independently for each successive input frame.

$$H(z) = \frac{G}{A(z)} = \frac{G}{1 + a(2)z^{-1} + \dots + a(p+1)z^{-p}}$$

where,

- $H(z)$  -- Transfer function of the estimated AR model
- $G$  -- Scalar gain
- $A(z)$  -- Polynomial coefficients of the AR model

Data Types: `single` | `double`

#### **G** — Model gain

scalar

Gain of the estimated AR model, returned as a scalar.

$$H(z) = \frac{G}{A(z)} = \frac{G}{1 + a(2)z^{-1} + \dots + a(p+1)z^{-p}}$$

where,

- $H(z)$  -- Transfer function of the estimated AR model
- $G$  -- Scalar gain
- $A(z)$  -- Polynomial coefficients of the AR model

Data Types: `single` | `double`

## Parameters

**Estimation order** — Order of AR model

4 (default) | positive integer

Specify the estimation order  $p$  of the all-pole AR model as a positive integer. To guarantee a nonsingular output, you must set  $p$  to be less than or equal to half the input vector length. Otherwise, the output can be singular.

## Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | <code>double</code>   <code>single</code> |
| <b>Multidimensional Signals</b> | No  |
| <b>Variable-Size Signals</b>    | No  |

## More About

### AR(p) Model

In an AR model of order  $p$ , the current output is a linear combination of the past  $p$  outputs plus a white noise input. The weights on the  $p$  past outputs minimize the mean squared prediction error of the autoregression.

Let  $y(n)$  be a wide-sense stationary random process that you obtain by filtering white noise of variance  $e$  with the system function  $A(z)$ . If  $P_y(e^{j\omega})$  is the power spectral density of  $y(n)$ , then

$$P_y(e^{j\omega}) = \frac{e}{|A(e^{j\omega})|^2} = \frac{e}{\left|1 + \sum_{k=1}^p a(k)e^{-j\omega k}\right|^2}.$$

Because the covariance method characterizes the input data using an all-pole model, the correct choice of the model order  $p$  is important.

### Compare AR Model Parameter Estimation Methods

This table compares the features of the Burg AR Estimator block to the Covariance AR Estimator, Modified Covariance AR Estimator, and the Yule-Walker AR Estimator blocks.

The Yule-Walker AR Estimator and Burg AR Estimator blocks return similar results for large frame sizes.

|                                      | <b>Burg AR Estimator</b>   | <b>Covariance AR Estimator</b>                                    | <b>Modified Covariance AR Estimator</b>   | <b>Yule-Walker AR Estimator</b>  |
|--------------------------------------|--|---|---|--|
| <b>Characteristics</b>               | Does not apply window to data  | Does not apply window to data                                     | Does not apply window to data   | Applies window to data   |
|                                      | Minimizes the forward and backward prediction errors in the least squares sense, with the AR coefficients constrained to satisfy the L-D recursion | Minimizes the forward prediction error in the least squares sense | Minimizes the forward and backward prediction errors in the least squares sense | Minimizes the forward prediction error in the least squares sense (also called "autocorrelation method")         |
| <b>Advantages</b>                    | Always produces a stable model   |   |   | Always produces a stable model   |
| <b>Disadvantages</b>                 |  | Can produce unstable models                                       | Can produce unstable models   | Performs relatively poorly for short data records  |
| <b>Conditions for Nonsingularity</b> |  | Order must be less than or equal to 1/2 the input frame size      | Order must be less than or equal to 2/3 the input frame size                    | Because of the biased estimate, the autocorrelation matrix is guaranteed to positive-definite, hence nonsingular |

## Version History

Introduced before R2006a

## References

- [1] Kay, S. M. *Modern Spectral Estimation: Theory and Application*. Englewood Cliffs, NJ: Prentice-Hall, 1988.
- [2] Marple, S. L., Jr., *Digital Spectral Analysis with Applications*. Englewood Cliffs, NJ: Prentice-Hall, 1987.

## Extended Capabilities

### C/C++ Code Generation

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

Generated code relies on the memcpy or memset function (`string.h`) under certain conditions.

## **See Also**

### **Functions**

arccov

### **Blocks**

Burg AR Estimator | Covariance Method | Modified Covariance AR Estimator | Yule-Walker AR Estimator



# Covariance Method

Power spectral density estimate using covariance method



## Libraries:

DSP System Toolbox / Estimation / Power Spectrum Estimation

## Description

The Covariance Method block estimates the power spectral density (PSD) of the input using the covariance method. This method fits an autoregressive (AR) model to the signal by minimizing the forward prediction error in the least squares sense.

The block computes the spectrum from the FFT of the estimated AR model parameters.

## Ports

### Input

**Input** — Input

column vector | unoriented vector

Specify the input as a column vector or an unoriented vector. This input represents a frame of consecutive time samples from a single-channel signal.

Data Types: `single` | `double`

### Output

**Output** — Power spectral density estimate

column vector

Power spectral density estimate of the signal at  $N_{fft}$  equally spaced frequency points, returned as a column vector. The frequency points are in the range  $[0, F_s)$ , where  $F_s$  is the sampling rate of the signal.

Data Types: `single` | `double`

## Parameters

**Estimation order** — Order of AR model

6 (default) | positive integer

Specify the estimation order of the AR model (all-pole model) as a positive integer. The block computes the spectrum from the FFT of the estimated AR model parameters. To guarantee a nonsingular output, the value of the **Estimation order** parameter must be less than or equal to half the input vector length.

**Inherit FFT length from estimation order** — Inherit FFT length from estimation order

off (default) | on

When you select the **Inherit FFT length from estimation order** parameter, the FFT length  $N_{fft}$  is one greater than the estimation order. To specify the number of points on which to perform the FFT, clear the **Inherit FFT length from estimation order** parameter. You can then specify a power-of-two FFT length using the **FFT length** parameter. The block zero-pads or wraps the input to  $N_{fft}$  before computing the FFT.

**FFT length** — FFT length

256 (default) | positive integer greater than or equal to 2

Enter the number of data points  $N_{fft}$  on which to perform the FFT as a positive integer greater than or equal to 2. When  $N_{fft}$  is larger than the input frame size, the block zero-pads each frame as needed. When  $N_{fft}$  is smaller than the input frame size, the block wraps each frame as needed.

**Dependencies**

To enable this parameter, clear the **Inherit FFT length from input dimensions** parameter.

**Inherit sample time from input** — Inherit sample time from input

on (default) | off

When you select the **Inherit sample time from input** parameter, the block computes the frequency data from the sample period of the input signal. For the block to produce a valid output, the following conditions must hold:

- The input to the block is the original signal with no samples added or deleted (by insertion of zeros, for example).
- The sample period of the time-domain signal in the simulation equals the sample period of the original time series.

If these conditions do not hold, clear the **Inherit sample time from input** parameter. You can then specify a sample time using the **Sample time of original time series** parameter.

**Sample time of original time series** — Sample time of original time-domain signal

1 (default) | positive scalar

Specify the sample time of the original time-domain signal as a positive scalar.

**Dependencies**

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

**Block Characteristics**

|                                 |                 |
|---------------------------------|-----------------|
| <b>Data Types</b>               | double   single |
| <b>Multidimensional Signals</b> | No              |

|                              |    |
|------------------------------|----|
| <b>Variable-Size Signals</b> | No |
|------------------------------|----|

## More About

### Compare Power Spectral Density Estimation Methods

The Burg Method and Yule-Walker Method blocks return similar results for large frame sizes.

This table compares the features of the Burg Method block to the Covariance Method, Modified Covariance Method, and the Yule-Walker Method blocks.

|                        | <b>Burg</b>  | <b>Covariance</b>   | <b>Modified Covariance</b>  | <b>Yule-Walker</b>   |
|------------------------|--|---|---|--|
| <b>Characteristics</b> | Does not apply window to data  | Does not apply window to data   | Does not apply window to data   | Applies window to data   |
|                        | Minimizes the forward and backward prediction errors in the least squares sense, with the AR coefficients constrained to satisfy the L-D recursion | Minimizes the forward prediction error in the least squares sense                   | Minimizes the forward and backward prediction errors in the least squares sense   | Minimizes the forward prediction error in the least squares sense (also called autocorrelation method) |
| <b>Advantages</b>      | High resolution for short data records   | Better resolution than Yule-Walker for short data records (more accurate estimates) | High resolution for short data records  | Performs as well as other methods for large data records   |
|                        | Always produces a stable model   | Able to extract frequencies from data consisting of $p$ or more pure sinusoids      | Able to extract frequencies from data consisting of $p$ or more pure sinusoids<br>Does not suffer spectral line-splitting | Always produces a stable model   |
| <b>Disadvantages</b>   | Peak locations highly dependent on initial phase   | Can produce unstable models   | Can produce unstable models   | Performs relatively poorly for short data records  |
|                        | Can suffer spectral line-splitting for sinusoids in noise, or when order is very large   | Frequency bias for estimates of sinusoids in noise                                  | Peak locations slightly dependent on initial phase  | Frequency bias for estimates of sinusoids in noise   |

|                                      | <b>Burg</b>  | <b>Covariance</b>  | <b>Modified Covariance</b>                                   | <b>Yule-Walker</b>  |
|--------------------------------------|--|--|--|---|
|                                      | Frequency bias for estimates of sinusoids in noise |  | Minor frequency bias for estimates of sinusoids in noise     |   |
| <b>Conditions for Nonsingularity</b> |  | Order must be less than or equal to 1/2 the input frame size | Order must be less than or equal to 2/3 the input frame size | Because of the biased estimate, the autocorrelation matrix is guaranteed to be positive-definite, hence nonsingular |

## Version History

Introduced before R2006a

## References

- [1] Kay, S. M. *Modern Spectral Estimation: Theory and Application*. Englewood Cliffs, NJ: Prentice-Hall, 1988.
- [2] Marple, S. L. Jr., *Digital Spectral Analysis with Applications*. Englewood Cliffs, NJ: Prentice-Hall, 1987.
- [3] Orfanidis, S. J. *Introduction to Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1995.

## Extended Capabilities

### C/C++ Code Generation

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

## See Also

Burg Method | Covariance AR Estimator | Modified Covariance Method | Short-Time FFT | Yule-Walker Method

### Topics

“Spectral Analysis”

# Cross-Spectrum Estimator

Estimate cross-power spectrum density



## Library

Estimation / Power Spectrum Estimation

dspsect3

## Description

The Cross-Spectrum Estimator block outputs the frequency cross-power spectrum density of two real or complex input signals,  $x$  and  $y$ , via Welch's method of averaged modified periodograms. The input signals must be of the same size and data type.

The Cross-Spectrum Estimator block computes the current power spectrum estimate by averaging the last  $N$  power spectrum estimates, where  $N$  is the number of spectral averages defined in **Number of spectral averages**. The block buffers the input data into overlapping segments. You can set the length of the data segment and the amount of data overlap through the parameters set in the block dialog box. The block computes the power spectrum based on the parameters set in the block dialog box.

Each column of the input signal is treated as a separate channel. If the input is a two-dimensional signal, the first dimension represents the channel length (or frame size) and the second dimension represents the number of channels. If the input is a one-dimensional signal, then it is interpreted as a single channel.

## Parameters

### Window length source

Source of the window length value. You can set this parameter to:

- Same as input frame length (default) — Window length is set to the frame size of the input.
- Specify on dialog — Window length is the value specified in **Window length**.

This parameter is nontunable.

### Window length

Length of the window, in samples, used to compute the spectrum estimate, specified as a positive integer scalar greater than 2. This parameter applies when you set **Window length source** to **Specify on dialog**. The default is 1024. This parameter is nontunable.

**Window Overlap (%)**

Percentage of overlap between successive data windows, specified as a scalar in the range [0, 100). The default is 0. This parameter is nontunable.

**Averaging method**

Specify the averaging method as `Running` or `Exponential`. In the running averaging method, the block computes an equally weighted average of a specified number of spectrum estimates defined by the **Number of spectral averages** parameter. In the exponential method, the block computes the average over samples weighted by an exponentially decaying forgetting factor.

**Number of spectral averages**

Number of spectral averages, specified as a positive integer scalar. The default is 1. The spectrum estimator computes the current power spectrum estimate by averaging the last  $N$  power spectrum estimates, where  $N$  is the number of spectral averages defined in **Number of spectral averages**. This parameter is nontunable.

This parameter applies when **Averaging method** is set to `Running`.

**Specify forgetting factor from input port**

Select this check box to specify the forgetting factor from an input port. When you do not select this check box, the forgetting factor is specified through the **Forgetting factor** parameter.

This parameter applies when **Averaging method** is set to `Exponential`.

**Forgetting factor**

Specify the exponential weighting forgetting factor as a scalar value greater than zero and smaller than or equal to one. The default is 0.9.

This parameter applies when you set **Averaging method** to `Exponential` and clear the **Specify forgetting factor from input port** parameter.

**FFT length source**

Source of the FFT length value. You can set this parameter to:

- `Auto` (default) — FFT length is set to the frame size of the input.
- `Property` — FFT length is the value specified in **FFT length**.

This parameter is nontunable.

**FFT length**

Length of the FFT used to compute the spectrum estimates, specified as a positive integer scalar. This parameter applies when you set **FFT length source** to `Property`. The default is 1024. This parameter is nontunable.

**Window function**

Window function for the cross-spectrum estimator, specified as one of `Chebyshev` | `Flat Top` | `Hamming` | `Hann` | `Kaiser` | `Rectangular`. The default is `Hann`. This parameter is nontunable.

**Sidelobe attenuation of window (dB)**

Side lobe attenuation of the window, specified as real positive scalar. This parameter applies when you set **Window function** to `Chebyshev` or `Kaiser`. The default is 60. This parameter is nontunable.

**Frequency range**

Frequency range of the cross-spectrum estimator. You can set this parameter to:

- **centered** (default) — The cross-spectrum estimator computes the centered two-sided spectrum of complex or real input signals,  $x$  and  $y$ . The length of the cross-spectrum estimate is equal to the FFT length. The spectrum estimate is computed over the frequency range  $[-\text{SampleRate}/2 \text{ SampleRate}/2]$  when the FFT length is even and  $[-\text{SampleRate}/2 \text{ SampleRate}/2]$  when FFT length is odd.
- **onesided** — The cross-spectrum estimator computes the one-sided spectrum of real input signals,  $x$  and  $y$ . When the FFT length,  $NFFT$  is even, length of the cross-spectrum estimate is  $(NFFT/2) + 1$ , and is computed over the frequency range  $[0 \text{ SampleRate}/2]$ . When the FFT length,  $NFFT$  is odd, length of the cross-spectrum estimate is  $(NFFT + 1)/2$ , and is computed over the frequency range  $[0 \text{ SampleRate}/2]$ .
- **twosided** — The cross-spectrum estimator computes the two-sided spectrum of complex or real input signals,  $x$  and  $y$ . The length of the cross-spectrum estimate is equal to the FFT length. The spectrum estimate is computed over the frequency range  $[0 \text{ SampleRate}]$ , where  $\text{SampleRate}$  is the sample rate of the input signal.

This parameter is nontunable.

### Inherit sample rate from input

When you select this check box, the block's sample rate is computed as  $N/T_s$ , where  $N$  is the frame size of the input signal, and  $T_s$  is the sample time of the input signal. When you clear this check box, the block sample rate is the value specified in **Sample rate (Hz)**. By default, this check box is selected.

### Sample rate (Hz)

Sample rate of the input signal, specified as a positive scalar value. The default is 44100. This parameter applies when you clear the **Inherit sample rate from input** check box. This parameter is nontunable.

### Simulate using

Type of simulation to run. You can set this parameter to:

- **Code generation** (default)

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

- **Interpreted execution**

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

## Supported Data Types

| Port   | Supported Data Types   |
|--------|--|
| Input  | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> </ul> |
| Output | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> </ul> |

## Algorithms

### Welch's Method of Averaged Modified Periodograms

Give two signal inputs,  $x$  and  $y$ :

- 1 Multiply the inputs by the window and scale the result by the window power.
- 2 Compute FFT of the signals,  $X$  and  $Y$ , and multiply  $X$  with  $\text{conj}(Y)$  using  $Z = X.\text{*conj}(Y)$ .
- 3 Compute the current cross power spectrum estimate by taking the moving average of the last  $N$  number of  $Z$ 's and scaling the answer by the sample rate.

For further information on the algorithms, refer to the "Averaging Method" on page 2-1343 section in Spectrum Analyzer.

## Version History

Introduced in R2015a

## References

- [1] Hayes, Monson H. *Statistical Digital Signal Processing and Modeling*. Hoboken, NJ: John Wiley & Sons, 1996.
- [2] Kay, Steven M. *Modern Spectral Estimation: Theory and Application*. Englewood Cliffs, NJ: Prentice Hall, 1999.
- [3] Stoica, Petre, and Randolph L. Moses. *Spectral Analysis of Signals*. Englewood Cliffs, NJ: Prentice Hall, 2005.
- [4] Welch, P. D. "The Use of Fast Fourier Transform for the Estimation of Power Spectra: A Method Based on Time Averaging Over Short Modified Periodograms". *IEEE Transactions on Audio and Electroacoustics*. Vol. 15, No. 2, June 1967, pp. 70-73.

## Extended Capabilities

### C/C++ Code Generation

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

## See Also

### Objects

`dsp.CrossSpectrumEstimator`

### Blocks

Discrete Transfer Function Estimator | Periodogram | Spectrum Analyzer



# Cumulative Product

Cumulative product of channel, column, or row elements



## Library

Math Functions / Math Operations

dspmathops

## Description

The Cumulative Product block computes the cumulative product along the specified dimension of the input or across time (running product).

The input can be a vector or matrix.

### Input and Output Characteristics

#### Valid Input

The Cumulative Product block accepts vector or matrix inputs containing real or complex values.

#### Valid Reset Signal

The optional reset port, `Rst`, accepts scalar values, which can be any built-in Simulink data type including `boolean`. The rate of the input to the `Rst` port must be the same or slower than that of the input data signal. The sample time of the input to the `Rst` port must be a positive integer multiple of the input sample time.

### Computing the Running Product Along Channels of the Input

When you set the **Multiply input along** parameter to `Channels (running product)`, the block computes the cumulative product of the elements in each input channel. The running product of the current input takes into account the running product of all previous inputs. In this mode, you must also specify a value for the **Input processing** parameter. When you set the **Input processing** parameter to `Columns as channels (frame based)`, the block computes the running product along each column of the current input. When you set the **Input processing** parameter to `Elements as channels (sample based)`, the block computes a running product for each element of the input across time. See the following sections for more information:

- “Computing the Running Product for Each Column of the Input” on page 2-330
- “Computing the Running Product for Each Element of the Input” on page 2-330
- “Resetting the Running Product” on page 2-331

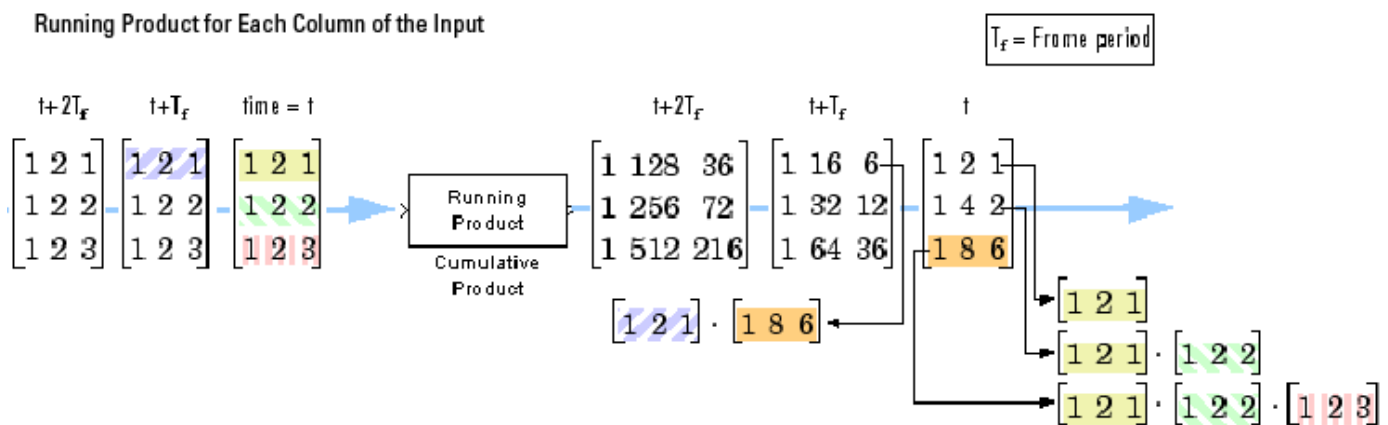
### Computing the Running Product for Each Column of the Input

When you set the **Input processing** parameter to **Columns as channels (frame based)**, the block treats each input column as an independent channel. As the following figure and equation illustrate, the output has the following characteristics:

- The first row of the first output is the same as the first row of the first input.
- The first row of each subsequent output is the element-wise product of the first row of the current input (time  $t$ ), and the last row of the previous output (time  $t - T_f$ , where  $T_f$  is the frame period).
- The output has the same size, dimension, data type, and complexity as the input.

Given an  $M$ -by- $N$  matrix input,  $u$ , the output,  $y$ , is an  $M$ -by- $N$  matrix whose first row has elements

$$y_{1,j}(t) = u_{1,j}(t) \cdot y_{M,j}(t - T_f)$$



### Computing the Running Product for Each Element of the Input

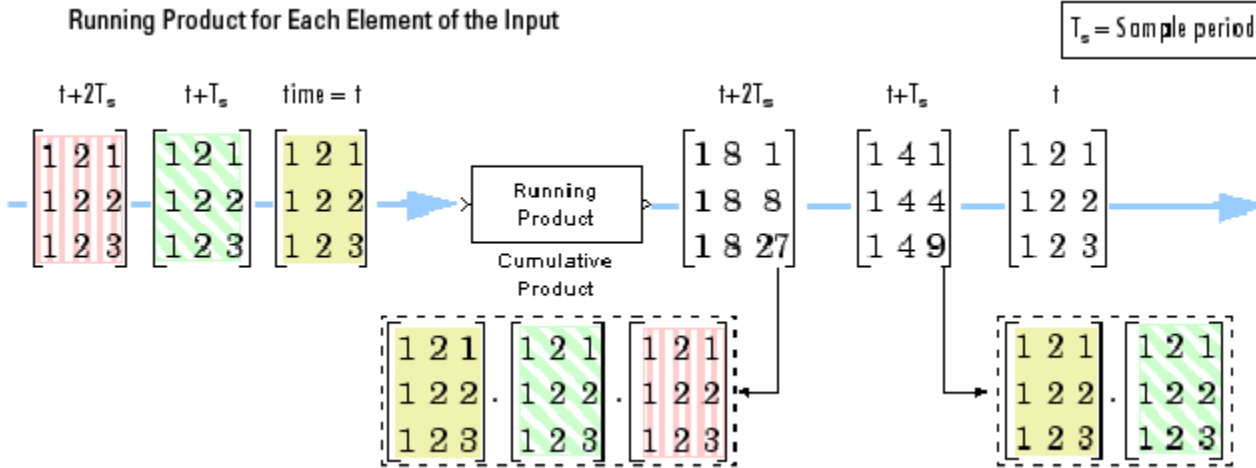
When you set the **Input processing** parameter to **Elements as channels (sample based)**, the block treats each element of the input matrix as an independent channel. As the following figure and equation illustrate, the output has the following characteristics:

- The first output is the same as the first input.
- Each subsequent output is the element-wise product of the current input (time  $t$ ) and the previous output (time  $t - T_s$ , where  $T_s$  is the sample period).
- The output has the same size, dimension, data type, and complexity as the input.

Given an  $M$ -by- $N$  matrix input,  $u$ , the output,  $y$ , is an  $M$ -by- $N$  matrix with the elements

$$y_{i,j}(t) = u_{i,j}(t) \cdot y_{i,j}(t - T_s) \quad \begin{matrix} 1 \leq i \leq M \\ 1 \leq j \leq N \end{matrix}$$

For convenience, the block treats length- $M$  unoriented vector inputs as  $M$ -by-1 column vectors when multiplying along channels. In such cases, the output is a length- $M$  unoriented vector.



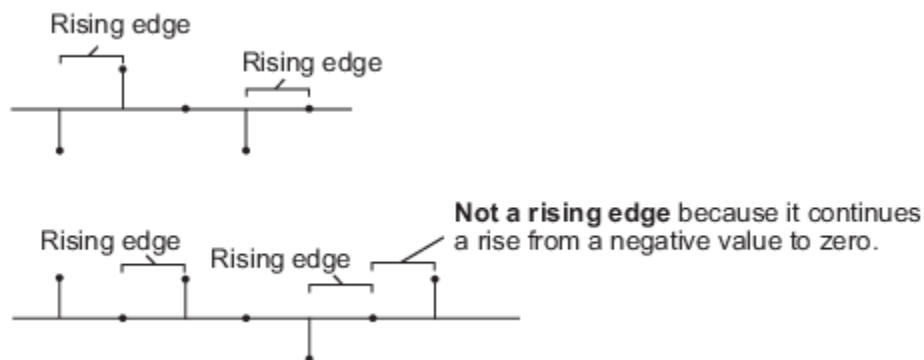
**Resetting the Running Product**

When you are computing the running product, you can configure the block to reset the running product whenever it detects a reset event at the optional Rst port. The rate of the input to the Rst port must be the same or slower than that of the input data signal. The sample time of the input to the Rst port must be a positive integer multiple of the input sample time. The input to the Rst port can be of the Boolean data type.

If a reset event occurs while the block is performing sample-based processing, the block initializes the current output to the values of the current input. If a reset event occurs while the block is performing frame-based processing, the block initializes the first row of the current output to the values in the first row of the current input.

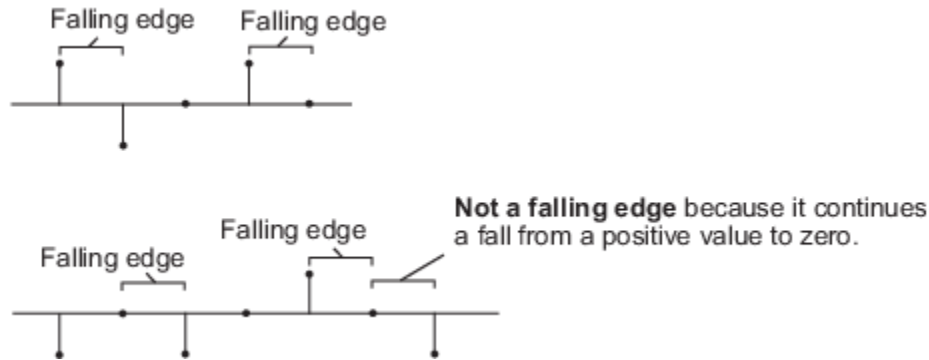
The **Reset port** parameter specifies the reset event, which can be one of the following:

- None disables the Rst port.
- Rising edge — Triggers a reset operation when the Rst input does one of the following:
  - Rises from a negative value to a positive value or zero
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure)



- Falling edge — Triggers a reset operation when the Rst input does one of the following:

- Falls from a positive value to a negative value or zero
- Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero (see the following figure)



- **Either edge** — Triggers a reset operation when the Rst input is a Rising edge or Falling edge (as described above)
- **Non-zero sample** — Triggers a reset operation at each sample time that the Rst input is not zero

---

**Note** When you run simulations in Simulink MultiTasking mode, reset signals have a one-sample latency. When the block detects a reset event, a one-sample delay occurs at the reset port rate before the block applies the reset. For more information on latency and the Simulink tasking modes, see “Excess Algorithmic Delay (Tasking Latency)” and “Time-Based Scheduling and Code Generation” (Simulink Coder).

---

### Multiplying Along Columns

When you set the **Multiply input along** parameter to **Columns**, the block computes the cumulative product of each column of the input. In this mode, the current cumulative product is independent of the cumulative products of previous inputs.

```
y = cumprod(u) % Equivalent MATLAB code
```

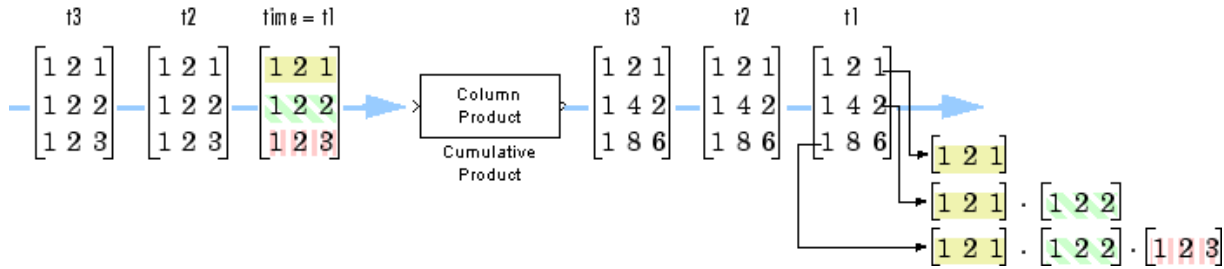
The output has the same size, dimension, data type, and complexity as the input. The  $m$ th output row is the element-wise product of the first  $m$  input rows.

Given an  $M$ -by- $N$  input,  $u$ , the output,  $y$ , is an  $M$ -by- $N$  matrix whose  $j$ th column has elements

$$y_{i,j} = \prod_{k=1}^i u_{k,j} \quad 1 \leq i \leq M$$

When multiplying along columns, the block treats length- $M$  unoriented vector inputs as  $M$ -by-1 column vectors.

**Product Along Columns**



**Multiplying Along Rows**

When you set the **Multiply input along** parameter to Rows, the block computes the cumulative product of the row elements. In this mode, the current cumulative product is independent of the cumulative products of previous inputs.

```
y = cumprod(u,2) % Equivalent MATLAB code
```

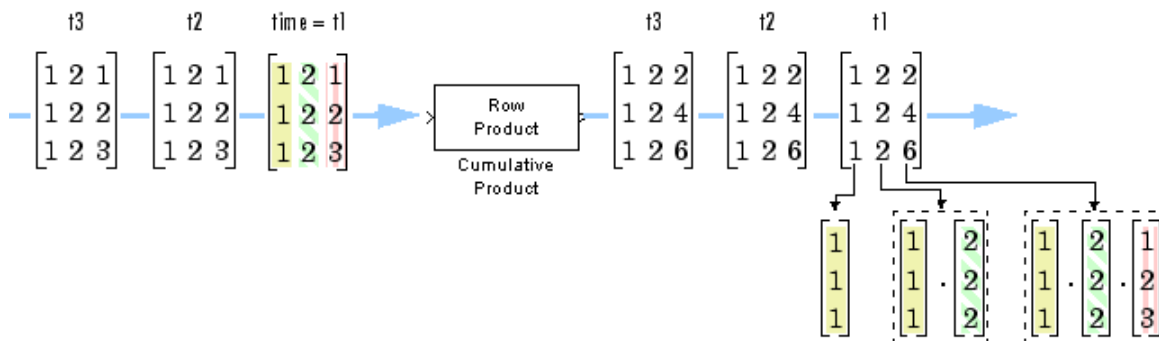
The output has the same size, dimension, and data type as the input. The *n*th output column is the element-wise product of the first *n* input columns.

Given an *M*-by-*N* matrix input, *u*, the output, *y*, is an *M*-by-*N* matrix whose *i*th row has elements

$$y_{i,j} = \prod_{k=1}^j u_{i,k} \quad 1 \leq j \leq N$$

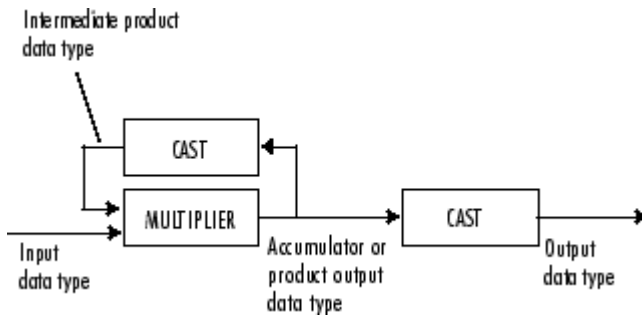
When you multiply along rows, the block treats length-*N* unoriented vector inputs as 1-by-*N* row vectors.

**Product Along Rows**



**Fixed-Point Data Types**

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



The output of the multiplier is in the product output data type when at least one of the inputs to the multiplier is real. When both of the inputs to the multiplier are complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication performed, see “Multiplication Data Types”. You can set the accumulator, product output, intermediate product, and output data types in the block dialog as discussed in “Parameters” on page 2-334.

## Parameters

### Main Tab

#### Multiply input along

Specify the dimension along which to compute the cumulative product. You can choose to multiply along Channels (running product), Columns, or Rows. For more information, see the following sections:

- “Computing the Running Product Along Channels of the Input” on page 2-329
- “Multiplying Along Columns” on page 2-332
- “Multiplying Along Rows” on page 2-333

#### Input processing

Specify how the block should process the input when computing the running product along the channels of the input. You can set this parameter to one of the following options:

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

This parameter is available only when you set the **Multiply input along** parameter to Channels (running product).

#### Reset port

Determines the reset event that causes the block to reset the product along channels. The rate of the input to the Rst port must be the same or slower than that of the input data signal. The sample time of the input to the Rst port must be a positive integer multiple of the input sample time. This parameter appears only when you set the **Multiply input along** parameter to Channels (running product). For more information, see “Resetting the Running Product” on page 2-331.

#### Data Types Tab

---

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

---

### Rounding mode

Specify the rounding mode for fixed-point operations as one of the following:

- Floor
- Ceiling
- Convergent
- Nearest
- Round
- Simplest
- Zero

For more details, see rounding mode.


### Saturate on integer overflow

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

### Intermediate product

Specify the intermediate product data type. As shown in “Fixed-Point Data Types” on page 2-333, the output of the multiplier is cast to the intermediate product data type before the next element of the input is multiplied into it. You can set it to:

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


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

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

### Product output

Specify the product output data type. See “Fixed-Point Data Types” on page 2-333 and “Multiplication Data Types” for illustrations depicting the use of the product output data type in this block. You can set it to:

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


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

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

### Accumulator

Specify the accumulator data type. See “Fixed-Point Data Types” on page 2-333 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

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

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


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

### Output

Specify the output data type. See “Fixed-Point Data Types” on page 2-333 for illustrations depicting the use of the output data type in this block. You can set it to:

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

If both inputs are unsigned, all data types including the output data type is unsigned. If one of the inputs is signed, internal and output data types are signed.

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

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

### Minimum

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

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

### Maximum

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

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

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

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



## Intermediate Product Data Type Assistant Parameters

### Mode

Select how you would like to specify the data type properties of the **Intermediate product** data type. You can choose:

- **Inherit** — Lets you specify a rule for inheriting a data type, for example, `Inherit: Inherit via internal rule`
- **Fixed point** — Lets you specify the fixed-point attributes of the data type.
- **Expression** — Lets you specify an expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

### Signedness

Specify the **Signedness** for the **Intermediate product** data type.

### Scaling

Specify the **Scaling** for the **Intermediate product** data type.

For more information see “Scaling” in the DSP System Toolbox User's Guide.

### Word length

Specify the **Word length** for the **Intermediate product** data type.

### Fraction length

Specify the **Fraction length** for the **Intermediate product** data type.

### Data type override

Specify the data type override mode. You can select one of the following options:

- **Inherit** — Inherits the data type override setting specified for the model.
- **Off** — Ignores the data type override setting specified for the model and uses the fixed-point data type you specify

This parameter appears only when you set the **Mode** parameter to **Built in** or **Fixed Point**. For more information, see “Specify Data Types Using Data Type Assistant” (Simulink).

## Product Output Data Type Assistant Parameters

### Mode

Select how you would like to specify the data type properties of the **Product output** data type. You can choose:

- **Inherit** — Lets you specify a rule for inheriting a data type, for example, `Inherit: Inherit via internal rule`

- **Fixed point** — Lets you specify the fixed-point attributes of the data type.
- **Expression** — Lets you specify an expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

**Signedness**

Specify the **Signedness** for the **Product output** data type.

**Scaling**

Specify the **Scaling** for the **Product output** data type.

For more information see “Scaling” in the DSP System Toolbox User's Guide.

**Word length**

Specify the **Word length** for the **Product output** data type.

**Fraction length**

Specify the **Fraction length** for the **Product output** data type.

**Data type override**

Specify the data type override mode. You can select one of the following options:

- **Inherit** — Inherits the data type override setting specified for the model.
- **Off** — Ignores the data type override setting specified for the model and uses the fixed-point data type you specify

This parameter appears only when you set the **Mode** parameter to **Built in** or **Fixed Point**. For more information, see “Specify Data Types Using Data Type Assistant” (Simulink).

**Accumulator Data Type Assistant Parameters****Mode**

Select how you would like to specify the data type properties of the **Accumulator** data type. You can choose:

- **Inherit** — Lets you specify a rule for inheriting a data type, for example, `Inherit: Inherit via internal rule`
- **Fixed point** — Lets you specify the fixed-point attributes of the data type.
- **Expression** — Lets you specify an expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

**Signedness**

Specify the **Signedness** for the **Accumulator** data type.

## Scaling

Specify the **Scaling** for the **Accumulator** data type.

For more information see “Scaling” in the DSP System Toolbox User's Guide.

## Word length

Specify the **Word length** for the **Accumulator** data type.

## Fraction length

Specify the **Fraction length** for the **Accumulator** data type.

## Data type override

Specify the data type override mode. You can select one of the following options:

- **Inherit** — Inherits the data type override setting specified for the model.
- **Off** — Ignores the data type override setting specified for the model and uses the fixed-point data type you specify

This parameter appears only when you set the **Mode** parameter to **Built in** or **Fixed Point**. For more information, see “Specify Data Types Using Data Type Assistant” (Simulink).

## Output Data Type Assistant Parameters

### Mode

Select how you would like to specify the data type properties of the **Output** data type. You can choose:

- **Inherit** — Lets you specify a rule for inheriting a data type, for example, **Inherit via internal rule**
- **Fixed point** — Lets you specify the fixed-point attributes of the data type.
- **Expression** — Lets you specify an expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

### Signedness

Specify the **Signedness** for the **Output** data type.

### Scaling

Specify the **Scaling** for the **Output** data type.

For more information see “Scaling” in the DSP System Toolbox User's Guide.

### Word length

Specify the **Word length** for the **Output** data type.

#### Fraction length

Specify the **Fraction length** for the **Output** data type.

#### Data type override

Specify the data type override mode. You can select one of the following options:

- **Inherit** — Inherits the data type override setting specified for the model.
- **Off** — Ignores the data type override setting specified for the model and uses the fixed-point data type you specify

This parameter appears only when you set the **Mode** parameter to **Built in** or **Fixed Point**. For more information, see “Specify Data Types Using Data Type Assistant” (Simulink).

## Supported Data Types

| Input and Output Ports       | Supported Data Types  |
|------------------------------|---|
| Data input port, <b>In</b>   | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul>                               |
| Reset input port, <b>Rst</b> | All built-in Simulink data types: <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul> |
| Output port                  | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul>                               |

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

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

Generated code relies on the memcpy or memset function (`string.h`) under certain conditions.

### Fixed-Point Conversion

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

## See Also

### Functions

`cumprod`

### Blocks

Cumulative Sum | Matrix Product

## Cumulative Sum

Cumulative sum of channel, column, or row elements



### Libraries:

DSP System Toolbox / Math Functions / Math Operations

## Description

The Cumulative Sum block computes the cumulative sum along the specified dimension of the input or across time (running sum).

## Ports

### Input

**In** — Input signal  
vector | matrix

Input, specified as a vector or as matrix inputs containing real or complex values.

This port is unnamed until you select a non-None value for the **Reset** port parameter.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fixed point`

**Rst** — Reset port  
scalar

The optional reset port, **Rst**, accepts scalar values, which can be any built-in Simulink data type including `boolean`. The rate of the input to the **Rst** port must be the same or slower than that of the input data signal. The sample time of the input to the **Rst** port must be a positive integer multiple of the input sample time.

This port is unnamed until you select a non-None value for the **Reset** port parameter.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean`

### Output

**Port\_1** — Output signal  
vector | matrix

Cumulative sum of input, specified as a vector or a matrix.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fixed point`

## Parameters

### Main

**Sum input along** — Dimension to sum along

Channels (running sum) (default) | Columns | Rows

Specify the dimension along which to compute the cumulative summations. You can choose to sum along Channels (running sum), Columns, or Rows. For more information, see these sections:

- “Computing the Running Sum Along Channels of the Input” on page 2-346
- “Summing Along Columns” on page 2-348
- “Summing Along Rows” on page 2-349

**Input processing** — Method to process the input

Columns as channels (frame based) (default) | Elements as channels (sample based)

Specify how the block processes the input when computing the running sum along the channels of the input. You can set this parameter to one of these options:

- Columns as channels (frame based) — The block treats each column of the input as a separate channel.
- Elements as channels (sample based) — The block treats each element of the input as a separate channel.

### Dependencies

This parameter is available only when you set the Sum input along parameter to Channels (running sum).

**Reset port** — Reset type

None (default) | Rising edge | Falling edge | Either edge | Non-zero sample

Determines the reset event that causes the block to reset the sum along channels. The rate of the input to the **Rst** port must be the same or slower than that of the input data signal. The sample time of the input to the **Rst** port must be a positive integer multiple of the input sample time. For more information, see “Resetting the Running Sum” on page 2-347.

### Dependencies

This parameter is available only when you set the Sum input along parameter to Channels (running sum).

### Data Types

---

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

---

**Rounding mode** — Rounding mode

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

Specify the rounding mode for fixed-point operations as one of the following:

- Floor
- Ceiling
- Convergent
- Nearest
- Round
- Simplest
- Zero

For more details, see rounding mode.

**Saturate on integer overflow** — Saturate for fixed-point operation

off (default) | on

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

**Accumulator** — Data type of accumulator

Inherit: Same as first input (default) | fixdt([],16,0)

**Accumulator** specifies the data type of the output of an accumulation operation in the Cumulative Sum block. For illustrations on how to use the accumulator data type in this block, see the 'Fixed-Point Conversion' section in “Extended Capabilities” on page 2-0 .

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

Alternatively, you can set the **Accumulator** data type by using the **Data Type Assistant**. Click the

**Show data type assistant** button  .

For more information, see “Specify Data Types Using Data Type Assistant” (Simulink) in *Simulink User's Guide* (Simulink).


**Output** — Data type of output

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

**Output** specifies the data type of the output of the Cumulative Sum block. For more information on the output data type, see the 'Fixed-Point Conversion' section in “Extended Capabilities” on page 2-0 .



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

Alternatively, you can set the **Output** data type by using the **Data Type Assistant**. Click the **Show data type assistant** button .

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

**Output Minimum** — Minimum value the block can output  
`[]` (default) | scalar

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

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

**Output Maximum** — Maximum value block can output  
`[]` (default) | scalar

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

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

**Lock data type settings against changes by the fixed-point tools** — Prevent fixed-point tools from overriding data types  
`off` (default) | `on`

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

## Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | <code>double</code>   <code>fixed point</code>   <code>integer</code>   <code>single</code> |
| <b>Direct Feedthrough</b>       | <code>no</code>   |
| <b>Multidimensional Signals</b> | <code>no</code>   |
| <b>Variable-Size Signals</b>    | <code>no</code>   |
| <b>Zero-Crossing Detection</b>  | <code>no</code>   |

## More About

### Computing the Running Sum Along Channels of the Input

When you set the `Sum input along` parameter to `Channels (running sum)`, the block computes the cumulative sum of the elements in each input channel. The running sum of the current input takes into account the running sum of all previous inputs. In this mode, you must also specify a value for the `Input processing` parameter.

When you set the **Input processing** parameter to:

- `Columns as channels (frame based)` -- The block computes the running sum along each column of the current input.
- `Elements as channels (sample based)` -- The block computes a running sum for each element of the input across time.

### Computing the Running Sum for Each Column of the Input

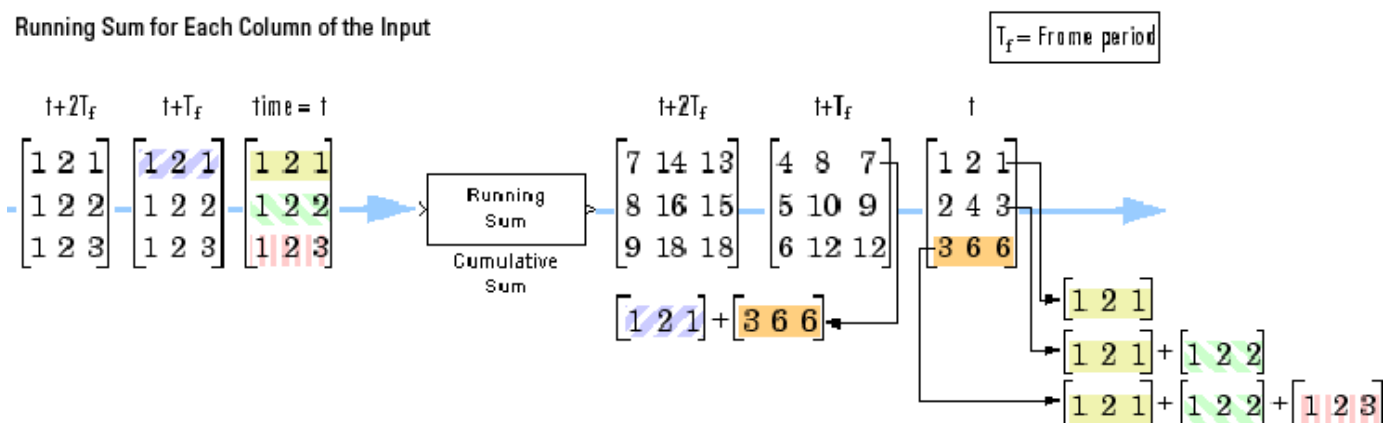
When you set the **Input processing** parameter to `Columns as channels (frame based)`, the block treats each input column as an independent channel. As the following figure and equation illustrate, the output has the following characteristics:

- The first row of the first output is the same as the first row of the first input.
- The first row of each subsequent output is the sum of the first row of the current input (time  $t$ ), and the last row of the previous output (time  $t - T_f$ , where  $T_f$  is the frame period).
- The output has the same size, dimension, data type, and complexity as the input.

Given an  $M$ -by- $N$  matrix input,  $u$ , the output,  $y$ , is an  $M$ -by- $N$  matrix whose first row has elements

$$y_{1,j}(t) = u_{1,j}(t) + y_{M,j}(t - T_f)$$

Running Sum for Each Column of the Input



### Computing the Running Sum for Each Element of the Input

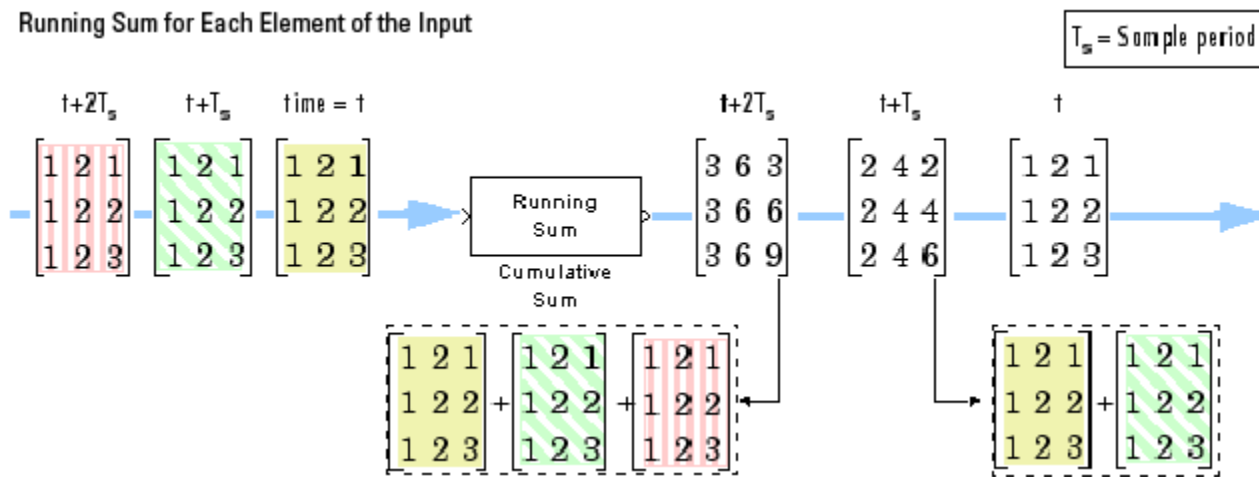
When you set the **Input processing** parameter to `Elements as channels (sample based)`, the block treats each element of the input matrix as an independent channel. As the following figure and equation illustrate, the output has these characteristics:

- The first output is the same as the first input.
- Each subsequent output is the sum of the current input (time  $t$ ) and the previous output (time  $t - T_s$ , where  $T_s$  is the sample period).
- The output has the same size, dimension, data type, and complexity as the input.

Given an  $M$ -by- $N$  matrix input,  $u$ , the output,  $y$ , is an  $M$ -by- $N$  matrix with the elements

$$y_{i,j}(t) = u_{i,j}(t) + y_{i,j}(t - T_s) \quad \begin{array}{l} 1 \leq i \leq M \\ 1 \leq j \leq N \end{array}$$

### Running Sum for Each Element of the Input



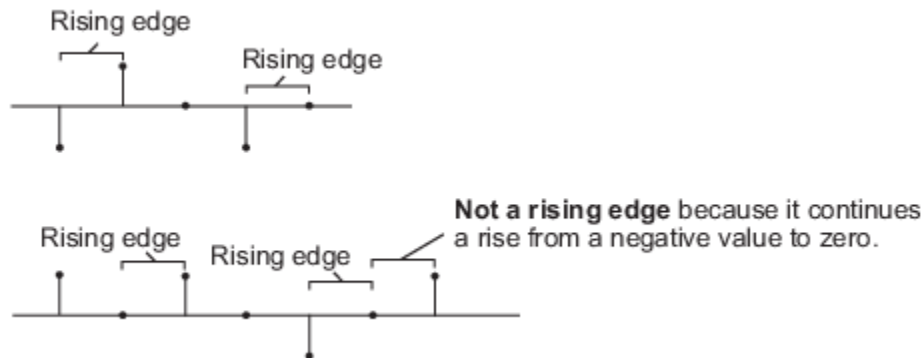
### Resetting the Running Sum

When you are computing the running sum, you can configure the block to reset the running sum whenever it detects a reset event at the optional **Rst** port. The rate of the input to the **Rst** port must be the same or slower than that of the input data signal. The sample time of the input to the **Rst** port must be a positive integer multiple of the input sample time. The reset sample time must be a positive integer multiple of the input sample time. The input to the **Rst** port can be boolean.

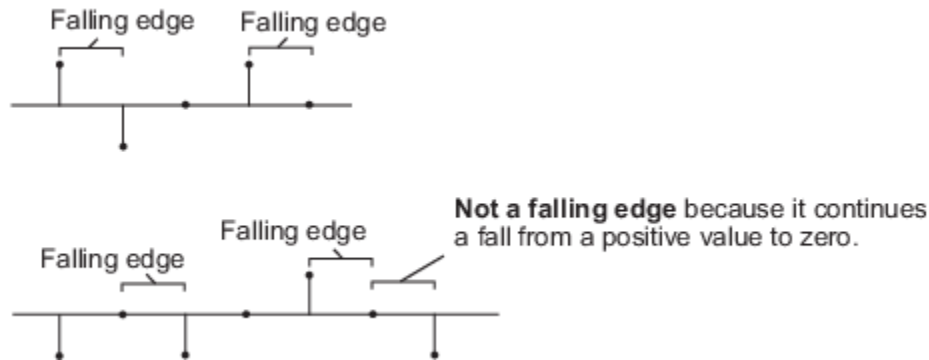
If a reset event occurs while the block is performing sample-based processing, the block initializes the current output to the values of the current input. If a reset event occurs while the block is performing frame-based processing, the block initializes the first row of the current output to the values in the first row of the current input.

The **Reset** port parameter specifies the reset event, which can be one of the following:

- **None** disables the **Rst** port.
- **Rising edge** — Triggers a reset operation when the **Rst** input does one of the following:
  - Rises from a negative value to a positive value or zero
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero



- **Falling edge** — Triggers a reset operation when the **Rst** input does one of the following:
  - Falls from a positive value to a negative value or zero
  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero



- **Either edge** — Triggers a reset operation when the **Rst** input is a **Rising edge** or **Falling edge**
- **Non-zero sample** — Triggers a reset operation at each sample time that the **Rst** input is not zero

---

**Note** When you run simulations in the Simulink **MultiTasking** mode, reset signals have a one-sample latency. When the block detects a reset event, a one-sample delay occurs at the reset port rate before the block applies the reset. For more information on latency and the Simulink tasking modes, see “Excess Algorithmic Delay (Tasking Latency)” and “Time-Based Scheduling and Code Generation” (Simulink Coder).

---

### Summing Along Columns

When you set the `Sum` input along parameter to `Columns`, the block computes the cumulative sum of each column of the input. In this mode, the current cumulative sum is independent of the cumulative sums of previous inputs.

```
y = cumsum(u) % Equivalent MATLAB code
```

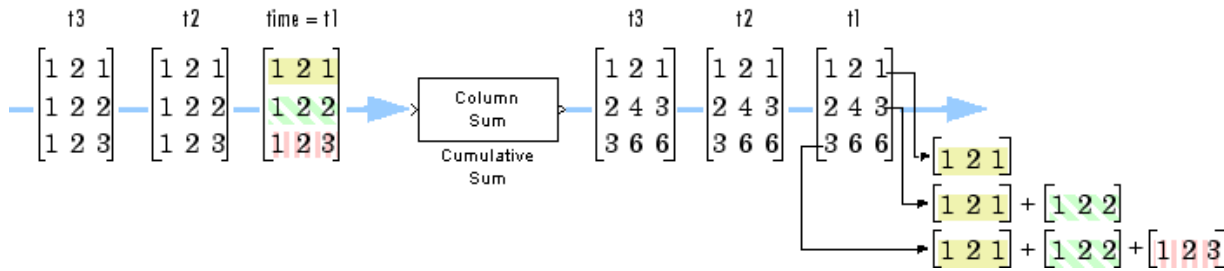
The output has the same size, dimension, data type, and complexity as the input. The  $m$ th output row is the sum of the first  $m$  input rows.

Given an  $M$ -by- $N$  input,  $u$ , the output,  $y$ , is an  $M$ -by- $N$  matrix whose  $j$ th column has elements

$$y_{i,j} = \sum_{k=1}^j u_{k,j} \quad 1 \leq i \leq M$$

The block treats length- $M$  unoriented vector inputs as  $M$ -by-1 column vectors when summing along columns.

**Sum Along Columns**



**Summing Along Rows**

When you set the Sum input along parameter to Rows, the block computes the cumulative sum of the row elements. In this mode, the current cumulative sum is independent of the cumulative sums of previous inputs.

```
y = cumsum(u,2) % Equivalent MATLAB code
```

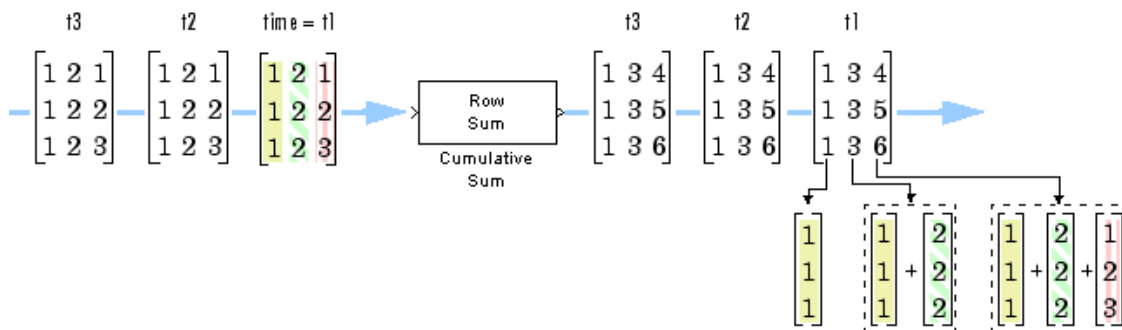
The output has the same size, dimension, and data type as the input. The  $n$ th output column is the sum of the first  $n$  input columns.

Given an  $M$ -by- $N$  input,  $u$ , the output,  $y$ , is an  $M$ -by- $N$  matrix whose  $i$ th row has elements

$$y_{i,j} = \sum_{k=1}^j u_{i,k} \quad 1 \leq j \leq N$$

When you sum along rows, the block treats length- $N$  unoriented vector inputs as 1-by- $N$  row vectors.

**Sum Along Rows**



**Version History**  
Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

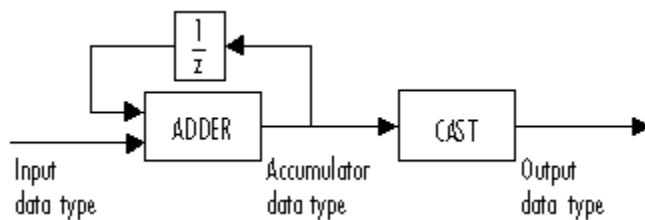
Generate C and C++ code using Simulink® Coder™.

Generated code relies on the memcpy or memset function (string.h) under certain conditions.

### Fixed-Point Conversion

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

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



You can set the accumulator and output data types in the block dialog box. See “Parameters” on page 2-343.

## See Also

### Functions

cumsum

### Blocks

Cumulative Product | Difference | Matrix Sum

# Dataflow Subsystem

Subsystem whose execution domain is set to Dataflow



**Libraries:**  
DSP System Toolbox / Dataflow

## Description

The Dataflow Subsystem block is a Subsystem block preconfigured with the execution domain set to dataflow. A Dataflow Subsystem uses synchronous dataflow as a model of computation, which is data-driven and statically scheduled.

Dataflow Subsystems help to

- Improve simulation throughput with multithreaded execution

Dataflow domains leverage the multicore CPU architecture of the host computer and can improve simulation speed significantly. It automatically partitions your model and simulates the system using multiple threads. By adding latency to your system, you can further increase concurrency and improve simulation throughput of your model.

- Automatically infer signal sizes for frame-based multirate models

See “Dataflow Domain” for more information.

## Ports

### Input

**In** — Signal input to dataflow subsystem  
scalar | vector | matrix

Placing an Inport block in a subsystem adds an external input port to the block. The port label on the subsystem block is the name of the Inport block.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | Boolean | fixed point | enumerated | bus

### Output

**Out** — Signal output from dataflow subsystem  
scalar | vector | matrix

Placing an Outport block in a subsystem adds an output port from the block. The port label on the subsystem block is the name of the Outport block.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | Boolean | fixed point | enumerated | bus

## Parameters

The Dataflow Subsystem block uses the same parameters as the Subsystem block. For parameter descriptions and programmatic use information, see Subsystem, Atomic Subsystem, Nonvirtual Subsystem, CodeReuse Subsystem.

---

**Note** Dataflow subsystems cannot be atomic subsystems.

---

**Latency** — Latency of dataflow subsystem  
0 (default) | scalar integer

To increase the throughput of a system, it can be advantageous to increase the latency of the system. Specify the **Latency** value in the **Execution** tab of the Property Inspector. For more information, see “Latency”.

### Programmatic Use

**Block Parameter:** Latency

**Type:** character vector

**Values:** scalar integer

**Default:** '0'

**Automatic frame size calculation** — Automatically calculate frame sizes and insert buffers  
0 (default) | 1

When the **Automatic frame size calculation** parameter is enabled, dataflow domains automatically calculate frame sizes and insert buffers into your model, avoiding signal size propagation errors in multirate signal processing systems. For more information, see “Automatic Frame Size Calculation”.

### Programmatic Use

**Block Parameter:** AutoFrameSizeCalculation

**Type:** character vector

**Values:** 'off' | 'on'

**Default:** 'off'

## Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | Boolean <sup>a</sup>   bus <sup>a</sup>   double <sup>a</sup>   enumerated <sup>a</sup>   fixed point <sup>a</sup>   half <sup>a</sup>   integer <sup>a</sup>   single <sup>a</sup>   string <sup>a</sup> |
| <b>Direct Feedthrough</b>       | no  |
| <b>Multidimensional Signals</b> | yes <sup>a</sup>  |
| <b>Variable-Size Signals</b>    | yes <sup>a</sup>  |
| <b>Zero-Crossing Detection</b>  | no  |

<sup>a</sup> Actual data type or capability support depends on block implementation.

## Version History

Introduced in R2018b



## Extended Capabilities

### C/C++ Code Generation

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

Dataflow subsystems can generate single-core or multi-core code. For more information on setting up your dataflow subsystem for code generation, see “Multicore Simulation and Code Generation of Dataflow Domains”.

## See Also

### Topics

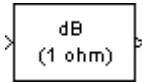
“Dataflow Domain”

“Multicore Simulation and Code Generation of Dataflow Domains”

“Model Multirate Signal Processing Systems Using Dataflow”

## dB Conversion

Convert magnitude data to decibels (dB or dBm)



### Libraries:

DSP System Toolbox / Math Functions / Math Operations

## Description

The dB Conversion block converts a linearly scaled power or amplitude input to dB or dBm. The reference power is 1 Watt for conversions to dB and 1 mWatt for conversions to dBm. The block's **Input signal** parameter specifies whether the input is a power signal or a voltage signal, and the **Convert to** parameter controls the scaling of the output. When selected, the **Add eps to input to protect against "log(0) = -inf"** parameter adds a value of `eps` to all power and voltage inputs. When this parameter is not selected, zero-valued inputs produce `-Inf` at the output.

## Ports

### Input

**Port\_1** — Input signal

scalar | vector | matrix

Input signal to be converted, specified as a real scalar, vector, or matrix.

When the "Input signal" on page 2-0 parameter is set to:

- **Power** — The input is treated as a power signal, specified in Watts.
- **Amplitude** — The input is treated as a voltage signal, specified in Volts.

Data Types: single | double

### Output

**Port\_1** — Output signal

scalar | vector | matrix

The converted output signal, returned as a scalar, vector, or matrix. The output is the same size, and data type as the input. The output  $y$  is calculated as follows.

When the "Input signal" on page 2-0 parameter is set to **Power** and **Convert to** parameter is set to:

- **dB** — The block performs this dB conversion:  

$$y = 10 \cdot \log_{10}(u)$$
- **dBm** — The block performs this dBm conversion:  

$$y = 10 \cdot \log_{10}(u) + 30$$

The dBm conversion is equivalent to performing the dB operation after converting the input to milliwatts.

Input  $u$  is a real, nonnegative, power signal, specified in Watts.

When the **Input signal** parameter is set to **Amplitude** and **Convert to** parameter is set to:

- **dB** — The block performs this dB conversion:  

$$y = 10 \cdot \log_{10}(\text{abs}(u)^2/R)$$
- **dBm** — The block performs the following dBm conversion:

$$y = 10 \cdot \log_{10}(\text{abs}(u)^2/R) + 30$$

Input  $u$  is a real voltage signal, specified in Volts.

The dBm conversion is equivalent to performing the dB operation after converting the  $(\text{abs}(u)^2/R)$  result to milliwatts. The scale factor,  $R$ , is specified in Ohms.

Data Types: `single` | `double`

## Parameters

**Convert to** — Signal to convert  
 dB (default) | dBm

The logarithmic scaling to which the input is converted, **dB** or **dBm**. The reference power is 1 Watt for conversions to dB and 1 mW for conversions to dBm.

Amplitude conversions use the scale factor specified in ohms by the **Load resistance** parameter,  $R$ , to convert the voltage input to units of power (watts) before converting to dB or dBm.

When the “Input signal” on page 2-0 parameter is set to **Power** and **Convert to** parameter is set to:

- **dB** — The block performs this dB conversion:  

$$y = 10 \cdot \log_{10}(u)$$
- **dBm** — The block performs this dBm conversion:

$$y = 10 \cdot \log_{10}(u) + 30$$

The dBm conversion is equivalent to performing the dB operation after converting the input to milliwatts.

Input  $u$  is a real, nonnegative, power signal, specified in Watts.

**Input signal** is set to **Amplitude** and **Convert to** parameter is set to:

- **dB** — The block performs this dB conversion:  

$$y = 10 \cdot \log_{10}(\text{abs}(u)^2/R)$$
- **dBm** — The block performs this dBm conversion:

$$y = 10 \cdot \log_{10}(\text{abs}(u)^2/R) + 30$$

The dBm conversion is equivalent to performing the dB operation after converting the  $(\text{abs}(u)^2/R)$  result to mW. The scale factor  $R$  is specified in Ohms.

Input  $u$  is a real voltage signal, specified in Volts.

**Tunable:** Yes

**Input signal** — Type of signal  
Amplitude (default) | Power

The type of input signal, Power or Amplitude.

When **Input signal** is set to:

- Power -- Input  $u$  is a real, nonnegative, power signal (units of Watts).
- Amplitude -- Input  $u$  is a real voltage signal (units of Volts).

**Load resistance (ohms)** — Scale factor  
1 (default) | positive scalar | positive integer

The scale factor  $R$ , specified in ohms. The block uses the scale factor to convert voltage input to units of power (watts) before converting to dB or dBm.

**Tunable:** Yes

#### Dependencies

To enable this parameter, set the "Input signal" on page 2-0 parameter to Amplitude.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

Complex Number Support: Yes

**Add eps to input to protect against "log(0) = -inf"** — Add eps  
off (default) | on

Select this parameter to add eps to all input values (power or voltage).

**Tunable:** Yes

## Block Characteristics

|                                 |                 |
|---------------------------------|-----------------|
| <b>Data Types</b>               | double   single |
| <b>Multidimensional Signals</b> | No              |
| <b>Variable-Size Signals</b>    | No              |

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

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

## See Also

### Blocks

Math Function

### Functions

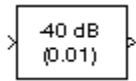
log10

### Topics

dB Gain

## dB Gain

Apply decibel gain



## Library

Math Functions / Math Operations

dspmathops

## Description

The dB Gain block multiplies the input by the decibel values specified in the **Gain** parameter. For an  $M$ -by- $N$  input matrix  $u$  with elements  $u_{ij}$ , the **Gain** parameter can be a real  $M$ -by- $N$  matrix with elements  $g_{ij}$  to be multiplied element-wise with the input, or a real scalar.

$$y_{ij} = u_{ij}10^{(g_{ij}/k)}$$

The value of  $k$  is 10 for power signals (select **Power** as the **Input signal** parameter) and 20 for voltage signals (select **Amplitude** as the **Input signal** parameter).

The value of the equivalent linear gain

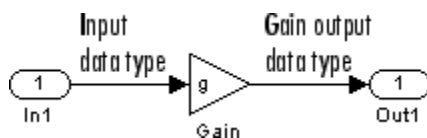
$$g_{ij}^{lin} = 10^{(g_{ij}/k)}$$

is displayed in the block icon below the dB gain value. The output is the same size as the input.

The dB Gain block supports real and complex floating-point and fixed-point data types.

### Fixed-Point Data Types

The following diagram shows the data types used within the dB Gain subsystem block for fixed-point signals.



The settings for the fixed-point parameters of the Gain block in the diagram above are as follows:

- Integer rounding mode: **Floor**
- Saturate on integer overflow — unselected
- Parameter data type mode — **Inherit via internal rule**
- Output data type mode — **Inherit via internal rule**

See the Gain reference page for more information.

## Parameters

### Gain

The dB gain to apply to the input, a scalar or a real  $M$ -by- $N$  matrix. Tunable (Simulink).

### Input signal

The type of input signal: Power or Amplitude. Tunable (Simulink).

---

**Note** This block does not support tunability in generated code.

---

## Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point (signed and unsigned)
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

## See Also

|               |                    |
|---------------|--------------------|
| dB Conversion | DSP System Toolbox |
| Math Function | Simulink           |
| log10         | MATLAB             |

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

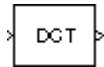
Generate C and C++ code using Simulink® Coder™.

### Fixed-Point Conversion

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

## DCT

Discrete cosine transform (DCT) of input



**Libraries:**  
DSP System Toolbox / Transforms

### Description

The DCT block computes the unitary discrete cosine transform (DCT) of the input signal  $u$ . When the input is an  $M$ -by- $N$  matrix, the block computes the DCT of each channel in the matrix.

Here is the equivalent MATLAB code.

```
y = dct(u)
```

This block supports Simulink virtual buses.

### Ports

#### Input

**Input 1** — Input signal  
vector | matrix |  $N$ -D array

Specify the input signal as a vector, matrix, or an  $N$ -D array.

For all  $N$ -D input arrays, the block computes the DCT across the first dimension. The size of the first dimension (frame size) must be a power of two. To work with other frame sizes, use the Pad block to pad or truncate the frame size to a power-of-two length.

When the input to the DCT block is an  $M$ -by- $N$  matrix, the block treats each input column as an independent channel containing  $M$  consecutive samples.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fixed point | bus

#### Output

**Output 1** — DCT output  
vector | matrix |  $N$ -D array

The block outputs the DCT of the input signal as a vector, matrix, or an  $N$ -D array.

When the block outputs an  $M$ -by- $N$  matrix, the  $l^{\text{th}}$  column of the matrix contains the length- $M$  DCT of the corresponding input column.

$$y(k, l) = w(k) \sum_{m=1}^M u(m, l) \cos \frac{\pi(2m-1)(k-1)}{2M}, \quad k = 1, \dots, M$$

where



$$w(k) = \begin{cases} \frac{1}{\sqrt{M}}, & k = 1 \\ \sqrt{\frac{2}{M}}, & 2 \leq k \leq M \end{cases}$$

When the input is fixed-point (signed and unsigned), the output is signed fixed-point.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `fixed point` | `bus`

## Parameters

### Main Tab

#### Sine and cosine computation — Sine and cosine computation

`Table lookup (default)` | `Trigonometric fcn`

Set the block to compute sines and cosines by either looking up sine and cosine values in a speed-optimized table (`Table lookup`), or by making sine and cosine function calls (`Trigonometric fcn`).

To compute sines and cosines, set the parameter to one of these values:

- `Table lookup` -- The block computes and stores the trigonometric values before the simulation starts, and retrieves them during the simulation. When you generate code from the block, the processor running the generated code stores the trigonometric values computed by the block in a speed-optimized table, and retrieves the values during code execution.

The block runs much more quickly, but requires extra memory for storing the precomputed trigonometric values.

- `Trigonometric fcn` -- The block computes sine and cosine values during the simulation. When you generate code from the block, the processor running the generated code computes the sine and cosine values while the code runs.

The block runs more slowly, but does not need extra data memory. For code generation, the block requires a support library to emulate the trigonometric functions, increasing the size of the generated code.

### Data Types Tab

#### Rounding mode — Rounding mode

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

Select the rounding mode for fixed-point operations. The sine table values do not obey this parameter; they always round to `Nearest`.

#### Saturate on integer overflow — Saturate on integer overflow

`off (default)` | `on`

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

---

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

- **Product output** data type is `Inherit: Inherit via internal rule`.
- **Accumulator** data type is `Inherit: Inherit via internal rule`.

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

---

### Sine table — Sine table

`Inherit: Same word length as input (default) | fixd(1,16)`

Choose how you specify the word length of the values of the sine table. The fraction length of the sine table values always equals the word length minus one. You can set this parameter to:

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


The sine table values do not obey the **Rounding mode** and **Saturate on integer overflow** parameters. The block always saturates and rounds off the sine table values to `Nearest`.

### Product output — Product output

`Inherit: Inherit via internal rule (default) | Inherit: Same as input | fixd(1,16,0)`

Specify the product output data type. See “Fixed-Point Data Types” on page 2-364 and “Multiplication Data Types” for illustrations depicting the use of the product output data type in this block. You can set this parameter to:

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

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


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

### Accumulator — Accumulator

`Inherit: Inherit via internal rule (default) | Inherit: Same as input | Inherit: Same as product output | fixd(1,16,0)`

Specify the accumulator data type. See “Fixed-Point Data Types” on page 2-364 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

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

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

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

### Output — Data type of output

Inherit: `Inherit via internal rule` (default) | `Inherit: Same as input` | `fixd(1,16,0)`

Specify the output data type. See “Fixed-Point Data Types” on page 2-364 for illustrations depicting the use of the output data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`.


When you select `Inherit: Inherit via internal rule`, the block calculates the output word length and fraction length automatically. The internal rule first calculates an ideal output word length and fraction length using the following equations:

$$WL_{idealoutput} = WL_{input} + \text{floor}(\log_2(DCTlength - 1)) + 1$$

$$FL_{idealoutput} = FL_{input}$$

Using these ideal results, the internal rule then selects word lengths and fraction lengths that are appropriate for your hardware. For more information on this rule, see “Inherit via Internal Rule”.

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

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

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

### Output Minimum — Output minimum

`[]` (default) | scalar

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

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

### Output Maximum — Output maximum

`[]` (default) | scalar

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

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

**Lock data type settings against changes by the fixed-point tools** — Prevent fixed-point tools from overriding data types  
off (default) | on

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

## Block Characteristics

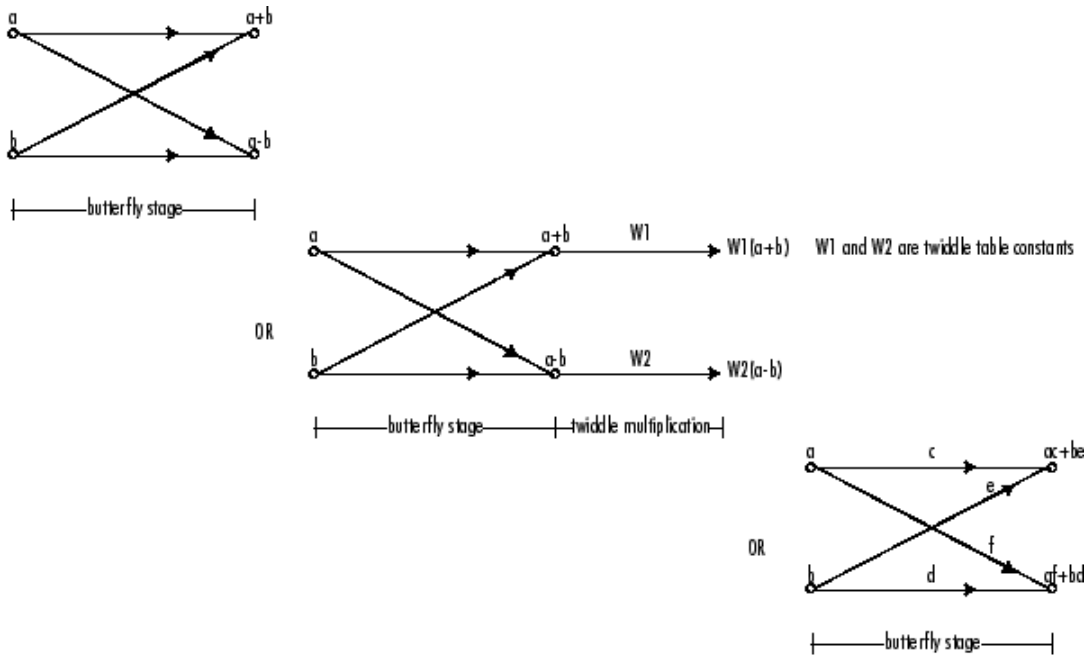
|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | double   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no                                      |
| <b>Multidimensional Signals</b> | yes                                     |
| <b>Variable-Size Signals</b>    | no                                      |
| <b>Zero-Crossing Detection</b>  | no                                      |

## More About

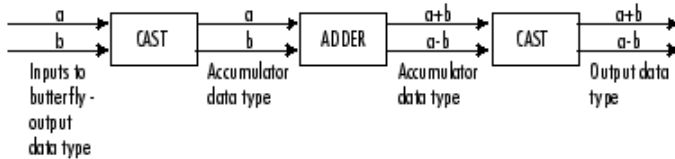
### Fixed-Point Data Types

The following diagrams show the data types used within the DCT block for fixed-point signals. You can set the sine table, accumulator, product output, and output data types displayed in the diagrams in the block parameters dialog box of the DCT block.

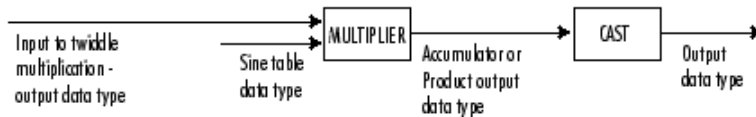
Inputs to the DCT block are first cast to the output data type and stored in the output buffer. Each butterfly stage processes signals in the accumulator data type, with the final output of the butterfly being cast back into the output data type.



**Butterfly Stage Data Types**



**Twiddle Multiplication Data Types**



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

---

**Note** When the block input is fixed point, all internal data types are signed fixed point.

---

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

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

Generated code relies on the `memcpy` or `memset` function (`string.h`) under certain conditions.

To comply with the embedded systems coding standards, generated code is non-recursive.

**Fixed-Point Conversion**

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

**See Also****Functions**

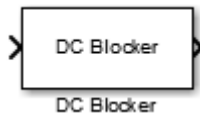
`dct`

**Blocks**

IDCT | Complex Cepstrum | FFT | Real Cepstrum

# DC Blocker

Block DC component



## Libraries:

DSP System Toolbox / Signal Operations

DSP System Toolbox HDL Support / Signal Operations

## Description

The DC Blocker block removes the DC component of the input signal. This block supports SIMD code generation. For details, see “Code Generation” on page 2-371.

## Ports

### Input

**Input** — Input signal

scalar | vector | matrix

Specify the input signal that contains the DC component as a scalar, vector, or a matrix. When the input is fixed-point, it is signed only.

Data Types: single | double | int8 | int16 | int32 | fixed point

### Output

**Output** — Output signal with no DC component

scalar | vector | matrix

Output signal with the DC component removed. When the output is fixed-point, it is signed only.

The size, data type, and complexity of the output signal are the same as that of the input signal.

Data Types: single | double | int8 | int16 | int32 | fixed point

## Parameters

**Algorithm for estimating DC offset** — Algorithm for estimating DC offset

IIR (default) | FIR | CIC | Subtract mean

Specify the algorithm used for estimating the DC offset. Select from the following:

- IIR uses a recursive estimate based on a narrow, lowpass elliptic filter. This algorithm typically uses less memory than FIR and is more efficient.
- FIR uses a nonrecursive, moving-average estimate. This algorithm typically uses more memory than IIR and is less efficient.
- CIC uses a lowpass filter that does not employ any multipliers. If the algorithm is CIC, then fixed-point data must be input to the DC Blocker.

- **Subtract mean** computes the means of the columns of the input matrix and subtracts the means from the input. This method does not retain state between inputs. For example, if the input is `[1 2 3 4; 3 4 5 6]`, then the DC Blocker block in **Subtract mean** mode outputs `[-1 -1 -1 -1; 1 1 1 1]`.

**Normalized bandwidth of lowpass IIR or CIC filter** — Normalized bandwidth

0.001 (default) | real scalar in the range (0 1)

Specify the normalized filter bandwidth as a real scalar greater than 0 and less than 1.

**Dependencies**

The DC Blocker uses this parameter only when the estimation algorithm is set to **IIR** or **CIC**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**Order of lowpass IIR elliptic filter** — Order of lowpass IIR elliptic filter

6 (default) | integer greater than 3

Specify the filter order of the lowpass IIR elliptic filter as an integer greater than 3.

**Dependencies**

The DC Blocker uses this parameter only when the estimation algorithm is set to **IIR**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**Number of past input samples for FIR algorithm** — Number of past input samples for FIR algorithm

50 (default) | positive integer

Specify, as a positive integer, the number of samples to use when the estimation algorithm is set to **FIR**.

**Dependencies**

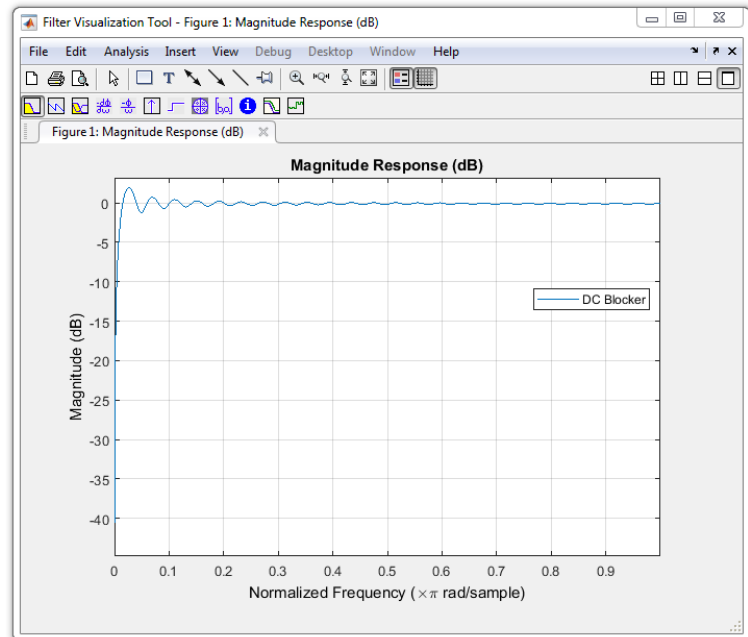
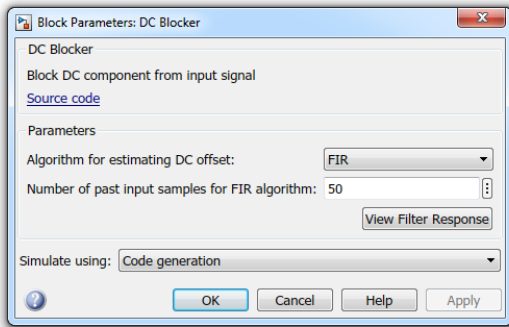
The DC Blocker uses this parameter only when the estimation algorithm is set to **FIR**.

Data Types: `single` | `double`

**View Filter Response** — Visualize frequency response  
button

Opens the `fvtool` and displays the magnitude response of the DC Blocker. The response is based on the block parameters. Changes made to these parameters update `fvtool`.





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

**Simulate using** — Type of simulation to run  
Code generation (default) | Interpreted execution

Specify the type of simulation to run as one of the following:

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

## Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | double   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no                                      |
| <b>Multidimensional Signals</b> | no                                      |
| <b>Variable-Size Signals</b>    | yes <sup>a</sup>                        |
| <b>Zero-Crossing Detection</b>  | no                                      |

<sup>a</sup>

## Algorithms

The DC blocker subtracts the DC component from the input signal. You can estimate the DC component by using the IIR, FIR, CIC, or subtract mean algorithm.

### IIR

Pass the input signal through an IIR lowpass elliptical filter.

The elliptical IIR filter has a passband ripple of 0.1 dB and a stopband attenuation of 60 dB. You specify the normalized bandwidth and the filter order.

### FIR

Pass the input signal through an FIR filter that uses a nonrecursive moving average from a finite number of past input samples.

The FIR filter coefficients are given as  $\text{ones}(1, \text{Length})/\text{Length}$ , where  $\text{Length}$  is the number of past input samples for FIR algorithm. The FIR filter structure is a direct form I transposed structure.

### CIC

Pass the input signal through a CIC filter. Because the CIC filter amplifies the signal, the filter gain is estimated and subtracted from the DC estimate.

The Cascaded Integrator-Comb (CIC) filter consists of two integrator-comb pairs. These pairs help to ensure that the peak of the first sidelobe of the filter response is attenuated by at least 25 dB relative to the peak of the main lobe. The normalized 3 dB bandwidth is used to calculate the differential delay. The delay is used to determine the gain of the CIC filter. The inverse of the filter gain is used as a multiplier, which is applied to the output of the CIC filter. This ensures that the aggregate gain of the DC estimate is 0 dB.

The following equation characterizes the aggregate magnitude response of the filter and the multiplier:

$$|H(e^{j\omega})| = \left| \frac{\sin(M\frac{\pi}{2}B_{norm})}{M\sin(\frac{\pi}{2}B_{norm})} \right|^N$$

- $B_{norm}$  is the normalized bandwidth such that  $0 < B_{norm} < 1$ .
- $M$  is the differential delay in samples.
- $N$  is the number of sections, equal to 2.

Set the differential delay,  $M$ , to the smallest integer such that  $|H(e^{j\omega})| < 1/\sqrt{2}$ . Once  $M$  is known, the gain of the CIC filter is calculated as  $M^N$ . Therefore, to precisely compensate for the filter gain, the multiplier is set to  $(1/M)^N$ .

### Subtract mean

Compute the mean value of each column of the input signal and subtract the mean from the input. For example, if the input is [1 2 3 4; 3 4 5 6], then a DC Blocker set to this mode outputs [-1 -1 -1 -1; 1 1 1 1].

## Version History

Introduced in R2014a

## References

- [1] Nezami, M.K. "Performance Assessment of Baseband Algorithms for Direct Conversion Tactical Software Defined Receivers: I/Q Imbalance Correction, Image Rejection, DC Removal, and Channelization." In MILCOM 2002. Proceedings, 369-76. Anaheim, CA, USA: IEEE, 2002.

## Extended Capabilities

### C/C++ Code Generation

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

The DC Blocker block supports SIMD code generation using Intel AVX2 technology when the input signal has a data type of `single` or `double`.

The SIMD technology significantly improves the performance of the generated code.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

---

**Note** For a DC Blocker optimized for HDL code generation, see "High Performance DC Blocker for FPGA" (DSP HDL Toolbox). The DSP HDL Toolbox block provides an HDL-optimized algorithm with hardware-friendly control signals and simulates the latency of the HDL algorithm in Simulink.

---

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

### HDL Architecture

This block has one default HDL architecture.

### HDL Block Properties

|                                  |  |
|----------------------------------|--|
| <b>ConstrainedOutputPipeline</b> | Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see "ConstrainedOutputPipeline" (HDL Coder). |
| <b>InputPipeline</b>             | Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "InputPipeline" (HDL Coder).                 |
| <b>OutputPipeline</b>            | Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "OutputPipeline" (HDL Coder).               |

**Complex Data Support**

This block supports code generation for complex signals.

**Fixed-Point Conversion**

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

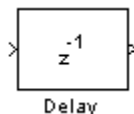
If the input is fixed point, it must be a signed integer or a signed fixed point value with a power-of-two slope and zero bias.

**See Also****Objects**

`dsp.DCBlocker` | `dsp.SOSFilter` | `dsp.FIRFilter`

## Delay

Delay discrete-time input by specified number of samples or frames



## Compatibility

---

**Note** The Delay block from the `dspsigops` library has been replaced by the Delay block from the Discrete library in Simulink. Existing instances of the `dspsigops` Delay block will be replaced with Simulink Delay block when there is an exact match in functionality between the two blocks. For new models, use the Delay block from the Discrete library in Simulink.

---

## Library

Signal Operations

`dspsigops`

## Description

The Delay block delays a discrete-time input by the number of samples or frames specified in the **Delay units** and **Delay** parameters. The **Delay** value must be an integer value greater than or equal to zero. When you enter a value of zero for the **Delay** parameter, any initial conditions you might have entered have no effect on the output.

The Delay block allows you to set the initial conditions of the signal that is being delayed. The initial conditions must be numeric.

### Frame-Based Processing

When you set the **Input processing** parameter to `Columns as channels (frame based)`, the block treats each column of the  $M$ -by- $N$  input matrix as an independent channel. The block delays each channel of the input as specified by the **Delay** parameter.

The **Delay** parameter can be a scalar integer by which the block equally delays all channels or a vector whose length is equal to the number of channels.

There are four different choices for initial conditions. The initial conditions can be the same or different for each channel. They can also be constant or varying along each channel. See the “Frame-Based Processing Examples” on page 2-375 section for more information.

### Sample-Based Processing

When you set the **Input processing** parameter to `Elements as channels (sample based)`, the block treats each element of the  $N$ -D input array as an independent channel. Thus, the total number

of channels in the input is equal to the product of the input dimensions. The dimension of the output is the same as that of the input.

The **Delay** parameter can be a scalar integer by which to equally delay all channels or an N-D array of the same dimensions as the input array, containing nonnegative integers that specify the number of sample intervals to delay each channel of the input.

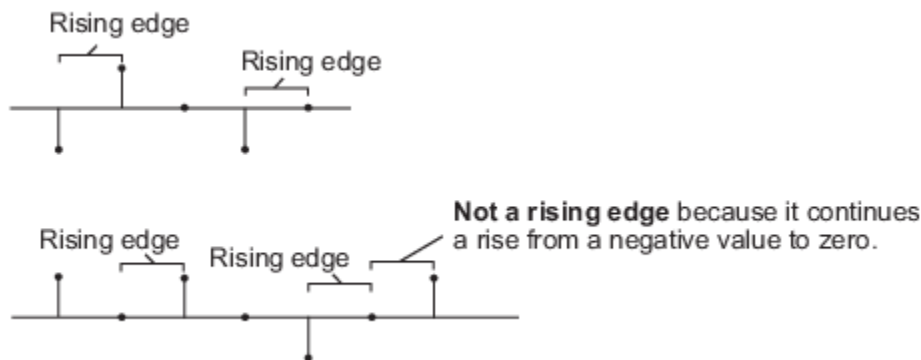
There are four different choices for initial conditions. The initial conditions can be the same or different for each channel. They can also be the same or different within a channel. See the “Sample-Based Processing Examples” on page 2-378 section for more information.

### Resetting the Delay

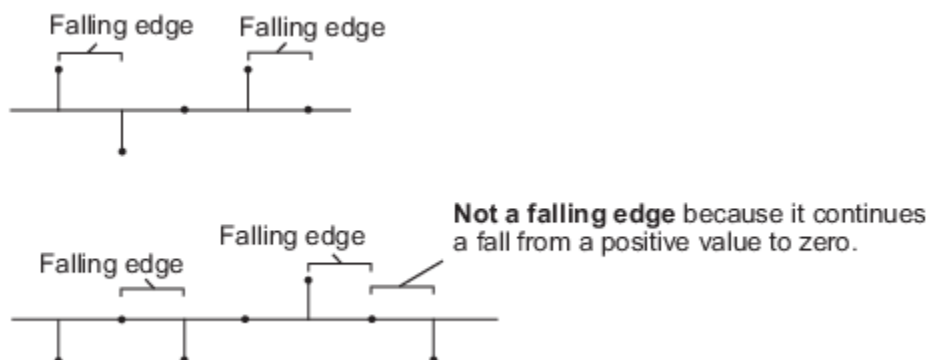
The Delay block resets the delay whenever it detects a reset event at the optional Rst port. The reset sample time must be a positive integer multiple of the input sample time.

The reset event is specified by the **Reset port** parameter, and can be one of the following:

- **None** disables the Rst port.
- **Rising edge** triggers a reset operation when the Rst input does one of the following:
  - Rises from a negative value to a positive value or zero
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure)



- **Falling edge** triggers a reset operation when the Rst input does one of the following:
  - Falls from a positive value to a negative value or zero
  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero (see the following figure)



- Either edge triggers a reset operation when the Rst input is Rising edge or Falling edge (as described earlier).
- Non-zero sample triggers a reset operation at each sample time that the Rst input is not zero.

---

**Note** When running simulations in the Simulink MultiTasking mode, reset signals have a one-sample latency. Therefore, when the block detects a reset event, there is a one-sample delay at the reset port rate before the block applies the reset. For more information on latency and the Simulink tasking modes, see “Excess Algorithmic Delay (Tasking Latency)” and “Time-Based Scheduling and Code Generation” (Simulink Coder).

---

This block supports Simulink virtual buses.

## Examples

### Frame-Based Processing Examples

There are four different choices for initial conditions. The initial conditions can be the same or different for each channel. They can also be constant or varying along each channel. The next sections describe the behavior of the block for each of these four cases:

- “Case 1 — Use the Same Initial Conditions for Each Channel and Within a Channel” on page 2-375
- “Case 2 — Use Different Initial Conditions for Each Channel and the Same Initial Conditions Within a Channel” on page 2-376
- “Case 3 — Use the Same Initial Conditions for Each Channel and Different Initial Conditions Within a Channel” on page 2-376
- “Case 4 — Use Different Initial Conditions for Each Channel and Within a Channel” on page 2-377

#### Case 1 — Use the Same Initial Conditions for Each Channel and Within a Channel

Enter a scalar value for the initial conditions. This value is used as the constant initial condition value for each of the channels.

For example, suppose your input is a matrix and you set the **Input processing** parameter to Columns as channels (frame based).

$$\begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 & 4 \\ 5 & 5 & 5 \\ 6 & 6 & 6 \end{bmatrix}, \begin{bmatrix} 7 & 7 & 7 \\ 8 & 8 & 8 \\ 9 & 9 & 9 \end{bmatrix}, \dots$$

You want the initial conditions of your three-channel signal to be identical and zero for the first frame:

- 1 Set the **Delay (frames)** parameter to 1.
- 2 Clear the **Specify different initial conditions for each channel** and the **Specify different initial conditions within a channel** check boxes.
- 3 Set the **Initial conditions** parameter to a scalar value of 0.

The output of the delay block is

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 & 4 \\ 5 & 5 & 5 \\ 6 & 6 & 6 \end{bmatrix}, \begin{bmatrix} 7 & 7 & 7 \\ 8 & 8 & 8 \\ 9 & 9 & 9 \end{bmatrix}, \dots$$

0, the scalar initial condition value, is used across the channels and within the channels for the first frame. This frame is the output at sample time zero.

### Case 2 — Use Different Initial Conditions for Each Channel and the Same Initial Conditions Within a Channel

The initial conditions must be a vector of length  $N$ , where  $N \geq 1$ .  $N$  is also equal to the number of channels in your signal. These initial condition values are used as the constant initial condition value for each of the channels.

For example, suppose your input is a matrix and you set the **Input processing** parameter to Columns as channels (frame based).

$$\begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 & 4 \\ 5 & 5 & 5 \\ 6 & 6 & 6 \end{bmatrix}, \begin{bmatrix} 7 & 7 & 7 \\ 8 & 8 & 8 \\ 9 & 9 & 9 \end{bmatrix}, \dots$$

You want the initial conditions of your three-channel signal to be [0 10 20] for the first frame:

- 1 Set the **Delay (frames)** parameter to 1.
- 2 Select the **Specify different initial conditions for each channel** check box.
- 3 Clear the **Specify different initial conditions within a channel** check box.
- 4 Set the **Initial conditions** parameter to [0 10 20].

The output of the delay block is

$$\begin{bmatrix} 0 & 10 & 20 \\ 0 & 10 & 20 \\ 0 & 10 & 20 \end{bmatrix}, \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 & 4 \\ 5 & 5 & 5 \\ 6 & 6 & 6 \end{bmatrix}, \begin{bmatrix} 7 & 7 & 7 \\ 8 & 8 & 8 \\ 9 & 9 & 9 \end{bmatrix}, \dots$$

The initial condition vector expands to create the frame that is output at sample time zero. Different initial conditions are used for each channel, but the same initial condition value is used with a channel.

### Case 3 — Use the Same Initial Conditions for Each Channel and Different Initial Conditions Within a Channel

In this case, the **Delay** parameter can be a scalar integer by which to equally delay all channels or a vector whose length is equal to the number of channels. All the values of this vector must be equal.

Enter the initial conditions as a vector. These values are used as the initial condition value along each of the channels to be delayed. The initial condition vector must have length equal to the value of the **Delay (frames)** parameter multiplied by the frame length. For example, if you want to delay your signal by two frames with frame length two and an initial condition value of 3, enter your initial condition vector as [3 3 3 3].

For example, suppose your input is a matrix and you set the **Input processing** parameter to Columns as channels (frame based).



$$\begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 & 4 \\ 5 & 5 & 5 \\ 6 & 6 & 6 \end{bmatrix}, \begin{bmatrix} 7 & 7 & 7 \\ 8 & 8 & 8 \\ 9 & 9 & 9 \end{bmatrix}, \dots$$

You want the initial conditions of your three-channel signal to be the same along each of the channels to be delayed:

- 1 Set the **Delay (frame)** parameter to 1.
- 2 Clear the **Specify different initial conditions for each channel** check box.
- 3 Select the **Specify different initial conditions within a channel** check box.
- 4 Set the **Initial conditions** parameter to [10 20 30].

The output of the delay block is

$$\begin{bmatrix} 10 & 10 & 10 \\ 20 & 20 & 20 \\ 30 & 30 & 30 \end{bmatrix}, \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 & 4 \\ 5 & 5 & 5 \\ 6 & 6 & 6 \end{bmatrix}, \begin{bmatrix} 7 & 7 & 7 \\ 8 & 8 & 8 \\ 9 & 9 & 9 \end{bmatrix}, \dots$$

The initial condition vector defines the initial condition values within each of the three channels. The same initial conditions are used for each channel, but different initial condition values are used with a channel.

#### Case 4 – Use Different Initial Conditions for Each Channel and Within a Channel

Enter a cell array for your initial condition values. Or, when you have a scalar delay value, you can enter the initial conditions as a matrix.

For example, suppose your input is a matrix and you set the **Input processing** parameter to Columns as channels (frame based).

$$\begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 & 4 \\ 5 & 5 & 5 \\ 6 & 6 & 6 \end{bmatrix}, \begin{bmatrix} 7 & 7 & 7 \\ 8 & 8 & 8 \\ 9 & 9 & 9 \end{bmatrix}, \dots$$

You want the initial conditions of your three-channel signal to be different for each channel and along each channel.

- 1 Set the **Delay (frames)** parameter to 1.
- 2 Select the **Specify different initial conditions for each channel** and the **Specify different initial conditions within a channel** check boxes.
- 3 Set the **Initial conditions** parameter to either [10 20 30; 40 50 60; 70 80 90] or {[10 40 70]; [20 50 80]; [30 60 90]}. Each cell of the cell array represents the delay along one channel.

Regardless of whether you use a matrix or cell array, the output of the delay block is

$$\begin{bmatrix} 10 & 20 & 30 \\ 40 & 50 & 60 \\ 70 & 80 & 90 \end{bmatrix}, \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 & 4 \\ 5 & 5 & 5 \\ 6 & 6 & 6 \end{bmatrix}, \begin{bmatrix} 7 & 7 & 7 \\ 8 & 8 & 8 \\ 9 & 9 & 9 \end{bmatrix} \dots$$

The initial condition matrix is the output at sample time zero. The elements of the initial condition cell array define the initial condition values within each channel. The first element, a

vector, represents the initial conditions within channel 1. The second element, a vector, represents the initial conditions within channel 2, and so on. Different initial conditions are used for each channel and within the channels.

### Sample-Based Processing Examples

There are four different choices for initial conditions. The initial conditions can be the same or different for each channel. They can also be the same or different along each channel. The next sections describe the behavior of the block for each of these four cases:

- “Case 1 — Use the Same Initial Conditions for Each Channel and Within a Channel” on page 2-378
- “Case 2 — Use Different Initial Conditions for Each Channel and the Same Initial Conditions Within a Channel” on page 2-378
- “Case 3 — Use the Same Initial Conditions for Each Channel and Different Initial Conditions Within a Channel” on page 2-379
- “Case 4 — Use Different Initial Conditions for Each Channel and Within a Channel” on page 2-380

#### Case 1 — Use the Same Initial Conditions for Each Channel and Within a Channel

Enter a scalar value for the initial conditions. This value is used as the constant initial condition value for each of the channels.

For example, suppose your input is a matrix and you set the **Input processing** parameter to `Elements as channels (sample based)`.

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}, \begin{bmatrix} 3 & 3 \\ 3 & 3 \end{bmatrix}, \dots$$

You want the initial conditions of your four-channel signal to be identical and zero for the first two samples:

- 1 Set the **Delay (samples)** parameter to 2.
- 2 Clear the **Specify different initial conditions for each channel** and **Specify different initial conditions within a channel** check boxes.
- 3 Set the **Initial conditions** parameter to a scalar value of 0.

The output of the delay block is

$$\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}, \begin{bmatrix} 3 & 3 \\ 3 & 3 \end{bmatrix}, \dots$$

0, the scalar initial condition value, is used for each channel and within the channels. It is the output at sample time zero and sample time one.

#### Case 2 — Use Different Initial Conditions for Each Channel and the Same Initial Conditions Within a Channel

The initial conditions must be an N-D array for N-D input. The initial conditions must have the same dimensions as the input data. These initial condition values are used as the constant initial condition value for each of the channels.

For example, suppose your input is a matrix and you set the **Input processing** parameter to `Elements as channels (sample based)`.

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}, \begin{bmatrix} 3 & 3 \\ 3 & 3 \end{bmatrix}, \dots$$

You want the initial conditions of your four-channel signal to be

$$\begin{bmatrix} 7 & 9 \\ 11 & 13 \end{bmatrix}$$

for the first two samples:

- 1 Set the **Delay (samples)** parameter to 2.
- 2 Select the **Specify different initial conditions for each channel** check box.
- 3 Clear the **Specify different initial conditions within a channel** check box.
- 4 Set the **Initial conditions** parameter to [7 9; 11 13].

The output of the delay block is

$$\begin{bmatrix} 7 & 9 \\ 11 & 13 \end{bmatrix}, \begin{bmatrix} 7 & 9 \\ 11 & 13 \end{bmatrix}, \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}, \begin{bmatrix} 3 & 3 \\ 3 & 3 \end{bmatrix}, \dots$$

The initial condition matrix is the output at sample time zero and sample time one. Different initial conditions are used for each channel; the same initial condition value is used within a channel.

### Case 3 — Use the Same Initial Conditions for Each Channel and Different Initial Conditions Within a Channel

In this case, for N-D sample-based inputs, the initial conditions parameter must be a vector whose length is equal to the delay value, specified by the **Delay** parameter. The values in this vector are used as the initial condition values along each of the channels to be delayed.

For example, suppose your input is a matrix and you set the **Input processing** parameter to **Elements as channels (sample based)**.

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}, \begin{bmatrix} 3 & 3 \\ 3 & 3 \end{bmatrix}, \dots$$

You want the initial conditions of your four channel signal to be the same along each of the channels to be delayed:

- 1 Set the **Delay (samples)** parameter to 2.
- 2 Clear the **Specify different initial conditions for each channel** check box.
- 3 Select the **Specify different initial conditions within a channel** check box.
- 4 Set the **Initial conditions** parameter to [10 20].

The output of the delay block is

$$\begin{bmatrix} 10 & 10 \\ 10 & 10 \end{bmatrix}, \begin{bmatrix} 20 & 20 \\ 20 & 20 \end{bmatrix}, \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}, \begin{bmatrix} 3 & 3 \\ 3 & 3 \end{bmatrix}, \dots$$

The first element of the initial conditions vector is the output, for all channels, at sample time zero. The second element of the initial conditions vector is the output, for all channels, at sample

time one. The same initial conditions are used for each channel, but different initial condition values are used within a channel.

#### Case 4 – Use Different Initial Conditions for Each Channel and Within a Channel

Enter a cell array for your initial condition values. The cell array must be the same size as your input signal. Each cell of the cell array represents the delay values for one channel, and must be a vector of size equal to the delay value. If you have a vector or scalar input and a scalar delay value, you can enter the initial conditions as a matrix.

For example, suppose your input is a matrix and you set the **Input processing** parameter to **Elements as channels (sample based)**.

$$[1 \ 1], [2 \ 2], [3 \ 3], \dots$$

You want the initial conditions of your two channel signal to be different for each channel and along each channel:

- 1 Set the **Delay (samples)** parameter to 2.
- 2 Select the **Specify different initial conditions for each channel** and **Specify different initial conditions within a channel** check boxes.
- 3 Set the **Initial conditions** parameter to `[10 20; 30 40]`.

The output of the delay block is

$$[10 \ 20], [30 \ 40], [1 \ 1], [2 \ 2], \dots$$

The first row of the initial conditions vector is the output at sample time zero. The second row of the initial conditions vector is the output at sample time one. Different initial conditions are used for each channel and within the channels.

In addition, suppose your input is a matrix and you set the **Input processing** parameter to **Elements as channels (sample based)**.

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}, \begin{bmatrix} 3 & 3 \\ 3 & 3 \end{bmatrix}, \dots$$

You want the initial conditions of your two-channel signal to be different for each channel and along each channel:

- 1 Set the **Delay (samples)** parameter to 2.
- 2 Select the **Specify different initial conditions for each channel** and the **Specify different initial conditions within a channel** check boxes.
- 3 Set the **Initial conditions** parameter to `{[11 15] [12 16]; [13 17] [14 18]}`. The dimensions of the cell array match the dimensions of the input. Also, each element of the cell array represents the initial conditions within one channel.

The output of the delay block is

$$\begin{bmatrix} 11 & 12 \\ 13 & 14 \end{bmatrix}, \begin{bmatrix} 15 & 16 \\ 17 & 18 \end{bmatrix}, \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}, \dots$$

Each element of the cell array represents the initial conditions within a channel. The first element, a vector, represents the initial conditions within channel 1. The second element, a

vector, represents the initial conditions within channel 2, and so on. Different initial conditions are used for each channel and within the channels.

## Parameters

### Input processing

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

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

---

**Note** The option **Inherit from input** (this choice will be removed - see release notes) will be removed in a future release. See “Frame-Based Processing” in the *DSP System Toolbox Release Notes* for more information.

---

### Delay units

Select whether you want to delay your input by a specified number of **Samples** or **Frames**. This parameter appears only when you set the **Input processing** parameter to **Columns as channels (frame based)**.

### Delay (samples) or Delay (frames)

See “Sample-Based Processing” on page 2-373 and “Frame-Based Processing” on page 2-373 for a description of what format to use for each configuration of the block dialog.

### Specify different initial conditions for each channel

Select this check box when you want the initial conditions to vary across the channels. When you do not select this check box, the initial conditions are the same across the channels.

### Specify different initial conditions within a channel

Select this check box when you want the initial conditions to vary within the channels. When you do not select this check box, the initial conditions are the same within the channels.

### Initial conditions

Enter a scalar, vector, matrix, or cell array of initial condition values, depending on your choice for the **Specify different initial conditions for each channel** and **Specify different initial conditions within a channel** check boxes. See “Sample-Based Processing” on page 2-373 and “Frame-Based Processing” on page 2-373 for a description of what format to use for each configuration of the block dialog.

### Reset port

Determines the reset event that causes the block to reset the delay. For more information, see “Resetting the Delay” on page 2-374.

## Supported Data Types

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

- Fixed point (signed and unsigned)
- Boolean
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

## **Version History**

**Introduced before R2006a**

### **Extended Capabilities**

#### **C/C++ Code Generation**

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

Generated code relies on the memcpy or memset function (`string.h`) under certain conditions.

#### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

For more information on implementations, properties, and restrictions for HDL code generation, see the "HDL Code Generation" section of the Delay page.

#### **Fixed-Point Conversion**

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

### **See Also**

`dsp.Delay` | Unit Delay | Variable Fractional Delay | Variable Integer Delay

# Delay Line

Rebuffer sequence of inputs



**Libraries:**

DSP System Toolbox / Signal Management / Buffers

## Description

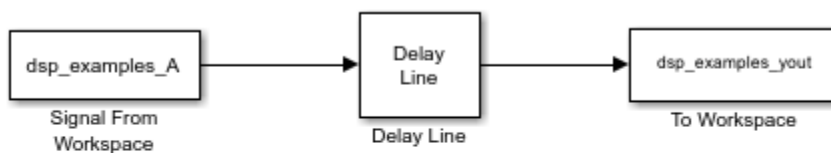
The Delay Line block rebuffers a sequence of  $M_i$ -by- $N$  matrix inputs into a sequence of  $M_o$ -by- $N$  matrix outputs, where  $M_o$  is the output frame size you specify in the **Delay line size** parameter. Depending on whether  $M_o$  is greater than, less than, or equal to the input frame size,  $M_i$ , the output frames can be underlapped or overlapped. The block always performs frame-based processing and rebuffers each of the  $N$  input channels independently.

## Examples

### Rebuffer Signal Using Delay Line

In this example, the Delay Line block rebuffers a signal with frame size 4 to a frame size of 3 using the Delay Line block.

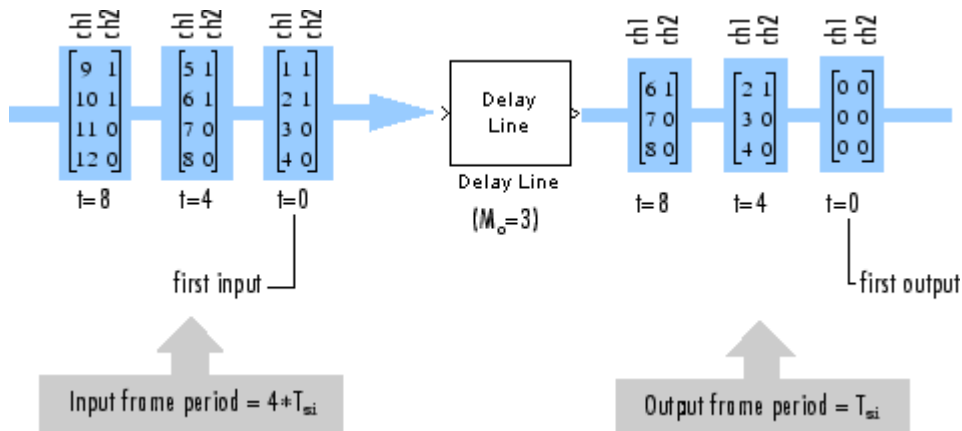
Open the model. The **Delay line size** parameter is set to 3.



Copyright 2006-2023 The MathWorks, Inc.

**Note:** This model creates workspace variables called "dsp\_examples\_A" and "dsp\_examples\_yout".

Run the model. The first output frame is all zeros because the **Initial conditions** parameter is set to zero. Because the input frame size of 4 is larger than the output frame size of 3, only the last three samples in each input frame are propagated to the corresponding output frame. The frame periods of the input and output are the same, and the output sample period is  $T_{si}(M_i/M_o)$ , or 4/3 the input sample period.



## Ports

### Input

**In** — Input signal  
vector | matrix

Specify the input signal as a vector or a matrix of size  $M_i$ -by- $N$ .

This port is unnamed until you select the **Show En\_Out port for selectively enabling output** parameter.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

**En\_Out** — Linearize the block output  
true | false

Specify whether to linearize the block output.

If the input value to the **En\_Out** port is a Boolean 1 (or true), the block linearizes the output and the output is valid. This setting allows the block to be more efficient when the tapped Delay Line output is not required at each sample time.

Note that when the input value to the **En\_Out** port is a Boolean 0 (or false), the block can give different results depending on the state of the model. The results can appear to match valid results or can be invalid, and they cannot be predicted. You should ignore the block output in all cases when the input to the **En\_Out** port is 0.

### Dependencies

To enable this port, select the **Show En\_Out port for selectively enabling output** parameter.

Data Types: Boolean

### Output

**Out** — Rebuffered output  
vector | matrix

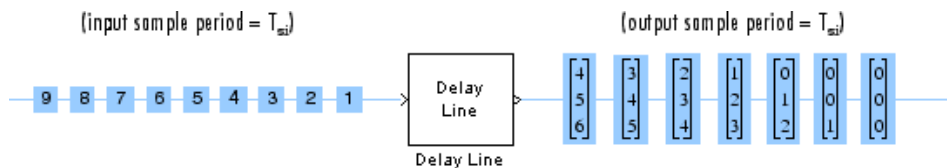
Rebuffered output, returned as a vector or a matrix of size  $M_o$ -by- $N$ .



When  $M_o > M_i$ , the output frame overlap is the difference between the output and input frame size,  $M_o - M_i$ . When  $M_o < M_i$ , the output is underlapped and the Delay Line block discards the first  $M_i - M_o$  samples of each input frame so that only the last  $M_o$  samples are buffered into the corresponding output frame. When  $M_o = M_i$ , the output data is identical to the input data, but is delayed by the latency of the block. Due to the block latency, the outputs are always delayed by one frame, the entries of which you specify in the **Initial conditions** parameter.

The output frame period is equal to the input frame period ( $T_{fo} = T_{fi}$ ). The output sample period,  $T_{so}$ , is therefore equal to  $T_{fi}/M_o$ , or equivalently,  $T_{si}(M_i/M_o)$

In the most typical use, each output differs from the preceding output by only one sample, as illustrated below for scalar input.



Note that the first output of the block in the example above is all zeros; this is because the **Initial Conditions** parameter is set to zero.

This port is unnamed until you select the **Show En\_Out port for selectively enabling output** parameter.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

## Parameters

**Delay line size** — Delay line size

64 (default) | positive integer

Specify the number of rows  $M_o$  in output matrix as a positive integer.

**Initial conditions** — Initial conditions

0 (default) | scalar | vector | matrix

Specify the value of the block initial output as a scalar, vector, or a matrix. The Delay Line block buffer is initialized to the value specified by the **Initial conditions** parameter. The block outputs this buffer at the first simulation step ( $t=0$ ).

When the block outputs a vector, the **Initial conditions** can be a vector of the same size, or a scalar value to be repeated across all elements of the initial output. When the block outputs a matrix, the **Initial conditions** can be a matrix of the same size, a vector (of length equal to the number of matrix rows) to be repeated across all columns of the initial output, or a scalar to be repeated across all elements of the initial output.

**Allow directfeedthrough** — Allow directfeedthrough

off (default) | on

When you select this check box, the input data is not delayed by an extra frame before it is available at the output buffer. Instead, the input data is available immediately at the output port of the block.

**Show En\_Out port for selectively enabling output** — Show En\_Out port for selectively enabling output

off (default) | on

When you select this check box, the **En\_Out** input port appears on the block icon. This block uses a circular buffer internally even though the output is linear. This means that for valid output, data from the circular buffer has to be linearized. The **En\_Out** port determines whether or not a valid output needs to be computed based on the value of its Boolean input. If the input value to the **En\_Out** port is 1, the block output is linearized, and thus is valid. Otherwise, the output is not linearized, and is invalid. This allows the block to be more efficient when the tapped Delay Line output is not required at each sample time.

Note that when the input value to the **En\_Out** port is 0, the block can give different results depending on the state of the model. The results can appear to match valid results or can be invalid, and they cannot be predicted. You should ignore the block output in all cases when the input to the **En\_Out** port is 0.

**Hold previous value when the output is disabled** — Hold previous value when the output is disabled

off (default) | on

Use this parameter to specify the block output at those time steps when the internal state buffer is not being linearized to output valid data.

When you do not select this check box, the block memory is free to be used by other parts of the model, and the signal on the output port is invalid. When you select this check box, the most recent valid value is held on the output port, and slightly more memory is used by the block.

#### Dependencies

To enable this parameter, select **Show En\_Out port for selectively enabling output**.

## Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | Boolean   double   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no  |
| <b>Multidimensional Signals</b> | no  |
| <b>Variable-Size Signals</b>    | no  |
| <b>Zero-Crossing Detection</b>  | no  |

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

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

Generated code relies on the `memcpy` or `memset` function (`string.h`) under certain conditions.

### Fixed-Point Conversion

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

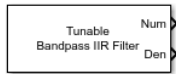
## See Also

### Blocks

Buffer

# Tunable Bandpass IIR Filter

Design IIR bandpass filter with tunable filter specifications



## Libraries:

DSP System Toolbox / Filtering / Filter Sources

## Description

Use the Tunable Bandpass IIR Filter block to design an IIR bandpass filter using the Butterworth, Chebyshev Type I and Chebyshev Type II design methods. You can tune the filter design specifications such as the filter order, 3-dB cutoff frequencies, passband ripple, and stopband attenuation while the simulation is running. When the values of these parameters change, the block redesigns the filter and outputs the numerator and denominator coefficients in the fourth-order section form.

## Examples

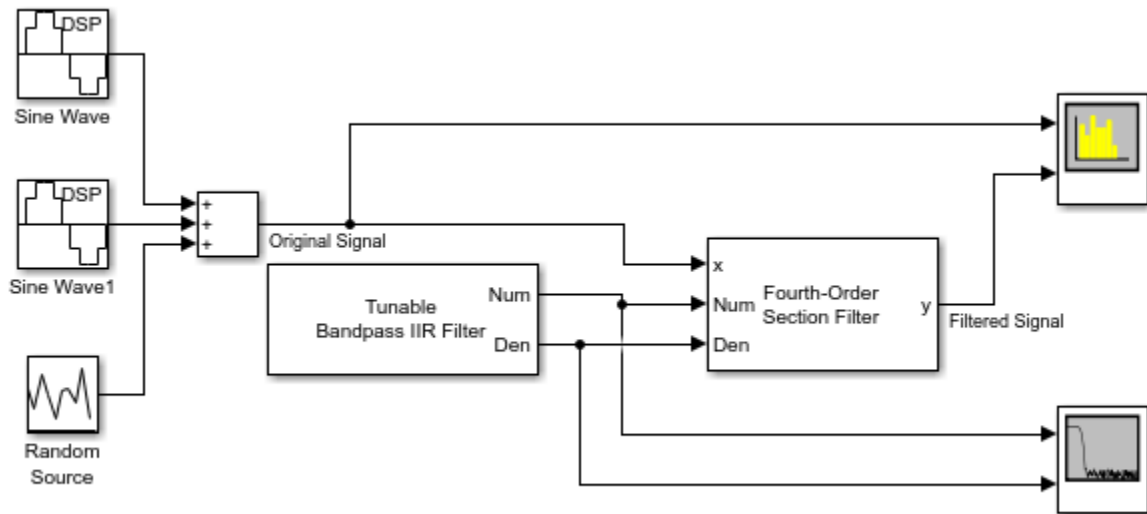
### Filter Noisy Signal Using Fourth-Order Section (FOS) Filter in Simulink

Filter a noisy sinusoidal signal using the Fourth-Order Section Filter block. The numerator and denominator coefficients of the fourth-order section filter are obtained using the Tunable Bandpass IIR Filter block. You can tune the frequency specifications of this filter during simulation.

### Open and Run the Model

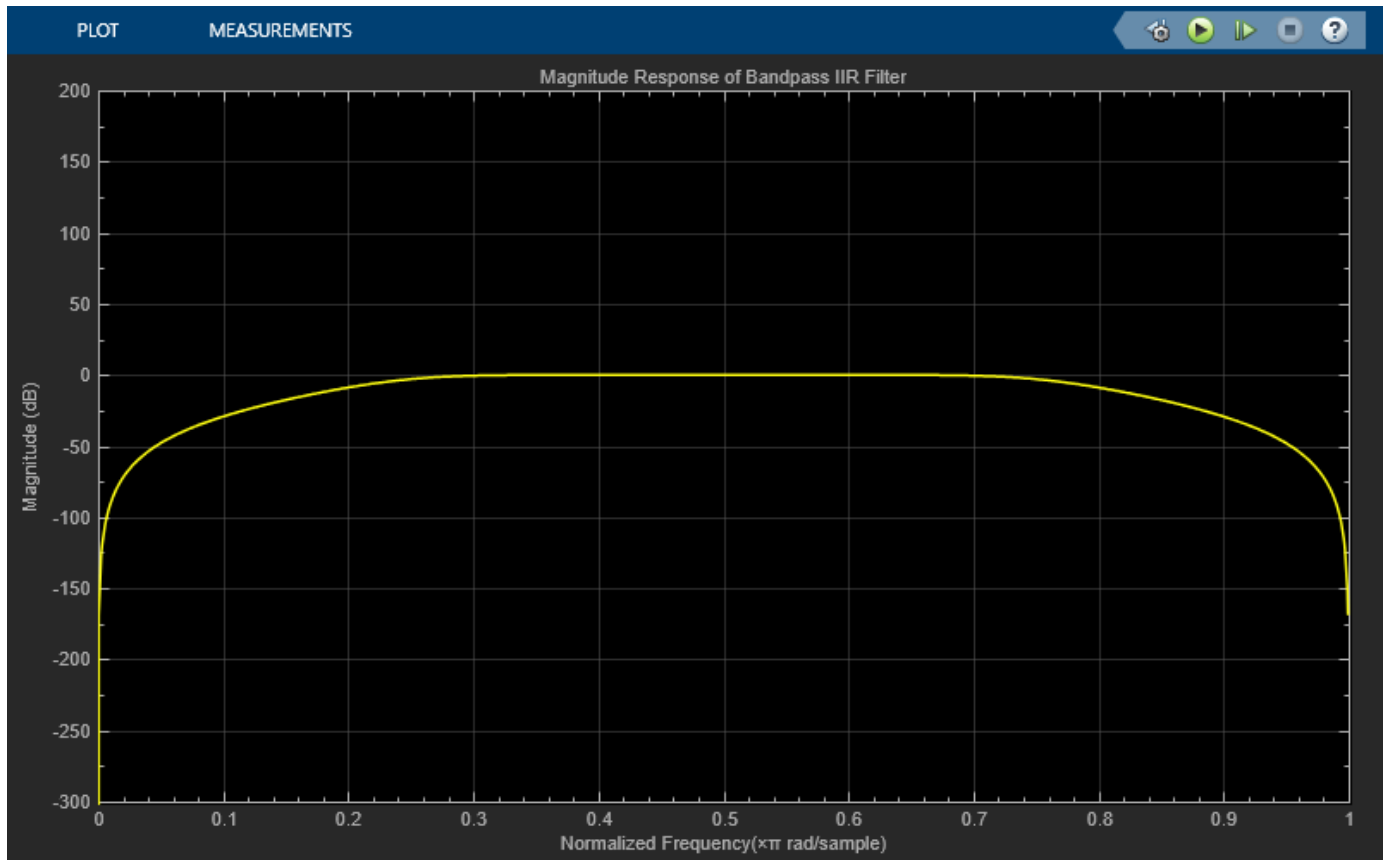
The input signal is a sum of two sine waves with the frequencies of 100 Hz and 350 Hz. The sample rate is 1000 Hz and the number of samples in each frame is 1024. Add zero-mean white Gaussian noise with a variance of  $1e-4$  to the sum of the sine waves.

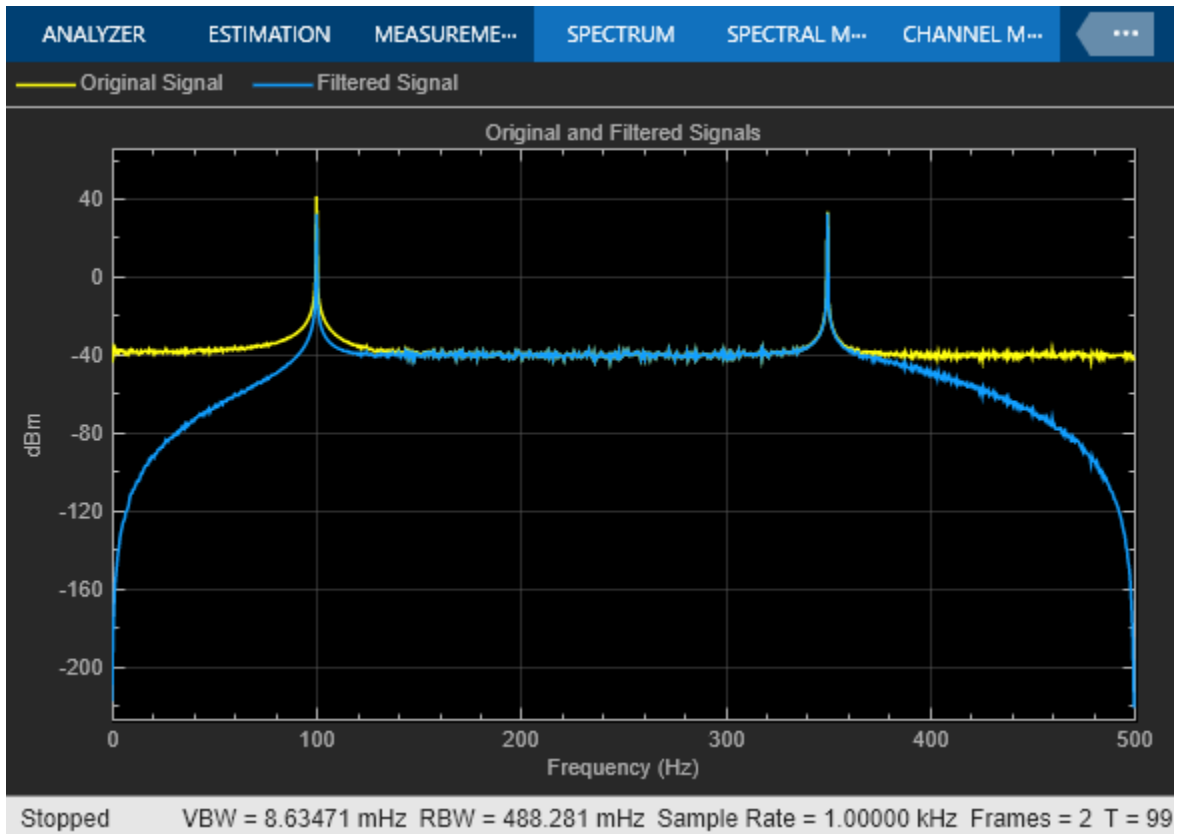
The Tunable Bandpass IIR Filter block designs a sixth order bandpass IIR filter with the first and second 3-dB cutoff frequencies at  $0.25 \pi$  rad/sample and  $0.75 \pi$  rad/sample, respectively. Visualize the frequency response of the filter using Filter Visualizer.



Copyright 2021-2023 The MathWorks, Inc.

Filter the noisy sinusoidal signal using the Fourth-Order Section Filter block. Visualize the original sinusoidal signal and the filtered signal using the Spectrum Analyzer.

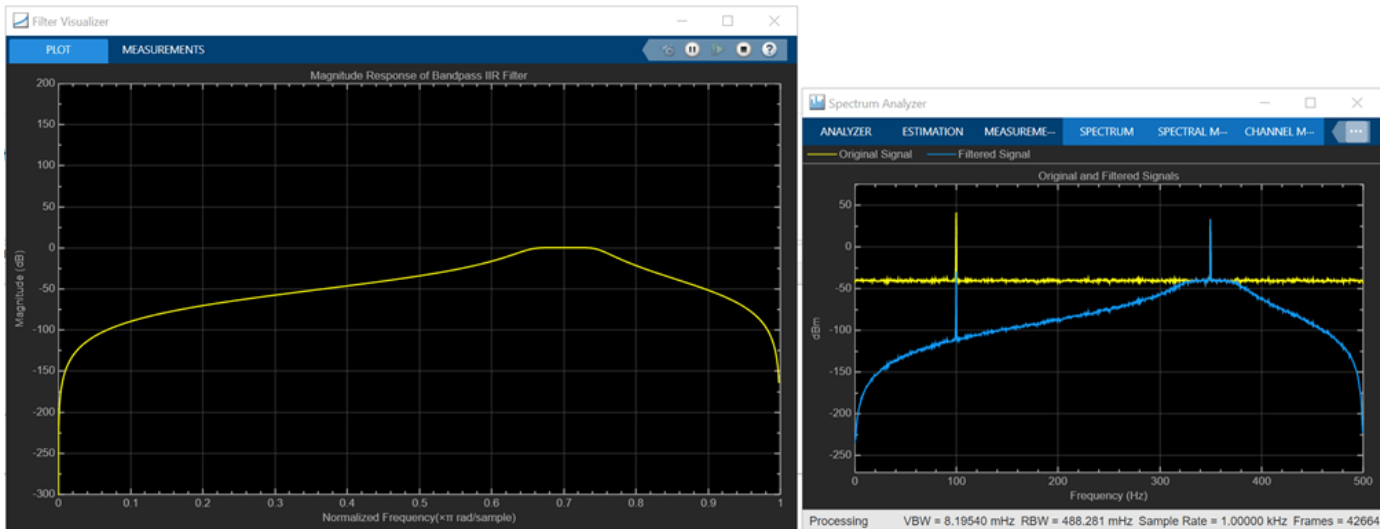




### Tuning Frequency Specification of FOS Filter

During simulation, you can tune the frequency specifications of the FOS filter by tuning the frequency parameters in the Tunable Bandpass IIR Filter block. The filter response updates accordingly.

Change the first 3-dB cutoff frequency to  $0.65 \pi$  rad/sample in the Tunable Bandpass IIR Filter block. The first tone of the sinusoidal signal is attenuated as it no longer falls in the passband frequency of the filter.



## Ports

### Input

**N** — Filter order  
even positive integer

Specify the filter order as an even positive integer.

### Dependency

To enable this port, select the **Specify filter order from input port** parameter.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**Fc1** — First 3-dB cutoff frequency  
nonnegative scalar  $\leq$  **Fc2**

Specify the first 3-dB cutoff frequency **Fc1** of the filter in normalized frequency units as a nonnegative scalar less than or equal to the second 3-dB cutoff frequency **Fc2**.

### Dependency

To enable this port, select the **Specify first 3-dB cutoff frequency from input port** parameter.

Data Types: single | double

**Fc2** — Second 3-dB cutoff frequency  
**Fc1**  $\leq$  positive scalar  $\leq$  1

Specify the second 3-dB cutoff frequency of the filter in normalized frequency units as a positive scalar greater than or equal to **Fc1** and less than or equal to 1.

### Dependency

To enable this port, select the **Specify second 3-dB cutoff frequency from input port** parameter.

Data Types: single | double

**Ap** — Passband ripple in dB  
nonnegative scalar

Specify the passband ripple of the Chebyshev Type I filter as a nonnegative scalar in dB.

**Dependency**

To enable this port:

- Set the **Design method** parameter to Chebyshev Type I.
- Select the **Specify the passband ripple from input port** parameter.

Data Types: single | double

**Ast** — Stopband attenuation in dB  
nonnegative scalar

Specify the stopband attenuation of the Chebyshev Type II filter as a nonnegative scalar in dB.

**Dependency**

To enable this port:

- Set the **Design method** parameter to Chebyshev Type II.
- Select the **Specify the stopband attenuation from input port** parameter.

Data Types: single | double

**Output**

**Num** — Numerator coefficients  
 $P$ -by-5 matrix

Numerator coefficients **b**, returned as a  $P$ -by-5 matrix (fourth-order section form), where  $P$  is the number of filter sections.

$$\mathbf{b} = \begin{bmatrix} b_{01} & b_{11} & b_{21} & b_{31} & b_{41} \\ b_{02} & b_{12} & b_{22} & b_{32} & b_{42} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ b_{0P} & b_{1P} & b_{2P} & b_{3P} & b_{4P} \end{bmatrix}$$

This equation represents the fourth-order section filter in the transfer function form.

$$H(z) = \prod_{k=1}^P H_k(z) = \prod_{k=1}^P \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2} + b_{3k}z^{-3} + b_{4k}z^{-4}}{a_{0k} + a_{1k}z^{-1} + a_{2k}z^{-2} + a_{3k}z^{-3} + a_{4k}z^{-4}}$$

where,

- **a** -- Denominator coefficients matrix. For more details on how to specify this matrix, see the **Den** output port.
- $k$  -- Row index.

The maximum number of filter sections equals  $\text{ceil}(N_{max}/4)$ , where  $N_{max}$  is the value of the **Filter maximum order (must be even)** parameter.



When the actual filter order  $N$  is less than the maximum filter order  $N_{max}$ , the last  $\text{ceil}(N_{max}/4) - \text{ceil}(N/4)$  sections are trivial with coefficients  $[b_0, b_1, b_2, b_3, b_4] = [1, 0, 0, 0, 0]$ .

The data type of this port depends on the value of the **Output data type** parameter.

Data Types: `single` | `double`

**Den** — Denominator coefficients

$P$ -by-5 matrix

Denominator coefficients **a**, returned as a  $P$ -by-5 matrix (fourth-order section form), where  $P$  is the number of filter sections.

$$\mathbf{a} = \begin{bmatrix} a_{01} & a_{11} & a_{21} & a_{31} & a_{41} \\ a_{02} & a_{12} & a_{22} & a_{32} & a_{42} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{0P} & a_{1P} & a_{2P} & a_{3P} & a_{4P} \end{bmatrix}$$

This equation represents the fourth-order section filter in the transfer function form.

$$H(z) = \prod_{k=1}^P H_k(z) = \prod_{k=1}^P \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2} + b_{3k}z^{-3} + b_{4k}z^{-4}}{a_{0k} + a_{1k}z^{-1} + a_{2k}z^{-2} + a_{3k}z^{-3} + a_{4k}z^{-4}}$$

where,

- **b** -- Numerator coefficients matrix. For more details on how to specify this matrix, see the **Num** output port.
- $k$  -- Row index.

The maximum number of filter sections equals  $\text{ceil}(N_{max}/4)$ , where  $N_{max}$  is the value of the **Filter maximum order (must be even)** parameter.

The leading denominator coefficient  $a_0$  is always 1.

When the actual filter order  $N$  is less than the maximum filter order  $N_{max}$ , the last  $\text{ceil}(N_{max}/4) - \text{ceil}(N/4)$  sections are trivial with coefficients  $[a_0, a_1, a_2, a_3, a_4] = [1, 0, 0, 0, 0]$ .

The data type of this port depends on the value of the **Output data type** parameter.

Data Types: `single` | `double`

## Parameters

**Design method** — Filter design method

Butterworth (default) | Chebyshev Type I | Chebyshev Type II

Specify the filter design method as:

- Butterworth
- Chebyshev Type I
- Chebyshev Type II

**Specify filter order from input port** — Flag to specify filter order from input port

off (default) | on

Select this parameter to specify the filter order from the input port **N**. When you clear this parameter, you can specify the filter order in the block dialog box using the **Filter order (must be even)** parameter.

**Filter order (must be even)** — Filter order

6 (default) | even positive integer

Specify the filter order as an even positive integer less than or equal to the value of the **Filter maximum order (must be even)** parameter.

**Tunable:** Yes

**Dependency**

To enable this parameter, clear the **Specify filter order from input port** parameter.

**Filter maximum order (must be even)** — Maximum order of filter

10 (default) | even positive integer

Specify the maximum order of the filter as an even positive integer. The value you specify in the **Filter order (must be even)** parameter must be less than or equal to the value you specify in the **Filter maximum order (must be even)** parameter.

**Specify first 3-dB cutoff frequency from input port** — Specify first 3-dB cutoff frequency from input port

off (default) | on

Select this parameter to specify the first 3-dB filter cutoff frequency from the input port **Fc1**. When you clear this parameter, you can specify the first 3-dB cutoff frequency in the block dialog box using the **First 3-dB cutoff frequency** parameter.

**First 3-dB cutoff frequency** — First 3-dB cutoff frequency

0.25 (default) | positive scalar

Specify the first 3-dB cutoff frequency of the filter in normalized frequency units as a positive scalar less than or equal to the second 3-dB cutoff frequency.

**Tunable:** Yes

**Dependency**

To enable this parameter, clear the **Specify first 3-dB cutoff frequency from input port** parameter.

**Specify second 3-dB cutoff frequency from input port** — Specify second 3-dB cutoff frequency from input port

off (default) | on

Select this parameter to specify the second 3-dB filter cutoff frequency from the input port **Fc2**. When you clear this parameter, you can specify the second 3-dB cut-off frequency in the block dialog box using the **Second 3-dB cutoff frequency** parameter.

**Second 3-dB cutoff frequency** — Second 3-dB cutoff frequency

0.75 (default) | positive scalar

Specify the second 3-dB cutoff frequency of the filter in normalized frequency units as a positive scalar greater than or equal to the first 3-dB cutoff frequency and less than or equal to 1.

**Tunable:** Yes

**Dependency**

To enable this parameter, clear the **Specify second 3-dB cutoff frequency from input port** parameter.

**Specify the passband ripple from input port** — Specify passband ripple from input port

off (default) | on

Select this parameter to specify passband ripple from the input port **Ap**. When you clear this parameter, you can specify the passband ripple in the block dialog box using the **Passband ripple (dB)** parameter.

**Dependency**

To enable this parameter, set the **Design method** parameter to Chebyshev Type I.

**Passband ripple (dB)** — Passband ripple in dB

1 (default) | positive scalar

Specify the passband ripple of the Chebyshev Type I filter as a positive scalar in dB.

**Tunable:** Yes

**Dependency**

To enable this parameter:

- Set the **Design method** parameter to Chebyshev Type I.
- Clear the **Specify the passband ripple from input port** parameter.

**Specify the stopband attenuation from input port** — Specify stopband attenuation from input port

off (default) | on

Select this parameter to specify stopband attenuation from the input port **Ast**. When you clear this parameter, you can specify the stopband attenuation in the block dialog box using the **Stopband attenuation (dB)** parameter.

**Dependency**

To enable this parameter, set the **Design method** parameter to Chebyshev Type II.

**Stopband attenuation (dB)** — Stopband attenuation in dB

60 (default) | positive scalar

Specify the stopband attenuation of the Chebyshev Type II filter as a positive scalar in dB.

**Tunable:** Yes

**Dependency**

To enable this parameter:

- Set the **Design method** parameter to Chebyshev Type II.
- Clear the **Specify the stopband attenuation from input port** parameter.

**Sample time** — Sample time in seconds

-1 (default) | positive scalar | Inf

Specify the sample time as -1 (inherited), Inf (constant sample time), or a positive scalar. This parameter indicates when during the simulation the block produces the output and updates its internal state. For more details, see “What Is Sample Time?” (Simulink).

**Output data type** — Data type of filter coefficients

double (default) | single

Specify the data type of the filter coefficients that the block outputs through the **Num** and **Den** ports as double or single.

**Simulate using** — Type of simulation to run

Code generation (default) | Interpreted execution

Specify the type of simulation to run as one of the following:

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

## Block Characteristics

|                                 |                           |
|---------------------------------|---------------------------|
| <b>Data Types</b>               | double   integer   single |
| <b>Direct Feedthrough</b>       | no                        |
| <b>Multidimensional Signals</b> | no                        |
| <b>Variable-Size Signals</b>    | no                        |

|                                |    |
|--------------------------------|----|
| <b>Zero-Crossing Detection</b> | no |
|--------------------------------|----|

## Version History

Introduced in R2023a

## References

[1] Orfanidis, Sophocles J. *Introduction to Signal Processing*. Prentice Hall, 1996.

## Extended Capabilities

### C/C++ Code Generation

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

## See Also

### Functions

butter | cheby1 | cheby2

### Blocks

Tunable Bandstop IIR Filter | Bandpass Filter | Complex Bandpass Decimator

# Tunable Bandstop IIR Filter

Design IIR bandstop filter with tunable filter specifications

**Libraries:**

DSP System Toolbox / Filtering / Filter Sources

## Description

Use the Tunable Bandstop IIR Filter block to design an IIR bandstop filter using the Butterworth, Chebyshev Type I and Chebyshev Type II design methods. You can tune the filter design specifications such as the filter order, 3-dB cutoff frequencies, passband ripple, and stopband attenuation while the simulation is running. When the values of these parameters change, the block redesigns the filter and outputs the numerator and denominator coefficients in the fourth-order section form.

## Examples

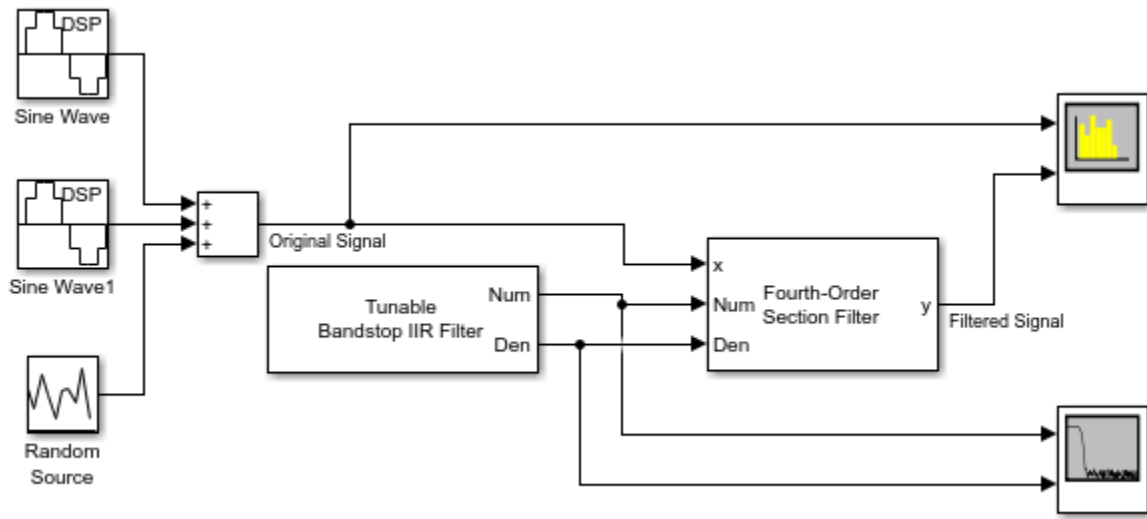
### Filter Noisy Signal Using Fourth-Order Section (FOS) Filter in Simulink

Filter a noisy sinusoidal signal using the Fourth-Order Section Filter block. The numerator and denominator coefficients of the fourth-order section filter are obtained using the Tunable Bandstop IIR Filter block. You can tune the frequency specifications of this filter during simulation.

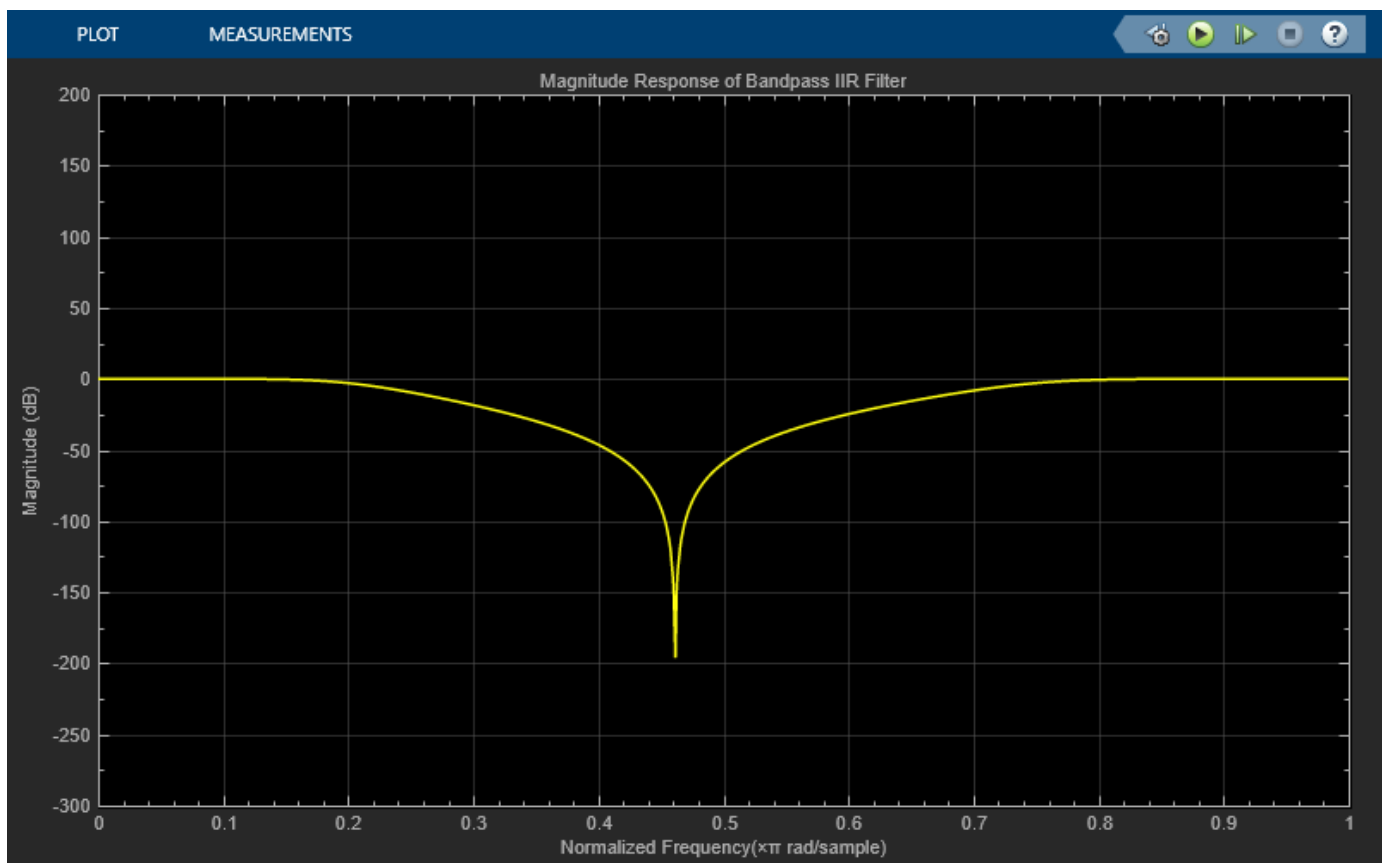
#### Open and Run the Model

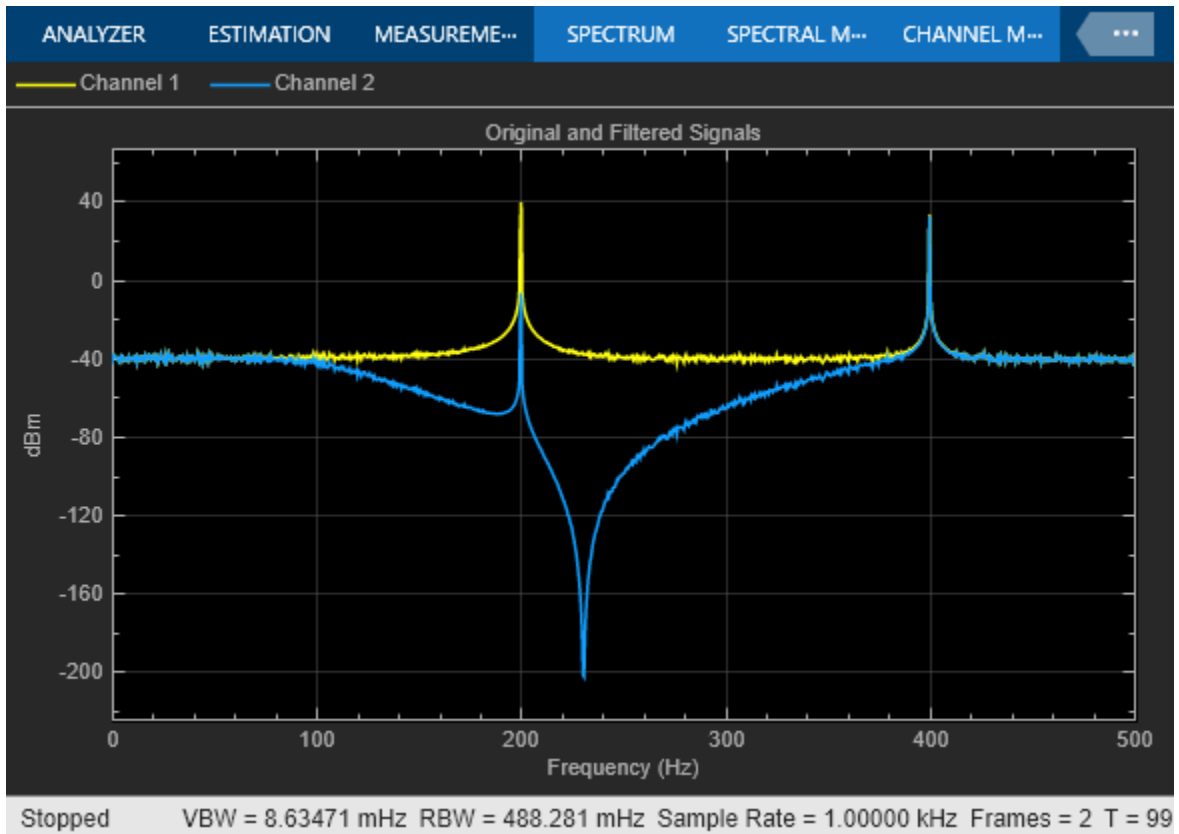
The input signal is a sum of two sine waves with the frequencies of 200 Hz and 400 Hz. The sample rate is 1000 Hz and the number of samples in each frame is 1024. Add zero-mean white Gaussian noise with a variance of  $1e-4$  to the sum of the sine waves.

The Tunable Bandstop IIR Filter block designs a sixth order bandpass IIR filter with the first and second 3-dB cutoff frequencies at  $0.2 \pi$  rad/sample and  $0.75 \pi$  rad/sample, respectively. Visualize the frequency response of the filter using Filter Visualizer.



Filter the noisy sinusoidal signal using the Fourth-Order Section Filter block. Visualize the original sinusoidal signal and the filtered signal using the Spectrum Analyzer. The first tone is attenuated as it falls in the stopband region of the filter while the second tone remains unaffected as it falls in the passband region of the filter.



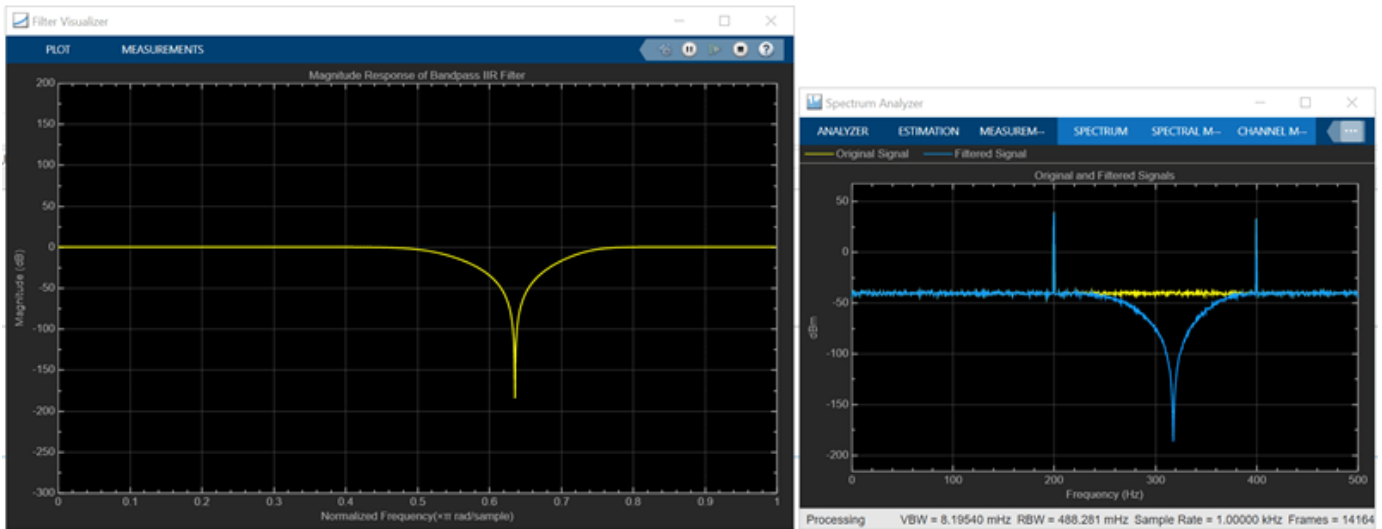


### Tuning Frequency Specification of FOS Filter

During simulation, you can tune the frequency specifications of the FOS filter by tuning the frequency parameters in the Tunable Bandstop IIR Filter block. The filter response updates accordingly.

Change the first 3-dB cutoff frequency to  $0.5 \pi$  rad/sample in the Tunable Bandstop IIR Filter block. The first tone of the sinusoidal signal now falls in the passband region and is therefore unattenuated.





## Ports

### Input

**N** — Filter order  
even positive integer

Specify the filter order as an even positive integer.

### Dependency

To enable this port, select the **Specify filter order from input port** parameter.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**Fc1** — First 3-dB cutoff frequency  
nonnegative scalar  $\leq$  **Fc2**

Specify the first 3-dB cutoff frequency **Fc1** of the filter in normalized frequency units as a nonnegative scalar less than or equal to the second 3-dB cutoff frequency **Fc2**.

### Dependency

To enable this port, select the **Specify first 3-dB cutoff frequency from input port** parameter.

Data Types: `single` | `double`

**Fc2** — Second 3-dB cutoff frequency  
**Fc1**  $\leq$  positive scalar  $\leq$  1

Specify the second 3-dB cutoff frequency of the filter in normalized frequency units as a positive scalar greater than or equal to **Fc1** and less than or equal to 1.

### Dependency

To enable this port, select the **Specify second 3-dB cutoff frequency from input port** parameter.

Data Types: `single` | `double`

**Ap** — Passband ripple in dB  
nonnegative scalar

Specify the passband ripple of the Chebyshev Type I filter as a nonnegative scalar in dB.

**Dependency**

To enable this port:

- Set the **Design method** parameter to Chebyshev Type I.
- Select the **Specify the passband ripple from input port** parameter.

Data Types: single | double

**Ast** — Stopband attenuation in dB  
nonnegative scalar

Specify the stopband attenuation of the Chebyshev Type II filter as a nonnegative scalar in dB.

**Dependency**

To enable this port:

- Set the **Design method** parameter to Chebyshev Type II.
- Select the **Specify the stopband attenuation from input port** parameter.

Data Types: single | double

**Output**

**Num** — Numerator coefficients  
 $P$ -by-5 matrix

Numerator coefficients **b**, returned as a  $P$ -by-5 matrix (fourth-order section form), where  $P$  is the number of filter sections.

$$\mathbf{b} = \begin{bmatrix} b_{01} & b_{11} & b_{21} & b_{31} & b_{41} \\ b_{02} & b_{12} & b_{22} & b_{32} & b_{42} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ b_{0P} & b_{1P} & b_{2P} & b_{3P} & b_{4P} \end{bmatrix}$$

This equation represents the fourth-order section filter in the transfer function form.

$$H(z) = \prod_{k=1}^P H_k(z) = \prod_{k=1}^P \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2} + b_{3k}z^{-3} + b_{4k}z^{-4}}{a_{0k} + a_{1k}z^{-1} + a_{2k}z^{-2} + a_{3k}z^{-3} + a_{4k}z^{-4}}$$

where,

- **a** -- Denominator coefficients matrix. For more details on how to specify this matrix, see the **Den** output port.
- $k$  -- Row index.

The maximum number of filter sections equals  $\text{ceil}(N_{max}/4)$ , where  $N_{max}$  is the value of the **Filter maximum order (must be even)** parameter.

When the actual filter order  $N$  is less than the maximum filter order  $N_{max}$ , the last  $\text{ceil}(N_{max}/4) - \text{ceil}(N/4)$  sections are trivial with coefficients  $[b_0, b_1, b_2, b_3, b_4] = [1, 0, 0, 0, 0]$ .

The data type of this port depends on the value of the **Output data type** parameter.

Data Types: `single` | `double`

**Den** — Denominator coefficients

$P$ -by-5 matrix

Denominator coefficients **a**, returned as a  $P$ -by-5 matrix (fourth-order section form), where  $P$  is the number of filter sections.

$$\mathbf{a} = \begin{bmatrix} a_{01} & a_{11} & a_{21} & a_{31} & a_{41} \\ a_{02} & a_{12} & a_{22} & a_{32} & a_{42} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{0P} & a_{1P} & a_{2P} & a_{3P} & a_{4P} \end{bmatrix}$$

This equation represents the fourth-order section filter in the transfer function form.

$$H(z) = \prod_{k=1}^P H_k(z) = \prod_{k=1}^P \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2} + b_{3k}z^{-3} + b_{4k}z^{-4}}{a_{0k} + a_{1k}z^{-1} + a_{2k}z^{-2} + a_{3k}z^{-3} + a_{4k}z^{-4}}$$

where,

- **b** -- Numerator coefficients matrix. For more details on how to specify this matrix, see the **Num** output port.
- $k$  -- Row index.

The maximum number of filter sections equals  $\text{ceil}(N_{max}/4)$ , where  $N_{max}$  is the value of the **Filter maximum order (must be even)** parameter.

The leading denominator coefficient  $a_0$  is always 1.

When the actual filter order  $N$  is less than the maximum filter order  $N_{max}$ , the last  $\text{ceil}(N_{max}/4) - \text{ceil}(N/4)$  sections are trivial with coefficients  $[a_0, a_1, a_2, a_3, a_4] = [1, 0, 0, 0, 0]$ .

The data type of this port depends on the value of the **Output data type** parameter.

Data Types: `single` | `double`

## Parameters

**Design method** — Filter design method

Butterworth (default) | Chebyshev Type I | Chebyshev Type II

Specify the filter design method as:

- Butterworth
- Chebyshev Type I
- Chebyshev Type II

**Specify filter order from input port** — Flag to specify filter order from input port

off (default) | on

Select this parameter to specify the filter order from the input port **N**. When you clear this parameter, you can specify the filter order in the block dialog box using the **Filter order (must be even)** parameter.

**Filter order (must be even)** — Filter order

6 (default) | even positive integer

Specify the filter order as an even positive integer less than or equal to the value of the **Filter maximum order (must be even)** parameter.

**Tunable:** Yes

**Dependency**

To enable this parameter, clear the **Specify filter order from input port** parameter.

**Filter maximum order (must be even)** — Maximum order of filter

10 (default) | even positive integer

Specify the maximum order of the filter as an even positive integer. The value you specify in the **Filter order (must be even)** parameter must be less than or equal to the value you specify in the **Filter maximum order (must be even)** parameter.

**Specify first 3-dB cutoff frequency from input port** — Specify first 3-dB cutoff frequency from input port

off (default) | on

Select this parameter to specify the first 3-dB filter cutoff frequency from the input port **Fc1**. When you clear this parameter, you can specify the first 3-dB cutoff frequency in the block dialog box using the **First 3-dB cutoff frequency** parameter.

**First 3-dB cutoff frequency** — First 3-dB cutoff frequency

0.25 (default) | positive scalar

Specify the first 3-dB cutoff frequency of the filter in normalized frequency units as a positive scalar less than or equal to the second 3-dB cutoff frequency.

**Tunable:** Yes

**Dependency**

To enable this parameter, clear the **Specify first 3-dB cutoff frequency from input port** parameter.

**Specify second 3-dB cutoff frequency from input port** — Specify second 3-dB cutoff frequency from input port

off (default) | on

Select this parameter to specify the second 3-dB filter cutoff frequency from the input port **Fc2**. When you clear this parameter, you can specify the second 3-dB cut-off frequency in the block dialog box using the **Second 3-dB cutoff frequency** parameter.

**Second 3-dB cutoff frequency** — Second 3-dB cutoff frequency

0.75 (default) | positive scalar

Specify the second 3-dB cutoff frequency of the filter in normalized frequency units as a positive scalar greater than or equal to the first 3-dB cutoff frequency and less than or equal to 1.

**Tunable:** Yes

**Dependency**

To enable this parameter, clear the **Specify second 3-dB cutoff frequency from input port** parameter.

**Specify the passband ripple from input port** — Specify passband ripple from input port

off (default) | on

Select this parameter to specify passband ripple from the input port **Ap**. When you clear this parameter, you can specify the passband ripple in the block dialog box using the **Passband ripple (dB)** parameter.

**Dependency**

To enable this parameter, set the **Design method** parameter to Chebyshev Type I.

**Passband ripple (dB)** — Passband ripple in dB

1 (default) | positive scalar

Specify the passband ripple of the Chebyshev Type I filter as a positive scalar in dB.

**Tunable:** Yes

**Dependency**

To enable this parameter:

- Set the **Design method** parameter to Chebyshev Type I.
- Clear the **Specify the passband ripple from input port** parameter.

**Specify the stopband attenuation from input port** — Specify stopband attenuation from input port

off (default) | on

Select this parameter to specify stopband attenuation from the input port **Ast**. When you clear this parameter, you can specify the stopband attenuation in the block dialog box using the **Stopband attenuation (dB)** parameter.

**Dependency**

To enable this parameter, set the **Design method** parameter to Chebyshev Type II.

**Stopband attenuation (dB)** — Stopband attenuation in dB

60 (default) | positive scalar

Specify the stopband attenuation of the Chebyshev Type II filter as a positive scalar in dB.

**Tunable:** Yes

**Dependency**

To enable this parameter:

- Set the **Design method** parameter to Chebyshev Type II.
- Clear the **Specify the stopband attenuation from input port** parameter.

**Sample time** — Sample time in seconds

-1 (default) | positive scalar | Inf

Specify the sample time as -1 (inherited), Inf (constant sample time), or a positive scalar. This parameter indicates when during the simulation the block produces the output and updates its internal state. For more details, see “What Is Sample Time?” (Simulink).

**Output data type** — Data type of filter coefficients

double (default) | single

Specify the data type of the filter coefficients that the block outputs through the **Num** and **Den** ports as double or single.

**Simulate using** — Type of simulation to run

Code generation (default) | Interpreted execution

Specify the type of simulation to run as one of the following:

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

## Block Characteristics

|                                 |                           |
|---------------------------------|---------------------------|
| <b>Data Types</b>               | double   integer   single |
| <b>Direct Feedthrough</b>       | no                        |
| <b>Multidimensional Signals</b> | no                        |
| <b>Variable-Size Signals</b>    | no                        |

|                                |    |
|--------------------------------|----|
| <b>Zero-Crossing Detection</b> | no |
|--------------------------------|----|

## Version History

Introduced in R2023a

## References

[1] Orfanidis, Sophocles J. *Introduction to Signal Processing*. Prentice Hall, 1996.

## Extended Capabilities

### C/C++ Code Generation

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

## See Also

### Functions

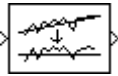
butter | cheby1 | cheby2

### Blocks

Tunable Bandpass IIR Filter | Bandstop Filter | Complex Bandpass Decimator

## Detrend

Remove linear trend from vectors



**Libraries:**

DSP System Toolbox / Statistics

### Description

The Detrend block removes a linear trend from the length- $M$  input vector,  $u$ , by subtracting the straight line that best fits the data in the least squares sense.

The least squares line,  $\hat{u} = ax + b$ , is the line with parameters  $a$  and  $b$  that minimizes the quantity

$$\sum_{i=1}^M (u_i - \hat{u}_i)^2$$

for  $M$  evenly-spaced values of  $x$ , where  $u_i$  is the  $i$ th element in the input vector. The output,  $y = u - \hat{u}$ , is always an  $M$ -by-1 column vector.

### Ports

#### Input

**Input** — Input data  
vector

Input data, specified as a vector. The block removes the straight line that best fits the data in the least squares sense.

Data Types: `single` | `double`

#### Output

**Output** — Output data  
column vector

Output data, returned as a column vector. The block returns the remaining data after removing the straight line that best fits the data in the least squares sense.

Data Types: `single` | `double`

### Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | <code>double</code>   <code>single</code> |
| <b>Multidimensional Signals</b> | No  |
| <b>Variable-Size Signals</b>    | No  |



## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

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

Generated code relies on the memcpy or memset function (`string.h`) under certain conditions.

## See Also

### Blocks

Cumulative Sum | Difference | Least Squares Polynomial Fit | Unwrap

### Functions

detrend

## Difference

Compute element-to-element difference along specified dimension of input



### Libraries:

DSP System Toolbox / Math Functions / Math Operations

## Description

The Difference block computes the difference between adjacent elements in rows, columns, or a specified dimension of the input array  $u$ . You can configure the block to compute the difference only within the current input, or across consecutive inputs (running difference).

## Ports

### Input

**Port\_1** — Input signal

vector | matrix |  $N$ -D array

Input signal, specified as a vector, matrix, or an  $N$ -D array.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

Complex Number Support: Yes

### Output

**Port\_1** — Output signal

vector | matrix |  $N$ -D array

The difference between adjacent rows, columns, or a specified dimension of the input, returned as a vector, matrix, or an  $N$ -D array.

The output is the same data type and complexity as the input, but the dimension that the difference was calculated on is one less. The length of the other dimensions is unchanged.

For more details on how the output is computed, see the “Difference along” on page 2-0 parameter.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

Complex Number Support: Yes

## Parameters

### Main Tab

**Running difference** — Running difference

No (default) | Yes

Specify whether the block computes a running difference.

- **No** -- The block computes the difference between adjacent elements in the specified dimension of the current input. In this mode, the block can compute the difference along the columns, rows, or a specified dimension of the input depending on the “Difference along” on page 2-0 parameter.
- **Yes** -- The block computes the running difference along the columns of the input. See “Running Operation” on page 2-414 for more information.

**Difference along** — Difference dimension  
Columns (default) | Rows | Specified dimension

Specify whether the block computes the difference along the columns, rows, or a specified dimension of the input.

- **Columns** -- The block computes differences between adjacent elements in each column of the input. Equivalent MATLAB code is given by:

```
y = diff(u)
```

For  $M$ -by- $N$  inputs, the output is an  $(M - 1)$ -by- $N$  matrix whose  $j$ th column has these elements:

$$y_{i,j} = u_{i+1,j} - u_{i,j} \quad 1 \leq i \leq (M - 1)$$

- **Rows** -- The block computes differences between adjacent elements in each row of the input. Equivalent MATLAB code is given by:

```
y = diff(u, [], 2)
```

The output is an  $M$ -by- $(N-1)$  matrix whose  $i$ th row has the following elements:

$$y_{i,j} = u_{i,j+1} - u_{i,j} \quad 1 \leq j \leq (N - 1)$$

- **Specified dimension** -- The behavior of the block is an extension of the row-wise differencing described earlier. The block computes differences between adjacent elements along the dimension you specify in the **Dimension** parameter. Equivalent MATLAB code is given by:

```
y = diff(u, [], d)
```

where  $d$  is the dimension.

The output is an array whose length in the specified dimension is one less than that of the input, and whose lengths in other dimensions are unchanged. For example, consider an  $M$ -by- $N$ -by- $P$ -by- $R$  input array with elements  $u(i,j,k,l)$  and assume that the value of the **Dimension** parameter is 3. The output of the block is an  $M$ -by- $N$ -by- $(P-1)$ -by- $R$  array with the following elements:

$$y_{i,j,k,l} = u_{i,j,k+1,l} - u_{i,j,k,l} \quad 1 \leq k \leq (P - 1)$$

**Dimension** — One-based dimension  
1 (default) | 2 | 3

Specify the one-based dimension along which to compute element-to-element differences.

### Dependencies

To enable this parameter, select **Specified dimension** for the **Difference along** parameter.

## Data Types Tab

### Rounding mode — Method of rounding operation

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

Specify the rounding mode for fixed-point operations as one of the following:

- Floor
- Ceiling
- Convergent
- Nearest
- Round
- Simplest
- Zero

For more details, see Rounding Modes.

### Saturate on integer overflow — Method of overflow action

off (default) | on


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

### Accumulator — Accumulator data type

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

**Accumulator** specifies the data type of the output of an accumulation operation in the Difference block.

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

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

For more information on the data type assistant, see “Specify Data Types Using Data Type Assistant” (Simulink).


For illustrations on how to use the accumulator data type in this block, see Fixed-Point Conversion in “Extended Capabilities” on page 2-0 .

**Output** — Output data type

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

**Output** specifies the data type of the output of the Difference block.

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

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

For more information, see “Control Data Types of Signals” (Simulink).

For more information on the output data type, see Fixed-Point Conversion in “Extended Capabilities” on page 2-0 .

**Output Minimum** — Minimum value that block can output

[] (default) | scalar

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

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

**Output Maximum** — Maximum value that block can output

[] (default) | scalar

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

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

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

off (default) | on

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

---

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

---

## Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | double   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no                                      |
| <b>Multidimensional Signals</b> | no                                      |
| <b>Variable-Size Signals</b>    | yes                                     |
| <b>Zero-Crossing Detection</b>  | no                                      |

## More About

### Running Operation

When you set the **Running difference** parameter to Yes, the block computes the running difference along the columns of the input.

For an  $M$ -by- $N$  input matrix, the output is an  $M$ -by- $N$  matrix whose  $j$ th column has the following elements:

$$y_{i,j} = u_{i+1,j} - u_{i,j} \quad 2 \leq i \leq (M - 1)$$

In the **Running difference** mode, the first element of the output for each column is the first input element minus the last input element of the previous frame. For the first frame, the block subtracts zero from the first input element.

$$y_{1,j}(t) = u_{1,j}(t) - u_{M,j}(t - T_f)$$

## Version History

Introduced before R2006a

## Extended Capabilities

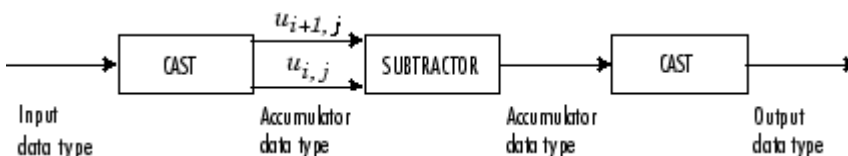
### C/C++ Code Generation

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

### Fixed-Point Conversion

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

This diagram shows the data types used within the Difference block for fixed-point signals.



You can set the accumulator and output data types in the block dialog as discussed in “Parameters” on page 2-410 .

## **See Also**

### **Functions**

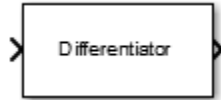
diff

### **Blocks**

Cumulative Sum

## Differentiator Filter

Direct form FIR fullband differentiator filter



## Library

Filtering / Filter Designs

dspfdesign

## Description

The Differentiator Filter block applies a fullband differentiator filter on the input signal to differentiate all its frequency components. The block uses an FIR equiripple filter design to design the differentiator filter. The ideal frequency response of the differentiator is  $D(\omega) = j\omega$  for  $-\pi \leq \omega \leq \pi$ .

You can design the filter with minimum order or with a specifies order.

The input signal can be a real- or complex-valued column vector or matrix. If the input signal is a matrix, each column of the matrix is treated as an independent channel.

This block supports variable-size input, enabling you to change the channel length during simulation. The output port properties, such as data type, complexity, and dimension, are identical to the input port properties. The block supports fixed-point operations.

This block also supports SIMD code generation. For details, see “Code Generation” on page 2-421.

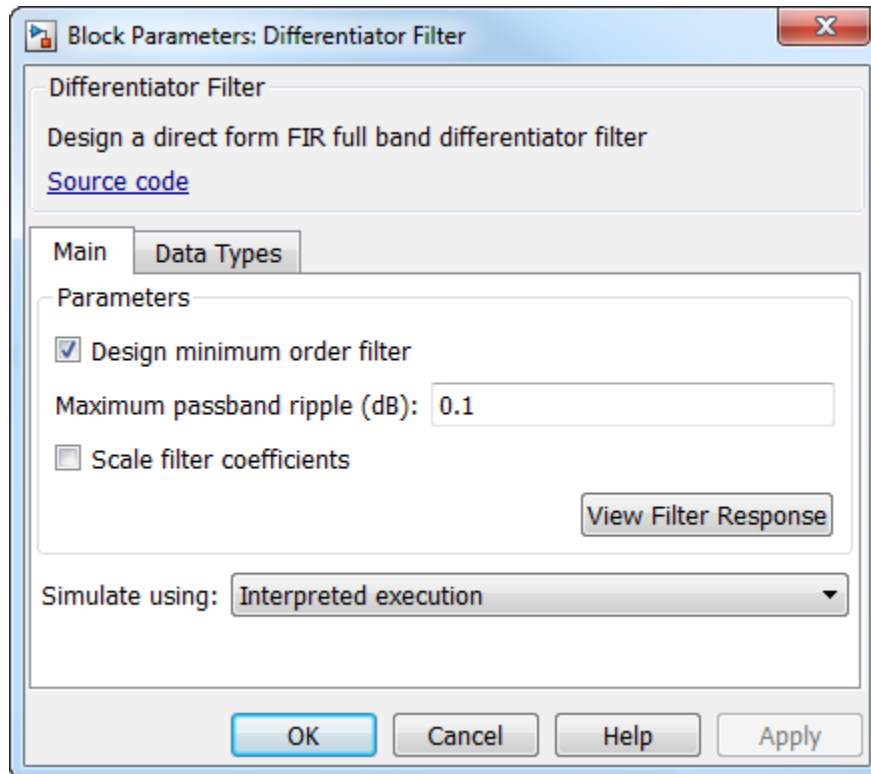
## Examples

- “Group Delay Estimation in Simulink”



## Dialog Box

### Main Tab



#### Design minimum order filter

When you select this check box, the block designs a filter with the minimum order, with the passband ripple specified in **Maximum passband ripple (dB)**. When you clear this check box, specify the order of the filter in **Filter order**.

By default, this check box is selected.

#### Filter order

Filter order of the differentiator filter, specified as an odd positive scalar integer. You can specify the filter order only when **Design minimum order filter** check box is not selected. The default is 31.

#### Maximum passband ripple (dB)

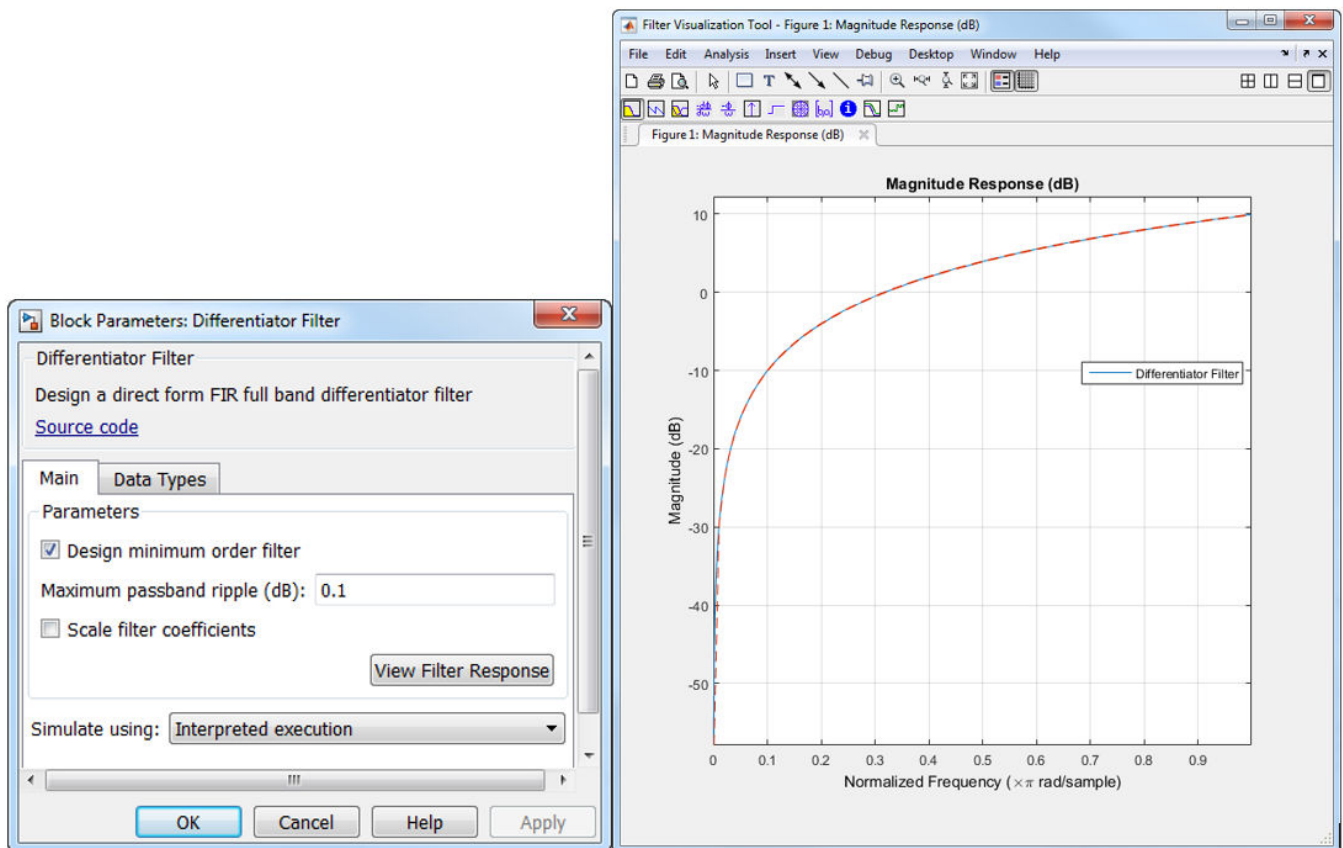
Maximum ripple of the filter response in the passband, specified as a real positive scalar in dB. The default is 0.1.

#### Scale filter coefficients

When you select this check box, the filter coefficients are scaled to preserve the input dynamic range. By default, this check box is not selected.

#### View Filter Response

Opens the Filter Visualization Tool (fvtool) and displays the magnitude and phase response of the Differentiator Filter block. The response is based on the block dialog box parameters. Changes made to these parameters update FVTool.



To update the magnitude response while FVTool is running, modify the dialog box parameters and click **Apply**.

### Simulate using

Type of simulation to run. You can set this parameter to:

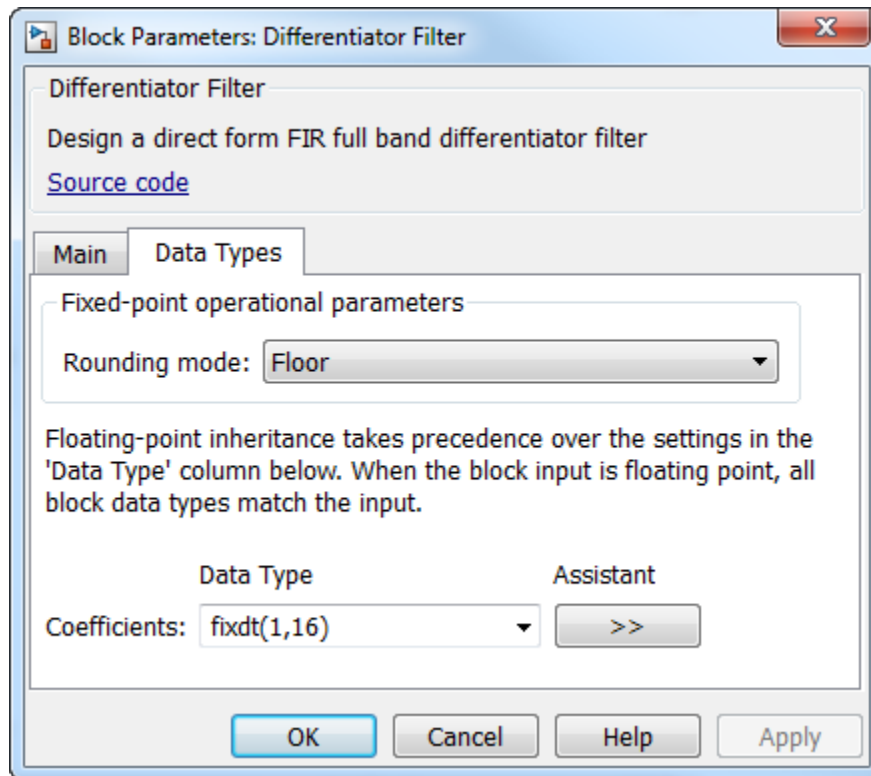
- Interpreted execution (default)

Simulate model using the MATLAB interpreter. This option shortens startup time and has faster simulation speed than Code generation.

- Code generation

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

## Data Types Tab




### Rounding mode

Rounding method for the output fixed-point operations. The rounding methods are Ceiling, Convergent, Floor, Nearest, Round, Simplest, and Zero. The default is Floor.

### Coefficients

Fixed-point data type of the coefficients, specified as one of the following:

- `fixdt(1,16)` (default) — Signed fixed-point data type of word length 16, with binary point scaling. The block determines the fraction length automatically from the coefficient values such that the coefficients occupy the maximum representable range without overflowing.
- `fixdt(1,16,0)` — Signed fixed-point data type of word length 16 and fraction length 0. You can change the fraction length to any other integer value.
- `<data type expression>` — Specify the data type using an expression that evaluates to a data type object, for example, numeric type (`fixdt([ ],16,15)`). Specify the sign mode of this data type as `[ ]` or `true`.
- Refresh Data Type — Refresh to the default data type.

Click the **Show data type assistant** button  to display the data type assistant, which helps you set the stage input parameter.

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

The word length of the output is same as the word length of the input. The fraction length of the output is computed such that the entire dynamic range of the output can be represented without

overflow. For details on how the block computes the fraction length, see “Fixed-Point Precision Rules for Avoiding Overflow in FIR Filters”.

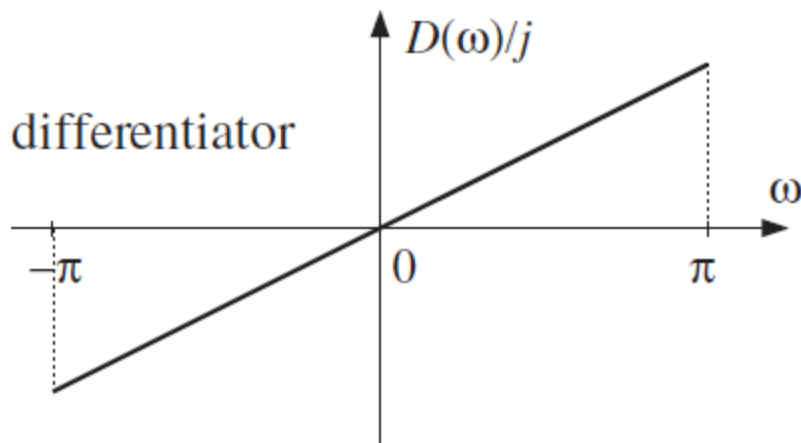
## Supported Data Types

| Port   | Supported Data Types   |
|--------|--|
| Input  | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed or unsigned)</li> </ul> |
| Output | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed or unsigned)</li> </ul> |

## Algorithms

### Differentiator Filter

Differentiator computes the derivative of a signal. The frequency response of an ideal differentiator filter is given by  $D(\omega) = j\omega$ , defined over the Nyquist interval  $-\pi \leq \omega \leq \pi$ .



The frequency response is antisymmetric and is linearly proportional to the frequency.

`dsp.Differentiator` object acts as a differentiator filter. This object condenses the two-step process into one. For the minimum order design, the object uses generalized Remez FIR filter design algorithm. For the specified order design, the object uses the Parks-McClellan optimal equiripple FIR filter design algorithm. The filter is designed as a linear phase Type-IV FIR filter with a Direct form structure.

The ideal differentiator has an antisymmetric impulse response given by  $d(n) = -d(-n)$ . Hence  $d(0) = 0$ . The differentiator must have zero response at zero frequency.

### Linear-Phase FIR Differentiator Filter

The impulse response of an antisymmetric linear-phase FIR filter is given by  $h(n) = -h(M - 1 - n)$ , where  $M$  is the length of the filter. Because the filter is antisymmetric, you can use this type of FIR filter to design the linear-phase FIR differentiators.

Consider the design of linear-phase FIR differentiators based on the Chebyshev approximation criterion.

If  $M$  is odd, the real-valued frequency response of the FIR filter,  $H_r(\omega)$ , has the characteristics that  $H_r(0) = 0$  and  $H_r(\pi) = 0$ . This filter satisfies the condition of zero response at zero frequency. However, it is not fullband because  $H_r(\pi) = 0$ . This differentiator has a linear response over the limited frequency range  $[0, 2\pi f_p]$ , where  $f_p$  is the bandwidth of the differentiator. The absolute error between the desired response and the Chebyshev approximation increases as  $\omega$  increases from 0 to  $2\pi f_p$ .

If  $M$  is even, the real-valued frequency response of the FIR filter,  $H_r(\omega)$ , has the characteristics that  $H_r(0) = 0$  and  $H_r(\pi) \neq 0$ . This filter satisfies the condition of zero response at zero frequency. It is fullband and this design results in a significantly smaller approximation error than comparable odd-length differentiators. Hence, even-length (odd order) differentiators are preferred in practical systems.

## Version History

Introduced in R2015b

## Extended Capabilities

### C/C++ Code Generation

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

The Differentiator Filter block supports SIMD code generation using Intel AVX2 technology when the input signal has a data type of `single` or `double`.

The SIMD technology significantly improves the performance of the generated code.

### Fixed-Point Conversion

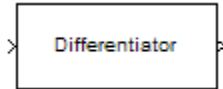
Design and simulate fixed-point systems using Fixed-Point Designer™.

## See Also

`dsp.Differentiator` | Highpass Filter | Variable Bandwidth FIR Filter | Variable Bandwidth IIR Filter | Biquad Filter

## Differentiator Filter (Obsolete)

Design differentiator filter



### Compatibility

---

**Note** The Differentiator Filter (Obsolete) block has been replaced by the Differentiator Filter block. Existing instances of the Differentiator Filter (Obsolete) block will continue to operate. For new models, use the Differentiator Filter block.

---

### Library

Filtering / Filter Designs

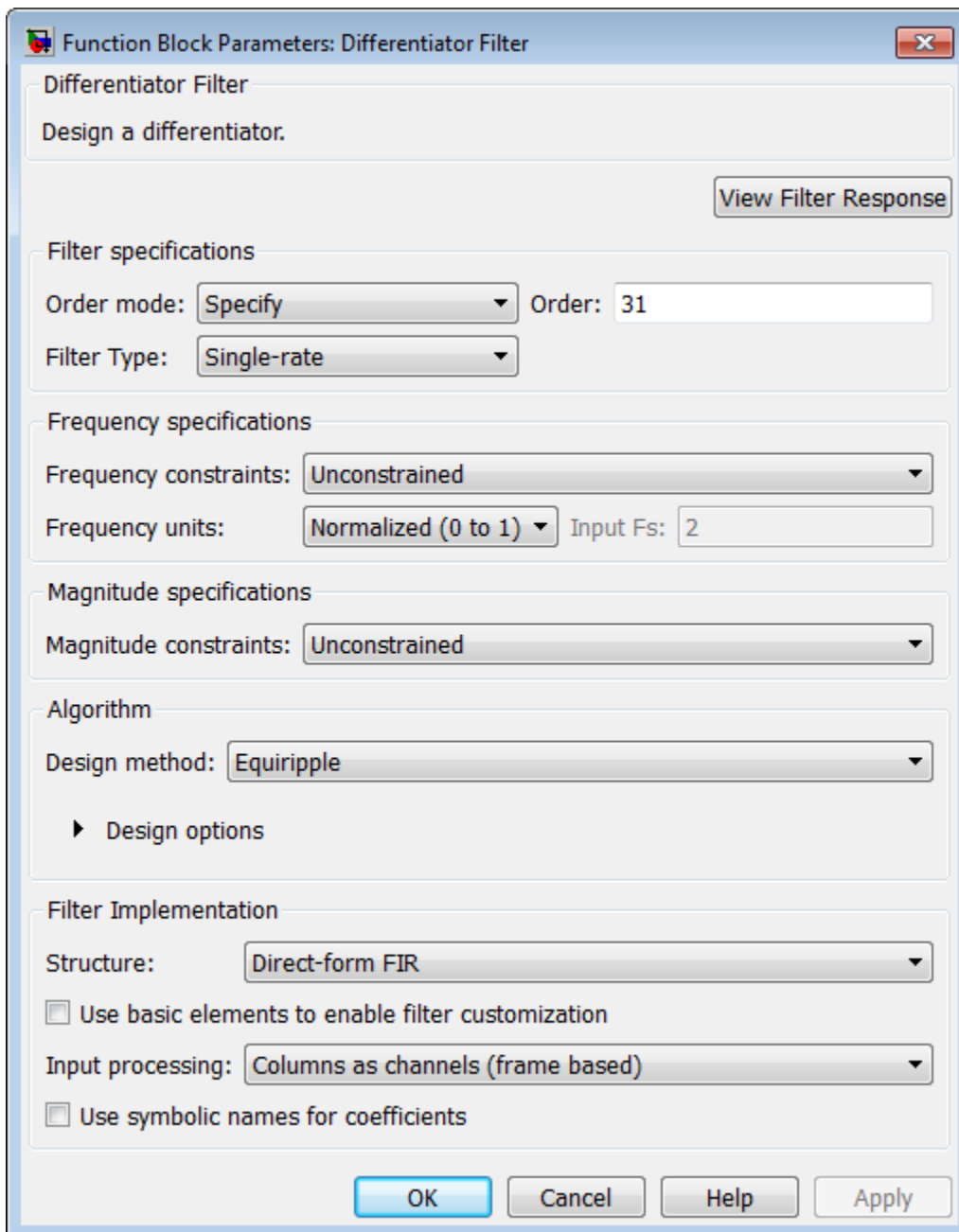
dspfdesign

### Description

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

### Dialog Box

See “Differentiator Filter Design — Main Pane” on page 5-613 for more information about the parameters of this block. The **Data Types** and **Code Generation** panes are not available for blocks in the DSP System Toolbox Filter Designs library.



### View filter response

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

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

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

## Filter Specifications

### Order mode

Select either **Minimum** or **Specify** (the default). Selecting **Specify** enables the **Order** option so you can enter the filter order.

### Order

Enter the filter order. This option is enabled only if you set the **Order mode** to **Specify**. The default order is 31.

### Filter type

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

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

### Decimation Factor

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

### Interpolation Factor

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

## Frequency Specifications

The parameters in this group allow you to specify your filter response curve.

### Frequency constraints

This option is only available when you specify the order of the filter design. Supported options are **Unconstrained** and **Passband edge and stopband edge**.

### Frequency units

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

### Input Fs

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

### Fpass

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



**Fstop**

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

**Magnitude Specifications**

Parameters in this group specify the filter response in the passbands and stopbands. These parameters are only available for minimum-order designs.

**Magnitude constraints**

This option is only available when you specify the order of your filter design. The available **Magnitude constraints** depend on the value of the **Frequency constraints** parameter. When you set the **Frequency constraints** parameter to Unconstrained, the **Magnitude constraints** parameter must also be Unconstrained. When you set the **Frequency constraints** parameter to Passband edge and stopband edge, the **Magnitude constraints** parameter can be Unconstrained, Passband ripple, or Stopband attenuation.

**Magnitude units**

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

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

**Apass**

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

**Astop2**

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

**Algorithm**

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

**Design Method**

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well.

**Design Options**

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

**Density factor**

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

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

**Wpass**

Passband weight. This option is only available for a specified-order design when **Frequency constraints** is equal to Passband edge and stopband edge and the **Design method** is Equiripple.

**Wstop**

Stopband weight. This option is only available for a specified-order design when **Frequency constraints** is equal to Passband edge and stopband edge and the **Design method** is Equiripple.

**Filter Implementation****Structure**

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

**Use basic elements to enable filter customization**

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

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

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

**Input processing**

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

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

---

**Note** The Inherited (this choice will be removed – see release notes) option will be removed in a future release. See “Frame-Based Processing” in the *DSP System Toolbox Release Notes* for more information.

---

### Rate options

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

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

### Use symbolic names for coefficients

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

### Supported Data Types

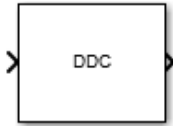
| Port   | Supported Data Types   |
|--------|--|
| Input  | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> </ul> |
| Output | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> </ul> |

### Version History

Introduced in R2006b

# Digital Down-Converter

Translate digital signal from intermediate frequency (IF) band to baseband and decimate it



## Libraries:

DSP System Toolbox / Signal Operations

## Description

The Digital Down-Converter (DDC) block converts a digitized real signal centered at an intermediate frequency (IF) to a baseband complex signal centered at zero frequency. The DDC block downsamples the frequency downconverted signal using a cascade of three decimation filters. This block designs the decimation filters according to the filter parameters you set in the block dialog box.

## Ports

### Input

**Input 1** — Data input  
column vector | matrix

Specify the data input as a column vector or a matrix. The number of rows in the input signal must be a multiple of the decimation factor that you specify in the **Decimation factor** parameter.

When the input data type is `double` or `single` precision, the output data type is the same as that of the input. When the input data type is a signed integer or fixed-point type, the output data type is defined by the **Output** parameter on the **Data Types** tab.

When the input is fixed-point, it must be signed only.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `fixed point`

**Osc** — Oscillator signal  
column vector | matrix

Specify the oscillator signal that the block uses to frequency down convert the input signal as a column vector or a matrix. The signal must be complex and its length must be equal to the length of the input.

When the oscillator input is fixed-point, it must be signed only.

### Dependencies

To enable this port, set the **Type of oscillator** parameter to `Input port`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `fixed point`

## Output

**Output 1** — Downconverted and downsampled signal  
column vector | matrix

The block outputs the downconverted and downsampled signal as a column vector or a matrix. The length of the output is equal to the length of the input divided by the value you specify in the **Decimation factor** parameter.

When the data type of the input is `double` or `single` precision, the data type of the output is the same as that of the input. When the input data type is fixed-point type or a signed integer, the output data type is defined by the **Output** parameter on the **Data Types** tab.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `fixed point`

## Parameters

### Main Tab

**Decimation factor** — Decimation factor

100 (default) | positive integer  $\geq 2$  | vector of positive integers

Specify the decimation factor as a positive integer  $\geq 2$ , or as a 1-by-2 or 1-by-3 vector of positive integers.

When you set this parameter to a scalar value, the block applies the same decimation factor to the three decimation filtering stages.

When you set this parameter to a 1-by-2 vector, the block applies the first and second values in the vector to the first and second filtering stages, respectively, and bypasses the third stage. Both elements of **Decimation factor** must be greater than 1.

When you set this parameter to a 1-by-3 vector, the block applies the values in the vector to the corresponding filtering stages. The first and second elements of **Decimation factor** must be greater than 1, and the third element must be 1 or 2.

**Minimum order filter design** — Design minimum-order filter

on (default) | off

When you select this check box, the block designs filters with the minimum possible order that meets the requirements specified in these parameters:

- **Passband ripple of cascade response (dB)**
- **Stopband attenuation of cascade response (dB)**
- **Two sided bandwidth of input signal (Hz)**
- **Source of stopband frequency**
- **Stopband frequency (Hz)**

When you clear this check box, the block designs filters with orders that you specify in **Number of sections of CIC decimator**, **Order of CIC compensation filter stage**, and **Order of third filter stage**. The filter designs meet the passband and stopband frequency specifications that you set in

**Two sided bandwidth of input signal (Hz), Source of stopband frequency, and Stopband frequency (Hz).** By default, this check box is selected.

**Number of sections of CIC decimator** — Number of sections in CIC decimator

3 (default) | positive integer

Specify the number of sections in the CIC decimator as a positive integer.

**Dependencies**

To enable this parameter, clear the **Minimum order filter design** parameter.

**Order of third filter stage** — Order of third filter stage

10 (default) | even positive integer

Specify the order of the third filter stage as an even positive integer. When you specify **Decimation factor** as a 1-by-2 vector, the block ignores the value in this parameter because the block bypasses the third filter stage.

**Dependencies**

To enable this parameter, clear the **Minimum order filter design** parameter.

**Source of stopband frequency** — Source of stopband frequency

Auto (default) | Property

Specify the source of the stopband frequency as Auto or Property.

When you set this parameter to Auto, the block places the cutoff frequency of the cascade filter response at approximately  $F_c = \text{SampleRate} / M/2$  Hz, where  $M$  is the total decimation factor specified in **Decimation factor**.  $\text{SampleRate}$  is computed as  $1/T_s$ , where  $T_s$  is the sample time of the input signal. The block computes the stopband frequency as  $F_{stop} = F_c + (TW / 2)$ .  $TW$  is the transition bandwidth of the cascade response, computed as  $2 \times (F_c - F_p)$ , where the passband frequency  $F_p$  equals  $\text{Bandwidth}/2$ .

When you set this parameter to Property, specify the stopband frequency in **Stopband frequency (Hz)**.

**Type of oscillator** — Type of oscillator

Sine wave (default) | NCO | Input port | None

Specify the oscillator type as one of the following:

- **Sine wave** (default) — The block performs frequency down conversion on the input signal using a complex exponential obtained from samples of a sinusoidal trigonometric function.
- **NCO** — The block performs frequency down conversion on the input signal with a complex exponential obtained using a numerically controlled oscillator (NCO).
- **Input port** — The block performs frequency down conversion on the input signal using the complex signal that you provide through the **Osc** input port.
- **None** — The block does not require the mixer stage and acts as a three-stage cascaded decimator.

**Center frequency of output signal (Hz)** — Center frequency of output signal

14e6 (default) | positive scalar

Specify the center frequency of the output signal in Hz as a double-precision positive scalar that is less than or equal to half the sample rate. The block downconverts the input signal from the passband center frequency, which you specify in **Center frequency of input signal (Hz)**, to 0 Hz.

**Dependency**

To enable this parameter, set **Type of oscillator** to Sine wave or NCO.

**Order of CIC compensation filter stage** — Order of CIC compensation filter stage

12 (default) | positive integer

Specify the order of the CIC compensation filter stage as a positive integer.

**Dependencies**

To enable this parameter, clear the **Minimum order filter design** parameter.

**Two-sided bandwidth of input signal (Hz)** — Two-sided bandwidth of input signal

200000 (default) | positive integer

Specify the two-sided bandwidth of the input signal in Hz as a positive integer. The block sets the passband frequency of the cascade of filters to half of the value you specify in this parameter. Set the value of this parameter to less than **Input sample rate/Decimation factor**. When you select the **Inherit sample rate from input** check box, set this value to less than  $((1/T_s) / \text{Decimation factor})$ , where  $T_s$  is the sample time of the input signal.

**Stopband frequency (Hz)** — Stopband frequency

150000 (default) | positive scalar

Specify the stopband frequency in Hz as a double-precision positive scalar.

**Dependencies**

To enable this parameter, set **Source of stopband frequency** to Property.

**Passband ripple of cascade response (dB)** — Passband ripple of cascade response

0.1 (default) | positive scalar

Specify the passband ripple of the cascade response in dB as a double-precision positive scalar. When you select the **Minimum order filter design**, the block designs the filters so that the cascade response meets the passband ripple that you specify in **Passband ripple of cascade response (dB)**.

**Dependencies**

To enable this parameter, select the **Minimum order filter design** parameter.

**Stopband attenuation of cascade response (dB)** — Stopband attenuation of cascade response

60 (default) | positive scalar

Specify the stopband attenuation of the cascade response in dB as a double-precision positive scalar. When you select the **Minimum order filter design** parameter, the block designs the filters such that the cascade response meets the stopband attenuation that you specify in this parameter.

**Dependencies**

To enable this parameter, select the **Minimum order filter design** parameter.

**Number of NCO accumulator bits** — Number of NCO accumulator bits

16 (default) | positive integer

Specify the number of NCO accumulator bits as an integer scalar in the range [1 128].

**Dependencies**

To enable this parameter, set **Type of oscillator** to NCO.

**Number of NCO quantized accumulator bits** — Number of NCO quantized accumulator bits

12 (default) | positive integer in the range [1 128]

Specify the number of NCO quantized accumulator bits as an integer scalar in the range [1 128]. This value must be less than the value you specify in **Number of NCO accumulator bits**.

**Dependencies**

To enable this parameter, set **Type of oscillator** to NCO.

**Dither control for NCO** — Dither control for NCO

on (default) | off

When you select this check box, the block applies dither to the NCO signal according to the number of dither bits you specify in **Number of NCO dither bits**. By default, this check box is selected.

**Dependencies**

To enable this parameter, set **Type of oscillator** to NCO.

**Number of NCO dither bits** — Number of NCO dither bits

4 (default) | positive integer

Specify the number of NCO dither bits as an integer scalar smaller than the number of accumulator bits in **Number of NCO accumulator bits**.

**Dependencies**

To enable this parameter, set **Type of oscillator** to NCO and select the **Dither control for NCO** parameter.

**Inherit sample rate from input** — Inherit sample rate from input

off (default) | on



When you select this check box, the block computes the sample rate as  $N/T_s$ , where  $N$  is the frame size of the input signal, and  $T_s$  is the sample time of the input signal. When you clear this check box, the block sets the sample rate to the value in **Input sample rate (Hz)**.

**Input sample rate (Hz)** — Sample rate of input signal

30e6 (default) | positive scalar

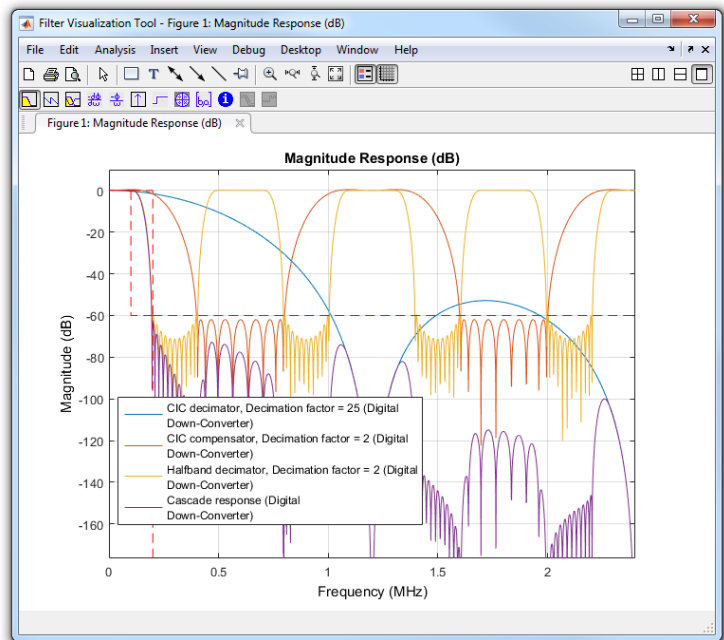
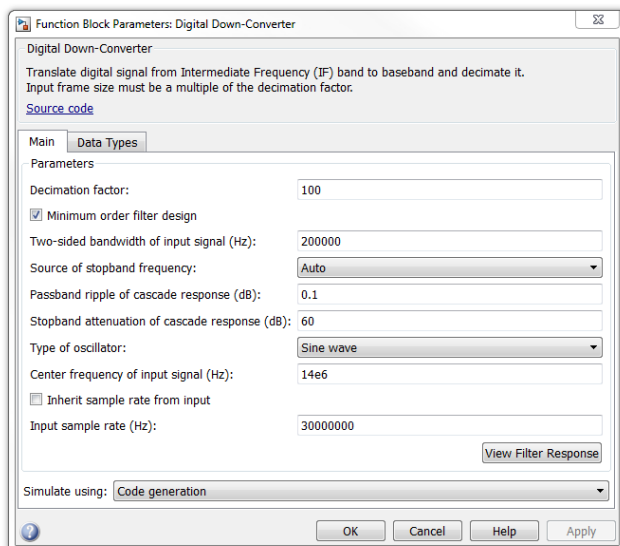
Specify the sample rate of the input signal in Hz as a positive scalar value greater than or equal to twice the value in **Center frequency of input signal (Hz)**.

### Dependencies

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

**View Filter Response** — View Filter Response button

Click this button to open the Filter Visualization Tool (FVTool) and display the magnitude and phase response of each stage as well as the cascade of stages in the Digital Down-Converter. The response is based on the values you specify in the block parameters dialog box. Changes made to these parameters update FVTool.



To update the magnitude response while FVTool is running, modify the parameters in the dialog box and click **Apply**.

**Simulate using** — Simulate using

Code generation (default) | Interpreted execution

Specify the type of simulation to run. You can set this parameter to:

- **Code generation** (default)

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

- **Interpreted execution**

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

## Data Types Tab

### Stage input — Stage input

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

Specify the data type of the input in the first, second, and third filter stages. You can set this parameter to one of the following:

- **Inherit: Same as input** (default) — The block inherits the **Stage input** data type from the input signal.
- `fixdt([],16,0)` — The block uses the fixed-point data type with binary-point scaling. Specify the sign mode of this data type as [] or `true`.
- An expression that evaluates to a data type, for example, `numerictype([],16,15)`. Specify the sign mode of this data type as [] or `true`.

For help with setting the stage input parameter, you can click the **Show data type assistant** button



to display the data type assistant.

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

### Output — Output

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

Specify the data type of the block output. You can set this parameter to:

- **Inherit: Same as input** (default) — The block Inherits the output datatype from the input.
- `fixdt([],16,0)` — The block uses the fixed-point data type with binary point scaling. Specify the sign mode of this data type as [] or `true`.
- An expression that evaluates to a data type, for example, `numerictype([],16,15)`. Specify the sign mode of this data type as [] or `true`.

For help with setting the **Output** parameter, you can click the **Show data type assistant** button



to display the data type assistant.

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

**Minimum** — Minimum

[] (default) | scalar

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

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

**Maximum** — Maximum

[] (default) | scalar

Maximum value of the block output. The default value is [] (unspecified). Simulink software uses this value to perform:

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

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

off (default) | on

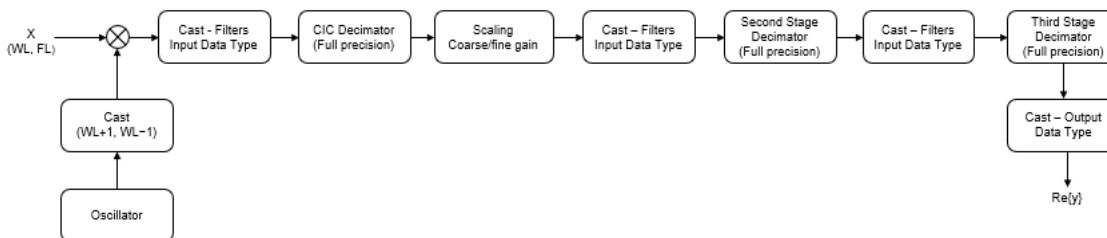
Select this parameter to prevent the fixed-point tools from overriding the data types you specify in the block dialog box.

**Block Characteristics**

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | double   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no                                      |
| <b>Multidimensional Signals</b> | no                                      |
| <b>Variable-Size Signals</b>    | no                                      |
| <b>Zero-Crossing Detection</b>  | no                                      |

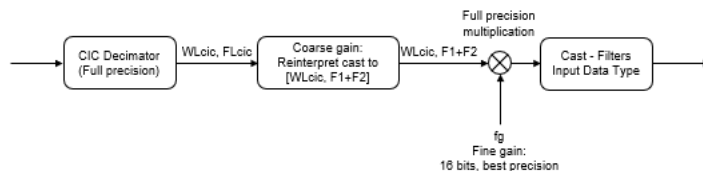
**More About****Fixed Point**

The following block diagram represents the DDC arithmetic with signed fixed-point inputs.



- WL is the word length of the input, and FL is the fraction length of the input.
- The input of each filter is cast to the filter input data type. In the `dsp.DigitalDownConverter` object, you can specify the filter input data type through the `FiltersInputDataType` and `CustomFiltersInputDataType` properties. In the Digital Down-Converter block, you can specify the filter input data type through the **Stage input** parameter.
- The oscillator output is cast to a word length equal to the input word length plus one. The fraction length is equal to the input word length minus one.
- The scaling at the output of the CIC decimator consists of coarse- and fine-gain adjustments. The coarse gain is achieved using the `reinterpretcast` function on the CIC decimator output. The fine gain is achieved using full-precision multiplication.

The following figure depicts the coarse-gain and fine-gain operations.

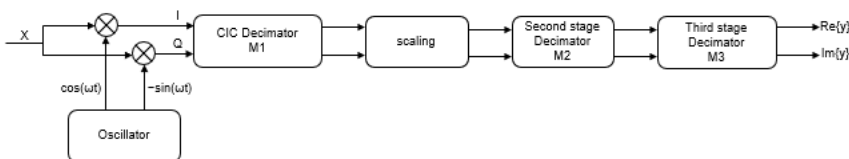


If the normalization gain is  $G$  (where  $0 < G \leq 1$ ), then:

- $WLCic$  is the word length of the CIC decimator output and  $FLCic$  is the fraction length of the CIC decimator output.
- $F1 = \text{abs}(\text{nextpow2}(G))$ , indicating the part of  $G$  achieved using bit shifts (coarse gain).
- $F2 = \text{fraction length specified by the filter input data type}$ .
- $fg = \text{fi}((2^{F1}) * G, \text{true}, 16)$ , which indicates that the remaining gain cannot be achieved with a bit shift (fine gain).

## Algorithms

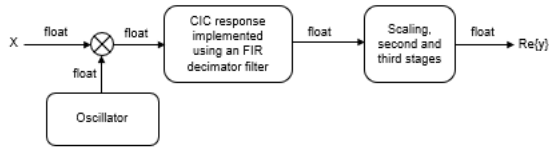
The digital down converter downconverts the input signal by multiplying it with a complex exponential that has the specified center frequency. The algorithm downsamples the frequency downconverted signal using a cascade of three decimation filters. In this case, the filter cascade consists of a CIC decimator, a CIC compensator, and a third FIR decimation stage. The following block diagram shows the architecture of the digital down converter.



The scaling section normalizes the CIC gain and the oscillator power. It can also contain a correction factor to achieve the desired ripple specification. When you specify an oscillator signal through the input port, the normalization factor does not include the oscillator power factor. Depending on how you set the decimation factor, the block bypasses the third filter stage. When the input data type is double or single, the algorithm implements an  $N$ -section CIC decimation filter as an FIR filter with a response that corresponds to a cascade of  $N$  boxcar filters. The algorithm emulates a CIC filter with

an FIR filter so that you can run simulations with floating-point data. When the input data type is fixed-point, the algorithm implements a true CIC filter with actual comb and integrator sections.

This block diagram represents the DDC arithmetic with single or double-precision, floating-point inputs.



For details about fixed-point operation, see “Fixed Point” on page 2-435.

## Version History

Introduced in R2015a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

If the input is fixed point, it must be a signed integer or a signed fixed point value with a power-of-two slope and zero bias.

## See Also

### Objects

`dsp.DigitalDownConverter` | `dsp.DigitalUpConverter`

### Blocks

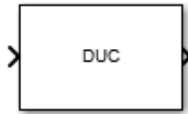
Digital Up-Converter

### Topics

“Digital Up and Down Conversion for Family Radio Service in MATLAB”

# Digital Up-Converter

Interpolate and translate digital signal from baseband to intermediate frequency (IF) band



**Libraries:**  
DSP System Toolbox / Signal Operations

## Description

The Digital Up-Converter (DUC) block converts a complex digital baseband signal to a real passband signal.

The DUC block upsamples the input signal using a cascade of three interpolation filters. The block frequency upconverts the upsampled signal by multiplying it by the specified center frequency of the output signal. This block designs the interpolation filters according to the filter parameters you set in the block dialog box.

## Ports

### Input

**Input 1** — Input signal  
column vector

Specify the input signal as a column vector of real or complex values.

When the input data type is `double` or `single` precision, the output data type is the same as that of the input. When the input data type is a signed integer or fixed-point, the output data type is defined by the **Output** parameter on the **Data Types** tab.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `fixed point`

### Output

**Output 1** — Upconverted and upsampled signal  
column vector

The block outputs the upconverted and upsampled signal as a column vector of real values. The length of the output is equal to the length of the input multiplied by the value that you specify in the **Interpolation factor** parameter.

When the input data type is `double` or `single` precision, the output data type is the same as that of the input. When the input data type is a signed integer or fixed-point, the output data type is defined by the **Output** parameter on the **Data Types** tab.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `fixed point`

## Parameters

### Main Tab

**Interpolation factor** — Interpolation factor

100 (default) | positive integer > 1 | vector of positive integers

Specify the interpolation factor as a positive integer > 1, or as a 1-by-2 or 1-by-3 vector of positive integers.

When you set this parameter to a scalar value, the block applies the same interpolation factor to the three interpolation filtering stages.

When you set this parameter to a 1-by-2 vector, the block applies the second and third values in the vector to the second and third filtering stages, respectively, and bypasses the first filter stage. Both elements of the **Interpolation factor** must be greater than 1.

When you set this parameter to a 1-by-3 vector, the block applies the values in the vector to the corresponding filtering stages. The second and third elements of **Interpolation factor** must be greater than 1, and the first element must be 1 or 2.

**Minimum order filter design** — Design minimum-order filter

on (default) | off

When you select this check box, the block designs filters with the minimum possible order that meets the requirements specified in these parameters:

- **Passband ripple of cascade response (dB)**
- **Stopband attenuation of cascade response (dB)**
- **Two sided bandwidth of input signal (Hz)**
- **Source of stopband frequency**
- **Stopband frequency (Hz)**

When you clear this check box, the block designs filters with orders that you specify in **Order of first filter stage**, **Order of CIC compensation filter stage**, and **Number of sections of CIC interpolator**. The filter designs meet the passband and stopband frequency specifications that you set in **Two sided bandwidth of input signal (Hz)**, **Source of stopband frequency**, and **Stopband frequency (Hz)**. By default, this check box is selected.

**Order of first filter stage** — Order of first filter stage

10 (default) | positive even integer

Specify the order of the first filter stage as an even positive integer. When you specify **Interpolation factor** as a 1-by-2 vector, the block ignores the value in this parameter because the block bypasses the first filter stage.

### Dependencies

To enable this parameter, clear the **Minimum order filter design** parameter.

**Number of sections of CIC interpolator** — Number of sections of CIC interpolator

3 (default) | positive integer

Specify the number of sections in the CIC interpolator as a positive integer.

**Dependencies**

This parameter appears when you clear the **Minimum order filter design** parameter.

**Source of stopband frequency** — Source of stopband frequency

Auto (default) | Property

Specify the source of the stopband frequency as Auto or Property.

When you set this parameter to *Auto*, the block places the cutoff frequency of the cascade filter response at approximately  $F_c = \text{SampleRate}/2$  Hz, and computes the stopband frequency as  $F_{stop} = F_c + TW/2$ . *SampleRate* is computed as  $1/T_s$ , where  $T_s$  is the sample time of the input signal. *TW* is the transition bandwidth of the cascade response, computed as  $2 \times (F_c - F_p)$ , and the passband frequency,  $F_p$ , equals *Bandwidth*/2.

When you set this parameter to *Property*, specify the stopband frequency in the **Stopband frequency (Hz)** parameter.

**Type of oscillator** — Type of oscillator

Sine wave (default) | NCO

Specify the oscillator type as one of the following:

- *Sine wave* (default) — The block performs frequency upconversion on the output of the interpolation filter cascade using a complex exponential signal obtained from samples of a sinusoidal trigonometric function.
- *NCO* — The block performs frequency upconversion with a complex exponential obtained using a numerically controlled oscillator (NCO).

**Center frequency of output signal (Hz)** — Center frequency of output signal

14e6 (default) | positive scalar

Specify the center frequency of the output signal in Hz as a double-precision positive scalar. The value of this parameter must be less than or equal to half the product of the *SampleRate* times the total interpolation factor. *SampleRate* is computed as  $1/T_s$ , where  $T_s$  is the sample time of the input signal. The block up converts the input signal so that the output spectrum centers at the frequency you specify in **Center frequency of output signal (Hz)**.

**Order of CIC compensation filter stage** — Order of CIC compensation filter stage

12 (default) | positive integer

Specify the order of the CIC compensation filter stage as a positive integer.

**Dependencies**

To enable this parameter, clear the **Minimum order filter design** parameter.



**Two sided bandwidth of input signal (Hz)** — Two sided bandwidth of input signal

200000 (default) | positive integer

Specify the two-sided bandwidth of the input signal in Hz as a positive integer. The block sets the passband frequency of the cascade of filters to half of the value you specify in this parameter.

**Stopband frequency (Hz)** — Stopband frequency

150000 (default) | positive scalar

Specify the stopband frequency in Hz as a double-precision positive scalar.

#### Dependencies

To enable this parameter, set the **Source of stopband frequency** to Property.

**Passband ripple of cascade response (dB)** — Passband ripple of cascade response

0.1 (default) | positive scalar

Specify the passband ripple of the cascade response in dB as a double-precision positive scalar. When you select the **Minimum order filter design** parameter, the block designs the filters so that the cascade response meets the passband ripple that you specify in **Passband ripple of cascade response (dB)**.

#### Dependencies

To enable this parameter, select the **Minimum order filter design** parameter.

**Stopband attenuation of cascade response (dB)** — Stopband attenuation of cascade response

60 (default) | positive scalar

Specify the stopband attenuation of the cascade response in dB as a double-precision positive scalar. When you select the **Minimum order filter design** parameter, the block designs the filters such that the cascade response meets the stopband attenuation that you specify in this parameter.

#### Dependencies

To enable this parameter, select the **Minimum order filter design** parameter.

**Number of NCO accumulator bits** — Number of NCO accumulator bits

16 (default) | positive integer

Specify the number of NCO accumulator bits as an integer scalar in the range [1 128].

#### Dependencies

To enable this parameter, set **Type of oscillator** to NCO.

**Number of NCO quantized accumulator bits** — Number of NCO quantized accumulator bits

12 (default) | positive integer

Specify the number of NCO quantized accumulator bits as an integer scalar in the range [1 128]. This value must be less than the value you specify in **Number of NCO accumulator bits**.

#### Dependencies

To enable this parameter, set **Type of oscillator** to NCO.

#### Dither control for NCO — Dither control for NCO

on (default) | off

When you select this parameter, the block applies dither to the NCO signal according to the number of dither bits you specify in **Number of NCO dither bits**.

#### Dependencies

To enable this parameter, set **Type of oscillator** to NCO.

#### Number of NCO dither bits — Number of NCO dither bits

4 (default) | positive integer

Specify the number of NCO dither bits as an integer scalar smaller than the number of accumulator bits in **Number of NCO accumulator bits**.

#### Dependencies

To enable this parameter, set **Type of oscillator** to NCO and select the **Dither control for NCO** parameter.

#### Inherit sample rate from input — Inherit sample rate from input

off (default) | on

When you select this parameter, the block computes the sample rate as  $N/T_s$ , where  $N$  is the frame size of the input signal, and  $T_s$  is the sample time of the input signal. When you clear this parameter, the block sets the sample rate to the value in **Input sample rate (Hz)**.

#### Input sample rate (Hz) — Sample rate of input signal

300e3 (default) | positive scalar

Specify the sample rate of the input signal in Hz as a positive scalar. The value of this parameter multiplied by the total interpolation factor must be greater than or equal to twice the value in **Center frequency of output signal (Hz)**.

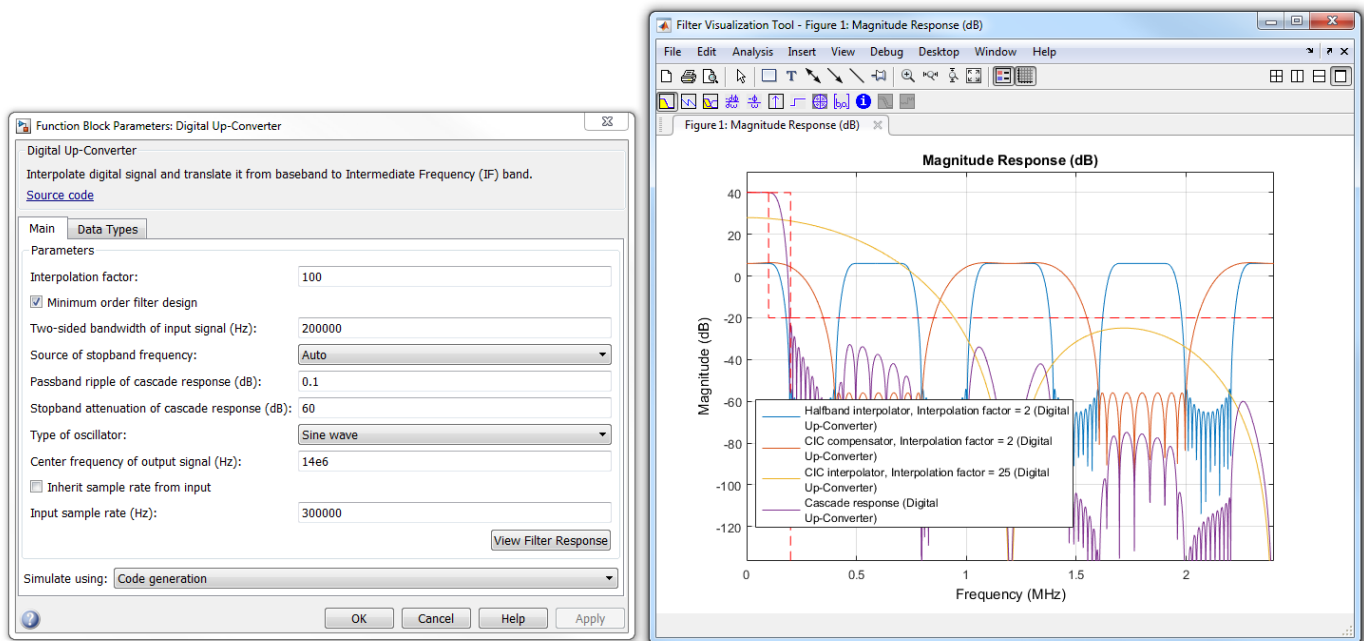
#### Dependencies

To enable this parameter, clear the **Inherit sample rate from input** parameter.

#### View Filter Response — View Filter Response

button

Click this button to open the Filter Visualization Tool (FVTool) and display the magnitude and phase response of each stage as well as the cascade of stages in the Digital Up-Converter. The response is based on the values you specify in the block parameters dialog box.. Changes made to these parameters update FVTool.



To update the magnitude response while FVTool is running, modify the parameters in the dialog box and click **Apply**.

**Simulate using** — Simulate using

Code generation (default) | Interpreted execution

Specify the type of simulation to run. You can set this parameter to:

- Code generation (default)

Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than Interpreted execution.

- Interpreted execution

Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than Code generation.

## Data Types Tab

**Stage output** — Stage output

Inherit: Same as input (default) | `fixdt([],16,0)`

Specify the data type of the output in the first, second, and third filter stages. You can set this parameter to:

- Inherit: Same as input (default) — The block inherits the **Stage output** data type from the input signal.

- `fixdt([], 16, 0)` — The block uses the fixed-point data type with binary-point scaling. Specify the sign mode of this data type as `[]` or `true`.
- An expression that evaluates to a data type, for example, `numerictype([], 16, 15)`. Specify the sign mode of this data type as `[]` or `true`.

The block casts the data at the output of each filter stage according to the value you set in this parameter. For the CIC stage, the casting is done after the signal has been scaled by the normalization factor.

For help with setting the stage output parameter, you can click the **Show data type assistant** button



to display the data type assistant.

See “Specify Data Types Using Data Type Assistant” (Simulink) in *Simulink User's Guide* (Simulink) for more information.

### Output — Output

Inherit: Same as input (default) | `fixdt([], 16, 0)`

Specify the data type of the block output. You can set this parameter to:

- `Inherit: Same as input (default)` — The block Inherits the output datatype from the input.
- `fixdt([], 16, 0)` — The block uses the fixed-point data type with binary-point scaling. Specify the sign mode of this data type as `[]` or `true`.
- An expression that evaluates to a data type, for example, `numerictype([], 16, 15)`. Specify the sign mode of this data type as `[]` or `true`.

For help with setting the **Output** parameter, you can click the **Show data type assistant** button



to display the data type assistant..

See “Specify Data Types Using Data Type Assistant” (Simulink) in *Simulink User's Guide* (Simulink) for more information.

### Minimum — Minimum

`[]` (default) | scalar

Specify the minimum value of the block output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Specify Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

### Maximum — Maximum

`[]` (default) | scalar

Specify the maximum value of the block output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Specify Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

**Lock data type settings against changes by the fixed-point tools** — Prevent fixed-point tools from overriding data types

off (default) | on

Select this parameter to prevent the fixed-point tools from overriding the data types you specify in the block dialog box.

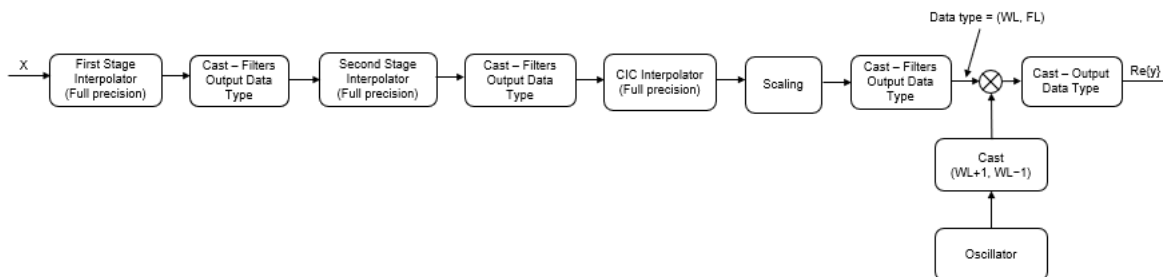
## Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | double   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no                                      |
| <b>Multidimensional Signals</b> | no                                      |
| <b>Variable-Size Signals</b>    | no                                      |
| <b>Zero-Crossing Detection</b>  | no                                      |

## More About

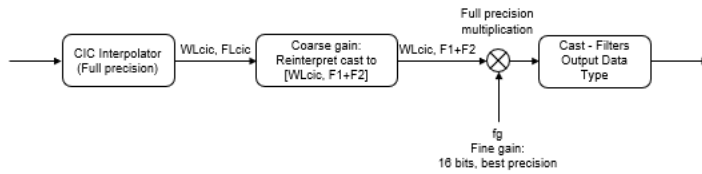
### Fixed Point

The block diagram represents the DUC arithmetic with signed fixed-point inputs.



- $WL$  is the word length of the input, and  $FL$  is the fraction length of the input.
- The output of each filter is cast to the filter output data type. In the `dsp.DigitalUpConverter` object, you can specify the filter output data type through the `FiltersOutputDataType` and `CustomFiltersOutputDataType` properties. In the Digital Up-Converter block, you can specify the filter output data type through the **Stage output** parameter. The casting of the CIC output occurs after the scaling factor is applied.
- The oscillator output is cast to a word length equal to the filter output data type word length plus one. The fraction length is equal to the filter output data type word length minus one.
- The scaling at the output of the CIC interpolator consists of coarse-gain and fine-gain adjustments. The coarse gain is achieved using the `reinterpretcast` function on the CIC interpolator output. The fine gain is achieved using full-precision multiplication.

The figure shows the coarse-gain and fine-gain operations.

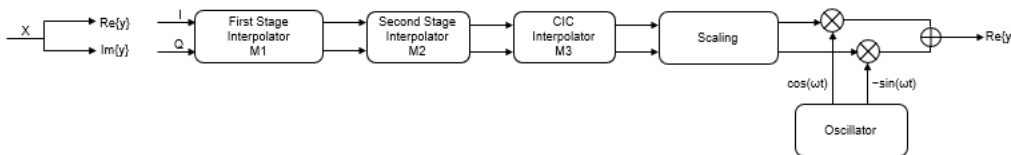


If the normalization gain is  $G$  (where  $0 < G \leq 1$ ), then:

- $WL_{cic}$  is the word length of the CIC interpolator output, and  $FL_{cic}$  is the fraction length of the CIC interpolator output.
- $F1 = \text{abs}(\text{nextpow2}(G))$ , indicating the part of  $G$  achieved by using bit shifts (coarse gain).
- $F2$  is the fraction length specified by the filter output data type.
- $fg = \text{fi}((2^{F1}) * G, \text{true}, 16)$ , which indicates that the remaining gain cannot be achieved with a bit shift (fine gain).

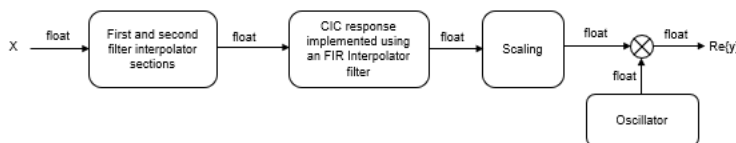
## Algorithms

The digital up converter upsamples the input signal using a cascade of three interpolation filters. This algorithm frequency-upconverts the upsampled signal by multiplying it with a complex exponential that has the specified center frequency. In this case, the filter cascade consists of an FIR interpolation stage, a second stage for CIC compensation, and a CIC interpolator. The block diagram shows the architecture of the digital up converter.



The scaling section normalizes the CIC gain and the oscillator power. It can also contain a correction factor to achieve the desired ripple specification. Depending on how you set the interpolation factor, the block bypasses the first filter stage. When the input data type is floating point, the algorithm implements an  $N$ -section CIC interpolation filter as a FIR filter with a response that corresponds to a cascade of  $N$  boxcar filters. The algorithm emulates a CIC filter with an FIR filter so that you can run simulations with floating-point data. When the input data type is a fixed-point type, the algorithm implements a true CIC filter with actual comb and integrator sections.

This block diagram represents the DUC arithmetic with floating-point inputs.



For details about fixed-point operation, see “Fixed Point” on page 2-445.

## Version History

Introduced in R2015a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

If the input is fixed point, it must be a signed integer or a signed fixed point value with a power-of-two slope and zero bias.

## See Also

### Objects

`dsp.DigitalDownConverter` | `dsp.DigitalUpConverter`

### Blocks

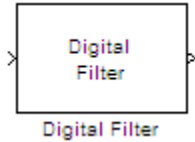
Digital Down-Converter

### Topics

“Digital Up and Down Conversion for Family Radio Service in MATLAB”

## Digital Filter (Obsolete)

Filter each channel of input over time using static or time-varying digital filter implementations



## Library

Filtering / Filter Implementations

dsparch4

## Description

---

**Note** Use of Digital Filter block in future releases is not recommended. Existing instances will continue to operate, but certain functionality will be disabled. See “Functionality being removed or replaced for blocks and System objects”. We strongly recommend using one of Discrete FIR Filter, Discrete Filter, Biquad Filter, or Allpole Filter in new designs.

---

You can use the Digital Filter block to efficiently implement a floating-point or fixed-point filter for which you know the coefficients, or that is already defined in a `dfilt` object. The block independently filters each channel of the input signal with a specified digital IIR or FIR filter. The block can implement *static filters* with fixed coefficients, as well as *time-varying filters* with coefficients that change over time. You can tune the coefficients of a static filter during simulation.

This block filters each channel of the input signal independently over time. You must set the **Input processing** parameter to specify how the block interprets the input signal. You can select one of the following options:

- **Columns as channels (frame based)** — When you select this option, the block treats each column of the input as an independent channel.
- **Elements as channels (sample based)** — When you select this option, the block treats each element of the input as an individual channel.

The output dimensions always match those of the input signal. The outputs of this block numerically match the outputs of the Digital Filter Design block and of the `dfilt` object.

---

**Note** The Digital Filter block has direct feedthrough, so if you connect the output of this block back to its input you get an algebraic loop. For more information on direct feedthrough and algebraic loops, see “Algebraic Loop Concepts” (Simulink).

---



## Sections of This Reference Page

- “Coefficient Source” on page 2-449
- “Supported Filter Structures” on page 2-449
- “Specifying Initial Conditions” on page 2-451
- “State Logging” on page 2-453
- “Fixed-Point Data Types” on page 2-454
- “Dialog Box” on page 2-454
- “Filter Structure Diagrams” on page 2-464
- “Supported Data Types” on page 2-491
- “See Also” on page 2-491

## Coefficient Source

The Digital Filter block can operate in three different modes. Select the mode in the **Coefficient source** group box.

- **Dialog parameters** Enter information about the filter such as structure and coefficients in the block mask.
- **Input port(s)** Enter the filter structure in the block mask, and the filter coefficients come in through one or more block ports. This mode is useful for specifying time-varying filters.
- **Discrete-time filter object (DFILT)** Specify the filter using a `dfilt` object.

## Supported Filter Structures

When you select **Discrete-time filter object (DFILT)**, the following `dfilt` structures are supported:

- `dfilt.df1`
- `dfilt.df1t`
- `dfilt.df2`
- `dfilt.df2t`
- `dfilt.df1sos`
- `dfilt.df1tsos`
- `dfilt.df2sos`
- `dfilt.df2tsos`
- `dfilt.dffir`
- `dfilt.dffirt`
- `dfilt.dfsymfir`
- `dfilt.dfasymfir`
- `dfilt.latticear`
- `dfilt.latticemamin`

When you select **Dialog parameters** or **Input port(s)**, the list of filter structures offered in the **Filter structure** parameter depends on whether you set the **Transfer function type** to IIR (poles & zeros), IIR (all poles), or FIR (all zeros), as summarized in the following table.

**Note** Each structure listed in the table below supports both fixed-point and floating-point signals.

The table also shows the vector or matrix of filter coefficients you must provide for each filter structure.

### Filter Structures and Filter Coefficients

| Transfer Function Type         | Supported Filter Structures                 | Filter Coefficient Specification  |
|--------------------------------|---|---|
| <b>IIR (poles &amp; zeros)</b> | Direct form I                               | <ul style="list-style-type: none"> <li>Numerator coefficients vector [b0, b1, b2, ..., bn]</li> <li>Denominator coefficients vector [a0, a1, a2, ..., am]</li> </ul> See Special Consideration for the Leading Denominator Coefficient on page 2-450. |
|                                | Direct form I transposed                    |   |
|                                | Direct form II                              |   |
|                                | Direct form II transposed                   |   |
|                                | Biquadratic direct form I (SOS)             | <ul style="list-style-type: none"> <li>M-by-6 second-order section (SOS) matrix.</li> <li>Scale values</li> </ul>   |
|                                | Biquadratic direct form I transposed (SOS)  |   |
|                                | Biquadratic direct form II (SOS)            |   |
|                                | Biquadratic direct form II transposed (SOS) |   |
| <b>IIR (all poles)</b>         | Direct form                                 | Denominator coefficients vector [a0, a1, a2, ..., am]   |
|                                | Direct form transposed                      |   |
|                                | Lattice AR                                  | Reflection coefficients vector [k1, k2, ..., kn]  |
| <b>FIR (all zeros)</b>         | Direct form                                 | Numerator coefficients vector [b0, b1, b2, ..., bn]   |
|                                | Direct form symmetric                       |   |
|                                | Direct form antisymmetric                   |   |
|                                | Direct form transposed                      |   |
|                                | Lattice MA                                  | Reflection coefficients vector [k1, k2, ..., kn]  |

#### Special Considerations for the Leading Denominator Coefficient

In some cases, the Digital Filter block requires the leading denominator coefficient ( $a_0$ ) to be 1. This requirement applies under the following conditions:

- The Digital Filter block is operating in a fixed-point mode. The block operates in a fixed-point mode when at least one of the following statements is true:
  - The input to the Digital Filter block has a fixed-point or integer data type.

- The **Fixed-point instrumentation mode** parameter under **Analysis > Fixed Point Tool** has a setting of Minimums, maximums and overflows.
- The **Coefficient source** has a setting of Dialog or Input port(s).

---

**Note** If you are working in one of the fixed-point situations described in the previous bullet, and the **Coefficient source** is set to Input port(s), you must select the **First denominator coefficient = 1, remove a0 term in the structure** check box.

---

- The **Transfer function type** and **Filter structure** parameters are set to one of the combinations described in the following table.

| Transfer function type         | Filter structure          |
|--------------------------------|---------------------------|
| <b>IIR (poles &amp; zeros)</b> | Direct form I             |
|                                | Direct form I transposed  |
|                                | Direct form II            |
|                                | Direct form II transposed |
| <b>IIR (all poles)</b>         | Direct form               |
|                                | Direct form transposed    |

The Digital Filter block produces an error if you use it in one of these configurations and your leading denominator coefficient ( $a_0$ ) does not equal 1. To resolve the error, set your leading denominator coefficient to 1 by scaling all numerator and denominator coefficients by a factor of  $a_0$ .

### Specifying Initial Conditions

In **Dialog parameters** and **Input port(s)** modes, the block initializes the internal filter states to zero by default, which is equivalent to assuming past inputs and outputs are zero. You can optionally use the **Initial conditions** parameter to specify nonzero initial conditions for the filter delays.

To determine the number of initial condition values you must specify, and how to specify them, see the following table on Valid Initial Conditions and Number of Delay Elements (Filter States). The **Initial conditions** parameter can take one of four forms as described in the following table.

## Valid Initial Conditions

| Initial Condition   | Examples   | Description   |
|---|--|---|
| Scalar  | 5<br><br>Each delay element for each channel is set to 5.  | The block initializes all delay elements in the filter to the scalar value.   |
| Vector<br>(for applying the same delay elements to each channel)            | For a filter with two delay elements: $[d_1 \ d_2]$<br><br>The delay elements for all channels are $d_1$ and $d_2$ .   | Each vector element specifies a unique initial condition for a corresponding delay element. The block applies the same vector of initial conditions to each channel of the input signal. The vector length must equal the number of delay elements in the filter (specified in the table Number of Delay Elements (Filter States)).   |
| Vector or matrix<br>(for applying different delay elements to each channel) | For a 3-channel input signal and a filter with two delay elements:<br>$[d_1 \ d_2 \ D_1 \ D_2 \ d_1 \ d_2]$ or<br>$\begin{bmatrix} d_1 & D_1 & d_1 \\ d_2 & D_2 & d_2 \end{bmatrix}$ <ul style="list-style-type: none"> <li>The delay elements for channel 1 are <math>d_1</math> and <math>d_2</math>.</li> <li>The delay elements for channel 2 are <math>D_1</math> and <math>D_2</math>.</li> <li>The delay elements for channel 3 are <math>d_1</math> and <math>d_2</math>.</li> </ul> | Each vector or matrix element specifies a unique initial condition for a corresponding delay element in a corresponding channel: <ul style="list-style-type: none"> <li>The vector length must be equal to the product of the number of input channels and the number of delay elements in the filter (specified in the table Number of Delay Elements (Filter States)).</li> <li>The matrix must have the same number of rows as the number of delay elements in the filter (specified in the table Number of Delay Elements (Filter States)), and must have one column for each channel of the input signal.</li> </ul> |
| Empty matrix  | $[ \ ]$<br>Each delay element for each channel is set to $\theta$ .  | The empty matrix, $[ \ ]$ , is equivalent to setting the <b>Initial conditions</b> parameter to the scalar value $\theta$ .   |

The number of delay elements (filter states) per input channel depends on the filter structure, as indicated in the following table.

**Number of Delay Elements (Filter States)**

| Filter Structure   | Number of Delay Elements per Channel   |
|--|--|
| Direct form<br>Direct form transposed<br>Direct form symmetric<br>Direct form antisymmetric  | #_of_filter_coeffs-1   |
| Direct form I<br>Direct form I transposed  | <ul style="list-style-type: none"> <li>• #_of_zeros-1</li> <li>• #_of_poles-1</li> </ul> |
| Direct form II<br>Direct form II transposed  | max(#_of_zeros, #_of_poles)-1  |
| Biquadratic direct form I (SOS)<br>Biquadratic direct form I transposed (SOS)<br>Biquadratic direct form II (SOS)<br>Biquadratic direct form II transposed (SOS) | 2 * #_of_filter_sections   |
| Lattice AR<br>Lattice MA   | #_of_reflection_coeffs   |

**State Logging**

Simulink enables you to log the states in your model to the MATLAB workspace. The following table indicates which filter structures of the Digital Filter block support the Simulink state logging feature. See “State” (Simulink) for more information.

| Transfer Function Type         | Filter Structure                            | State Logging Supported |
|--------------------------------|---|-------------------------|
| <b>IIR (poles &amp; zeros)</b> | Direct form I                               | No                      |
|                                | Direct form I transposed                    | Yes                     |
|                                | Direct form II                              | No                      |
|                                | Direct form II transposed                   | Yes                     |
|                                | Biquadratic direct form I (SOS)             | Yes                     |
|                                | Biquadratic direct form I transposed (SOS)  | Yes                     |
|                                | Biquadratic direct form II (SOS)            | Yes                     |
|                                | Biquadratic direct form II transposed (SOS) | Yes                     |
| <b>IIR (all poles)</b>         | Direct form                                 | No                      |
|                                | Direct form transposed                      | Yes                     |
|                                | Lattice AR                                  | Yes                     |
| <b>FIR (all zeros)</b>         | Direct form                                 | No                      |
|                                | Direct form symmetric                       | No                      |
|                                | Direct form antisymmetric                   | No                      |
|                                | Direct form transposed                      | Yes                     |
|                                | Lattice MA                                  | Yes                     |

## Fixed-Point Data Types

All structures supported by the Digital Filter block support fixed-point data types. You can specify intermediate fixed-point data types for quantities such as the coefficients, accumulator, and product output for each filter structure. See “Filter Structure Diagrams” on page 2-464 for diagrams depicting the use of these intermediate fixed-point data types in each filter structure.

## Dialog Box

### Coefficient Source

The Digital Filter block can operate in three different modes. Select the mode in the **Coefficient source** group box.

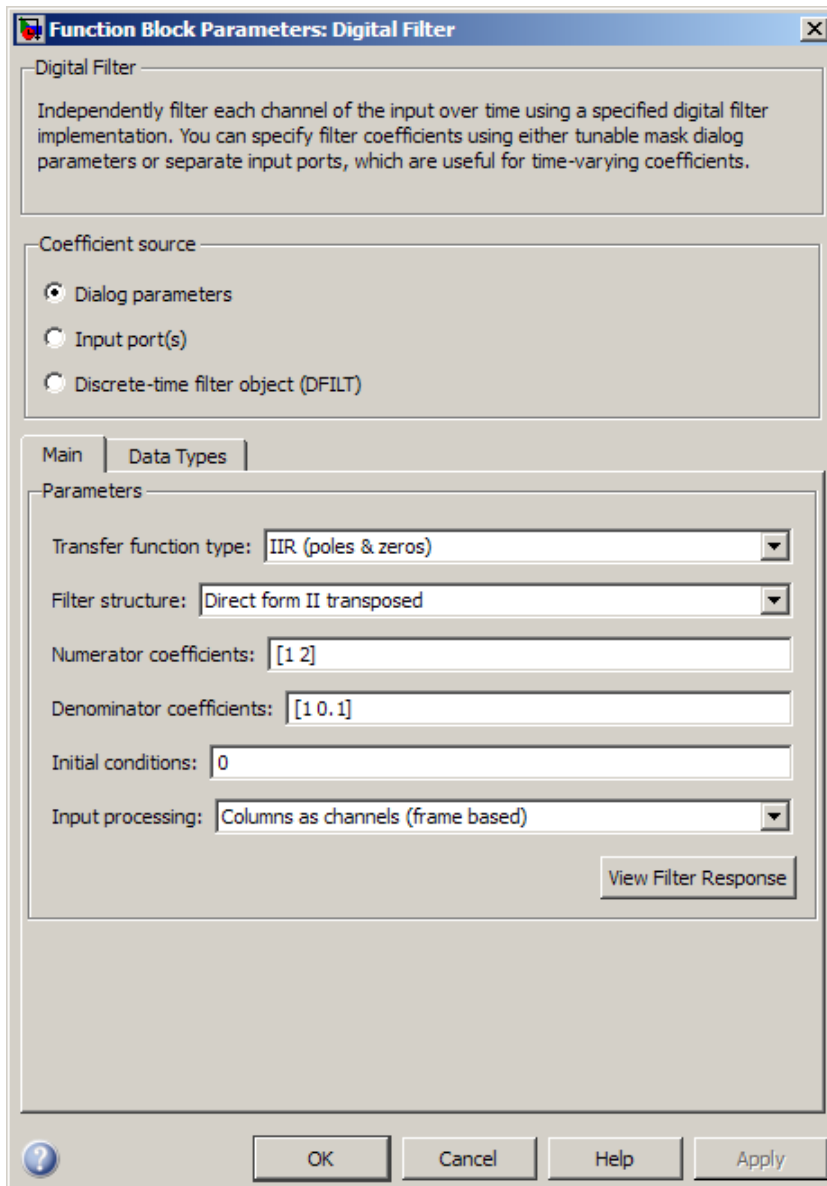
- **Dialog parameters** Enter information about the filter such as structure and coefficients in the block mask.
- **Input port(s)** Enter the filter structure in the block mask, and the filter coefficients come in through one or more block ports. This mode is useful for specifying time-varying filters.
- **Discrete-time filter object (DFILT)** Specify the filter using a `dfilt` object.

Different items appear on the Digital Filter block dialog depending on whether you select **Dialog parameters**, **Input port(s)**, or **Discrete-time filter object (DFILT)** in the **Coefficient source** group box. See the following sections for details:

- “Specify Filter Characteristics in Dialog and/or Through Input Ports” on page 2-454
- “Specify Discrete-Time Filter Object” on page 2-461

### Specify Filter Characteristics in Dialog and/or Through Input Ports

The **Main** pane of the Digital Filter block dialog appears as follows when **Dialog parameters** is specified in the **Coefficient source** group box. The parameters below can appear when **Dialog parameters** or **Input port(s)** is selected, as noted.



### Transfer function type

Select the type of transfer function of the filter; IIR (poles & zeros), IIR (all poles), or FIR (all zeros). See “Supported Filter Structures” on page 2-449 for more information.

### Filter structure

Select the filter structure. The selection of available structures varies depending the setting of the **Transfer function type** parameter. See “Supported Filter Structures” on page 2-449 for more information.

### Numerator coefficients

Specify the vector of numerator coefficients of the filter's transfer function.

This parameter is only visible when **Dialog parameters** is selected *and* when the selected filter structure lends itself to specification with numerator coefficients. Tunable (Simulink).

**Denominator coefficients**

Specify the vector of denominator coefficients of the filter's transfer function.

In some cases, the leading denominator coefficient ( $a_0$ ) must be 1. See *Special Consideration for the Leading Denominator Coefficient* on page 2-450 for more information.

This parameter is only visible when **Dialog parameters** is selected *and* when the selected filter structure lends itself to specification with denominator coefficients. Tunable (Simulink).

**Reflection coefficients**

Specify the vector of reflection coefficients of the filter's transfer function.

This parameter is only visible when **Dialog parameters** is selected *and* when the selected filter structure lends itself to specification with reflection coefficients. Tunable (Simulink).

**SOS matrix (Mx6)**

Specify an  $M$ -by-6 *SOS matrix* containing coefficients of a second-order section (SOS) filter, where  $M$  is the number of sections. You can use the `ss2sos` and `tf2sos` functions from Signal Processing Toolbox software to check whether your SOS matrix is valid.

This parameter is only visible when **Dialog parameters** is selected *and* when the selected filter structure is biquadratic. Tunable (Simulink).

**Scale values**

Specify the scale values to be applied before and after each section of a biquadratic filter.

- If you specify a scalar, that value is applied before the first filter section. The rest of the scale values are set to 1.
- You can also specify a vector with  $M + 1$  elements, assigning a different value to each scale. See “Filter Structure Diagrams” on page 2-464 for diagrams depicting the use of scale values in biquadratic filter structures.

This parameter is only visible when **Dialog parameters** is selected *and* when the selected filter structure is biquadratic. Tunable (Simulink).

**First denominator coefficient = 1, remove  $a_0$  term in the structure**

Select this parameter to reduce the number of computations the block must make to produce the output by omitting the  $1 / a_0$  term in the filter structure. The block output is invalid if you select this parameter when the first denominator filter coefficient is *not* always 1 for your time-varying filter.

This parameter is only enabled when the **Input port(s)** is selected *and* when the selected filter structure lends itself to this specification.

**Coefficient update rate**

Specify how often the block updates time-varying filters; once per sample or once per frame.

This parameter appears only when the following conditions are met:

- You specify **Input port(s)** in the Coefficient source group box.
- You set the **Input processing** parameter to `Columns as channels (frame based)`.

**Initial conditions**

Specify the initial conditions of the filter states. To learn how to specify initial conditions, see “Specifying Initial Conditions” on page 2-451.



### Initial conditions on zeros side

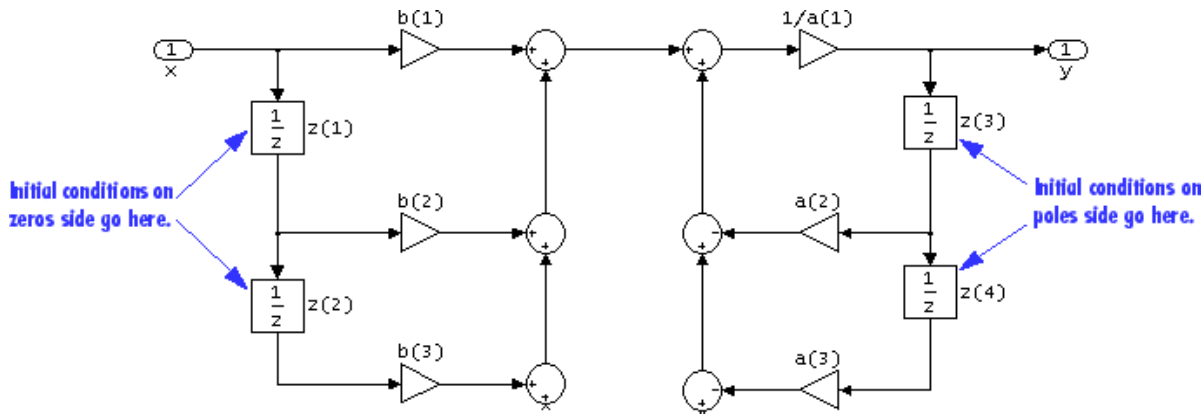
(Not shown in dialog above.) Specify the initial conditions for the filter states on the side of the filter structure with the zeros ( $b_0, b_1, b_2, \dots$ ); see the diagram below.

This parameter is enabled only when the filter has both poles and zeros, *and* when you select a structure such as direct form I, which has separate filter states corresponding to the poles ( $a_k$ ) and zeros ( $b_k$ ). To learn how to specify initial conditions, see “Specifying Initial Conditions” on page 2-451.

### Initial conditions on poles side

(Not shown in dialog above.) Specify the initial conditions for the filter states on the side of the filter structure with the poles ( $a_0, a_1, a_2, \dots$ ); see the diagram below.

This parameter is enabled only when the filter has both poles and zeros, *and* when you select a structure such as direct form I, which has separate filter states corresponding to the poles ( $a_k$ ) and zeros ( $b_k$ ). To learn how to specify initial conditions, see “Specifying Initial Conditions” on page 2-451.



### Input processing

Specify how the block should process the input. You can set this parameter to one of the following options:

- Columns as channels (frame based) — When you select this option, the block treats each column of the input as a separate channel.
- Elements as channels (sample based) — When you select this option, the block treats each element of the input as a separate channel.

---

**Note** The Inherited (this choice will be removed - see release notes) option will be removed in a future release. See “Frame-Based Processing” in the *DSP System Toolbox Release Notes* for more information.

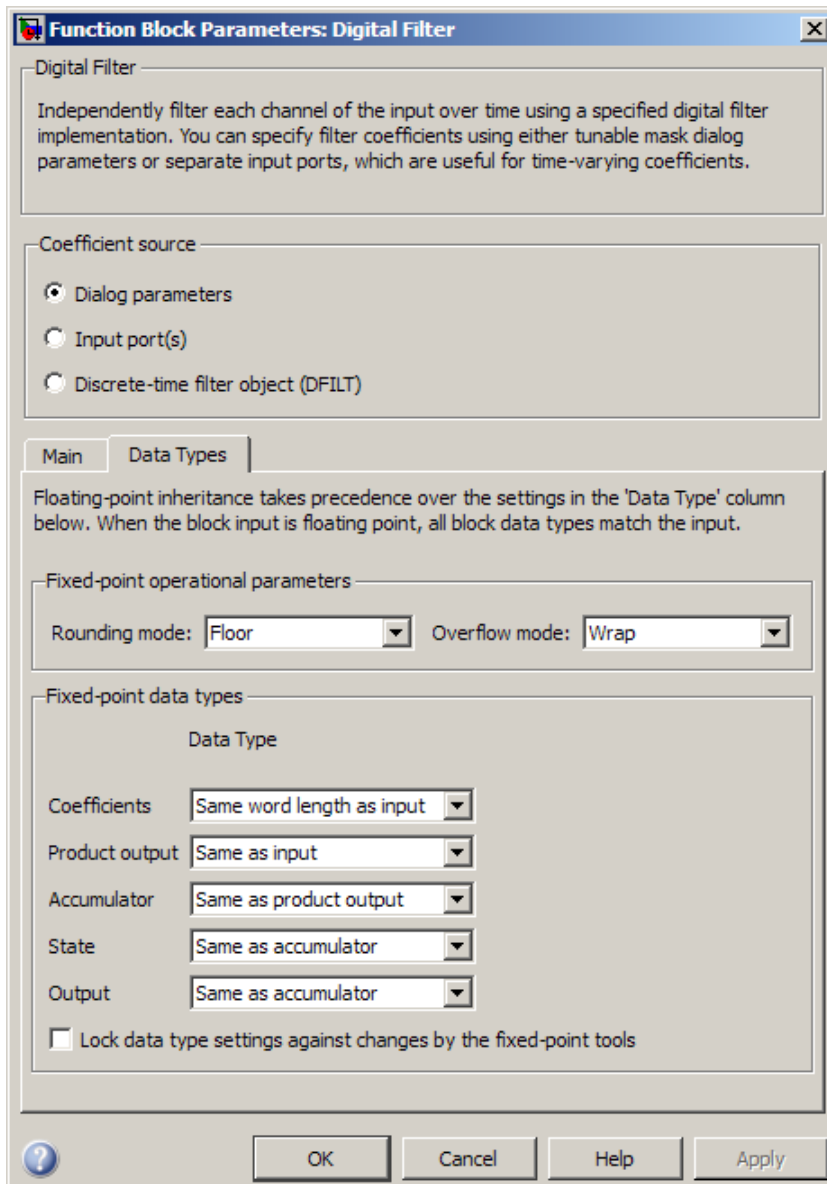
---

### View filter response

This button opens the Filter Visualization Tool (**FVTool**) from the Signal Processing Toolbox product and displays the filter response of the filter defined by the block. For more information on FVTool, see the Signal Processing Toolbox documentation.

**Note** If you specify a filter in the **Filter** parameter, you must click the **Apply** button to apply the filter before using the **View filter response** button.

The **Data Types** pane of the Digital Filter block dialog appears as follows when **Dialog parameters** is specified in the **Coefficient source** group box. The parameters below can appear when **Dialog parameters** or **Input port(s)** is selected, depending on the filter structure and whether the coefficients are being entered via ports or on the block mask.



### Rounding mode

Select the rounding mode for fixed-point operations. The filter coefficients do not obey this parameter; they always round to Nearest.

**Overflow mode**

Select the overflow mode for fixed-point operations. The filter coefficients do not obey this parameter; they are always saturated.

**Section I/O**

Choose how you specify the word length and the fraction length of the fixed-point data type going into and coming out of each section of a biquadratic filter. See “Filter Structure Diagrams” on page 2-464 for illustrations depicting the use of the section I/O data type in this block.

This parameter is only visible when the selected filter structure is biquadratic:

- When you select `Same as input`, these characteristics match those of the input to the block.
- When you select `Binary point scaling`, you can enter the word and fraction lengths of the section input and output, in bits.
- When you select `Slope and bias scaling`, you can enter the word lengths, in bits, and the slopes of the section input and output. This block requires power-of-two slope and a bias of zero.

**Tap sum**

Choose how you specify the word length and the fraction length of the tap sum data type of a direct form symmetric or direct form antisymmetric filter. See “Filter Structure Diagrams” on page 2-464 for illustrations depicting the use of the tap sum data type in this block.

This parameter is only visible when the selected filter structure is either `Direct form symmetric` or `Direct form antisymmetric`:

- When you select `Same as input`, these characteristics match those of the input to the block.
- When you select `Binary point scaling`, you can enter the word length and the fraction length of the tap sum accumulator, in bits.
- When you select `Slope and bias scaling`, you can enter the word length, in bits, and the slope of the tap sum accumulator. This block requires power-of-two slope and a bias of zero.

**Multiplicand**

Choose how you specify the word length and the fraction length of the multiplicand data type of a direct form I transposed or biquadratic direct form I transposed filter. See “Filter Structure Diagrams” on page 2-464 for illustrations depicting the use of the multiplicand data type in this block.

This parameter is only visible when the selected filter structure is either `Direct form I transposed` or `Biquad direct form I transposed (SOS)`:

- When you select `Same as output`, these characteristics match those of the output to the block.
- When you select `Binary point scaling`, you can enter the word length and the fraction length of the multiplicand data type, in bits.
- When you select `Slope and bias scaling`, you can enter the word length, in bits, and the slope of the multiplicand data type. This block requires power-of-two slope and a bias of zero.

**Coefficients**

Choose how you specify the word length and the fraction length of the filter coefficients (numerator and/or denominator). See “Filter Structure Diagrams” on page 2-464 for illustrations depicting the use of the coefficient data types in this block:

- When you select **Same word length as input**, the word length of the filter coefficients match that of the input to the block. In this mode, the fraction length of the coefficients is automatically set to the binary-point only scaling that provides you with the best precision possible given the value and word length of the coefficients.
- When you select **Specify word length**, you can enter the word length of the coefficients, in bits. In this mode, the fraction length of the coefficients is automatically set to the binary-point only scaling that provides you with the best precision possible given the value and word length of the coefficients.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the coefficients, in bits. If applicable, you can enter separate fraction lengths for the numerator and denominator coefficients.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the coefficients. If applicable, you can enter separate slopes for the numerator and denominator coefficients. This block requires power-of-two slope and a bias of zero.
- The filter coefficients do not obey the **Rounding mode** and the **Overflow mode** parameters; they are always saturated and rounded to Nearest.

### Product output

Use this parameter to specify how you would like to designate the product output word and fraction lengths. See “Filter Structure Diagrams” on page 2-464 and “Multiplication Data Types” for illustrations depicting the use of the product output data type in this block:

- When you select **Same as input**, these characteristics match those of the input to the block.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the product output, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the product output. This block requires power-of-two slope and a bias of zero.

### Accumulator

Use this parameter to specify how you would like to designate the accumulator word and fraction lengths. See “Filter Structure Diagrams” on page 2-464 and “Multiplication Data Types” for illustrations depicting the use of the accumulator data type in this block:

- When you select **Same as input**, these characteristics match those of the input to the block.
- When you select **Same as product output**, these characteristics match those of the product output.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the accumulator, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the accumulator. This block requires power-of-two slope and a bias of zero.

### State

Use this parameter to specify how you would like to designate the state word and fraction lengths. See “Filter Structure Diagrams” on page 2-464 for illustrations depicting the use of the state data type in this block.

This parameter is not visible for direct form and direct form I filter structures.

- When you select **Same as input**, these characteristics match those of the input to the block.

- When you select `Same as accumulator`, these characteristics match those of the accumulator.
- When you select `Binary point scaling`, you can enter the word length and the fraction length of the accumulator, in bits.
- When you select `Slope and bias scaling`, you can enter the word length, in bits, and the slope of the accumulator. This block requires power-of-two slope and a bias of zero.

### Output

Choose how you specify the output word length and fraction length:

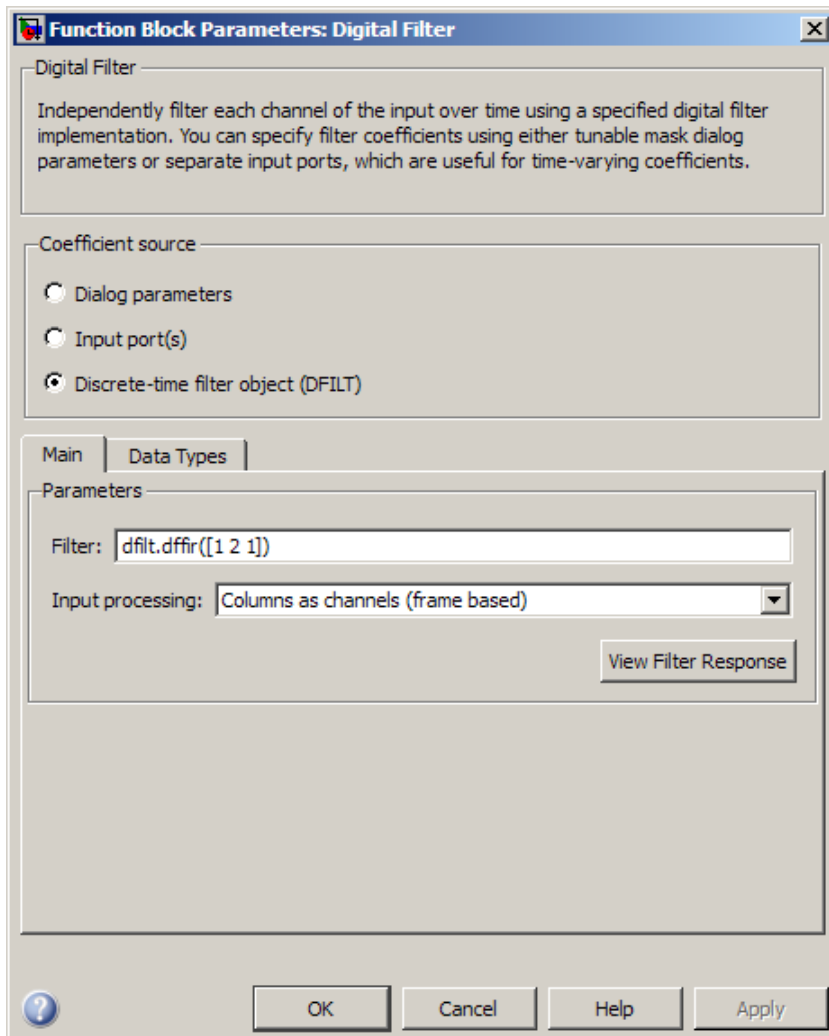
- When you select `Same as input`, these characteristics match those of the input to the block.
- When you select `Same as accumulator`, these characteristics match those of the accumulator.
- When you select `Binary point scaling`, you can enter the word length and the fraction length of the output, in bits.
- When you select `Slope and bias scaling`, you can enter the word length, in bits, and the slope of the output. This block requires power-of-two slope and a bias of zero.

### Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

### Specify Discrete-Time Filter Object

The **Main** pane of the Digital Filter block dialog appears as follows when **Discrete-time filter object (DFILT)** is specified in the **Coefficient source** group box.



### Filter

Specify the discrete-time filter object (`dfilt`) that you would like the block to implement. You can do this in one of three ways:

- You can fully specify the `dfilt` object in the block mask, as shown in the default value.
- You can enter the variable name of a `dfilt` object that is defined in any workspace.
- You can enter a variable name for a `dfilt` object that is not yet defined.

### Input processing

Specify how the block should process the input. You can set this parameter to one of the following options:

- **Columns as channels (frame based)** — When you select this option, the block treats each column of the input as a separate channel.
- **Elements as channels (sample based)** — When you select this option, the block treats each element of the input as a separate channel.

---

**Note** The Inherited (this choice will be removed - see release notes) option will be removed in a future release. See “Frame-Based Processing” in the *DSP System Toolbox Release Notes* for more information.

---

### View filter response

This button opens the Filter Visualization Tool (**FVTool**) from the Signal Processing Toolbox product and displays the filter response of the `dfilt` object specified in the **Filter** parameter. For more information on FVTool, see the Signal Processing Toolbox documentation.

---

**Note** If you specify a filter in the **Filter** parameter, you must click the **Apply** button to apply the filter before using the **View filter response** button.

---

The **Data Types** pane of the Digital Filter block dialog appears as follows when **Discrete-time filter object (DFILT)** is specified in the **Coefficient source** group box.



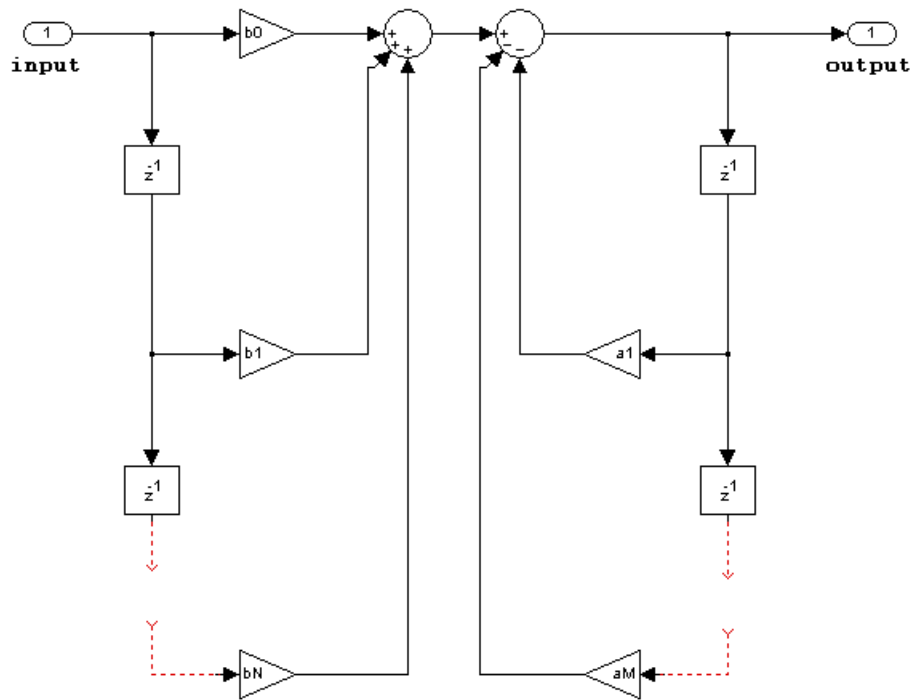
The fixed-point settings of the filter object specified on the **Main** pane are displayed on the **Data Types** pane. You cannot change these settings directly on the block mask. To change the fixed-point settings you must edit the filter object directly.

## Filter Structure Diagrams

The diagrams in the following sections show the filter structures supported by the Digital Filter block. They also show the data types used in the filter structures for fixed-point signals. You can set the coefficient, output, accumulator, product output, and state data types shown in these diagrams in the block dialog. This is discussed in “Dialog Box” on page 2-454.

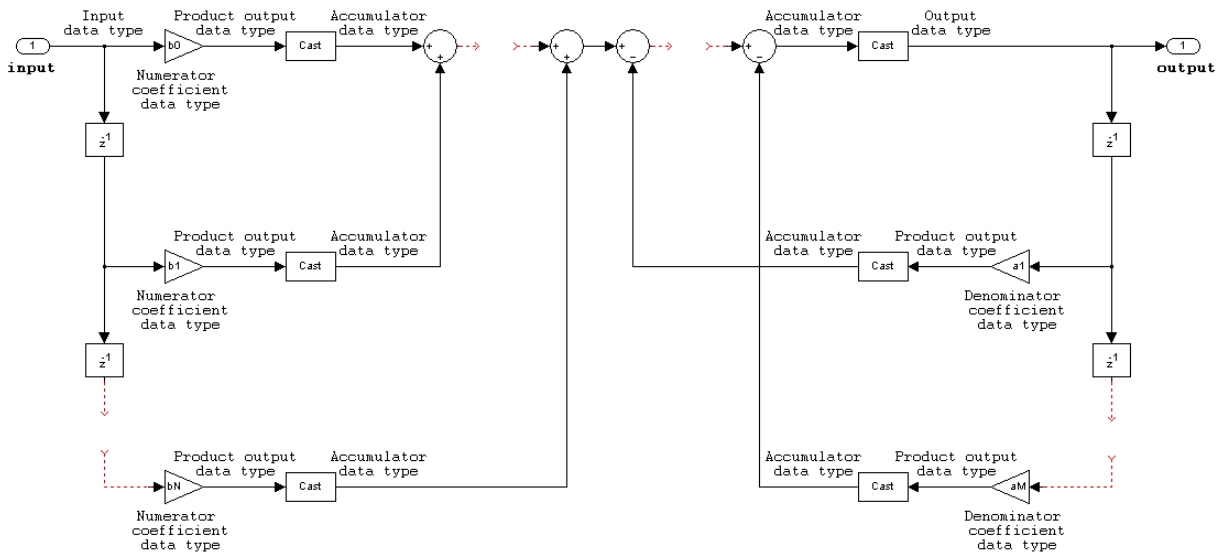
- “IIR direct form I” on page 2-465
- “IIR direct form I transposed” on page 2-466
- “IIR direct form II” on page 2-468
- “IIR direct form II transposed” on page 2-469
- “IIR biquadratic direct form I” on page 2-471
- “IIR biquadratic direct form I transposed” on page 2-472
- “IIR biquadratic direct form II” on page 2-474
- “IIR biquadratic direct form II transposed” on page 2-475
- “IIR (all poles) direct form” on page 2-477
- “IIR (all poles) direct form transposed” on page 2-479
- “IIR (all poles) direct form lattice AR” on page 2-481
- “FIR (all zeros) direct form” on page 2-482
- “FIR (all zeros) direct form symmetric” on page 2-483
- “FIR (all zeros) direct form antisymmetric” on page 2-485
- “FIR (all zeros) direct form transposed” on page 2-487
- “FIR (all zeros) lattice MA” on page 2-488



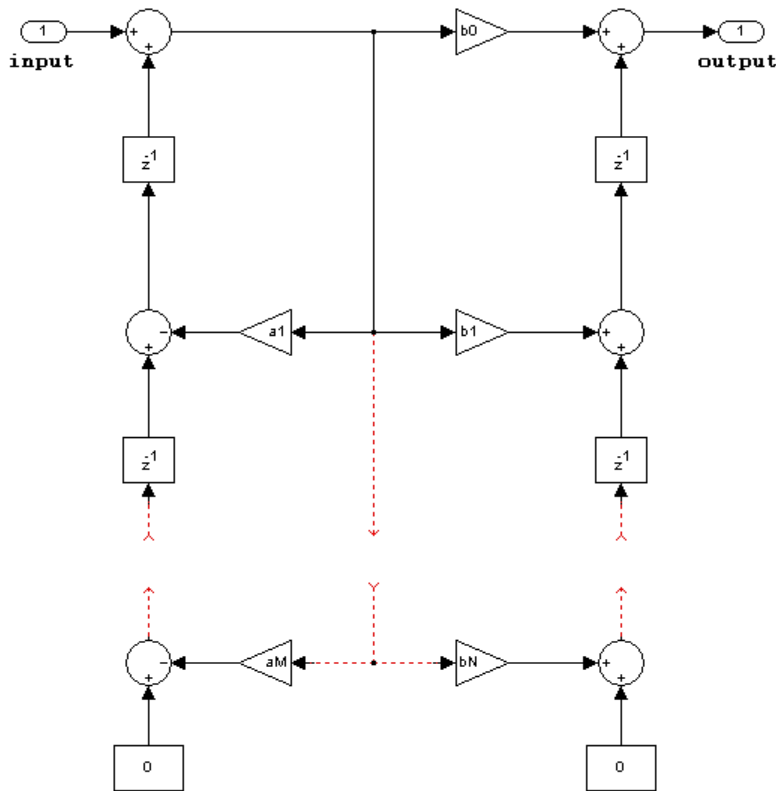
**IIR direct form I**

The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs can be real or complex.
- Numerator and denominator coefficients can be real or complex.
- Numerator and denominator coefficients must be the same complexity as each other.
  - When the numerator and denominator coefficients are specified via input ports and have different complexities from each other, you get an error.
  - When the numerator and denominator coefficients are specified in the dialog and have different complexities from each other, the block does not error, but instead processes the filter as if two sets of complex coefficients are provided. The coefficient set that is real-valued is treated as if it is a complex vector with zero-valued imaginary parts.
- Numerator and denominator coefficients must have the same word length. They can have different fraction lengths.
- The State data type cannot be specified on the block mask for this structure, because the input and output states have the same data types as the input and output buffers.



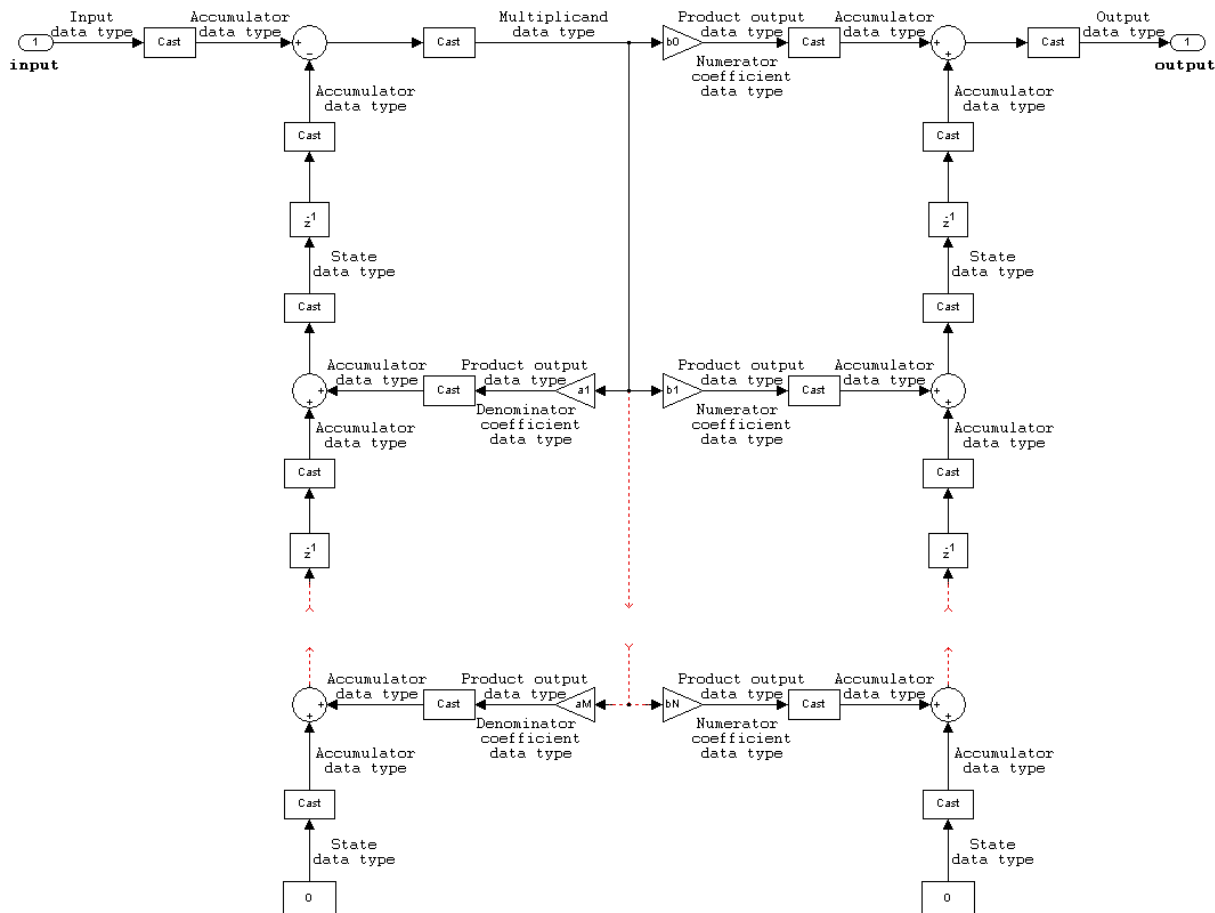
**IIR direct form I transposed**



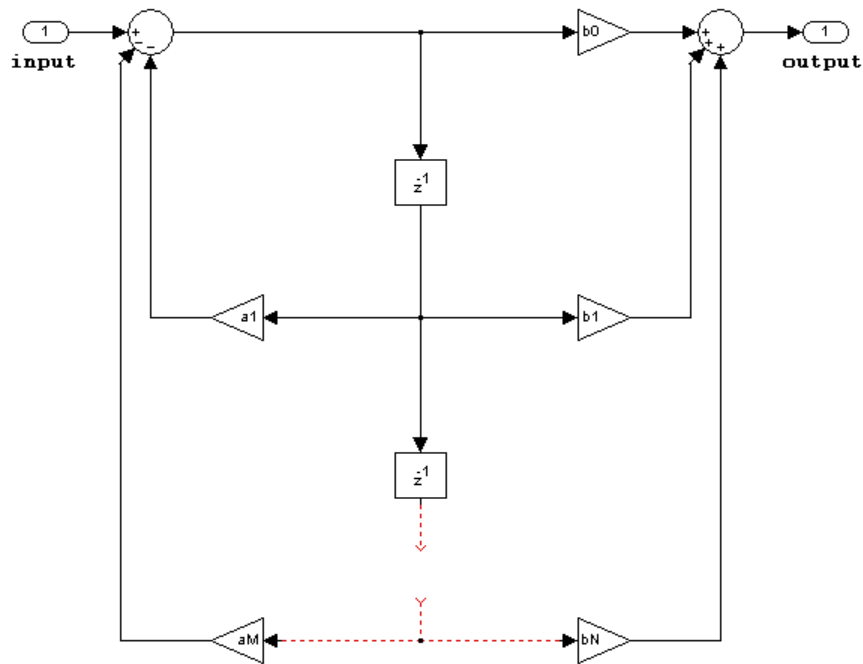
The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs can be real or complex.
- Numerator and denominator coefficients can be real or complex.

- Numerator and denominator coefficients must be the same complexity as each other.
  - When the numerator and denominator coefficients are specified via input ports and have different complexities from each other, you get an error.
  - When the numerator and denominator coefficients are specified in the dialog and have different complexities from each other, the block does not error, but instead processes the filter as if two sets of complex coefficients are provided. The coefficient set that is real-valued is treated as if it is a complex vector with zero-valued imaginary parts.
- States are complex when either the input or the coefficients are complex.
- Numerator and denominator coefficients must have the same word length. They can have different fraction lengths.

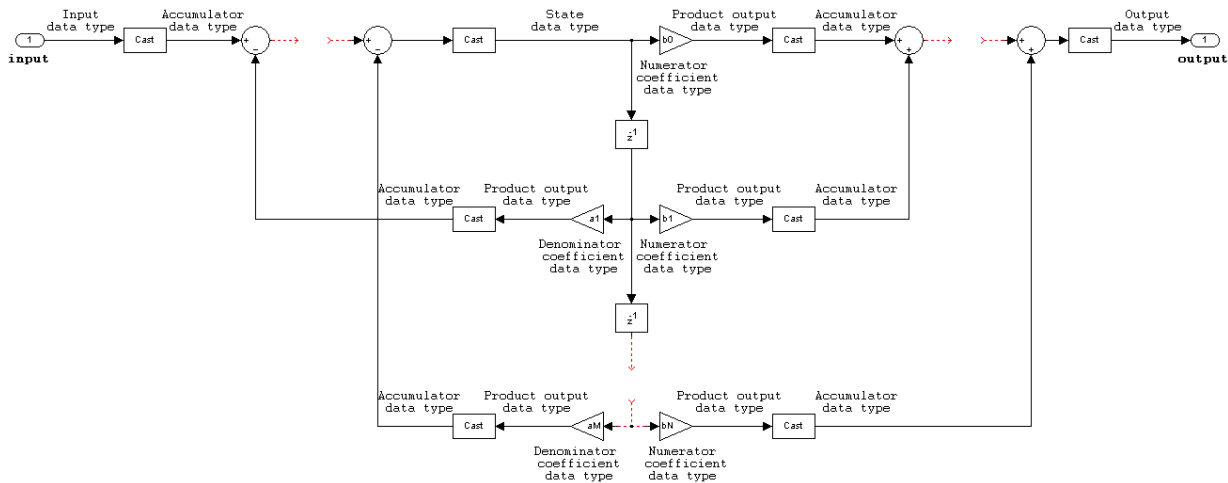


## IIR direct form II

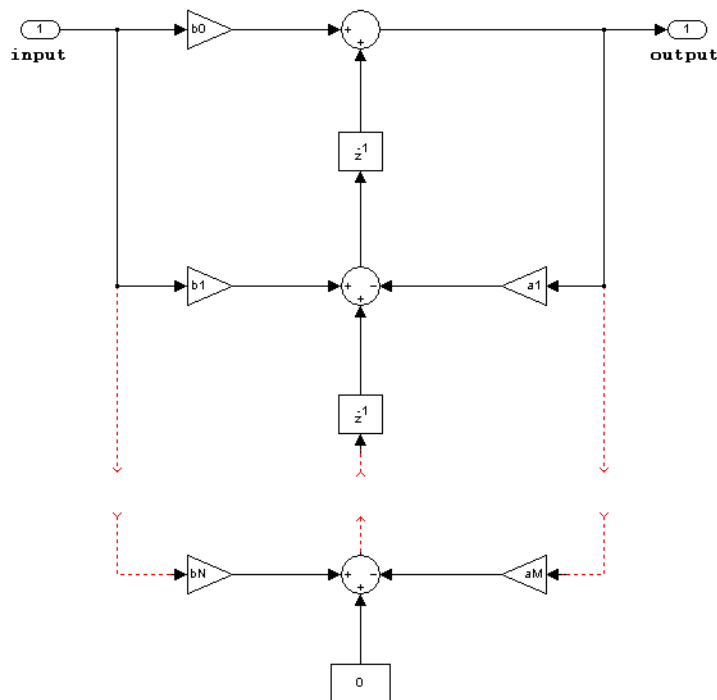


The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs can be real or complex.
- Numerator and denominator coefficients can be real or complex.
- Numerator and denominator coefficients must be the same complexity as each other.
  - When the numerator and denominator coefficients are specified via input ports and have different complexities from each other, you get an error.
  - When the numerator and denominator coefficients are specified in the dialog and have different complexities from each other, the block does not error, but instead processes the filter as if two sets of complex coefficients are provided. The coefficient set that is real-valued is treated as if it is a complex vector with zero-valued imaginary parts.
- States are complex when either the inputs or the coefficients are complex.
- Numerator and denominator coefficients must have the same word length. They can have different fraction lengths.



### IIR direct form II transposed

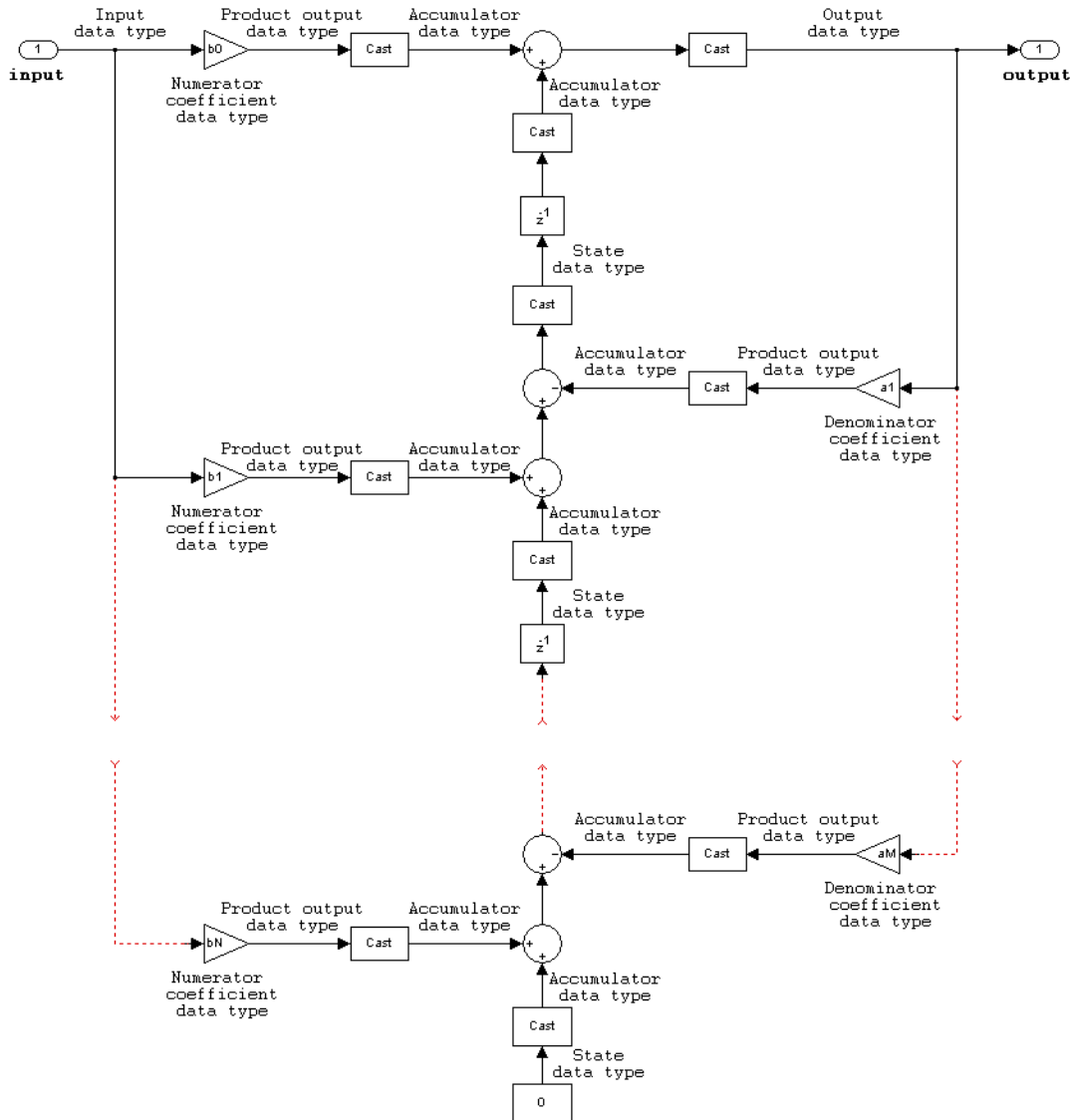


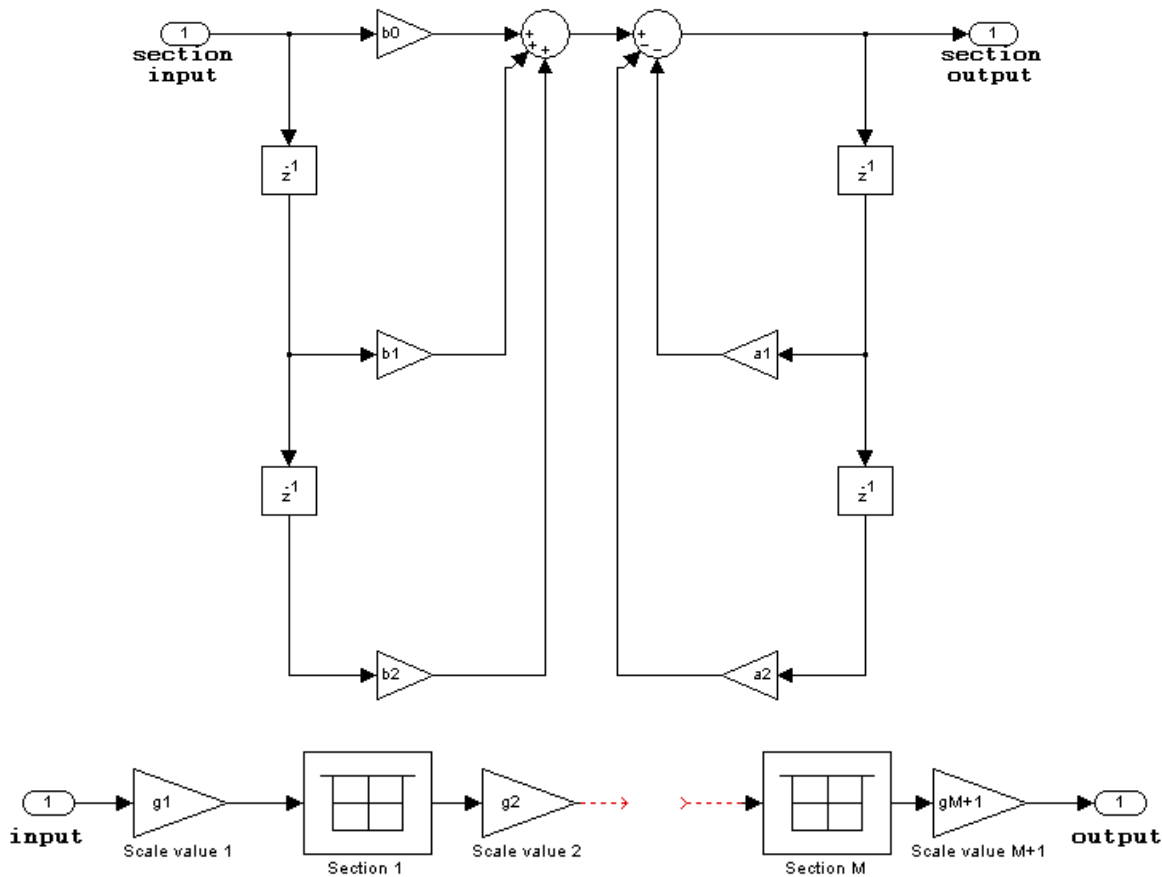
The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs can be real or complex.
- Numerator and denominator coefficients can be real or complex.
- Numerator and denominator coefficients must be the same complexity as each other.
  - When the numerator and denominator coefficients are specified via input ports and have different complexities from each other, you get an error.
  - When the numerator and denominator coefficients are specified in the dialog and have different complexities from each other, the block does not error, but instead processes the filter

as if two sets of complex coefficients are provided. The coefficient set that is real-valued is treated as if it is a complex vector with zero-valued imaginary parts.

- States are complex when either the inputs or the coefficients are complex.
- Numerator and denominator coefficients must have the same word length. They can have different fraction lengths.

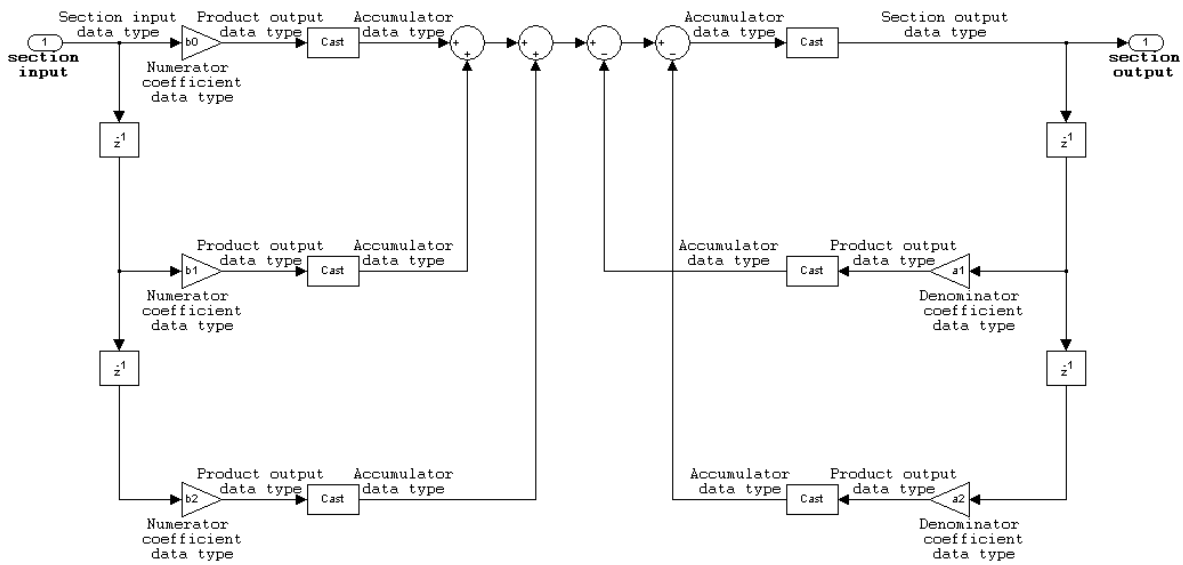


**IIR biquadratic direct form I**

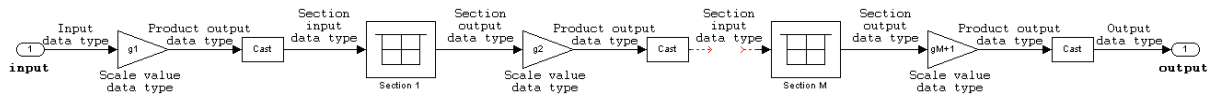
The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs and coefficients can be real or complex.
- Numerator and denominator coefficients can be real or complex.
- Specify the coefficients by a  $M$ -by-6 matrix in the block mask. You cannot specify coefficients by input ports for this filter structure.
- When the  $a_0$  element of any row is not equal to one, that row is normalized by  $a_0$  prior to filtering.
- States are complex when either the inputs or the coefficients are complex.
- You cannot specify the state data type on the block mask for this structure, because the input and output states have the same data types as the input.
- Scale values must have the same complexity as the coefficient SOS matrix.
- The scale value parameter must be a scalar or a vector of length  $M+1$ , where  $M$  is the number of sections.
- The **Section I/O** parameter determines the data type for the section input and output data types. The section input and stage output data type must have the same word length but can have different fraction lengths.

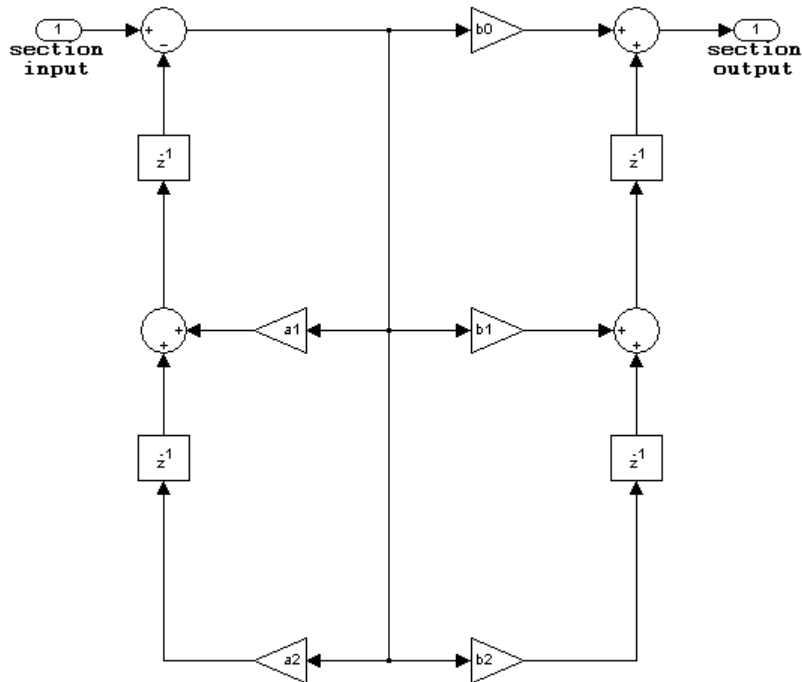
The following diagram shows the data types for one section of the filter.



The following diagram shows the data types between filter sections.

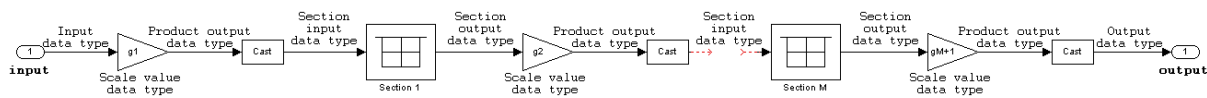


**IIR biquadratic direct form I transposed**

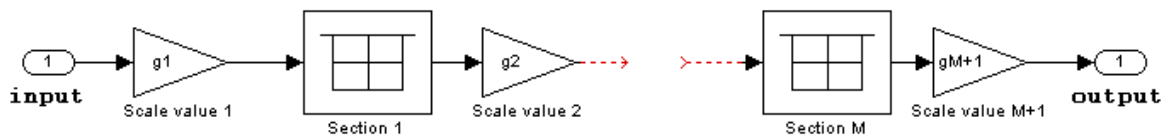
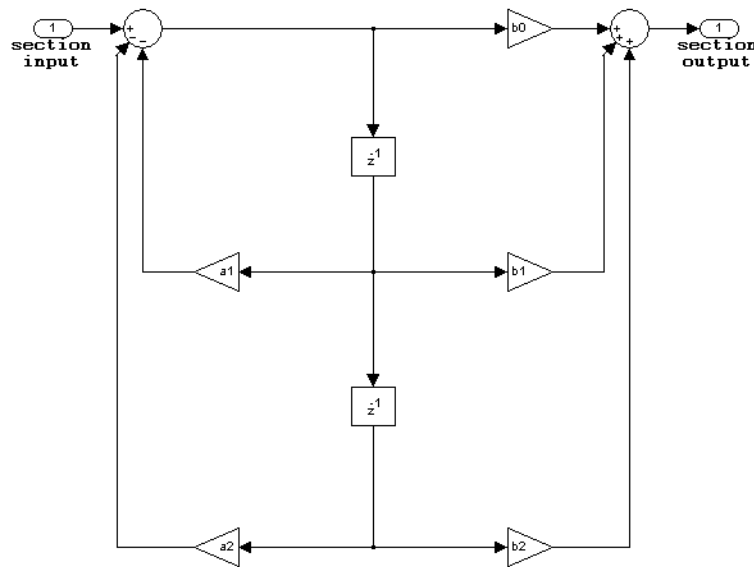








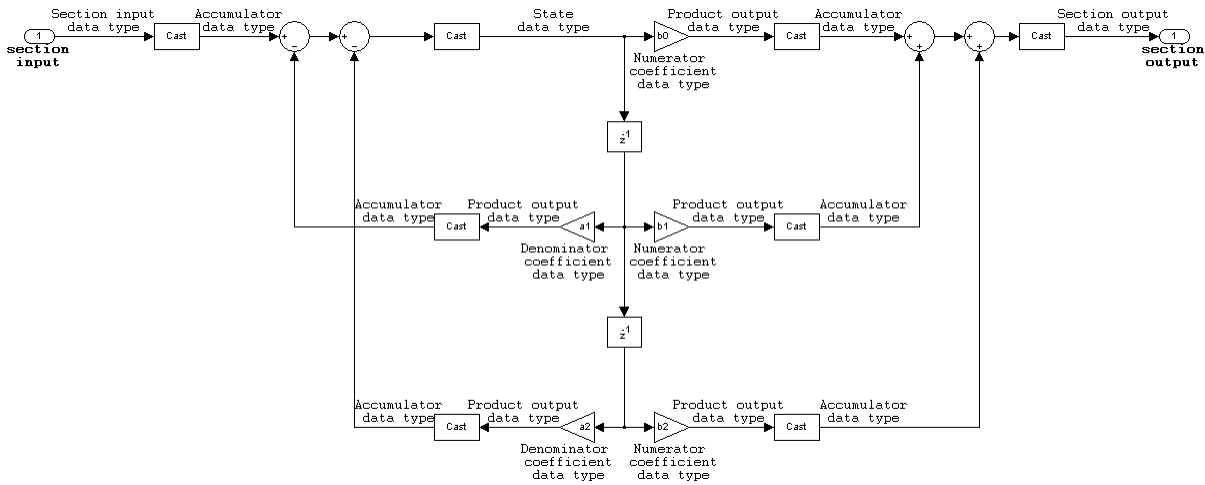
### IIR biquadratic direct form II



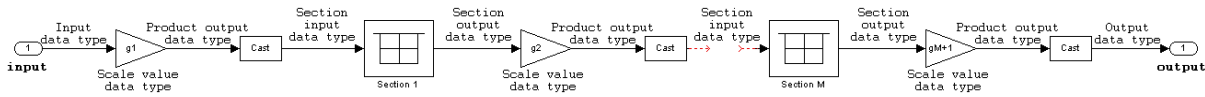
The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs and coefficients can be real or complex.
- Numerator and denominator coefficients can be real or complex.
- Specify the coefficients by a  $M$ -by-6 matrix in the block mask. You cannot specify coefficients by input ports for this filter structure.
- When the  $a_0$  element of any row is not equal to one, that row is normalized by  $a_0$  prior to filtering.
- States are complex when either the inputs or the coefficients are complex.
- Scale values must have the same complexity as the coefficient SOS matrix.
- The scale value parameter must be a scalar or a vector of length  $M+1$ , where  $M$  is the number of sections.
- The **Section I/O** parameter determines the data type for the section input and output data types. The section input and section output data type must have the same word length but can have different fraction lengths.

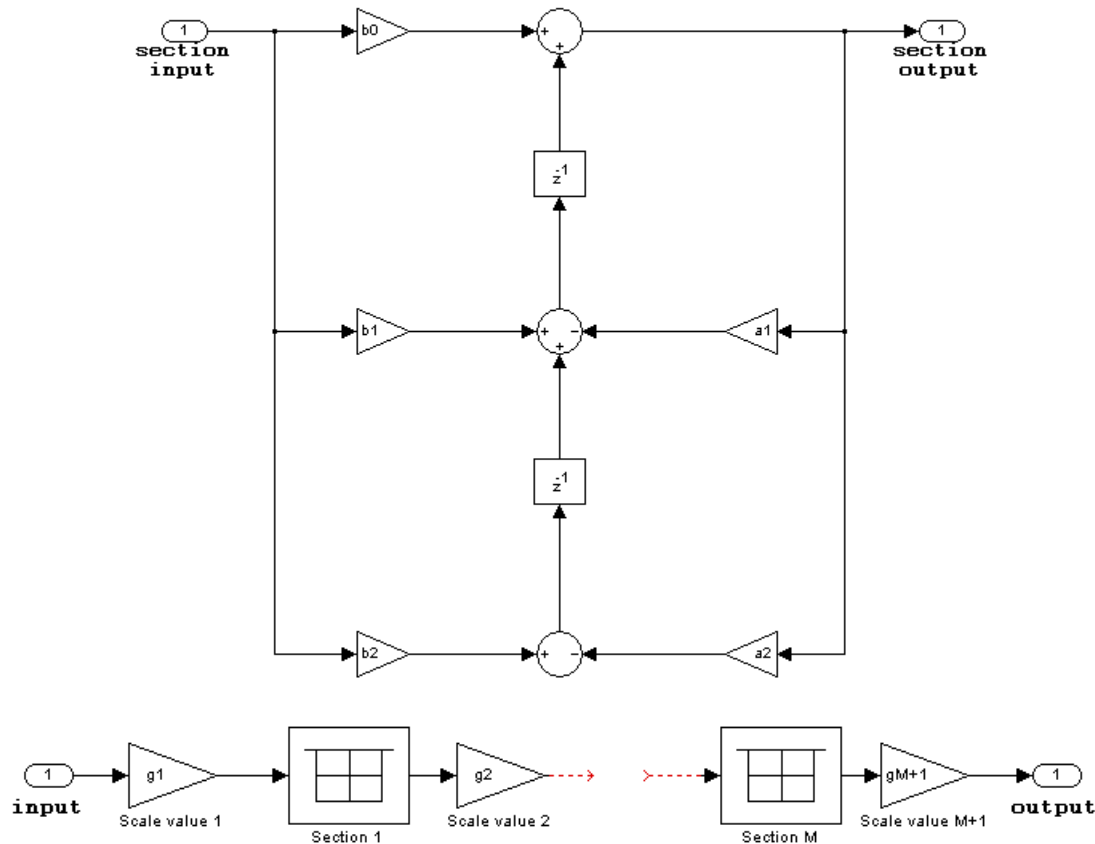
The following diagram shows the data types for one section of the filter.



The following diagram shows the data types between filter sections.



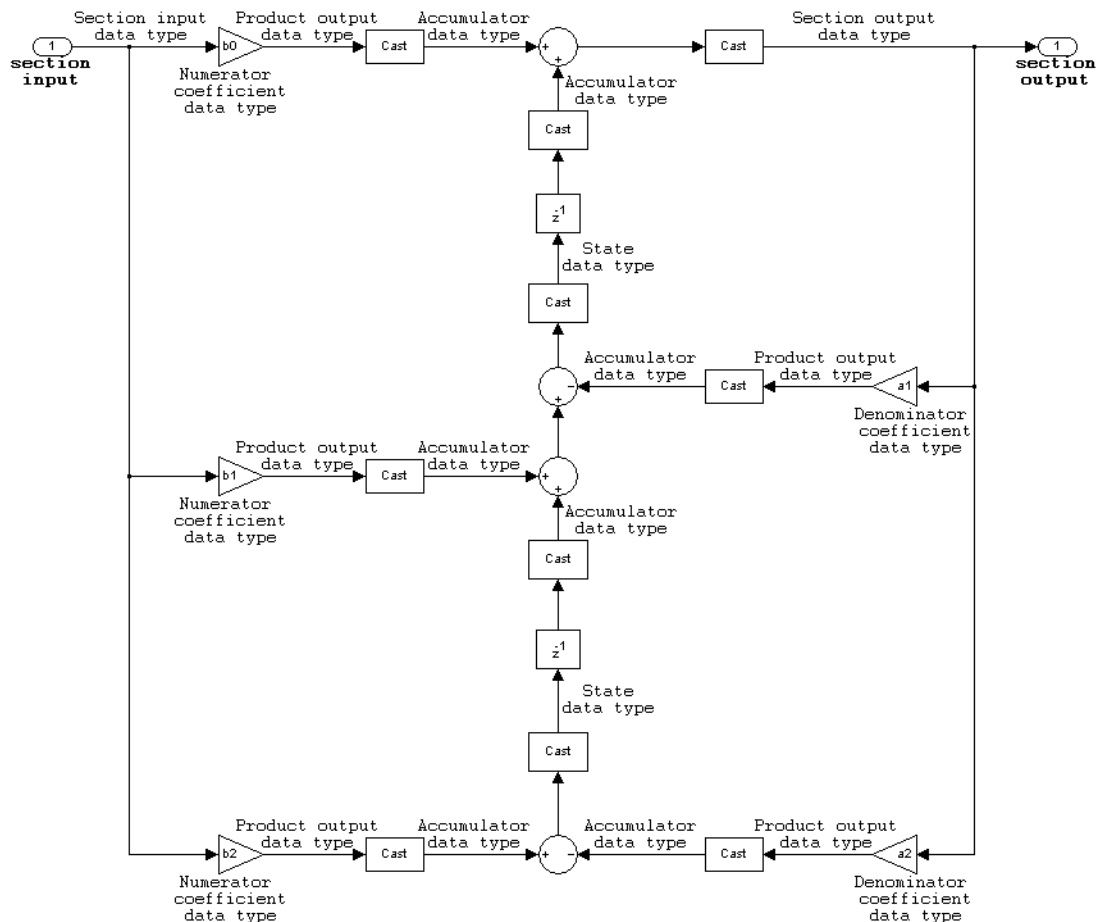
**IIR biquadratic direct form II transposed**



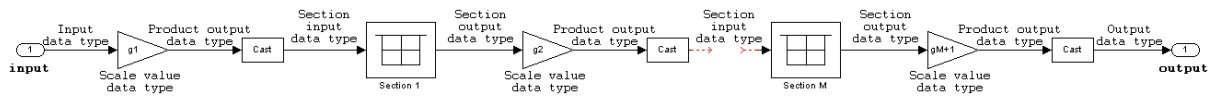
The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs and coefficients can be real or complex.
- Numerator and denominator coefficients can be real or complex.
- Specify the coefficients by a  $M$ -by-6 matrix in the block mask. You cannot specify coefficients by input ports for this filter structure.
- When the  $a_0$  element of any row is not equal to one, that row is normalized by  $a_0$  prior to filtering.
- States are complex when either the inputs or the coefficients are complex.
- Scale values must have the same complexity as the coefficient SOS matrix.
- The scale value parameter must be a scalar or a vector of length  $M+1$ , where  $M$  is the number of sections.
- The **Section I/O** parameter determines the data type for the section input and output data types. The section input and section output data type must have the same word length but can have different fraction lengths.

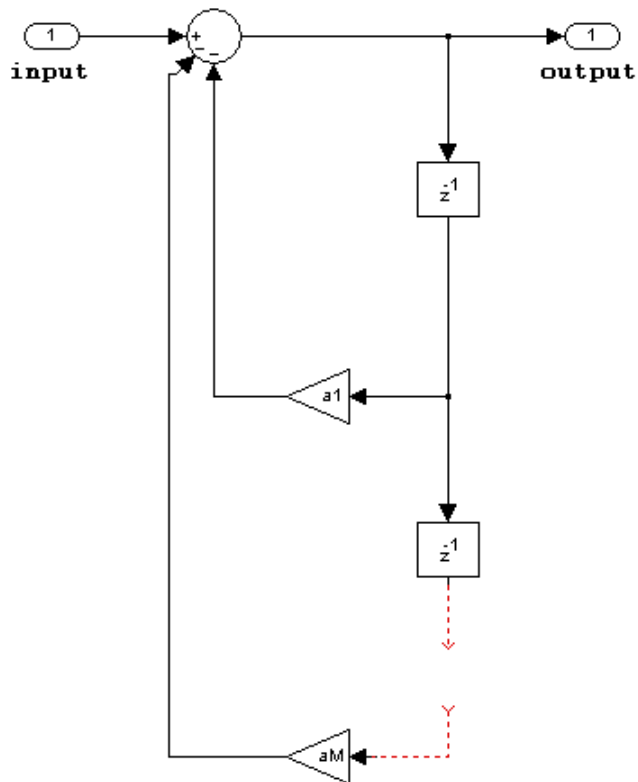
The following diagram shows the data types for one section of the filter.



The following diagram shows the data types between filter sections.

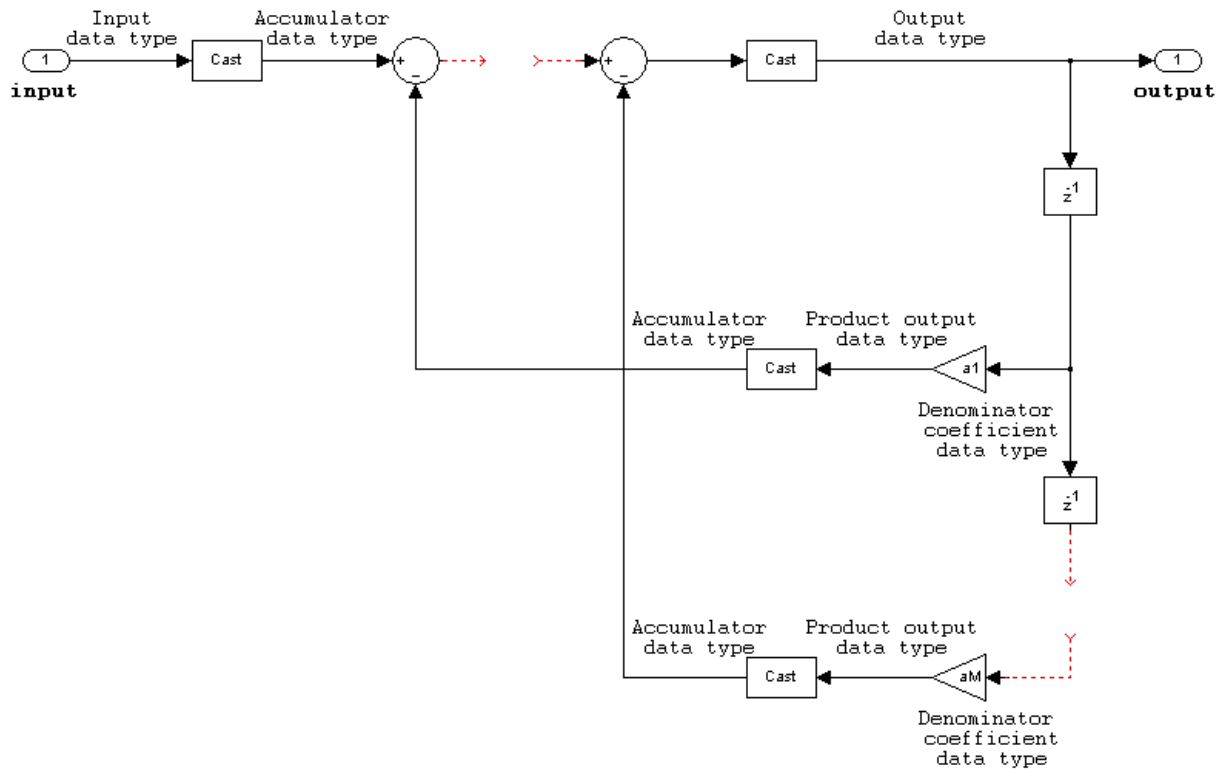


### IIR (all poles) direct form

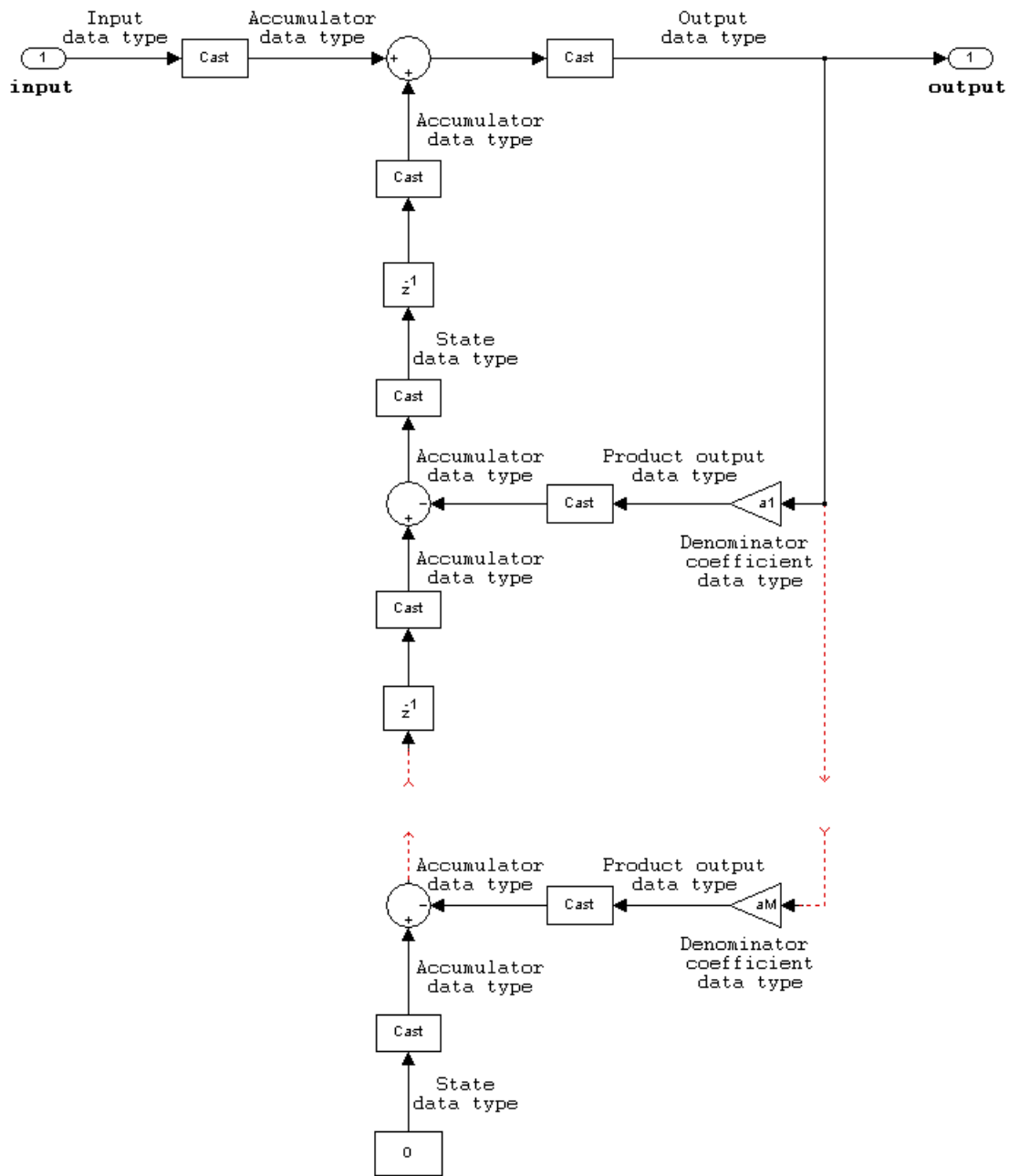


The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs and coefficients can be real or complex.
- Denominator coefficients can be real or complex.
- You cannot specify the state data type on the block mask for this structure, because the input and output states have the same data types as the input.



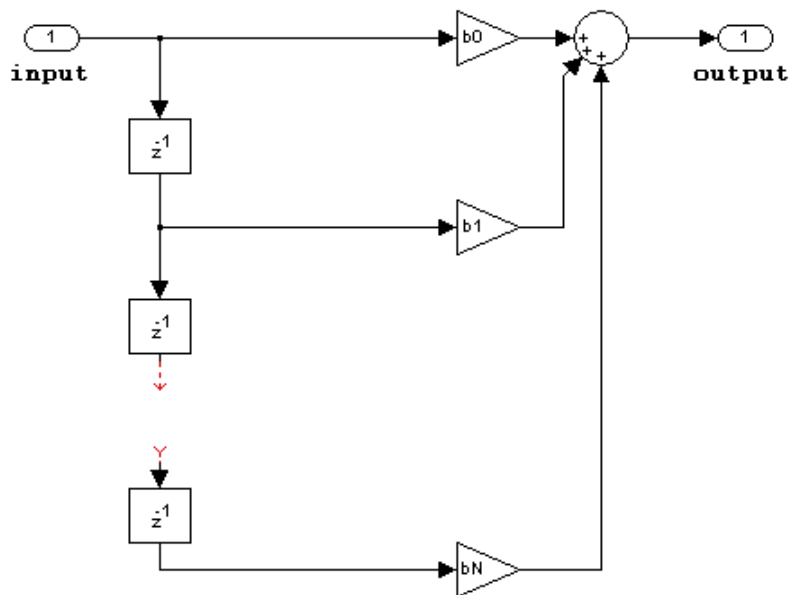






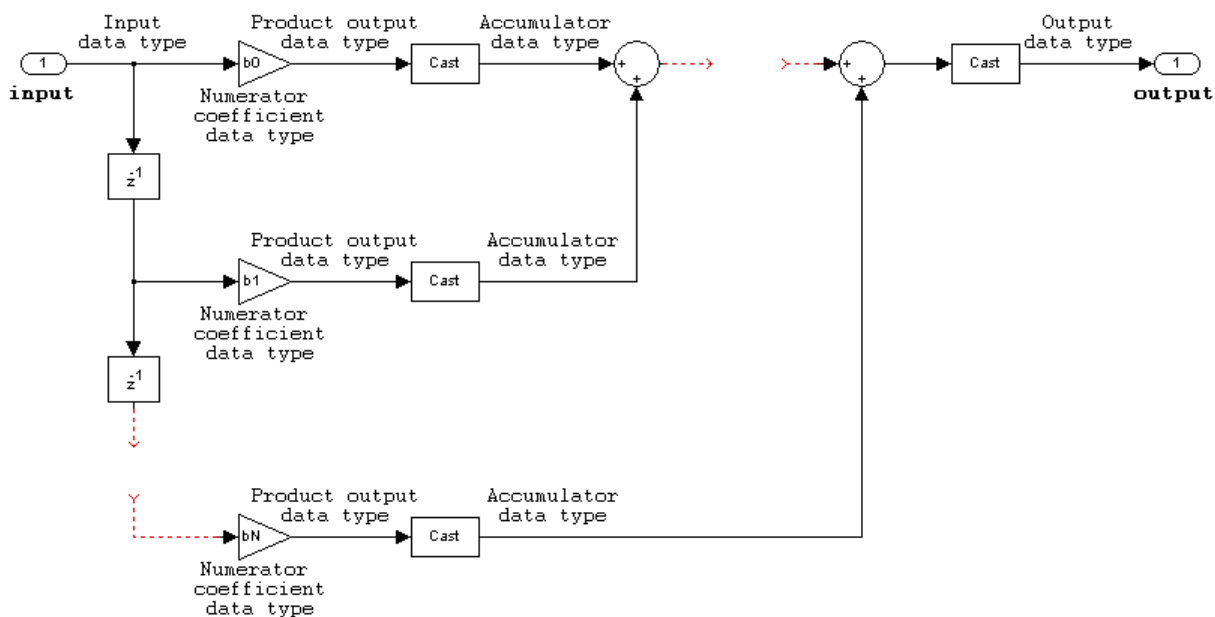


## FIR (all zeros) direct form

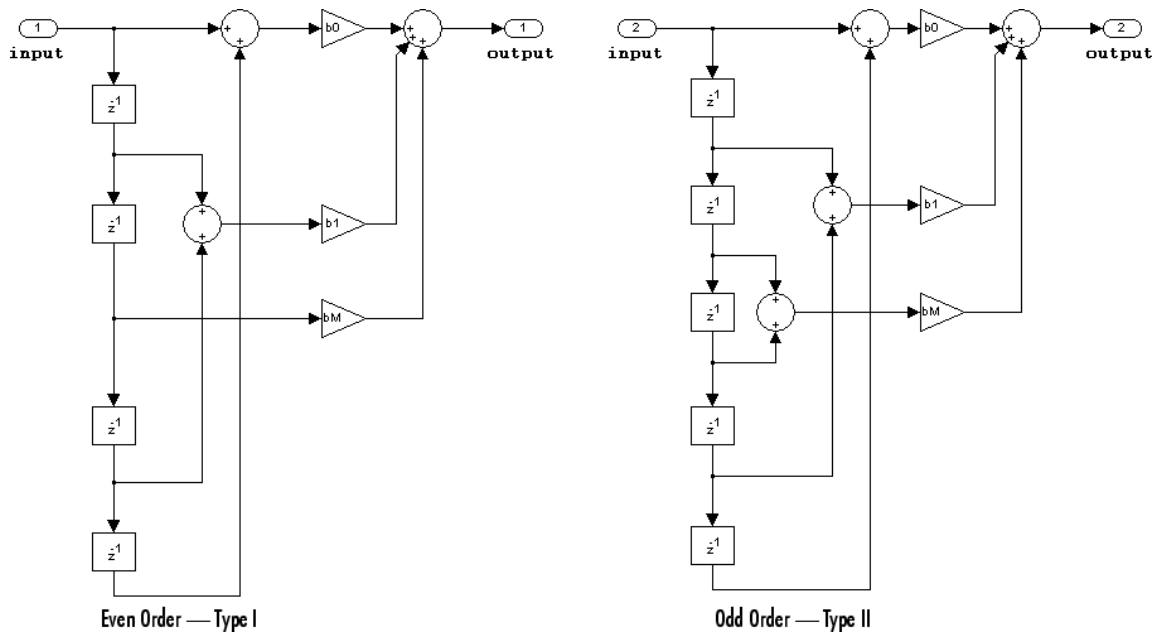


The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs can be real or complex.
- Numerator coefficients can be real or complex.
- You cannot specify the state data type on the block mask for this structure, because the input and output states have the same data types as the input.

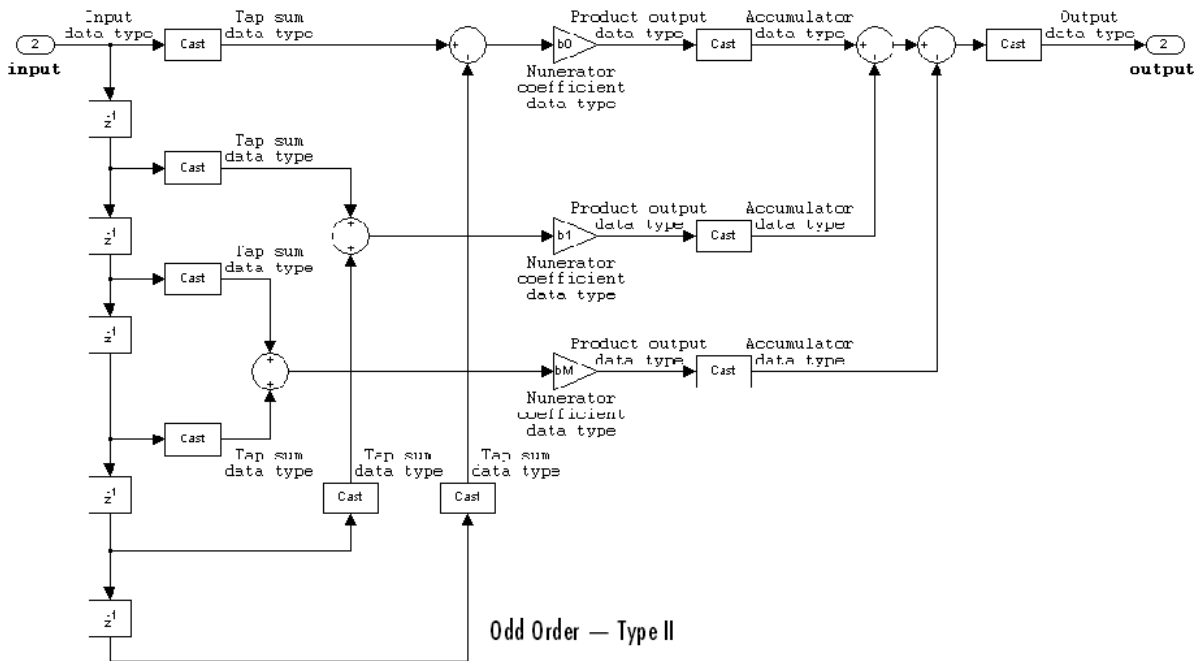
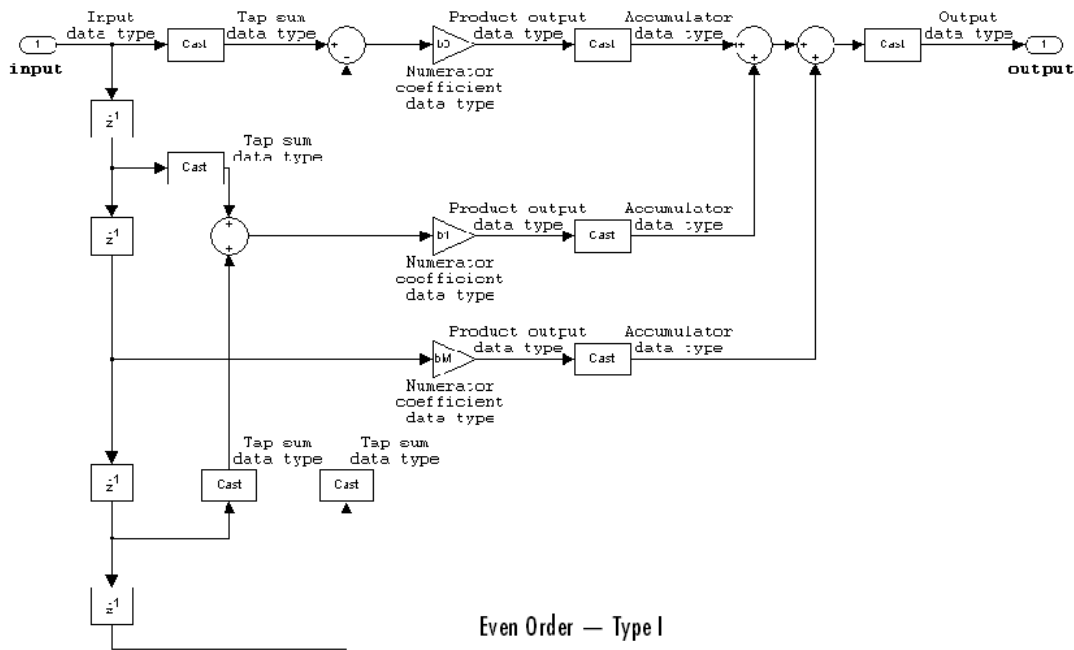


### FIR (all zeros) direct form symmetric

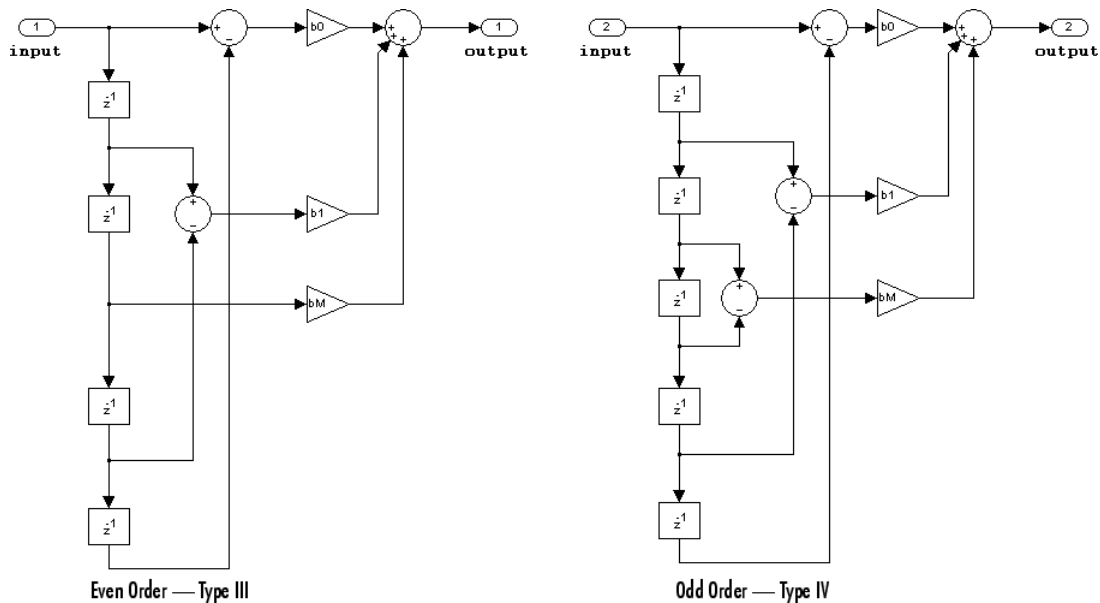


The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs can be real or complex.
- Numerator coefficients can be real or complex.
- You cannot specify the state data type on the block mask for this structure, because the input and output states have the same data types as the input.
- It is assumed that the filter coefficients are symmetric. Only the first half of the coefficients are used for filtering.
- The **Tap Sum** parameter determines the data type the filter uses when it sums the inputs prior to multiplication by the coefficients.

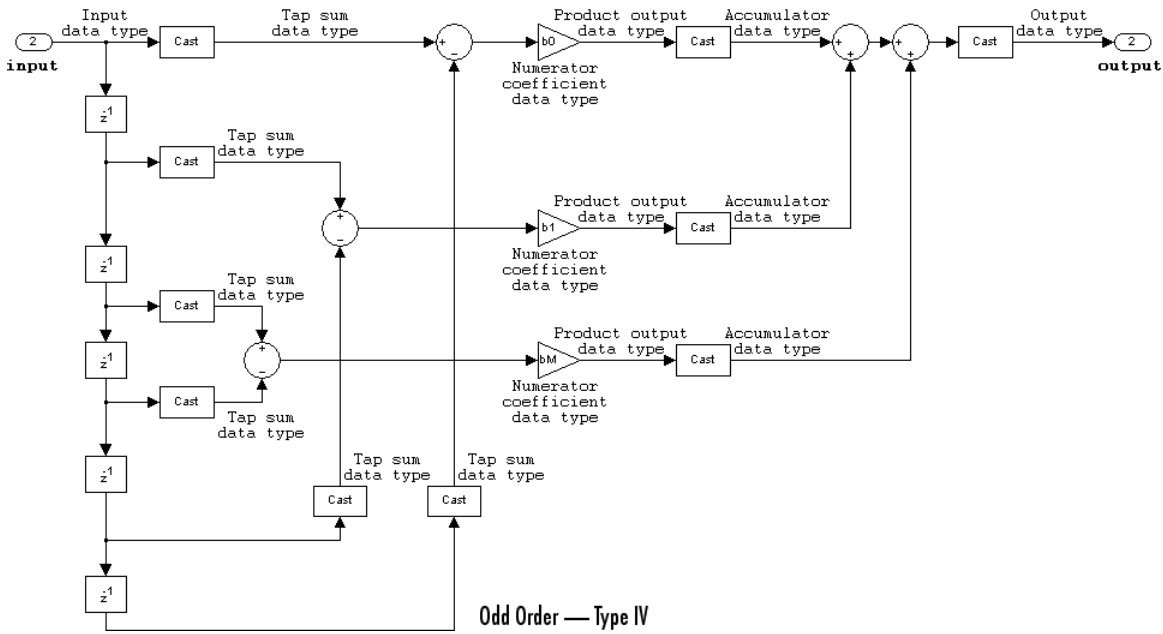
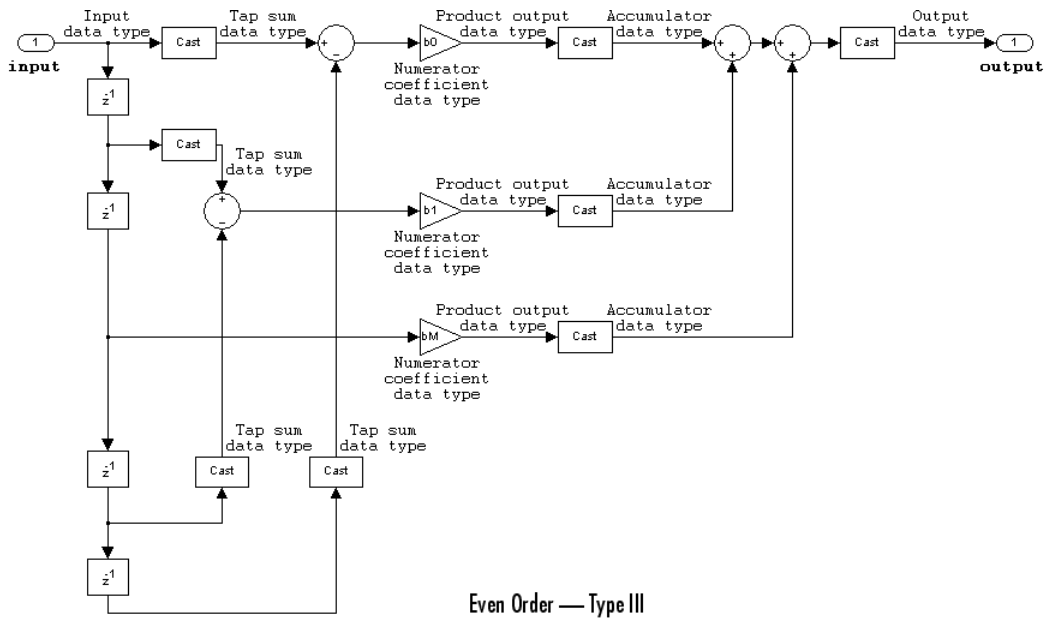


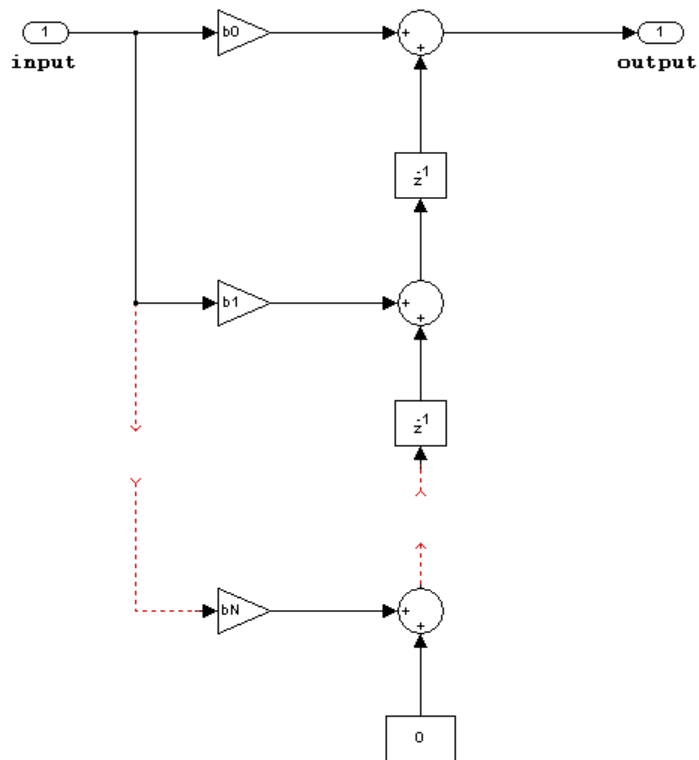
### FIR (all zeros) direct form antisymmetric



The following constraints are applicable when processing a fixed-point signal with this filter structure:

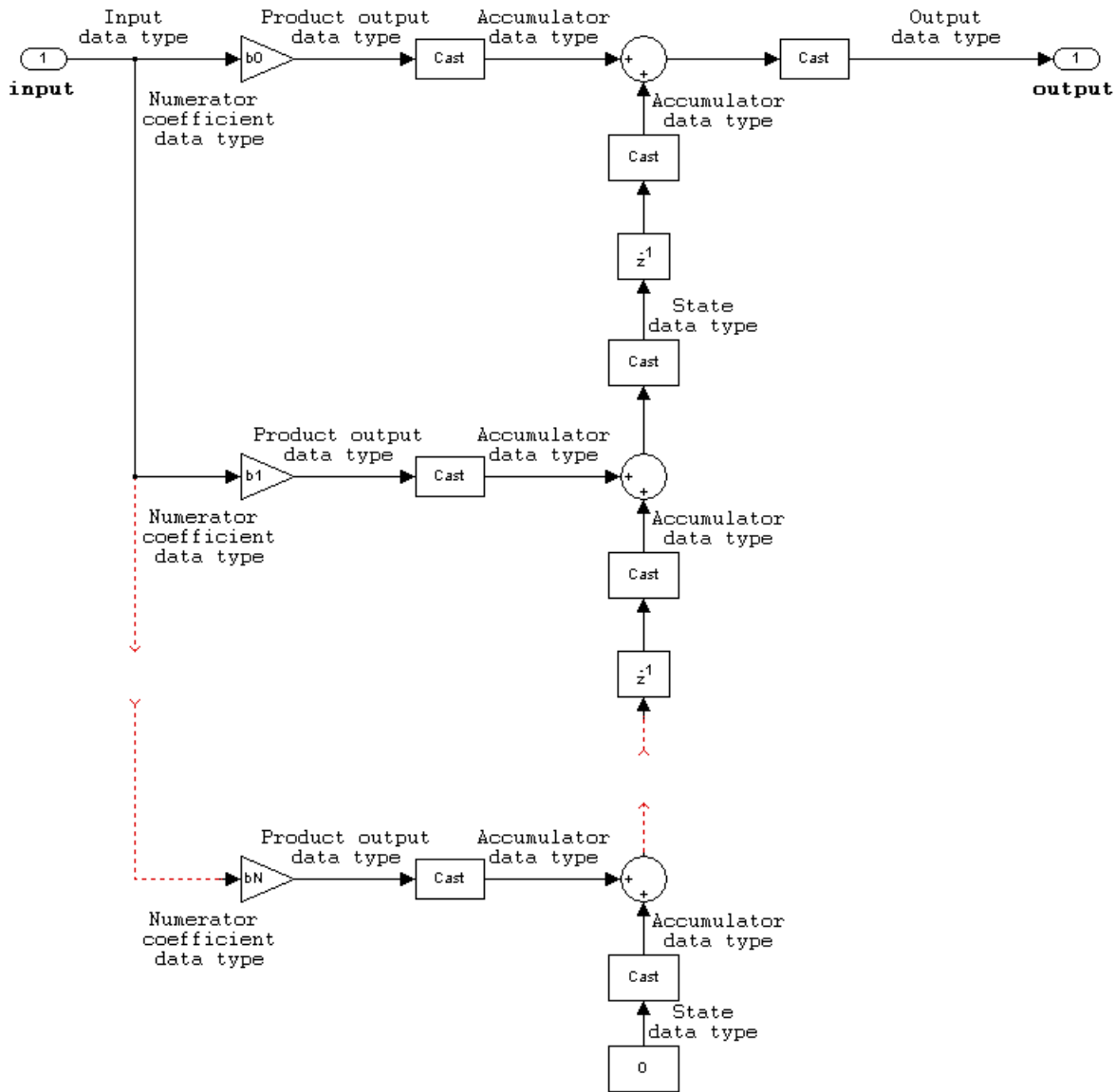
- Inputs can be real or complex.
- Numerator coefficients can be real or complex.
- You cannot specify the state data type on the block mask for this structure, because the input and output states have the same data types as the input.
- It is assumed that the filter coefficients are antisymmetric. Only the first half of the coefficients are used for filtering.
- The **Tap Sum** parameter determines the data type the filter uses when it sums the inputs prior to multiplication by the coefficients.



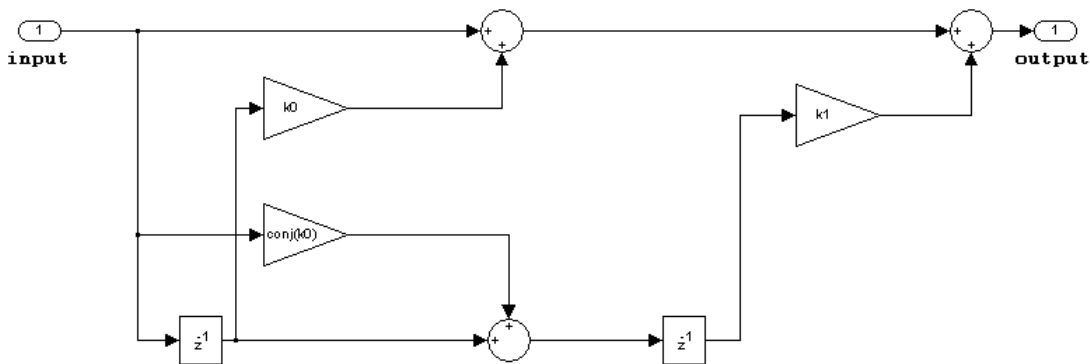
**FIR (all zeros) direct form transposed**

The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs can be real or complex.
- Coefficients can be real or complex.
- States are complex when either the inputs or the coefficients are complex.



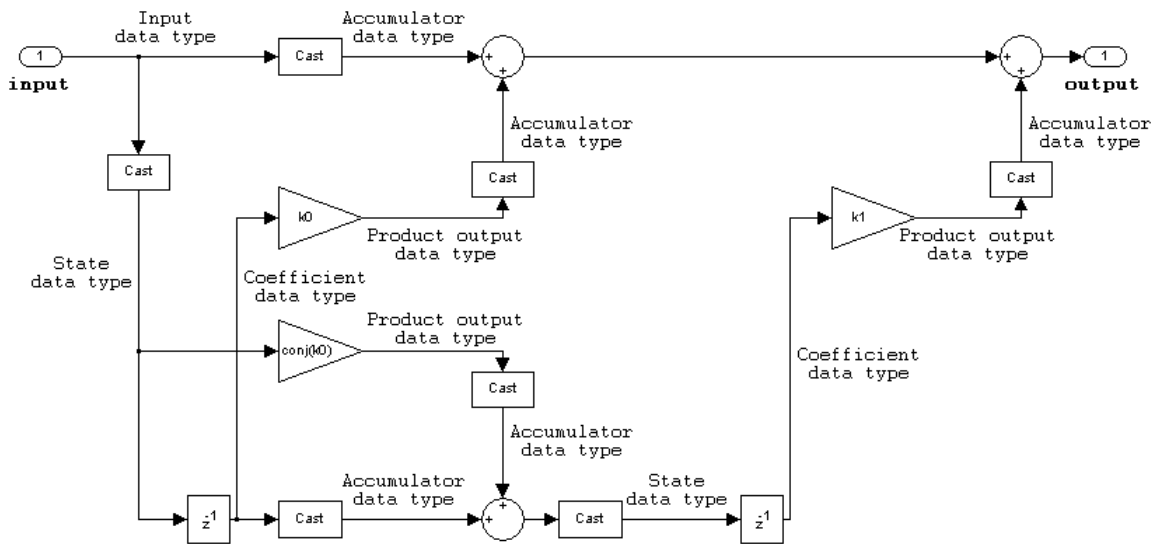
**FIR (all zeros) lattice MA**





The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs and coefficients can be real or complex.
- Coefficients can be real or complex.



## HDL Code Generation

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

---

**Note** Use of Digital Filter block in future releases is not recommended. Existing instances will continue to operate, but certain functionality will be disabled. See “Functionality being removed or replaced for blocks and System objects”. We strongly recommend using Discrete FIR Filter or Biquad Filter in new designs.

---

## HDL Architecture

When you specify `SerialPartition` and `ReuseAccum` for a Digital Filter block, observe the following constraints.

- If you specify **Dialog parameters** as the `Coefficient` source:
  - Set **Transfer function type** to FIR (all zeros).
  - Select **Filter structure** as one of: Direct form, Direct form symmetric, or Direct form asymmetric.

## Distributed Arithmetic Support

Distributed Arithmetic properties **DALUTPartition** and **DARadix** are supported for the default architecture, with FIR, Asymmetric FIR, and Symmetric FIR filter structures.

### AddPipelineRegisters Support

When you use **AddPipelineRegisters**, registers are placed based on the filter structure. The pipeline register placement determines the latency.

| Architecture                                   | Pipeline Register Placement  | Latency (clock cycles)   |
|--|--|--|
| FIR, Asymmetric FIR, and Symmetric FIR filters | A pipeline register is added between levels of a tree-based adder. | $\text{ceil}(\log_2(\text{FL}))$ .<br>FL is the filter length. |
| FIR Transposed                                 | A pipeline register is added after the products.                   | 1  |
| IIR SOS  | Pipeline registers are added between the filter sections.          | NS - 1.<br>NS is the number of sections.                       |

### HDL Filter Properties

|                                 |   |
|---------------------------------|---|
| <b>AddPipelineRegisters</b>     | Insert a pipeline register between stages of computation in a filter. See also AddPipelineRegisters (HDL Coder).  |
| <b>CoeffMultipliers</b>         | Specify the use of canonical signed digit (CSD) optimization to decrease filter area by replacing coefficient multipliers with shift-and-add logic. When you choose a fully parallel filter implementation, you can set <b>CoeffMultipliers</b> to <code>csd</code> or <code>factored-csd</code> . The default is <code>multipliers</code> , which retains multipliers in the HDL. See also CoeffMultipliers (HDL Coder). |
| <b>DALUTPartition</b>           | Specify distributed arithmetic partial-product LUT partitions as a vector of the sizes of each partition. The sum of all vector elements must be equal to the filter length. The maximum size for a partition is 12 taps. Set <b>DALUTPartition</b> to a scalar value equal to the filter length to generate DA code without LUT partitions. See also DALUTPartition (HDL Coder).   |
| <b>MultiplierInputPipeline</b>  | Specify the number of pipeline stages to add at filter multiplier inputs. See also MultiplierInputPipeline (HDL Coder).   |
| <b>MultiplierOutputPipeline</b> | Specify the number of pipeline stages to add at filter multiplier outputs. See also MultiplierOutputPipeline (HDL Coder).   |
| <b>ReuseAccum</b>               | Enable or disable accumulator reuse in a serial filter implementation. Set <b>ReuseAccum</b> to <code>on</code> to use a cascade-serial implementation. See also ReuseAccum (HDL Coder).  |

### HDL Block Properties

|                                  |  |
|----------------------------------|--|
| <b>ConstrainedOutputPipeline</b> | Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see "ConstrainedOutputPipeline" (HDL Coder). |
| <b>InputPipeline</b>             | Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "InputPipeline" (HDL Coder).                 |

|                       |  |
|-----------------------|--|
| <b>OutputPipeline</b> | Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder). |
|-----------------------|--|

### Complex Coefficients and Data Support

Except for decimator and interpolator filter structures, HDL Coder supports use of complex coefficients and complex input signals for all filter structures of the Digital Filter block.

### Restrictions

- You must set **Initial conditions** to zero. HDL code generation is not supported for nonzero initial states.
- HDL Coder does not support the Digital Filter block **Input port(s)** option for HDL code generation.

### Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point (signed only)
- 8-, 16-, and 32-bit signed integers

### See Also

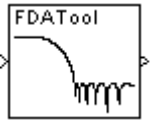
|                           |                           |
|---------------------------|---------------------------|
| Allpole Filter            | DSP System Toolbox        |
| Digital Filter Design     | DSP System Toolbox        |
| Biquad Filter             | DSP System Toolbox        |
| Discrete Filter           | Simulink                  |
| Discrete FIR Filter       | Simulink                  |
| Filter Realization Wizard | DSP System Toolbox        |
| filterDesigner            | DSP System Toolbox        |
| <b>FVTool</b>             | Signal Processing Toolbox |

## Version History

Introduced in R2014b

## Digital Filter Design

Design and implement digital FIR and IIR filters



### Libraries:

DSP System Toolbox / Filtering / Filter Designs

### Description

Use this block to design, analyze, and efficiently implement floating-point filters.

The Digital Filter Design block implements a digital Finite Impulse Response (FIR) or Infinite Impulse Response (IIR) filter that you design by using the **Filter Designer** (`filterDesigner`) app. This block provides the same filter implementation as the Discrete FIR Filter or Biquad Filter blocks.

You must specify whether the block performs frame-based or sample-based processing on the input by setting the **Input processing** parameter. The block applies the specified filter to each channel of a discrete-time input signal and outputs the result. The outputs of the block numerically match the outputs of the Discrete FIR Filter or Biquad Filter block and the MATLAB `filter` function. For more information, see “Getting Started with Filter Designer”.

These blocks also implement digital filters, but serve slightly different purposes:

- Discrete FIR Filter and Biquad Filter— Use to efficiently implement floating-point or fixed-point filters that you have already designed. These blocks provide the same exact filter implementation as the Digital Filter Design block.
- Filter Realization Wizard — Use to implement floating-point or fixed-point filters built from Sum, Gain, and Unit Delay blocks. You can either design the filter within this block, or import the coefficients of a filter that you designed elsewhere.

### Ports

#### Input

**Port\_1** — Input signal

vector | matrix

Signal to filter, specified as a vector or matrix of real values. When given matrix input, the block treats each column as an independent channel.

The sample rate,  $F_s$ , that you specify in the filter designer app must be identical to the sample rate of the input to the Digital Filter Design block. When the sampling frequencies do not match, the Digital Filter Design block returns a warning message and inherits the sample rate of the input block.

Data Types: `single` | `double`

## Output

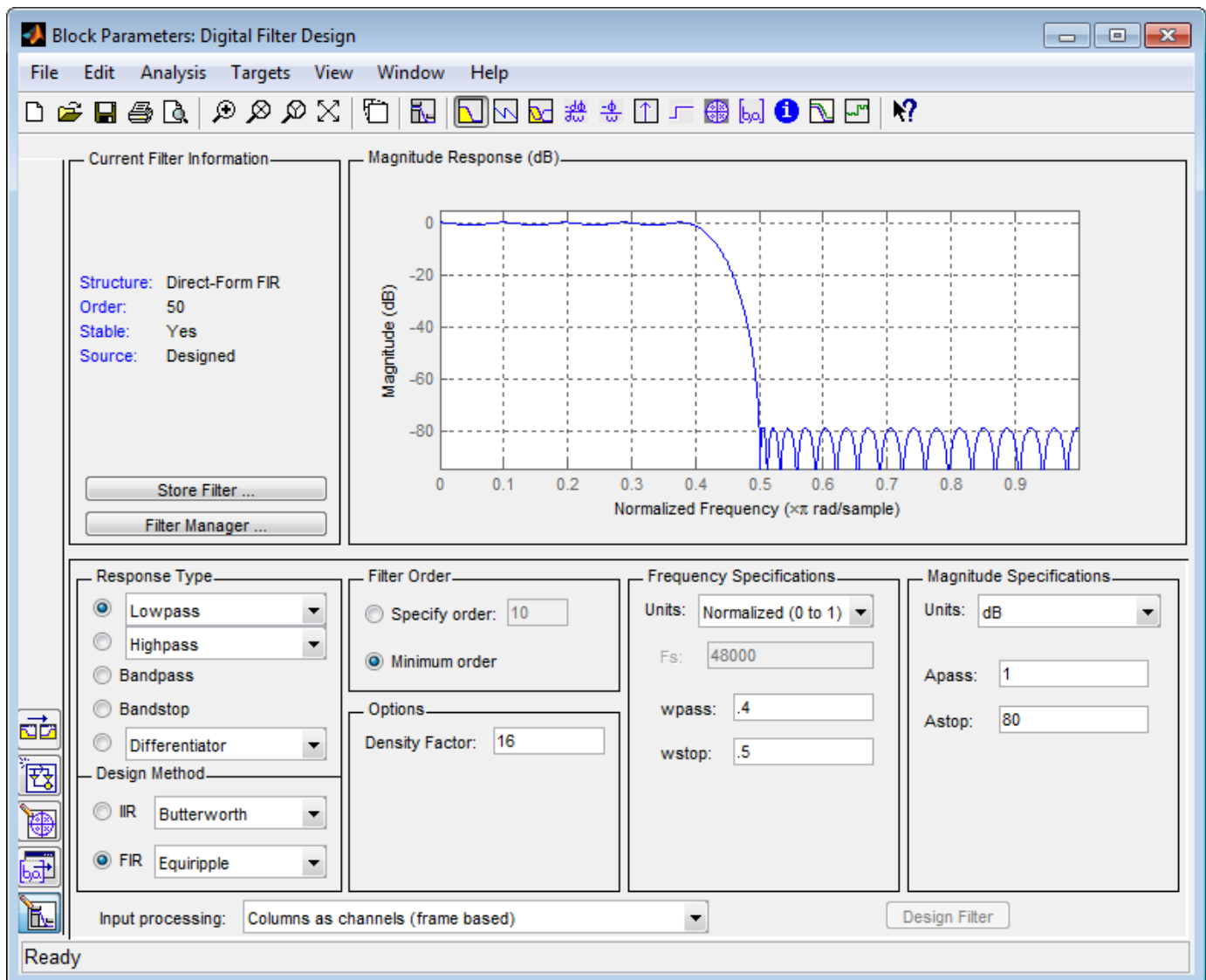
**Port 1** — Output signal  
vector | matrix

Filtered signal, returned as a vector or a matrix of real values.

Data Types: single | double

## Parameters

### Dialog Box



For more information about the parameters in this dialog box, see “Using Filter Designer”.

## Block Characteristics

|                                 |                 |
|---------------------------------|-----------------|
| <b>Data Types</b>               | double   single |
| <b>Direct Feedthrough</b>       | no              |
| <b>Multidimensional Signals</b> | no              |
| <b>Variable-Size Signals</b>    | no              |
| <b>Zero-Crossing Detection</b>  | no              |

## More About

### Designing the Filter

To open filter designer, double-click the Digital Filter Design block. Use the filter designer to design or import a digital FIR or IIR filter. To learn how to design filters with this block and filter designer, see:

- “Using Digital Filter Design Block”
- **Filter Designer** app reference page. For details, see `filterDesigner`.

### Tuning the Filter During Simulation

You can tune the filter specifications in filter designer during simulations if your changes do not modify the filter length or filter order. As you apply any filter changes in filter designer, the filter is updated.

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Generated code relies on the `memcpy` or `memset` function (`string.h`) under certain conditions.

The Digital Filter Design block supports SIMD code generation using Intel AVX2 technology under these conditions:

- **Input processing** is set to `Columns as channels (frame based)`.
- **Filter Structure** (in **Import Filter from Workspace** pane) is set to `Direct-Form FIR`. You can generate SIMD code even when the filter is a `Direct-Form FIR Transposed` filter. To create a `Direct-Form FIR Transposed` filter, select **Edit > Convert Structure**, and click **Direct-Form FIR Transposed**.
- Input signal has a data type of `single` or `double`.

The SIMD technology significantly improves the performance of the generated code.

## See Also

### Functions

filterDesigner | filter

### Tools

FVTool

### Blocks

Discrete FIR Filter | Biquad Filter | Analog Filter Design | Window Function

### Topics

“Filter Design”

“Filter Analysis”

“Using Digital Filter Design Block”

“Getting Started with Filter Designer”

# Discrete Impulse

Generate discrete impulse



**Libraries:**  
DSP System Toolbox / Sources

## Description

The Discrete Impulse block generates an impulse (the value 1) at output sample  $D+1$ , where you specify  $D$  using the **Delay** parameter ( $D \geq 0$ ). All output samples preceding and following sample  $D+1$  are zero.

When  $D$  is a length- $N$  vector, the block generates an  $M$ -by- $N$  matrix output representing  $N$  distinct channels, where you specify frame size  $M$  using the **Samples per frame** parameter. The impulse for the  $i$ th channel appears at sample  $D(i)+1$ .

The **Sample time** parameter value,  $T_s$ , specifies the output signal sample period. The resulting frame period is  $MT_s$ .

## Ports

### Output

**Port\_1** — Discrete impulse signal  
scalar | vector | matrix

Output signal containing a discrete impulse at output sample  $D(i)+1$ , where  $D$  is a scalar or vector specified by the **Delay** parameter. For more information, see “Description” on page 2-496.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | Boolean | fixed point

## Parameters

### Main

**Delay (samples)** — Number of zero-valued output samples

0 (default) | scalar | vector

The number of zero-valued output samples,  $D$ , preceding the impulse, specified as a scalar or vector of integer values, greater than or equal to zero. A length- $N$  vector specifies an  $N$ -channel output.

**Sample time** — Output sample period

1 (default) | positive scalar

The sample period,  $T_s$ , of the output signal specified as a positive finite scalar. The output frame period is  $MT_s$ .



**Samples per frame** — Samples per frame

1 (default) | positive integer


The number of samples,  $M$ , in each output frame, specified as a positive integer scalar.

**Data Types****Output data type** — Output data type

double (default) | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | boolean | fixdt(1,16) | fixdt(1,16,0) | Inherit: Inherit via back propagation | <data type expression>

Specify the output data type for this block. You can select one of the following:

- A rule that inherits a data type, for example, **Inherit: Inherit via back propagation**. When you select this option, the output data type and scaling matches that of the next downstream block.
- A built-in data type, such as **double**
- An expression that evaluates to a valid data type, for example, `fixdt(1,16)`

For help setting data type parameters, display the **Data Type Assistant** by clicking the **Show data type assistant** button .

See “Control Data Types of Signals” (Simulink) for more information.

**Data Type Assistant****Mode** — Data type mode

Inherit (default) | Built in | Fixed point | Expression

Select how you would like to specify the data type properties of the **Output data type**. You can choose:

- **Inherit** — Lets you specify a rule for inheriting a data type, for example, **Inherit: Same as input**
- **Built in**— Lets you specify a built-in data type, for example, **double**
- **Fixed point** — Lets you specify the fixed-point attributes of the data type.
- **Expression** — Lets you specify an expression that evaluates to a valid data type, for example, `fixdt(1,16)`

For more information, see “Specify Data Types Using Data Type Assistant” (Simulink).

**Signedness** — Signedness of fixed-point data

Signed (default) | Unsigned

Specify whether the fixed-point data is signed or unsigned. Signed data can represent positive and negative values, but Unsigned data represents positive values only. For more information, see “Specify Data Types Using Data Type Assistant” (Simulink).

**Scaling** — Method for scaling fixed-point data

Best precision (default) | Binary point

Specify the method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors. For more information, see “Specify Data Types Using Data Type Assistant” (Simulink).

**Dependencies**

To enable this parameter, set **Mode** to Fixed point

**Word length** — Bit size of the word that holds the quantized integer

16 (default) | integer from 2 to 128

Specify the bit size of the word that holds the quantized integer as a positive integer from 2 to 128. For more information, see “Specify Data Types Using Data Type Assistant” (Simulink).

**Dependencies**

To enable this parameter, set **Mode** to Fixed point

**Fraction length** — Fraction length for fixed-point data type

0 (default) | integer scalar

Specify the fraction length of the fixed-point data type as a positive or negative integer scalar.

**Dependencies**

To enable this parameter, set **Mode** to Fixed point and **Scaling** to Binary point.

**Data type override** — Data type override mode

Inherit (default) | Off

Select the data type override mode for this signal. You can select:

- **Inherit** — Inherits the data type override setting specified for the model.
- **Off** — Ignores the data type override setting specified for the model and uses the fixed-point data type you specify.

**Dependencies**

To enable this parameter, set **Mode** to Built in or Fixed point. For more information, see “Specify Data Types Using Data Type Assistant” (Simulink).

**Block Characteristics**

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | Boolean   double   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no  |
| <b>Multidimensional Signals</b> | no  |

|                                |    |
|--------------------------------|----|
| <b>Variable-Size Signals</b>   | no |
| <b>Zero-Crossing Detection</b> | no |

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

## See Also

### Blocks

Constant | Data Type Conversion | Multiphase Clock | N-Sample Enable | Signal From Workspace

### Functions

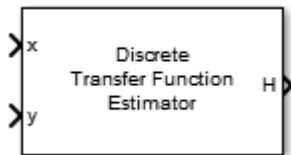
impz

### Topics

“Sample- and Frame-Based Concepts”

## Discrete Transfer Function Estimator

Compute estimate of frequency-domain transfer function of system



### Libraries:

DSP System Toolbox / Estimation / Power Spectrum Estimation

### Description

The Discrete Transfer Function Estimator block estimates the frequency-domain transfer function of a system using the Welch's method of averaged modified periodograms.

The block buffers the input data into overlapping segments. You can set the length of the data segment and the amount of data overlap through the parameters set in the block dialog box. The sample rate of the block is equal to  $1/T$ .  $T$  is the sample time of the inputs to the block.

The block first applies a window function to the two inputs,  $x$  and  $y$ , and then scales them by the window power. It takes the FFT of each signal, calling them  $X$  and  $Y$ . The block calculates  $P_{xx}$  which is the square magnitude of the FFT,  $X$ . The block then calculates  $P_{yx}$  which is  $X$  multiplied by the conjugate of  $Y$ . The output transfer function estimate,  $H$ , is calculated by dividing  $P_{yx}$  by  $P_{xx}$ .

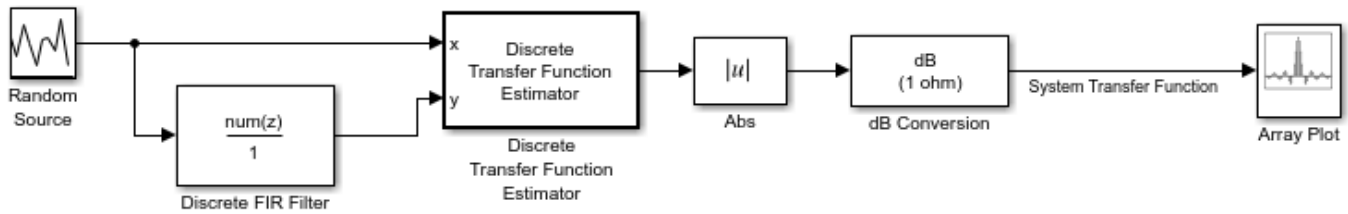
### Examples

#### Estimate Frequency-Domain Transfer Function of System

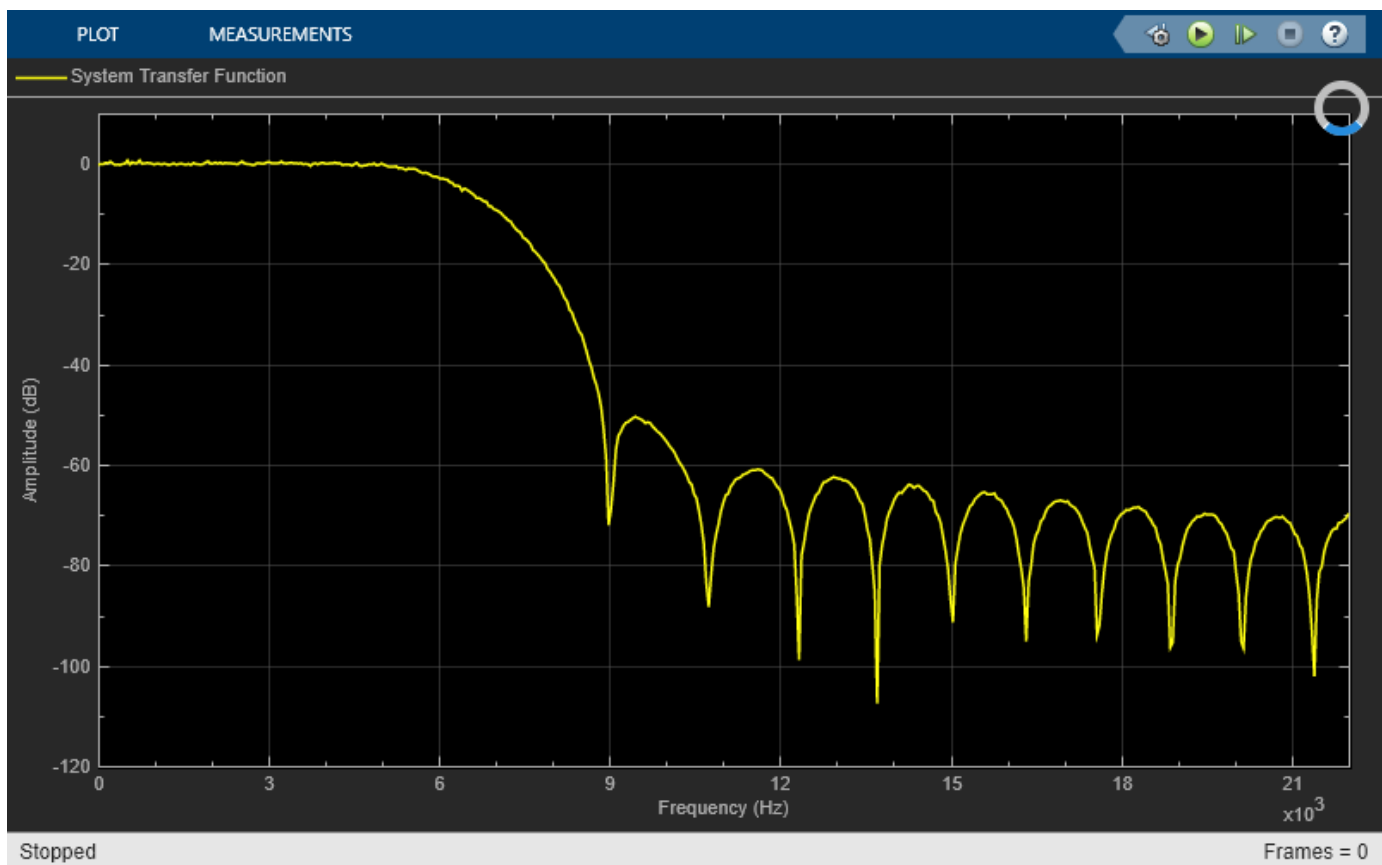
This example shows how to use the Discrete Transfer Function Estimator block to estimate the frequency-domain transfer function of a system.

Open the model. The Random Source block represents the system input signal. The sample rate of the system input is 44.1 KHz. The Random Source input passes through a low-pass filter with a normalized cutoff frequency of 0.3. The filtered signal represents the system output signal. Because the Discrete Transfer Function Estimator block outputs complex values, take the magnitude of the output to see a plot of the transfer function estimate.

Run the model. The transfer function plot displays the system transfer function, a low-pass filter that matches the frequency response of the Discrete FIR Filter block.



Copyright 2018-2023 The MathWorks, Inc.



## Ports

### Input

**x** — System input signal  
vector | matrix

Specify the system input signal as a vector or a matrix.

The inputs **x** and **y** must have the same size and data type.

Data Types: double | single

**y** — System output signal  
vector | matrix

Specify the system output signal as a vector or a matrix.

The inputs  $x$  and  $y$  must have the same size and data type.

Data Types: double | single

## Output

**Output** — Transfer function estimate  
vector | matrix

Transfer function estimate, returned as a vector or a matrix. For more details on how the block computes the transfer function estimate, see “Algorithms” on page 2-505.

Data Types: double | single

## Parameters

**Window length source** — Window length source

Same as input frame length (default) | Specify on dialog

Specify the source of the window length value. You can set this parameter to:

- Same as input frame length — Window length is set to the frame size of the input.
- Specify on dialog — Window length is the value you specify in **Window length**.

**Window length** — Window length

1024 (default) | positive integer  $\geq 2$

Specify the length of the window, in samples, used to compute the spectrum estimate as a positive integer greater than or equal to 2.

### Dependencies

To enable this parameter, set **Window length source** to Specify on dialog.

**Window overlap (%)** — Percentage of overlap between windows

0 (default) | nonnegative scalar

Specify the percentage of overlap between successive data windows as a nonnegative scalar in the range [0,100).

**Averaging method** — Averaging method

Running (default) | Exponential

Specify the averaging method as Running or Exponential. In the running averaging method, the block computes an equally weighted average of specified number of spectrum estimates defined by **Number of spectral averages** parameter. In the exponential method, the block computes the average over samples weighted by an exponentially decaying forgetting factor.

**Specify forgetting factor from input port** — Specify forgetting factor from input port

off (default) | on

Select this check box to specify the forgetting factor from an input port. When you do not select this check box, the forgetting factor is specified through the **Forgetting factor** parameter.

**Dependencies**

To enable this parameter, set **Averaging method** to Exponential.

**Forgetting factor** — Forgetting factor0.9 (default) | positive scalar  $\leq 1$ 

Specify the exponential weighting forgetting factor as a positive scalar  $\leq 1$ .

**Dependencies**

To enable this parameter, set **Averaging method** to Exponential and clear the **Specify forgetting factor from input port** parameter.

**Number of spectral averages** — Number of spectral averages

1 (default) | positive integer

Specify the number of spectral averages as a positive integer. The Transfer Function Estimator block computes the current estimate by averaging the last  $N$  estimates.  $N$  is the number of spectral averages.

**Dependencies**

To enable this parameter, set **Averaging method** to Running.

**FFT length source** — Source of FFT length value

Auto (default) | Property

Specify the source of the FFT length value as Auto or Property.

When the source of the FFT length is set to Auto, the block sets the FFT length to the input frame size. When the source of the FFT length is set to Property, you specify the FFT length in the **FFT length** parameter.

**FFT length** — FFT length

128 (default) | positive integer

Specify the length of the FFT that the Discrete Transfer Function Estimator block uses to compute spectral estimates as a positive integer.

**Window function** — Window function

Hann (default) | Chebyshev | Flat Top | Hamming | Kaiser | Rectangular

Specify a window function for the Discrete Transfer Function Estimator block. Possible values are:

- Hann
- Chebyshev
- Flat Top
- Hamming
- Kaiser
- Rectangular

**Sidelobe attenuation of window (dB)** — Sidelobe attenuation of window

60 (default) | positive scalar

Specify the sidelobe attenuation of the window as a positive scalar in dB.

#### Dependencies

To enable this parameter, set **Window function** to Kaiser or Chebyshev.

**Frequency range** — Frequency range of the transfer function estimate

Centered (default) | One-sided | Two-sided

Specify the frequency range of the transfer function estimate.

- **Centered** -- The block computes the centered two-sided transfer function of the real or complex input signals,  $x$  and  $y$ .
- **onesided** -- The block computes the one-sided transfer function of real input signals,  $x$  and  $y$ .
- **twosided** -- The block computes the two-sided transfer function of the real or complex input signals,  $x$  and  $y$ .

**Output magnitude squared coherence estimate** — Magnitude squared coherence estimate

off (default) | on

Select this check box to compute and output the magnitude squared coherence estimate using Welch's averaged, modified periodogram method. The magnitude squared coherence estimate indicates how well two inputs correspond to each other at each frequency.

**Simulate using** — Type of simulation to run

Code generation (default) | Interpreted execution

Type of simulation to run. You can set this parameter to:

- **Code generation** -- Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than **Interpreted execution**.
- **Interpreted execution** -- Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than **Code generation**.



## Block Characteristics

|                                 |                 |
|---------------------------------|-----------------|
| <b>Data Types</b>               | double   single |
| <b>Multidimensional Signals</b> | No              |
| <b>Variable-Size Signals</b>    | No              |

## Algorithms

### Welch's Method of Averaged Modified Periodograms

Give two signal inputs,  $x$  and  $y$ :

- 1 Multiply the inputs by the window and scale the result by the window power.
- 2 Take FFT of the signals,  $X$  and  $Y$ .
- 3 Compute the current power spectral density estimates,  $P_{xx}$ ,  $P_{yy}$ , and the current cross power spectral density estimate,  $P_{yx}$ , by taking the moving average of last  $N$  number of  $Z_1$ ,  $Z_2$ , and  $Z_3$  vectors, respectively:
  - $Z_1 = X \cdot \text{conj}(X)$
  - $Z_2 = Y \cdot \text{conj}(Y)$
  - $Z_3 = Y \cdot \text{conj}(X)$

For details on the moving average methods, see "Averaging Method" on page 4-1141.

The transfer function estimate is calculated by dividing  $P_{yx}$  by  $P_{xx}$ .

The magnitude squared coherence,  $C_{xy}$ , is defined by the following equation:

$$C_{xy} = \frac{(\text{abs}(P_{xy}) \cdot ^2)}{(P_{xx} \cdot P_{yy})}$$

## Version History

Introduced in R2014a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

## See Also

[dsp.TransferFunctionEstimator](#) | [Cross-Spectrum Estimator](#) | [Periodogram](#) | [Spectrum Analyzer](#)

### Topics

"Continuous-Time Transfer Function Estimation"

## Downsample

Resample input at lower rate by deleting samples



### Libraries:

DSP System Toolbox / Signal Operations

DSP System Toolbox HDL Support / Signal Operations

## Description

The Downsample block decreases the sampling rate of the input by deleting samples. When the block performs frame-based processing, it resamples the data in each column of the  $P_i$ -by- $Q$  input matrix independently. When the block performs sample-based processing, it treats each element of the input as a separate channel and resamples each channel of the input array across time. The resample rate is  $M$  times lower than the input sample rate, where  $M$  is the value of the downsample factor. The Downsample block resamples the input by discarding  $M-1$  consecutive samples following each sample that is output.

This block supports triggered subsystems when you set the **Rate options** parameter to Enforce single-rate processing.

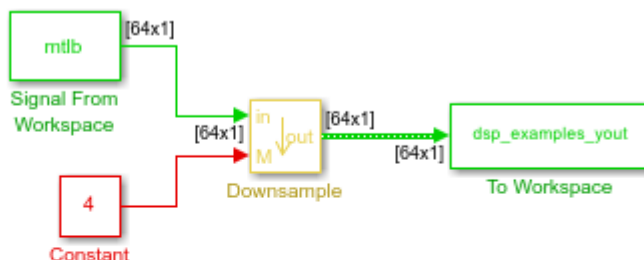
## Examples

### Downsample Signal in Single-Rate Frame-Based Processing Mode

In this example, the Downsample block decreases the sample rate by decreasing the frame size.

Open the model. The Downsample block resamples a single-channel input with a frame size of 64. In the dialog box of the Downsample block, the **Downsample factor source** parameter is set to Input port. This sets the **Input processing** and the **Rate options** parameters automatically to Columns as Channels (frame based) and Enforce single-rate processing, respectively. Specify a downsample factor of 4 from the input port.

Run the model. The block operates in the single-rate processing mode and downsamples the input by a factor of 4. The output of the block has a frame size of 64. As the block operates in the single-rate processing mode, the input and output signals of the Downsample block have the same frame rate. You can confirm this from the **Timing Legend**. To open the legend, click the **Debug** tab on the model toolstrip and click **Information Overlays > Timing Legend**.



Copyright 2008-2023 The MathWorks, Inc.

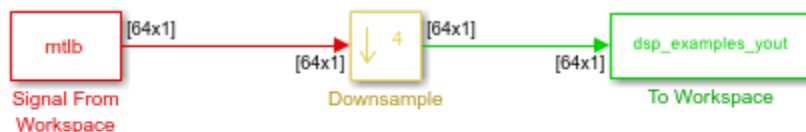
 Note: This model created workspace variables called "Fs", "mtlb", and "dsp\_examples\_yout".

### Downsample Signal In Multirate Frame-Based Processing Mode

In this example, the Downsample block decreases the sample rate by decreasing the frame rate.

Open the model. The input to the Downsample block is a single-channel signal with a frame period of one second. In the block dialog box, set the **Downsample factor, M** to 4 and **Rate options** to Allow multirate processing.

Run the model. The block operates in the multirate frame-based processing mode. With a downsample factor of 4, the output of the block has a frame period of 4 seconds. You can confirm this from the **Timing Legend**. To open the legend, click the **Debug** tab on the model toolstrip and click **Information Overlays > Timing Legend**. As the block operates in the multirate processing mode, the input and output signals of the Downsample block have the same frame size.



Copyright 2008-2023 The MathWorks, Inc.

 Note: This model creates workspace variables called "Fs", "mtlb", and "dsp\_examples\_yout".

### Downsample a Signal with One Frame Latency

Downsample a signal by a factor of 2 using the Downsample block.

## Open the System and Run the Model



Copyright 2008-2023 The MathWorks, Inc.



Note: This model creates a workspace variable called "dsp\_examples\_yout".

The Signal From Workspace block generates a two-channel signal with a frame size of 4. The timing legend shows the sample time of the signal before and after the downsampling operation. To see the legend, click the **Debug** tab > **Information Overlays** > **Timing Legend**.

Run the model. The sample time of the signal before the downsampling operation is half the sample time of the signal after the operation.

### One-Frame Latency

The **Input processing** parameter in the Downsample block is set to **Columns as channels** (frame based) and the input frame size (number of rows in the input)  $M_i$  is greater than 1. Hence, the latency of the signal is one frame. The **Initial conditions** parameter is set to  $[11 \ -11; 12 \ -12; 13 \ -13; 14 \ -14]$ . In all cases of one-frame latency, the  $M_i$  rows of the initial condition matrix appear in sequence as the first four output rows. Input sample  $D+1$  (that is, row  $D+1$  of the input matrix) appears in the output as sample  $M_i+1$ , followed by input sample  $D+1+K$ , input sample  $D+1+2K$ , and so on.

- $M_i$  - Number of input rows. In this example,  $M_i$  equals 4.
- $D$  - Sample offset parameter. In this example,  $D$  equals 1.
- $K$  - Downsample factor. In this example,  $K$  equals 2.

The Initial conditions value can be an  $M_i$ -by- $N$  matrix containing one value for each channel, or a scalar to be repeated across all elements of the  $M_i$ -by- $N$  matrix.

Here is the downsampled output signal written to the dsp\_examples\_yout variable in the base workspace.

```
dsp_examples_yout =
```

```

11   -11
12   -12
13   -13
14   -14
 2     -2
 4     -4
 6     -6
 8     -8
10    -10
12    -12
14    -14
16    -16
  
```

---

|     |      |
|-----|------|
| 18  | -18  |
| 20  | -20  |
| 22  | -22  |
| 24  | -24  |
| 26  | -26  |
| 28  | -28  |
| 30  | -30  |
| 32  | -32  |
| 34  | -34  |
| 36  | -36  |
| 38  | -38  |
| 40  | -40  |
| 42  | -42  |
| 44  | -44  |
| 46  | -46  |
| 48  | -48  |
| 50  | -50  |
| 52  | -52  |
| 54  | -54  |
| 56  | -56  |
| 58  | -58  |
| 60  | -60  |
| 62  | -62  |
| 64  | -64  |
| 66  | -66  |
| 68  | -68  |
| 70  | -70  |
| 72  | -72  |
| 74  | -74  |
| 76  | -76  |
| 78  | -78  |
| 80  | -80  |
| 82  | -82  |
| 84  | -84  |
| 86  | -86  |
| 88  | -88  |
| 90  | -90  |
| 92  | -92  |
| 94  | -94  |
| 96  | -96  |
| 98  | -98  |
| 100 | -100 |
| 0   | 0    |
| 0   | 0    |
| 0   | 0    |
| 0   | 0    |
| 0   | 0    |
| 0   | 0    |
| 0   | 0    |
| 0   | 0    |
| 0   | 0    |
| 0   | 0    |

0 0

## Ports

### Input

**in** — Data input

column vector | matrix |  $N$ -D array

Input data for which you want to decrease the sample rate, specified as a column vector or a matrix.

When you set the **Input processing** parameter to `Elements as channels (sample based)`, the input can be an  $N$ -D array.

The block supports variable-size input signals when you set **Input processing** to `Columns as channels (frame based)` and **Rate options** to `Enforce single-rate processing`, so the frame length (number of rows) and the number of channels (columns) can change during simulation. When the block accepts variable-size input signals, they can be of arbitrary frame length, that is, the input frame length does not have to be a multiple of the downsampling factor. When you specify fixed-size signals, the frame length can be arbitrary under certain conditions. For more details, see “Frame-Based Processing” on page 2-514 and “Sample-Based Processing” on page 2-517.

This port is unnamed until you set **Downsample factor source** to `Input port`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point`

**M** — Downsample factor

positive integer

Specify the downsample factor  $M$  as a positive integer.

### Dependency

To enable this port, set the **Downsample factor source** parameter to `Input port`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

### Output

**Port\_1** — Downsampled output

column vector | matrix

Downsampled output with a sampling rate that is  $1/M$  times the input sampling rate, returned as a column vector or a matrix.

When you set **Rate options** to:

- `Enforce single-rate processing` -- The block maintains the input sample rate and downsamples the signal by decreasing the output frame size by a factor of  $M$ .

The output has an upper bound size of  $\text{ceil}(P/M)$ -by- $Q$  for an input of size  $P$ -by- $Q$ .

- `Allow multirate processing` -- The block downsamples the signal such that the output sample rate is  $M$  times slower than the input sample rate.

The output frame size is the same as the input frame size.

For more details, see “Frame-Based Processing” on page 2-514 and “Sample-Based Processing” on page 2-517.

The output is a variable-size signal when any of these conditions are met:

- The input is a variable-size signal.
- The input is a fixed-size signal and the **Allow arbitrary frame length for fixed-size input signals** parameter is selected.
- Downsample factor is input through the port **M**

When the output is a variable-size signal, the block maintains the phase across consecutive frames of data. For more details, see “Phase Continuity” on page 2-516.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point`

## Parameters

**Downsample factor source** — Source of downsample factor

Dialog parameter (default) | Input port

Specify the downsample factor through one of these options:

- `Dialog parameter` -- The block enables the **Downsample factor, M** parameter in the block dialog box.
- `Input port` -- The block enables the input port **M** through which you specify the downsample factor.

**Downsample factor, M** — Downsampling factor

2 (default) | positive integer

Specify the integer factor  $M$  by which to decrease the input sample rate.

### Dependency

To enable this parameter, set **Downsample factor source** to `Dialog parameter`.

**Sample offset (0 to M–1)** — Sample offset

0 (default) | integer

The **Sample offset (0 to M–1)** parameter delays the output samples by an integer number of sample periods,  $D$ , where  $0 \leq D \leq (M-1)$ , so that you can select any of the  $M$  possible output phases. For example, when you downsample the sequence 1, 2, 3, ... by a factor of 4, you can select from four phases.

| Input Sequence | Sample Offset, $D$ | Output Sequence ( $M = 4$ )      |
|----------------|--------------------|----------------------------------|
| 1, 2, 3, ...   | 0                  | 1, 5, 9, 13, 17, 21, 25, 29, ... |

| Input Sequence | Sample Offset, $D$ | Output Sequence ( $M = 4$ )      |
|----------------|--------------------|----------------------------------|
| 1, 2, 3, ...   | 1                  | 0, 2, 6, 10, 14, 18, 22, 26, ... |
| 1, 2, 3, ...   | 2                  | 0, 3, 7, 11, 15, 19, 23, 27, ... |
| 1, 2, 3, ...   | 3                  | 0, 4, 8, 12, 16, 20, 24, 28, ... |

The initial zero in the last three output sequences in the table is a result of the **Initial conditions** parameter being set to its default value of 0. See “Latency” on page 2-517 for more information on the **Initial conditions** parameter.

When you input the downsample factor through the input port **M**, the **Sample offset (0 to M-1)** parameter is not enabled in the block dialog box. In this case, the sample offset  $D$  is automatically set to 0.

### Dependency

To enable this parameter, set **Downsample factor source** to Dialog parameter.

### Input processing — Method to process input

Columns as channels (frame based) (default) | Elements as channels (sample based)

Specify the method for input processing:

- **Columns as channels (frame based)** -- When you set the **Input processing** parameter to **Columns as channels (frame based)**, the block treats each of the  $Q$  input columns as an individual channel containing  $P_i$  sequential time samples. The block downsamples each channel independently by discarding  $M-1$  rows of the input matrix following each row that it outputs.  
  
For more information, see “What Is Frame-Based Processing?”.
- **Elements as channels (sample based)** -- When you set the **Input processing** parameter to **Elements as channels (sample based)**, the input can be a  $Q$ -D array. The Downsample block treats each element of the input as a separate channel, and resamples each channel of the input over time. The block downsamples the input array by discarding  $M-1$  samples following each sample that it passes through to the output. The input and output sizes of the Downsample block are identical.

For more information, see “What Is Sample-Based Processing?”.

### Dependency

To enable this parameter, set **Downsample factor source** to Dialog parameter.

When you set **Downsample factor source** to Input port, the **Input processing** parameter is automatically set to **Columns as channels (frame based)**.

### Rate options — Enforce single-rate or allow multirate processing

Enforce single-rate processing (default) | Allow multirate processing

Specify the method by which the block adjusts the rate at the output port to accommodate the reduced number of samples. The block performs single-rate or multirate processing depending on whether you set the **Input processing** parameter to **Elements as channels (sample based)** (sample-based processing mode) or **Columns as channels (frame based)** (frame-based processing mode).



- Elements as channels (sample based)

- Enforce single-rate processing

The block forces the output sample rate to match the input sample rate ( $T_{so} = T_{si}$ ) by repeating every  $M^{\text{th}}$  input sample  $M$  times at the output. In this mode, the block behaves like the Sample and Hold block with a repeating trigger event of period  $MT_{si}$ .

- Allow multirate processing

The sample period of the output is  $M$  times longer than the input sample period ( $T_{so} = MT_{si}$ ).

For more details, see “Sample-Based Processing” on page 2-517

- Columns as channels (frame based)

- Enforce single rate processing

The block generates the output at the slower (downsampled) rate using a proportionally smaller frame size than the input. The output has an upper bound size of  $\text{ceil}(P/M)$ , where  $P$  is the input frame length and  $M$  is the downsample factor. The input and output frame rates are equal. For more details, see “Frame-Based Processing” on page 2-514. For an example, see “Downsample Signal in Single-Rate Frame-Based Processing Mode” on page 2-506.

- Allow multirate processing

The block generates the output at the slower (downsampled) rate by using a proportionally longer frame period at the output port than at the input port. For downsampling by a factor of  $M$ , the output frame period is  $M$  times longer than the input frame period ( $T_{fo} = MT_{fi}$ ), but the input and output frame sizes are equal. For more details, see “Frame-Based Processing” on page 2-514. For an example, see “Downsample Signal In Multirate Frame-Based Processing Mode” on page 2-507.

### Dependency

To enable this parameter, set **Downsample factor source** to Dialog parameter.

When you set **Downsample factor source** to Input port, the **Rate options** parameter is automatically set to Enforce single-rate processing.

**Allow arbitrary frame length for fixed-size input signals** — Allow arbitrary frame length for fixed-size input signals

on (default) | off

Specify whether fixed-size input signals (size does not change during simulation) can have an arbitrary frame length, where the frame length does not have to be a multiple of the downsampling factor. The block uses this parameter setting only for fixed-size input signals and ignores this parameter if the input has a variable size.

When the input signal is a variable-size signal, the signal can have an arbitrary frame length, that is, the frame length does not have to be a multiple of the decimation factor.

For fixed-size input signals, if you:

- Select the **Allow arbitrary frame length for fixed-size input signals** parameter, the frame length of the signal does not have to be a multiple of the downsampling factor. If the input is not a

multiple of the downsampling factor, then the output is generally a variable-size signal. Therefore, to support an arbitrary input size, the block must also support variable-size operations, which you can enable by selecting the **Allow arbitrary frame length for fixed-size input signals** parameter.

- Clear the **Allow arbitrary frame length for fixed-size input signals** parameter, the input frame length must be a multiple of the downsampling factor.

### Dependency

To enable this parameter, set:

- **Downsample factor source** to Dialog parameter.
- **Input processing** to Columns as channels (frame based).
- **Rate options** to Enforce single-rate processing.

### Initial conditions — Initial value

0 (default) | real scalar | array

The initial block value for cases of nonzero latency. You can specify a scalar or an array of the same size as the input.

### Dependencies

This parameter does not appear when **Input processing** is set to Columns as channels (frame based) and **Rate options** is set to Enforce single-rate processing.

## Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | Boolean   double   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no  |
| <b>Multidimensional Signals</b> | no  |
| <b>Variable-Size Signals</b>    | yes   |
| <b>Zero-Crossing Detection</b>  | no  |

## More About

### Frame-Based Processing

When you set the **Input processing** parameter to Columns as channels (frame based) and use an input signal of size  $P$ -by- $Q$ , the block treats each of the  $Q$  input columns as an individual channel containing  $P$  sequential time samples. The block downsamples each channel independently by discarding  $M-1$  rows of the input matrix following each row that it outputs.

In this mode, the block can perform either single-rate or multirate processing. You can use the **Rate options** parameter to specify how the block adjusts the rate at the output to accommodate the reduced number of samples.

- When you set the **Rate options** parameter to Enforce single-rate processing, the block generates the output at the slower (downsampled) rate using a proportionally smaller frame size than the input.

The output has an upper bound size of  $\text{ceil}(P/M)$ , where  $P$  is the input frame length and  $M$  is the downsample factor.

In this mode, if you input a fixed-size signal (signal dimensions do not change during simulation) and select the **Allow arbitrary frame length for fixed-size input signals** parameter, the input frame length can be arbitrary and does not have to be a multiple of the downsampling factor. If you clear the **Allow arbitrary frame length for fixed-size input signals** parameter, the input frame length must be a multiple of the downsampling factor.

In this mode, if you input a variable-size signal (signal dimensions change during simulation), the **Allow arbitrary frame length for fixed-size input signals** appears on the block dialog box but does not have any impact on the input frame size requirements. You can input a variable-size signal of any frame size even if you do not select the **Allow arbitrary frame length for fixed-size input signals** parameter.

This table summarizes the support for arbitrary input frame length when you set **Input processing** to Columns as channels (frame based) and **Rate options** to Enforce single-rate processing.

| Input Signal               | Block Support for this Signal | Support for Arbitrary Input Frame Length   | Input Size   | Output Size                                     |
|----------------------------|-------------------------------|--|--------------|---|
| Fixed-size input signal    | Yes                           | When you select <b>Allow arbitrary frame length for fixed-size input signals</b> | $P$ -by- $Q$ | Upper bound size of $\text{ceil}(P/M)$ -by- $Q$ |
| Variable-size input signal | Yes                           | Always   | $P$ -by- $Q$ | Upper bound size of $\text{ceil}(P/M)$ -by- $Q$ |

- When you set the **Rate options** parameter to Allow multirate processing, the input and output of the Downsample block are of the same size, but the sample rate of the output is  $M$  times slower than that of the input. In this mode, the block treats a  $P$ -by- $Q$  matrix input as  $Q$  independent channels.

In this mode, the block accepts only fixed-size signals and these signals can have an arbitrary frame length.

This table summarizes the support for arbitrary input frame length when you set **Input processing** to Columns as channels (frame based) and **Rate options** to Allow multirate processing.

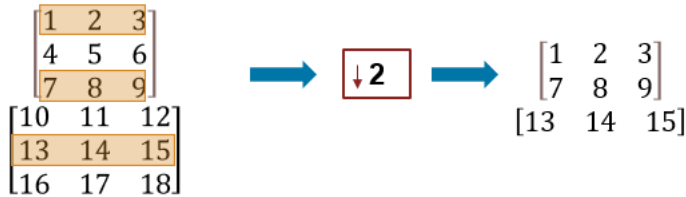
| Input Signal            | Block Support for this Signal | Support for Arbitrary Input Frame Length | Input Size   | Output Size  |
|-------------------------|-------------------------------|--|--------------|--------------|
| Fixed-size input signal | Yes                           | Always                                   | $P$ -by- $Q$ | $P$ -by- $Q$ |

| Input Signal               | Block Support for this Signal | Support for Arbitrary Input Frame Length | Input Size     | Output Size    |
|----------------------------|-------------------------------|--|----------------|----------------|
| Variable-size input signal | No                            | Not applicable                           | Not applicable | Not applicable |

**Phase Continuity**

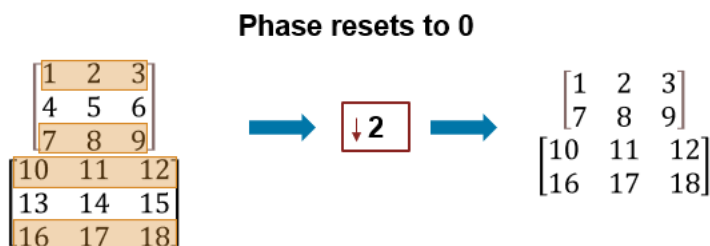
When you set the **Input processing** parameter to Columns as channels (frame based) and **Rate options** parameter to Enforce single-rate processing, under certain conditions, the Downsample block maintains the phase across consecutive frames of data.

**Phase is maintained**



| Input Conditions                         | Downsample Factor Source  | Support for Arbitrary Input Frame Length   | Output Signal        | Phase Continuity |
|--|---|--|----------------------|------------------|
| Fixed-size input signal                  | Specify in block dialog box   | When you select <b>Allow arbitrary frame length for fixed-size input signals</b> | Variable-size signal | Yes              |
| Variable-size input signal               | Specify in block dialog box   | Always   | Variable-size signal | Yes              |
| Fixed-size or variable-size input signal | Specify through input port<br>Downsample factor does not change during simulation | --   | Variable-size signal | Yes              |

Under certain conditions, the phase resets to 0 and is not maintained across consecutive frames of data.



| Input Conditions                         | Downsample Factor Source  | Support for Arbitrary Input Frame Length  | Output Signal        | Phase Continuity  |
|--|---|---|----------------------|-------------------|
| Fixed-size input signal                  | Specify in block dialog box   | When you clear <b>Allow arbitrary frame length for fixed-size input signals</b> | Fixed-size signal    | Phase is always 0 |
| Fixed-size or variable-size input signal | Specify through input port<br><br>At the event when downsample factor changes during simulation | --  | Variable-size signal | Phase resets to 0 |

### Sample-Based Processing

When you set the **Input processing** parameter to `Elements as channels (sample based)`, the block treats a  $P$ -by- $Q$  matrix input as  $P*Q$  independent channels, and resamples each channel over time. The output sample period ( $T_{so}$ ) is  $M$  times longer than the input sample period ( $T_{so} = M*T_{si}$ ), and the input and output sizes are identical.

In this mode, the block accepts only fixed-size signals and these signals can have an arbitrary frame length.

This table summarizes the support for arbitrary input frame length when you set **Input processing** to `Elements as channels (sample based)`.

| Input Signal               | Block Support for this Signal | Support for Arbitrary Input Frame Length | Input Size     | Output Size    |
|----------------------------|-------------------------------|--|----------------|----------------|
| Fixed-size input signal    | Yes                           | Always                                   | $P$ -by- $Q$   | $P$ -by- $Q$   |
| Variable-size input signal | No                            | Not applicable                           | Not applicable | Not applicable |

### Latency

Latency is the delay, measured in samples or frames, between the input and the output of the block.

The Downsample block has *zero-tasking latency* in the following cases:

- The downsample factor  $M$  is 1.
- The **Input processing** parameter is set to `Columns as channels (frame based)`, and the **Rate options** parameter is set to `Enforce single-rate processing`.
- The **Input processing** parameter is set to `Columns as channels (frame based)`, the **Rate options** parameter is set to `Allow multirate processing`, the **Sample offset (0 to M-1)** parameter  $D$  is set to 0, and the input frame size is equal to 1.
- The **Input processing** parameter is set to `Elements as channels (sample based)`, and the **Sample offset (0 to M-1)** parameter  $D$  is 0.

Zero-tasking latency means that the block propagates input sample  $D+1$  (received at  $t = 0$ ) as the first output sample, followed by input sample  $D+1+M$ , input sample  $D+1+2M$ , and so on. When there is zero-tasking latency, the block ignores the value of the **Initial conditions** parameter.

In all other cases, the latency is nonzero:

- When the **Input processing** parameter is set to `Elements as channels (sample based)`, the latency is one sample.
- When the **Input processing** parameter is set to `Columns as channels (frame based)` and the input frame size is greater than one, the latency is one frame.

In all cases of *one-sample latency*, the initial condition for each channel appears as the first output sample. Input sample  $D+1$  appears as the second output sample for each channel, followed by input sample  $D+1+M$ , input sample  $D+1+2M$ , and so on. The **Initial conditions** parameter can be an array of the same size as the input or a scalar to be applied to all signal channels.

In all cases of *one-frame latency*, the  $P_i$  rows of the initial condition matrix appear in sequence as the first  $P_i$  output rows. Input sample  $D+1$  (row  $D+1$  of the input matrix) appears in the output as sample  $P_i+1$ , followed by input sample  $D+1+M$ , input sample  $D+1+2M$ , and so on. The **Initial conditions** value can be an  $P_i$ -by- $Q$  matrix containing one value for each channel or a scalar to be repeated across all elements of the  $P_i$ -by- $Q$  matrix.

---

**Note** For more information on latency and the Simulink tasking modes, see “Excess Algorithmic Delay (Tasking Latency)” and “Time-Based Scheduling and Code Generation” (Simulink Coder).

---

## Version History

Introduced before R2006a

### R2022b: Support for arbitrary input frame length and variable input frame sizes

Starting in R2022b, this block supports input signals with arbitrary frame lengths when the:

- Input signal is a fixed-size signal (signal dimensions do not change during simulation) and the block allows for multirate processing
- Input signal is a fixed-size signal, block enforces single-rate processing, and you select the **Allow arbitrary frame length for fixed-size input signals** parameter (if enabled)
- Input signal is a variable-size signal (signal dimensions change during simulation)

When this block supports an arbitrary frame length input signal, the input frame length does not have to be a multiple of the downsampling factor.

For more details, see “Frame-Based Processing” on page 2-514 and “Sample-Based Processing” on page 2-517.

### **R2023a: Tune downsample factor through input port**

You can now specify the downsample factor through an input port when the Downsample block operates in the single-rate frame-based processing mode. When you specify the downsample factor through the input port, you can change the factor during simulation.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

Generated code relies on the `memcpy` or `memset` function (`string.h`) under certain conditions.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

### **Best Practices**

It is good practice to follow the Downsample block with a unit delay. Doing so prevents the code generator from inserting an extra bypass register in the HDL code.

See also “Multirate Model Requirements for HDL Code Generation” (HDL Coder).

### **HDL Architecture**

This block has one default HDL architecture.

### **HDL Block Properties**

|                                  |  |
|----------------------------------|--|
| <b>ConstrainedOutputPipeline</b> | Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder). |
| <b>InputPipeline</b>             | Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).                 |
| <b>OutputPipeline</b>            | Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).               |

**Complex Data Support**

This block supports code generation for complex signals.

**Restrictions**

- **Input processing** set to Columns as channels (frame based) is not supported.
- For **Input processing** set to Elements as channels (sample based), select Allow multirate processing. With this setting, if **Sample offset** is set to 0, **Initial conditions** has no effect on generated code.

**Fixed-Point Conversion**

Design and simulate fixed-point systems using Fixed-Point Designer™.

**See Also****Blocks**

FIR Decimation | FIR Rate Conversion | Repeat | Sample and Hold | Upsample

**Topics**

“Convert Sample and Frame Rates in Simulink Using Rate Conversion Blocks”

“Noise Cancellation in Simulink Using Normalized LMS Adaptive Filter”



## DSP Constant (Obsolete)

Generate discrete- or continuous-time constant signal



### Library

Sources

dspobslib

### Description

---

**Note** The DSP Constant block is still supported but is likely to be obsoleted in a future release. We strongly recommend replacing this block with the Constant block.

---

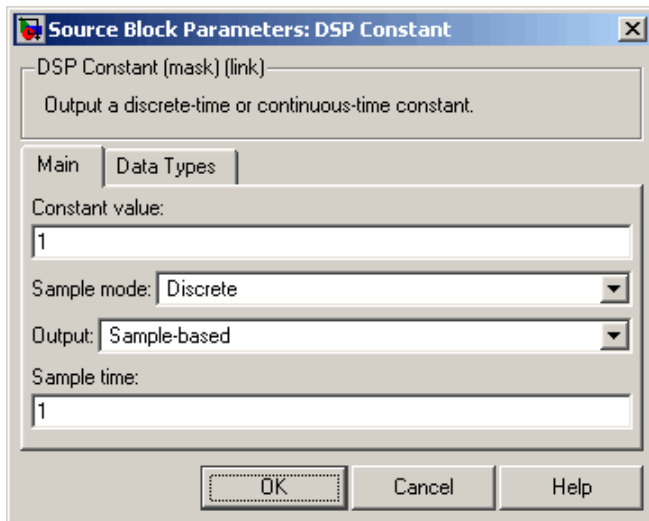
The DSP Constant block generates a signal whose value remains constant throughout the simulation. The **Constant value** parameter specifies the constant to output, and can be any valid MATLAB expression that evaluates to a scalar, vector, or matrix.

When **Sample mode** is set to **Continuous**, the output is a continuous-time signal. When **Sample mode** is set to **Discrete**, the **Sample time** parameter is visible, and the signal has the discrete output period specified by the **Sample time** parameter.

You can set the output signal to **Frame-based**, **Sample-based**, or **Sample-based** (interpret vectors as 1-D) with the **Output** parameter.

### Dialog Box

The **Main** pane of the DSP Constant block dialog box appears as follows.



### Constant value

Specify the constant to generate. This parameter is Tunable (Simulink); values entered here can be tuned, but their dimensions must remain fixed.

When you specify any data type information in this field, it is overridden by the value of the **Output data type** parameter in the **Data Types** pane, unless you select **Inherit** from 'Constant value'.

### Sample mode

Specify the sample mode of the output, **Discrete** for a discrete-time signal or **Continuous** for a continuous-time signal.

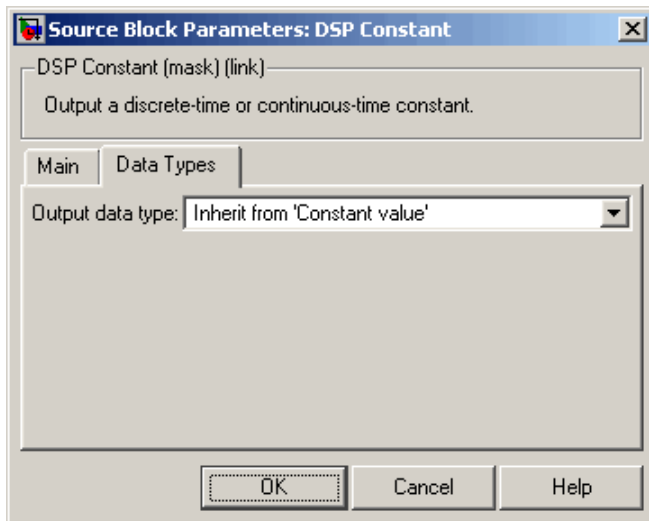
### Output

Specify whether the output is **Sample-based** (interpret vectors as 1-D), **Sample-based**, or **Frame-based**. When you select **Sample-based** and the output is a vector, its dimension is constrained to match the **Constant value** dimension (row or column). When you select **Sample-based** (interpret vectors as 1-D), however, the output has no specified dimensionality.

### Sample time

Specify the discrete sample period for sample-based outputs. When you select **Frame-based** for the **Output** parameter, this parameter is named **Frame period**, and is the discrete frame period for the frame-based output. This parameter is only visible when you select **Discrete** for the **Sample mode** parameter.

The **Data Types** pane of the DSP Constant block dialog box appears as follows.



### Output data type

Specify the output data type in one of the following ways:

- Choose one of the built-in data types from the list.
- Choose **Fixed-point** to specify the output data type and scaling in the **Signed**, **Word length**, **Set fraction length in output to**, and **Fraction length** parameters.
- Choose **User-defined** to specify the output data type and scaling in the **User-defined data type**, **Set fraction length in output to**, and **Fraction length** parameters.
- Choose **Inherit from 'Constant value'** to set the output data type and scaling to match the values of the **Constant value** parameter in the **Main** pane.
- Choose **Inherit via back propagation** to set the output data type and scaling to match the following block.

The value of this parameter overrides any data type information specified in the **Constant value** parameter in the **Main** pane, except when you select **Inherit from 'Constant value'**.

### Signed

Select to output a signed fixed-point signal. Otherwise, the signal is unsigned. This parameter is only visible when you select **Fixed-point** for the **Output data type** parameter.

### Word length

Specify the word length, in bits, of the fixed-point output data type. This parameter is only visible when you select **Fixed-point** for the **Output data type** parameter.

### User-defined data type

Specify any built-in or fixed-point data type. You can specify fixed-point data types using the following `fixdt` function. This parameter is only visible when you select **User-defined** for the **Output data type** parameter.

### Set fraction length in output to

Specify the scaling of the fixed-point output by either of the following two methods:

- Choose **Best precision** to have the output scaling automatically set such that the output signal has the best possible precision.

- Choose **User-defined** to specify the output scaling in the **Fraction length** parameter.

This parameter is only visible when you select **Fixed-point** for the **Output data type** parameter, or when you select **User-defined** and the specified output data type is a fixed-point data type.

### **Fraction length**

For fixed-point output data types, specify the number of fractional bits, or bits to the right of the binary point. This parameter is only visible when you select **Fixed-point** or **User-defined** for the **Output data type** parameter and **User-defined** for the **Set fraction length in output to** parameter.

## **Supported Data Types**

- Double-precision floating point
- Single-precision floating point
- Fixed point
- Boolean
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

## **See Also**

Constant

Signal From Workspace

Simulink

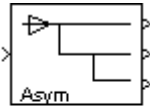
DSP System Toolbox

## **Version History**

**Introduced in R2008b**

## DWT

Compute discrete wavelet transform (DWT) of input or decompose signals into subbands with smaller bandwidths and slower sample rates



**Libraries:**  
DSP System Toolbox / Transforms



### Description

You can configure this block to compute the Discrete Wavelet Transform (DWT) of the input signal or decompose the signal into subbands with smaller bandwidths and slower sample rates. The block uses a series of highpass and lowpass FIR filters to repeatedly divide the input frequency range, as illustrated in “Multilevel Filter Banks” (the Asymmetric one).

You can specify the filter bank highpass and lowpass filters by providing vectors of filter coefficients. You can do so directly on the block mask. If you have a Wavelet Toolbox™ license, you can specify wavelet-based filters by selecting a wavelet from the **Filter** parameter. You must set the filter bank structure to asymmetric or symmetric, and specify the number of levels in the filter bank.

For the same input, the DWT configuration of this block does not produce the same results as the Wavelet Toolbox `dwt` function. Because DSP System Toolbox is designed for real-time implementation and Wavelet Toolbox is designed for analysis, the products handle boundary conditions and filter states differently. To make the output of the `dwt` function match the DWT output of this block, complete the following steps:

- 1 Set the boundary condition of the `dwt` function to zero-padding. To do so, type `dwtmode('zpd')` at the MATLAB command line.
- 2 To match the latency of the block (implemented using FIR filters), add zeros to the input of the `dwt` function. The number of zeros you add must be equal to the half-length of the filter.

---

**Note** The DWT block is same as the Dyadic Analysis Filter Bank block with different default settings. For more information on block ports and parameters, see the Dyadic Analysis Filter Bank block reference page.

---

### Block Characteristics

|                                 |                 |
|---------------------------------|-----------------|
| <b>Data Types</b>               | double   single |
| <b>Multidimensional Signals</b> | No              |
| <b>Variable-Size Signals</b>    | No              |

## Version History

Introduced before R2006a

## References

- [1] Fliege, N. J. *Multirate Digital Signal Processing: Multirate Systems, Filter Banks, Wavelets*. West Sussex, England: John Wiley & Sons, 1994.
- [2] Strang, G. and T. Nguyen. *Wavelets and Filter Banks*. Wellesley, MA: Wellesley-Cambridge Press, 1996.
- [3] Vaidyanathan, P. P. *Multirate Systems and Filter Banks*. Englewood Cliffs, NJ: Prentice Hall, 1993.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Generated code relies on the memcpy or memset function (`string.h`) under certain conditions.

## See Also

### Objects

`dsp.DyadicAnalysisFilterBank`

### Blocks

Dyadic Analysis Filter Bank | Two-Channel Analysis Subband Filter | IDWT

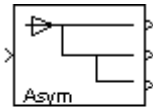
### Topics

“Multilevel Filter Banks”

“Calculate Channel Latencies Required for Wavelet Reconstruction”

# Dyadic Analysis Filter Bank

Decompose signals into subbands with smaller bandwidths and slower sample rates or compute discrete wavelet transform (DWT)



## Libraries:

DSP System Toolbox / Filtering / Multirate Filters



## Description

You can configure this block to decompose a broadband signal into a collection of subbands with smaller bandwidths and slower sample rates. The block uses a series of highpass and lowpass FIR filters to repeatedly divide the input frequency range, as illustrated in “Multilevel Filter Banks” (the Asymmetric one).

You can specify the filter bank highpass and lowpass filters by providing vectors of filter coefficients. You can do so directly on the block mask. If you have a Wavelet Toolbox license, you can specify wavelet-based filters by selecting a wavelet from the **Filter** parameter. You must set the filter bank structure to asymmetric or symmetric, and specify the number of levels in the filter bank.

For the same input, the DWT configuration of this block does not produce the same results as the Wavelet Toolbox `dwt` function. Because DSP System Toolbox is designed for real-time implementation and Wavelet Toolbox is designed for analysis, the products handle boundary conditions and filter states differently. To make the output of the `dwt` function match the DWT output of this block, complete the following steps:

- 1 Set the boundary condition of the `dwt` function to zero-padding. To do so, type `dwtmode('zpd')` at the MATLAB command line.
- 2 To match the latency of the block (implemented using FIR filters), add zeros to the input of the `dwt` function. The number of zeros you add must be equal to the half-length of the filter.

The Dyadic Analysis Filter Bank block is same as the DWT block, but with different default settings.

## Ports

### Input

**Input 1** — Data input  
column vector | matrix

Specify the data input as a column vector or a matrix.

The block always interprets the input signal as frames and operates along the columns. It treats each column of the input as an independent channel, and the number of rows of each channel must be a multiple of  $2^n$ , where  $n$  is the number of filter bank levels that you specify in the **Number of levels** parameter. For example, a frame size of 16 is appropriate for a three-level tree (16 is a multiple of  $2^3$ ).

For more information on why the input data must meet these requirements, see the figure in “Outputs of a 3-Level Asymmetric Dyadic Analysis Filter Bank” on page 2-529.

The block decomposes the input signal into either  $n + 1$  or  $2^n$  subbands. To decompose signals with a frame size that is not a multiple of  $2^n$ , use the Two-Channel Analysis Subband Filter block. You can connect multiple copies of the Two-Channel Analysis Subband Filter block to create a multilevel dyadic analysis filter bank.

Data Types: `single` | `double`

## Output

**Output 1** — Dyadic subband decomposition output  
column vector | matrix

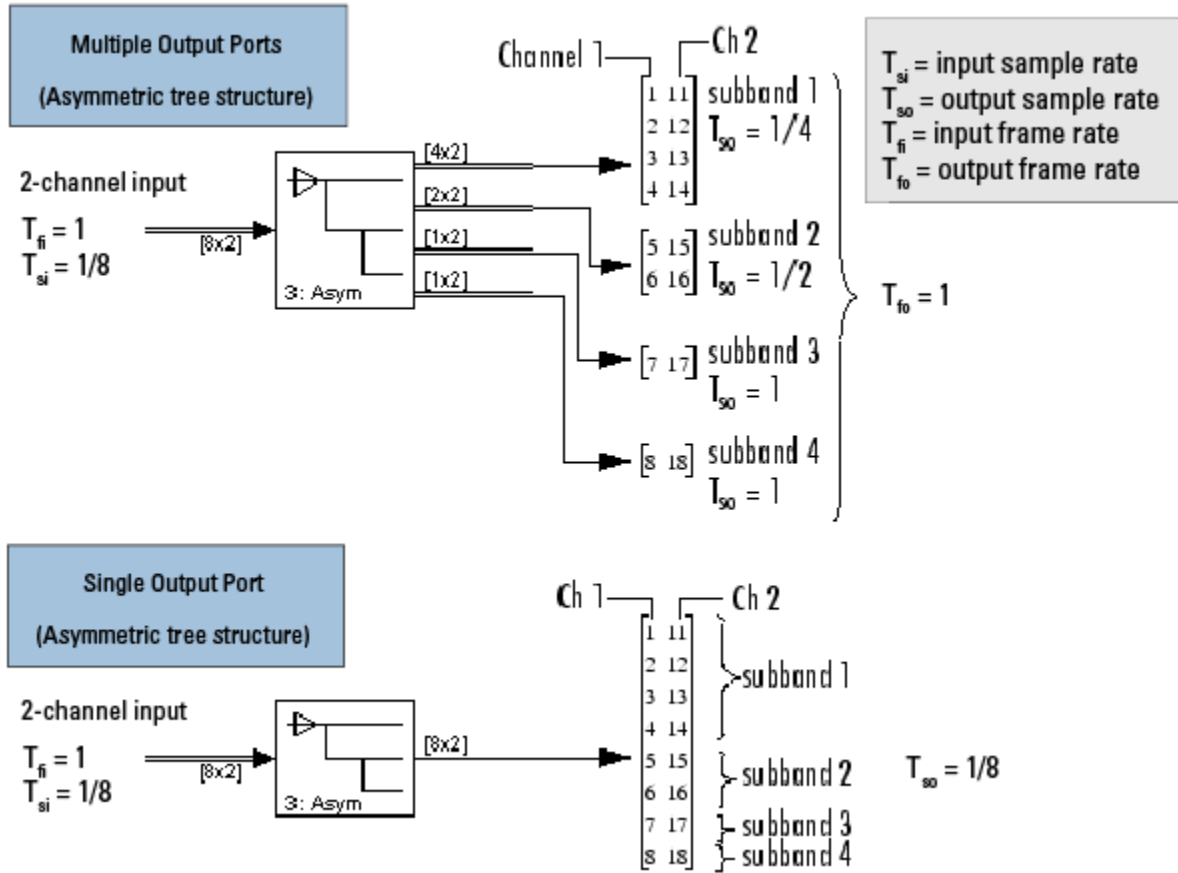
The block outputs the dyadic subband decomposition output as a column vector or a matrix. The decomposition output contains elements with the highest-frequency subband first followed by subbands in decreasing frequency.

The output characteristics vary depending on the block parameter settings, as summarized in the following list and figure:

- **Number of levels** parameter set to  $n$
- **Tree structure** parameter setting:
  - **Asymmetric** — Block produces  $n+1$  output subbands
  - **Symmetric** — Block produces  $2^n$  output subbands
- **Output** parameter setting can be **Multiple ports** or **Single port**. When you set the **Output** parameter to **Single port**, the block outputs one vector or matrix of concatenated subbands. The following figure illustrates the difference between the two settings for a 3-level asymmetric dyadic analysis filter bank. For an explanation of the illustrated output characteristics, see the table Output Characteristics for an n-Level Dyadic Analysis Filter Bank.

For more information about the filter bank levels and structures, see “Dyadic Analysis Filter Banks”.





**Outputs of a 3-Level Asymmetric Dyadic Analysis Filter Bank**

The following table summarizes the different output characteristics of the block when you set it to output from single or multiple ports.

## Output Characteristics for an n-Level Dyadic Analysis Filter Bank

|                                       | Single Output Port   | Multiple Output Ports   |
|---------------------------------------|--|---|
| <b>Output Description</b>             | Block concatenates all the subbands into one vector or matrix, and outputs the concatenated subbands from a single output port. Each output column contains subbands of the corresponding input channel. | Block outputs each subband from a separate output port. The topmost port outputs the subband with the highest frequencies. Each output column contains a subband for the corresponding input channel.   |
| <b>Output Frame Rate</b>              | <i>Not applicable</i>  | Same as input frame rate (However, the output frame sizes can vary, so the output sample rates can vary.)   |
| <b>Output Dimensions (Frame Size)</b> | Same number of rows and columns as the input.  | <p>The output has the same number of columns as the input. The number of output rows is the output frame size. For an input with frame size <math>M_i</math> output <math>y_k</math> has frame size <math>M_{o,k}</math>:</p> <ul style="list-style-type: none"> <li>• <b>Symmetric</b> — All outputs have the frame size <math>M_i / 2^n</math>.</li> <li>• <b>Asymmetric</b> — The frame size of each output (except the last) is half that of the output from the previous level. The outputs from the last two output ports have the same frame size since they originate from the same level in the filter bank.</li> </ul> $M_{o,k} = \begin{cases} M_i/2^k & (1 \leq k \leq n) \\ M_i/2^n & (k = n + 1) \end{cases}$ |
| <b>Output Sample Rate</b>             | Same as input sample rate.   | <p>Though the outputs have the same frame rate as the input, they have different frame sizes than the input. Thus, the output sample rates <math>F_{so,k}</math> are different from the input sample rate <math>F_{si}</math>:</p> <ul style="list-style-type: none"> <li>• <b>Symmetric</b> — All outputs have the sample rate <math>F_{si} / 2^n</math>.</li> <li>• <b>Asymmetric</b> —</li> </ul> $F_{so,k} = \begin{cases} F_{si}/2^k & (1 \leq k \leq n) \\ F_{si}/2^n & (k = n + 1) \end{cases}$  |

Data Types: single | double

## Parameters

The parameters displayed in the block dialog box vary depending on the setting of the **Filter** parameter. Only some of the parameters described below are visible in the dialog box at any one time.

**Filter** — Type of filter used in subband decomposition

User-defined (default) | Haar | Daubechies | Symlets | Coiflets | Biorthogonal | Reverse Biorthogonal | Discrete Meyer

Specify the highpass and lowpass filters in the filter bank by setting the **Filter** parameter to one of these options:

- **User defined** — Allows you to explicitly specify the filters with two vectors of filter coefficients in the **Lowpass FIR filter coefficients** and **Highpass FIR filter coefficients** parameters. The block uses the same lowpass and highpass filters throughout the filter bank. The two filters should be halfband filters, where each filter passes the frequency band that the other filter stops.
- **Wavelet** such as **Biorthogonal** or **Daubechies** — The block uses the specified wavelet to construct the lowpass and highpass filters using the Wavelet Toolbox `wfilters` function. Depending on the wavelet, the block enable either the **Wavelet order** or the **Filter order [synthesis / analysis]** parameter. The **Filter order [synthesis / analysis]** parameter allows you to specify different wavelet orders for the analysis and synthesis filter stages. You must have a Wavelet Toolbox license to use wavelets.

See this table for a list of the supported wavelets.

### Specifying Filters with the Filter Parameter and Related Parameters

| Filter                      | Sample Setting for Related Filter Specification Parameters  | Corresponding Wavelet Function Syntax |
|-----------------------------|---|---------------------------------------|
| <b>User-defined</b>         | Filters based on Daubechies wavelets with wavelet order 3: <ul style="list-style-type: none"> <li>• <b>Lowpass FIR filter coefficients</b> =<br/>[0.0352 -0.0854 -0.1350 0.4599<br/>0.8069 0.3327]</li> <li>• <b>Highpass FIR filter coefficients</b> =<br/>[-0.3327 0.8069 -0.4599<br/>-0.1350 0.0854 0.0352]</li> </ul> | None                                  |
| <b>Haar</b>                 | None  | <code>wfilters('haar')</code>         |
| <b>Daubechies</b>           | <b>Wavelet order</b> = 4  | <code>wfilters('db4')</code>          |
| <b>Symlets</b>              | <b>Wavelet order</b> = 3  | <code>wfilters('sym3')</code>         |
| <b>Coiflets</b>             | <b>Wavelet order</b> = 1  | <code>wfilters('coif1')</code>        |
| <b>Biorthogonal</b>         | <b>Filter order [synthesis / analysis]</b> =<br>[3/1]   | <code>wfilters('bior3.1')</code>      |
| <b>Reverse Biorthogonal</b> | <b>Filter order [synthesis / analysis]</b> =<br>[3/1]   | <code>wfilters('rbio3.1')</code>      |
| <b>Discrete Meyer</b>       | None  | <code>wfilters('dmey')</code>         |

**Filter order [synthesis / analysis]** — Analysis and synthesis filter orders for biorthogonal filters

[1 / 1] (default) | [1 / 3] | [1 / 5] | [2 / 2] | [2 / 4] | [2 / 6] | [2 / 8] | [3 / 1] |  
[3 / 3] | [3 / 5] | [3 / 7] | [3 / 9] | [4 / 4] | [5 / 5] | [6 / 8]

Set the order of the wavelet for the synthesis and analysis filter stages. For example, when you set the **Filter** parameter to `Biorthogonal` and set the **Filter order [synthesis / analysis]** parameter to `[2 / 6]`, the block calls the `wfilters` function with input argument `'bior2.6'`.

#### Dependencies

To enable this parameter, set the **Filter** parameter to `Biorthogonal` or `Reverse Biorthogonal`.

**Wavelet order** — Order for orthogonal wavelets

2 (default) | positive integer

Set the order of the wavelet selected in the **Filter** parameter.

#### Dependencies

To enable this parameter, set the **Filter** parameter to `Daubechies`, `Symlets`, or `Coiflets`.

**Lowpass FIR filter coefficients** — Lowpass FIR filter coefficients

`[0.0352 -0.0854 -0.1350 0.4599 0.8069 0.3327]` (default) | row vector

Specifies coefficients used by all the lowpass filters in the filter bank using a vector of filter coefficients (descending powers of  $z$ ). The lowpass filter should be a half-band filter that passes the frequency band stopped by the filter specified in the **Highpass FIR filter coefficients** parameter. The default values of this parameter specify a filter based on a Daubechies wavelet with wavelet order 3.

#### Dependencies

To enable this parameter, set the **Filter** parameter to `User-defined`.

**Highpass FIR filter coefficients** — Highpass FIR filter coefficients

`[-0.3327 0.8069 -0.4599 -0.1350 0.0854 0.0352]` (default) | row vector

Specifies coefficients used by all the highpass filters in the filter bank using a vector of filter coefficients (descending powers of  $z$ ). This parameter is enabled when you set **Filter** to `User defined`. The highpass filter should be a half-band filter that passes the frequency band stopped by the filter specified in the **Lowpass FIR filter coefficients** parameter. The default values of this parameter specify a filter based on a Daubechies wavelet with wavelet order 3.

#### Dependencies

To enable this parameter, set the **Filter** parameter to `User-defined`.

**Number of levels** — Number of filter bank levels used in analysis (decomposition)

2 (default) | integer greater than or equal to 1

Select the number of filter bank levels. An  $n$ -level asymmetric structure has  $n+1$  outputs, and an  $n$ -level symmetric structure has  $2^n$  outputs, as shown in “Multilevel Filter Banks”. The block icon changes depending on the value of this parameter.

**Tree structure** — Structure of filter bank

Asymmetric (default) | Symmetric

Select the structure of the filter bank: *Asymmetric*, or *Symmetric*. See “Multilevel Filter Banks” for more information.

The default setting of this parameter is *Asymmetric* for the Dyadic Analysis Filter Bank block, and *Symmetric* for the DWT block.

**Output** — Single port or multiple ports

Multiple ports (default) | Single port

Set to *Multiple ports* to output each output subband on a separate port. The topmost port outputs the subband with the highest frequency band. Set to *Single port* to concatenate the subbands into one vector or matrix and output the concatenated subbands on a single port. For more information, see the Output on page 2-0 port description.

The default setting of this parameter is *Multiple ports* for the Dyadic Analysis Filter Bank block and *Single port* for the DWT block.

## Block Characteristics

|                                 |                 |
|---------------------------------|-----------------|
| <b>Data Types</b>               | double   single |
| <b>Direct Feedthrough</b>       | no              |
| <b>Multidimensional Signals</b> | no              |
| <b>Variable-Size Signals</b>    | no              |
| <b>Zero-Crossing Detection</b>  | no              |

## More About

### Wavelets

The primary application for dyadic analysis filter banks and dyadic synthesis filter banks is coding for data compression using wavelets.

At the transmitting end, the output of the dyadic analysis filter bank is fed to a lossy compression scheme, which typically assigns the number of bits for each filter bank output in proportion to the relative energy in that frequency band. This represents the more powerful signal components by a greater number of bits than the less powerful signal components.



At the receiving end, the transmission is decoded and fed to a dyadic synthesis filter bank to reconstruct the original signal. The filter coefficients of the complementary analysis and synthesis stages are designed to cancel aliasing introduced by the filtering and resampling.

See “Calculate Channel Latencies Required for Wavelet Reconstruction” for an example using the Dyadic Analysis and Dyadic Synthesis Filter Bank blocks.

## **Version History**

**Introduced before R2006a**

## **References**

- [1] Fliege, N. J. *Multirate Digital Signal Processing: Multirate Systems, Filter Banks, Wavelets*. West Sussex, England: John Wiley & Sons, 1994.
- [2] Strang, G. and T. Nguyen. *Wavelets and Filter Banks*. Wellesley, MA: Wellesley-Cambridge Press, 1996.
- [3] Vaidyanathan, P. P. *Multirate Systems and Filter Banks*. Englewood Cliffs, NJ: Prentice Hall, 1993.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

Generated code relies on the memcpy or memset function (`string.h`) under certain conditions.

## **See Also**

### **Objects**

`dsp.DyadicAnalysisFilterBank`

### **Blocks**

DWT | Dyadic Synthesis Filter Bank | Two-Channel Analysis Subband Filter

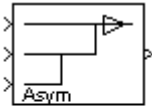
### **Topics**

“Multilevel Filter Banks”

“Calculate Channel Latencies Required for Wavelet Reconstruction”

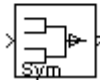
# Dyadic Synthesis Filter Bank

Reconstruct signals from subbands with smaller bandwidths and slower sample rates or compute inverse discrete wavelet transform (IDWT)



## Libraries:

DSP System Toolbox / Filtering / Multirate Filters



## Description

You can configure this block to compute the inverse discrete wavelet transform (IDWT) or reconstruct a signal from subbands with smaller bandwidths and slower sample rates. When the block computes the inverse discrete wavelet transform (IDWT) of the input, the output has the same dimensions as the input. Each column of the output is the IDWT of the corresponding input column. When reconstructing a signal, the block uses a series of highpass and lowpass FIR filters to reconstruct the signal from the input subbands, as illustrated in “Multilevel Filter Banks” (the Asymmetric one). The reconstructed signal has a wider bandwidth and faster sample rate than the input subbands.

You can specify the filter bank highpass and lowpass filters by providing vectors of filter coefficients. You can do so directly on the block dialog box. If you have a Wavelet Toolbox license, you can specify wavelet-based filters by selecting a wavelet from the **Filter** parameter. You must set the filter bank structure to asymmetric or symmetric, and specify the number of levels in the filter bank.

---

**Note** To use a dyadic synthesis filter bank to perfectly reconstruct the output of a dyadic analysis filter bank, the number of levels and tree structures of both filter banks must be the same. In addition, the filters in the synthesis filter bank must be designed to perfectly reconstruct the outputs of the analysis filter bank. Otherwise, the reconstruction is not perfect. For an example that shows perfect reconstruction, see “Wavelet Reconstruction and Noise Reduction”.

---

This block automatically computes wavelet-based perfect reconstruction filters when the wavelet selection in the **Filter** parameter of this block is the same as the **Filter** parameter setting of the corresponding Dyadic Analysis Filter Bank block. The use of wavelets requires a Wavelet Toolbox license. To learn how to design your own perfect reconstruction filters, see “References” on page 2-542.

---

## Ports

### Input

**Input 1** — Data input  
column vector | matrix

Specify the data input as a column vector or a matrix.

The input characteristics vary depending on the block parameter settings as summarized here:

- **Number of levels** parameter set to  $n$
- **Tree structure** parameter setting:
  - **Asymmetric** — Block accepts  $n+1$  input subbands
  - **Symmetric** — Block accepts  $2^n$  input subbands
  - **Input** parameter setting can be **Multiple ports** or **Single port**.

The inputs to this block are usually the outputs of a Dyadic Analysis Filter Bank block. Since the Dyadic Analysis Filter Bank block can output from either a single port or multiple ports, the Dyadic Synthesis Filter Bank block accepts inputs to either a single port or to multiple ports.

When you set the **Input** parameter to **Multiple ports**, you must provide each subband to the block through a different input port as a vector or matrix. You should input the highest frequency band through the topmost port. When you set the **Input** parameter to **Single port**, the block input must be a vector or matrix of concatenated subbands.

---

**Note** Any output of a Dyadic Analysis Filter Bank block whose parameter settings match the corresponding settings of this block is a valid input to this block. For example, the setting of the Dyadic Analysis Filter Bank block parameter **Output** must be the same as the **Input** parameter (**Single port** or **Multiple ports**) in this block.

---

#### **Valid Inputs for Input Set to Single Port**

- Input must be a vector or matrix of concatenated subbands.
- Each input column contains the subbands for an independent signal.
- Upper input rows contain the high-frequency subbands, and the lower rows contain the low-frequency subbands.

#### **Valid Inputs for Input Set to Multiple Ports**

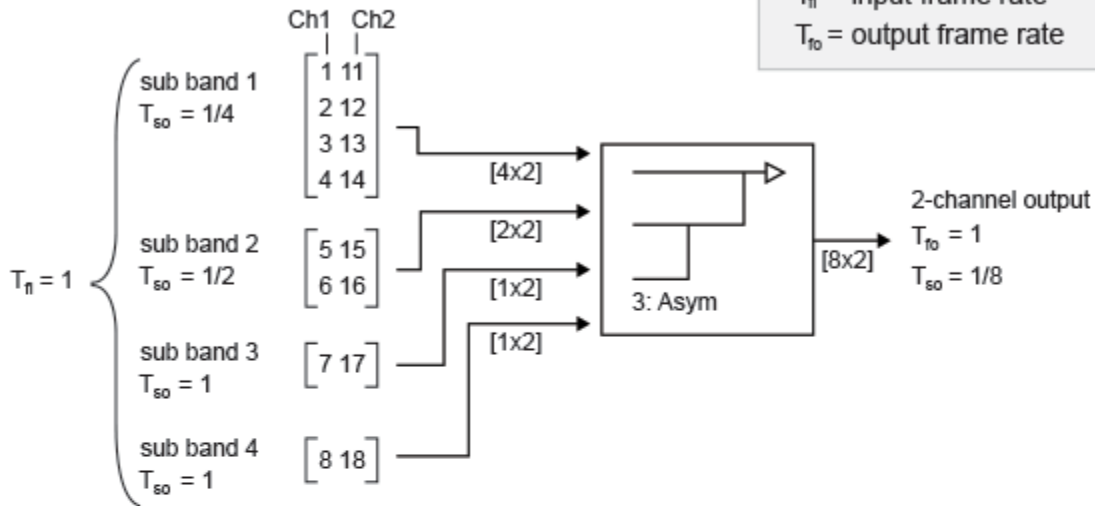
- Each subband must be provided as a vector or a matrix to separate block input ports.
- The columns of each input contains a subband for an independent signal.
- The input to the topmost input port is the subband containing the highest frequencies, and the input to the bottommost port is the subband containing the lowest frequencies.

This figure illustrates the difference between the two settings for a 3-level asymmetric dyadic synthesis filter bank.

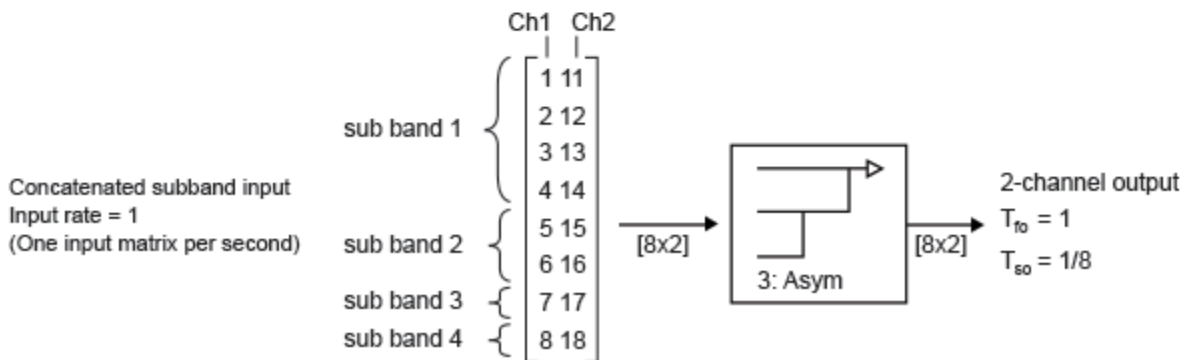


**Multiple Input Ports**  
(Asymmetric tree structure)

$T_{si}$  = input sample rate  
 $T_{so}$  = output sample rate  
 $T_{fi}$  = input frame rate  
 $T_{fo}$  = output frame rate



**Single Input Ports**  
(Asymmetric tree structure)



**Valid Inputs to a 3-Level Asymmetric Dyadic Synthesis Filter Bank**

For general information about the filter banks, see “Dyadic Synthesis Filter Banks”.

**Note** This block always performs frame-based processing, and its inputs must be of certain sizes. To use input subbands that do not fit the criteria of this block, use the Two-Channel Synthesis Subband Filter block. You can connect multiple copies of the Two-Channel Synthesis Subband Filter block to create a multilevel dyadic synthesis filter bank.

Data Types: single | double

## Output

### Output 1 — Reconstructed signal

column vector | matrix

The block outputs the reconstructed signal as a column vector or a matrix.

This table summarizes the output characteristics for both types of inputs. For an illustration of why the output characteristics exist, see the figure “Valid Inputs to a 3-Level Asymmetric Dyadic Synthesis Filter Bank” on page 2-537.

|                                | Input = Multiple ports   | Input = Single port<br>(Concatenated Subband Inputs)                  |
|--------------------------------|--|---|
| <b>Output Frame Rate</b>       | Same as the input frame rate.  | Same as the input rate (the rate of the concatenated subband inputs). |
| <b>Output Frame Dimensions</b> | <ul style="list-style-type: none"> <li>The output has the same number of columns as the inputs.</li> <li>The number of output rows depends on the tree structure of the filter bank:               <ul style="list-style-type: none"> <li><b>Asymmetric</b> — The number of output rows is twice the number of rows in the input to the topmost input port.</li> <li><b>Symmetric</b> — The number of output rows is the product of the number of input ports and the number of rows in an input to any input port.</li> </ul> </li> </ul> | The output has the same number of rows and columns as the input.      |

For general information about the filter banks, see “Dyadic Synthesis Filter Banks”.

Data Types: single | double

## Parameters

The parameters displayed in the block dialog box vary depending on the setting of the **Filter** parameter. Only some of the parameters described below are visible in the dialog box at any one time.

**Note** To use this block to reconstruct a signal decomposed by a Dyadic Analysis Filter Bank block, all the parameters in this block must be the same as the corresponding parameters in the Dyadic Analysis Filter Bank block (except the **Lowpass FIR filter coefficients** and **Highpass FIR filter coefficients**. See the descriptions of these parameters).

**Filter** — Type of filter used in filter bank

User-defined (default) | Haar | Daubechies | Symlets | Coiflets | Biorthogonal | Reverse Biorthogonal | Discrete Meyer

To determine the highpass and lowpass FIR filters in the filter bank:

- Select **User defined** to explicitly specify the filter coefficients in the **Lowpass FIR filter coefficients** and **Highpass FIR filter coefficients** parameters.
- Select a wavelet such as **Biorthogonal** or **Daubechies** to specify a wavelet-based filter. The block uses the Wavelet Toolbox `wfilters` function to construct the filters. Extra parameters such as **Wavelet order** or **Filter order [synthesis / analysis]** become enabled. For a list of the supported wavelets, see the table *Specifying Filters with the Filter Parameter and Related Parameters*.

You must specify the highpass and lowpass filters in the filter bank by setting the **Filter** parameter to one of the following options:

- **User defined** — Allows you to explicitly specify the filters with two vectors of filter coefficients in the **Lowpass FIR filter coefficients** and **Highpass FIR filter coefficients** parameters. The block uses the same lowpass and highpass filters throughout the filter bank. The two filters should be halfband filters, where each filter passes the frequency band that the other filter stops. To use this block to perfectly reconstruct a signal decomposed by a Dyadic Analysis Filter Bank block, the filters in this block *must* be designed to perfectly reconstruct the outputs of the analysis filter bank. To learn how to design your own perfect reconstruction filters, see “References” on page 2-542.
- **Wavelet** such as **Biorthogonal** or **Daubechies** — The block uses the specified wavelet to construct the lowpass and highpass filters using the Wavelet Toolbox function `wfilters`. Depending on the wavelet, the block might enable either the **Wavelet order** or **Filter order [synthesis / analysis]** parameter. (The latter parameter allows you to specify different wavelet orders for the analysis and synthesis filter stages.) To use this block to reconstruct a signal decomposed by a Dyadic Analysis Filter Bank block, you must set both blocks to use the same wavelets with the same order. You must have a Wavelet Toolbox license to use wavelets.

### Specifying Filters with the Filter Parameter and Related Parameters

| Filter                      | Sample Setting for Related Filter Specification Parameters  | Corresponding Wavelet Function Syntax |
|-----------------------------|---|---------------------------------------|
| <b>User-defined</b>         | Filters based on Daubechies wavelets with wavelet order 3: <ul style="list-style-type: none"> <li>• <b>Lowpass FIR filter coefficients</b> =<br/>[0.0352 -0.0854 -0.1350 0.4599<br/>0.8069 0.3327]</li> <li>• <b>Highpass FIR filter coefficients</b> =<br/>[-0.3327 0.8069 -0.4599<br/>-0.1350 0.0854 0.0352]</li> </ul> | None                                  |
| <b>Haar</b>                 | None  | <code>wfilters('haar')</code>         |
| <b>Daubechies</b>           | <b>Wavelet order</b> = 4  | <code>wfilters('db4')</code>          |
| <b>Symlets</b>              | <b>Wavelet order</b> = 3  | <code>wfilters('sym3')</code>         |
| <b>Coiflets</b>             | <b>Wavelet order</b> = 1  | <code>wfilters('coif1')</code>        |
| <b>Biorthogonal</b>         | <b>Filter order [synthesis / analysis]</b> =<br>[3/1]   | <code>wfilters('bior3.1')</code>      |
| <b>Reverse Biorthogonal</b> | <b>Filter order [synthesis / analysis]</b> =<br>[3/1]   | <code>wfilters('rbio3.1')</code>      |
| <b>Discrete Meyer</b>       | None  | <code>wfilters('dmey')</code>         |

**Filter order [synthesis / analysis]** — Wavelet order for synthesis filter stage

[1 / 1] (default) | [1 / 3] | [1 / 5] | [2 / 2] | [2 / 4] | [2 / 6] | [2 / 8] | [3 / 1] | [3 / 3] | [3 / 5] | [3 / 7] | [3 / 9] | [4 / 4] | [5 / 5] | [6 / 8]

The order of the wavelet for the synthesis and analysis filter stages. For example, when you set the **Filter** parameter to **Biorthogonal** and set the **Filter order [synthesis / analysis]** parameter to [2 / 6], the block calls the `wfilters` function with input argument 'bior2.6'.

**Dependencies**

To enable this parameter, set the **Filter** parameter to **Biorthogonal** or **Reverse Biorthogonal**.

**Wavelet order** — Wavelet order

2 (default) | positive integer

Set the order of the wavelet selected in the **Filter** parameter.

**Dependencies**

To enable this parameter, set the **Filter** parameter to **Daubechies**, **Symlets**, or **Coiflets**.

**Lowpass FIR filter coefficients** — Lowpass FIR filter coefficients

[0.3327 0.8069 0.4599 -0.1350 -0.0854 0.0352] (default)

Specifies coefficients used by all the lowpass filters in the filter bank using a vector of filter coefficients (descending powers of  $z$ ). This parameter is enabled when you set **Filter** to **User defined**. The lowpass filter should be a half-band filter that passes the frequency band stopped by the filter specified in the **Highpass FIR filter coefficients** parameter. To perfectly reconstruct a signal decomposed by the Dyadic Analysis Filter Bank, the filters in this block must be designed to perfectly reconstruct the outputs of the analysis filter bank. Otherwise, the reconstruction is not perfect. The default values of this parameter specify a perfect reconstruction filter for the default settings of the Dyadic Analysis Filter Bank (based on a Daubechies wavelet with wavelet order 3).

**Dependencies**

To enable this parameter, set the **Filter** parameter to **User-defined**.

**Highpass FIR filter coefficients** — Highpass FIR filter coefficients

[0.0352 0.0854 -0.1350 -0.4599 0.8069 -0.3327] (default)

Specifies coefficients used by all the highpass filters in the filter bank using a vector of filter coefficients (descending powers of  $z$ ). This parameter is enabled when you set **Filter** to **User defined**. The highpass filter should be a half-band filter that passes the frequency band stopped by the filter specified in the **Lowpass FIR filter coefficients** parameter. To perfectly reconstruct a signal decomposed by the Dyadic Analysis Filter Bank, the filters in this block must be designed to perfectly reconstruct the outputs of the analysis filter bank. Otherwise, the reconstruction is not perfect. The default values of this parameter specify a perfect reconstruction filter for the default settings of the Dyadic Analysis Filter Bank (based on a Daubechies wavelet with wavelet order 3).

**Dependencies**

To enable this parameter, set the **Filter** parameter to **User-defined**.

**Number of levels** — Number of filter bank levels

2 (default) | positive integer

Select the number of filter bank levels. An  $n$ -level asymmetric structure has  $n+1$  inputs, and an  $n$ -level symmetric structure has  $2^n$  inputs, as shown in “Multilevel Filter Banks”.

**Tree structure** — Structure of filter bank

Asymmetric (default) | Symmetric

Select the structure of the filter bank: `Asymmetric`, or `Symmetric`. See “Multilevel Filter Banks” for more details on these structures.

The default setting of this parameter is `Asymmetric` for the Dyadic Synthesis Filter Bank block, and `Symmetric` for the IDWT block.

**Input** — Input

Multiple ports (default) | Single port

Set to `Multiple ports` to accept each input subband at a separate port. The topmost port accepts the subband with the highest frequency band. Set to `Single port` to accept one vector or matrix of concatenated subbands at a single port. For more information, see `Input` on page 2-0 port description.

The default setting of this parameter is `Multiple ports` for the Dyadic Synthesis Filter Bank block, and `Single port` for the IDWT block.

When you set the **Input** parameter to `Multiple ports`, you must provide each subband to the block through a different input port as a vector or a matrix. You should input the highest frequency band through the topmost port. When you set the **Input** parameter to `Single port`, the block input must be a vector or matrix of concatenated subbands.

**Block Characteristics**

|                                 |                 |
|---------------------------------|-----------------|
| <b>Data Types</b>               | double   single |
| <b>Direct Feedthrough</b>       | no              |
| <b>Multidimensional Signals</b> | no              |
| <b>Variable-Size Signals</b>    | no              |
| <b>Zero-Crossing Detection</b>  | no              |

**Version History**

Introduced before R2006a

## References

- [1] Fliege, N. J. *Multirate Digital Signal Processing: Multirate Systems, Filter Banks, Wavelets*. West Sussex, England: John Wiley & Sons, 1994.
- [2] Strang, G. and T. Nguyen. *Wavelets and Filter Banks*. Wellesley, MA: Wellesley-Cambridge Press, 1996.
- [3] Vaidyanathan, P. P. *Multirate Systems and Filter Banks*. Englewood Cliffs, NJ: Prentice Hall, 1993.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Generated code relies on the `memcpy` or `memset` function (`string.h`) under certain conditions.

## See Also

### Objects

`dsp.DyadicSynthesisFilterBank`

### Blocks

Dyadic Analysis Filter Bank | IDWT | Two-Channel Synthesis Subband Filter

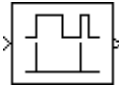
### Topics

“Multilevel Filter Banks”

“Calculate Channel Latencies Required for Wavelet Reconstruction”

# Edge Detector

Detect transition from zero to nonzero value



## Libraries:

DSP System Toolbox / Signal Management / Switches and Counters

## Description

The Edge Detector block generates an impulse (the value 1) in a given output channel when the corresponding channel of the input transitions from zero to a nonzero value. When the input does not transition from zero to a nonzero value, the block generates a zero in the corresponding output channel.

This block supports only discrete-time fixed-step signals. Continuous signals are not supported.

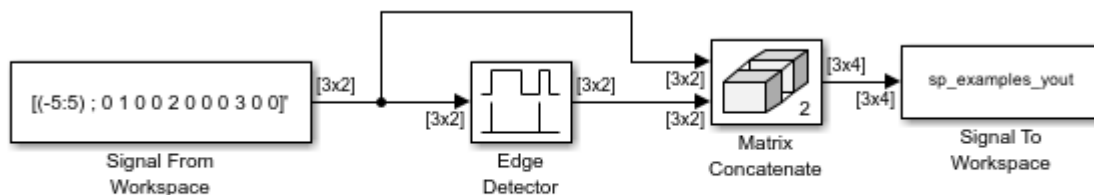
The output has the same dimension and sample rate as the input. When you set the **Input processing** parameter to `Columns as channels (frame based)`, the block counts an edge that is split across two consecutive frames in the frame that contains the nonzero value. For example, if there is a zero at the bottom of the first frame and a nonzero value at the top of the second frame, the block counts the edge in the second frame.

## Examples

### Detect Edges in Signal

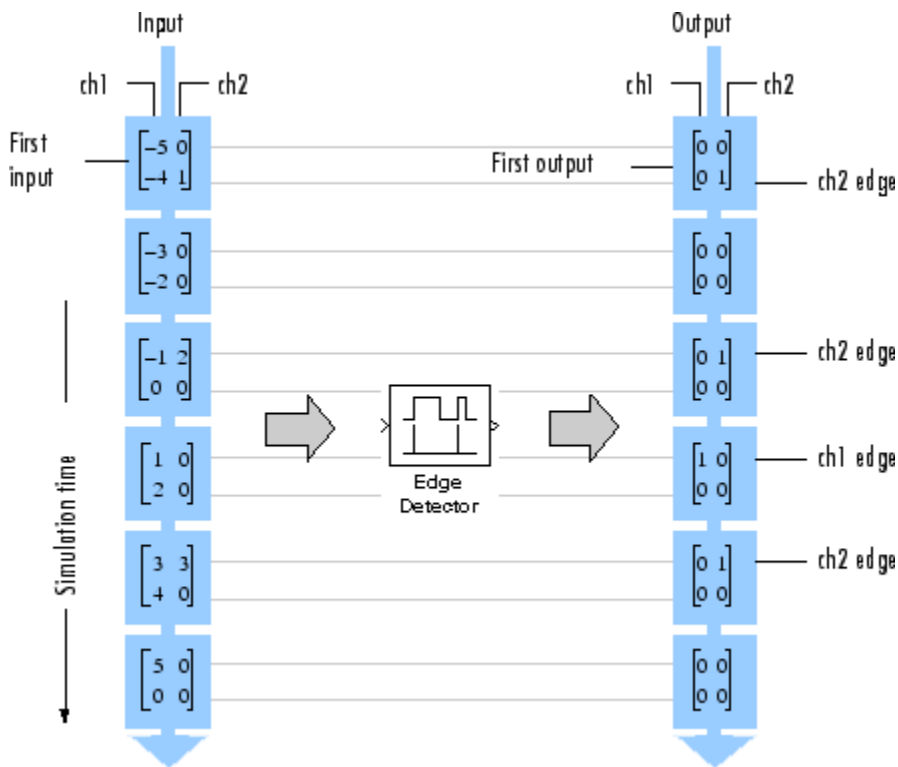
Detect edges in a signal using the Edge Detector block.

In the `ex_edgedetector_ref` model, the **Input processing** parameter of the Edge Detector block is set to `Columns as channels (frame based)`. Thus, the block interprets the 3-by-2 input as a multichannel signal with a frame size of 3. The Matrix Concatenate block concatenates the two input channels of the original signal with the two output channels of the Edge Detector block to create the four-channel workspace variable `sp_examples_yout`.



Copyright 2008-2010 The MathWorks, Inc.

As shown in the following figure, the block finds edges at sample 7 in channel 1, and at samples 2, 5, and 9 in channel 2.



Closing the model clears the "sp\_examples\_yout" variable from your workspace.

## Ports

### Input

**Input** — Input signal  
vector | matrix

Specify the input signal as a vector or a matrix.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated

### Output

**Output** — Output of edge detector  
vector | matrix

The block returns the output of the edge detector as a vector or a matrix.

The block might output Boolean values depending on the input data type, and whether Boolean support is enabled or disabled.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated



## Parameters

### Input processing — Input processing

Columns as channels (frame based) (default) | Elements as channels (sample based)

Specify how the block should process the input. You can set this parameter to one of the following options:

- Columns as channels (frame based) (default) — When you select this option, the block treats each column of the input as a separate channel.
- Elements as channels (sample based) — When you select this option, the block treats each element of the input as a separate channel.

## Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | Boolean   double   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no  |
| <b>Multidimensional Signals</b> | no  |
| <b>Variable-Size Signals</b>    | no  |
| <b>Zero-Crossing Detection</b>  | no  |

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Generated code relies on the memcpy or memset function (string.h) under certain conditions.

### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

## See Also

Counter | Event-Count Comparator

## Event-Count Comparator

Detect threshold crossing of accumulated nonzero inputs



## Library

Signal Management / Switches and Counters

dspswit3

## Description

The Event-Count Comparator block records the number of nonzero inputs to the Data port during the period that the block is enabled by a high signal (the value 1) at the Int port. Both inputs must be scalars.

When the number of accumulated nonzero inputs first equals the **Event threshold** setting, the block waits one additional sample interval, and then sets the output high (1). The block holds the output high until recording is restarted by a low-to-high (0-to-1) transition at the Int port.

The Event-Count Comparator block accepts real and complex floating-point and fixed-point inputs. However, because the block has discrete state, it does not support constant or continuous sample times. Therefore, at least one input or output port of the Event-Count Comparator block must be connected to a block whose **Sample time** parameter is discrete. The Event-Count Comparator block inherits this non-infinite discrete sample time.

## Parameters

### Event threshold

Specify the value against which to compare the number of nonzero inputs. Tunable (Simulink).

## Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point (signed and unsigned)
- Boolean
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers
- Enumerated

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Generated code relies on the `memcpy` or `memset` function (`string.h`) under certain conditions.

### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

## See Also

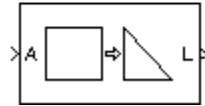
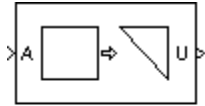
Counter | Edge Detector

## Topics

“Detect Threshold Crossings Using Event Count Comparator Block”

## Extract Triangular Matrix

Extract lower or upper triangle from input matrices



## Library

Math Functions / Matrices and Linear Algebra / Matrix Operations

dspmtx3

## Description

The Extract Triangular Matrix block creates a triangular matrix output from the upper or lower triangular elements of an  $M$ -by- $N$  input matrix. The block treats length- $M$  unoriented vector inputs as an  $M$ -by-1 matrix.

The **Extract** parameter selects between the two components of the input:

- **Upper** — Copies the elements on and above the main diagonal of the input matrix to an output matrix of the same size. The first *row* of the output matrix is therefore identical to the first *row* of the input matrix. The elements below the main diagonal of the output matrix are zero.
- **Lower** — Copies the elements on and below the main diagonal of the input matrix to an output matrix of the same size. The first *column* of the output matrix is therefore identical to the first *column* of the input matrix. The elements above the main diagonal of the output matrix are zero.

## Parameters

### Extract

The component of the matrix to copy to the output: upper triangle or lower triangle.

### Simulate using

**Code generation** (default) — Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than **Interpreted execution**.

**Interpreted execution** — Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than **Code generation**.

## Supported Data Types

| Port | Supported Data Types   |
|------|--|
| A    | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed and unsigned)</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul> |
| U    | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed and unsigned)</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul> |
| L    | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed and unsigned)</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul> |

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Generated code relies on the memcpy or memset function (string.h) under certain conditions.

### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

## See Also

### Functions

tril | triu

### Blocks

Autocorrelation LPC | Cholesky Factorization | Extract Diagonal | Forward Substitution | LDL Factorization | LU Factorization

**Topics**

“Extract Upper and Lower Triangles from Matrix”

# Farrow Rate Converter

Polynomial sample-rate converter with arbitrary conversion factor



## Libraries:

DSP System Toolbox / Signal Operations

## Description

The Farrow Rate Converter block converts the sample rate of an input signal using polynomial-fit sample-rate conversion. Polynomial-based filters are efficient at implementing fractional sample rate conversion. Farrow structures are implementations of polynomial-based filters. This block uses a Farrow structure to implement arbitrary rate-change factors efficiently. The rate-change factors can be irrational.

## Ports

### Input

**Port\_1** — Input signal

scalar | vector | matrix

Specify the input signal as a real or a complex-valued scalar, vector, or a matrix signal.

The block treats each column of the input signal as a separate channel. If the input is a two-dimensional signal, the first dimension represents the channel length (or frame size) and the second dimension represents the number of channels. If the input is a one-dimensional signal, then the block interprets it as a single-channel signal.

When you input a variable-size signal (frame length changes during simulation), the frame length of the signal can be arbitrary, that is, the input frame length does not have to be a multiple of the decimation factor. When you input a fixed-size signal (frame length does not change during simulation), the frame length can be arbitrary only when you select the **Allow arbitrary frame length for fixed-size input signals** parameter.

To determine the decimation factor, click the **View Info** button in the block dialog.

Data Types: single | double | int8 | int16 | int32 | int64 | fixed point

### Output

**Port\_1** — Rate converted signal

scalar | vector | matrix

The Farrow Rate Converter block outputs a signal that has the same data type and complexity as the input signal.

The output signal dimensions depend on the input signal dimensions, the overall interpolation and decimation factors of the Farrow Rate Converter block, and whether you select the **Allow arbitrary**

**frame length for fixed-size input signals** parameter. You can determine the overall interpolation factor  $L$  and the decimator factor  $M$  by clicking the **View Info** button in the block dialog box.

This table provides more details on the dimensions of the output signal when you input a fixed-size input signal.

### Fixed-Size Input Signal

| Input Signal  | Output Signal Dimensions   |
|---|--|
| $P$ -by-1 column vector, where $P$ is a multiple of the decimation factor $M$     | $(LP/M)$ -by-1, where $L$ is the interpolation factor  |
| $P$ -by-1 column vector, where $P$ is not a multiple of the decimation factor $M$ | $\text{ceil}(LP/M)$ -by-1 when you select <b>Allow arbitrary frame length for fixed-size input signals</b> .<br><br>If you do not select <b>Allow arbitrary frame length for fixed-size input signals</b> , the block errors.    |
| $P$ -by- $Q$ matrix, where $P$ is a multiple of the decimation factor $M$         | $(LP/M)$ -by- $Q$  |
| $P$ -by- $Q$ matrix, where $P$ is not a multiple of the decimation factor $M$     | $\text{ceil}(LP/M)$ -by- $Q$ when you select <b>Allow arbitrary frame length for fixed-size input signals</b> .<br><br>If you do not select <b>Allow arbitrary frame length for fixed-size input signals</b> , the block errors. |

This table gives more details on the dimensions of the output signal when you input a variable-size input signal. When you input a variable-size signal (frame length changes during simulation), the **Allow arbitrary frame length for fixed-size input signals** appears on the block dialog box but does not have any impact on the input frame length. You can input a variable-size signal of any frame length even if you do not select the **Allow arbitrary frame length for fixed-size input signals** parameter.

### Variable-Size Input Signal

| Input Signal | Output Signal Dimensions     |
|--------------|------------------------------|
| $P$ -by- $Q$ | $\text{ceil}(LP/M)$ -by- $Q$ |

Data Types: single | double | int8 | int16 | int32 | int64 | fixed point

## Parameters

### Main Tab

**Sample rate of input signal (Hz)** — Sample rate of input signal in Hz

48e3 (default) | positive scalar

Specify the sample rate of the input signal as a positive scalar in Hz. The input sample rate must be greater than the bandwidth of interest.



**Sample rate of output signal (Hz)** — Sample rate of output signal in Hz

96e3 (default) | positive scalar in Hz

Specify the sample rate of the output signal as a positive scalar in Hz. The output sample rate must be higher or lower than the input sample rate.

**Tolerance for output sample rate** — Maximum tolerance for output sample rate

0 (default) | positive scalar

Specify the maximum tolerance for the output sample rate as a positive scalar in the range [00.5].

The actual output sample rate varies but is within the specified range. For example, if you set the **Tolerance for output sample rate** to 0.01, then the actual output sample rate is in the range given by sample rate of output signal  $\pm 1\%$ . This flexibility allows for a simpler filter design.

**Filter Specification method** — Method for specifying filter coefficients

Polynomial order (default) | Coefficients

Specify the method the block uses to determine the polynomial interpolator coefficients as one of the following:

- **Polynomial order** — Specify the order of the Lagrange interpolation filter polynomial through the **Polynomial order** parameter.
- **Coefficients** — Specify the polynomial coefficients directly through the **Coefficients** parameter.

**Coefficients** — Filter polynomial coefficients

[-1 1; 1 0] (default) | real-valued square matrix

Specify the filter polynomial coefficients as a real-valued square matrix.

#### Dependencies

To enable this parameter, set **Filter specification method** to **Coefficients**.

**Polynomial order** — Order of Lagrange-interpolation-filter polynomial

3 (default) | positive integer less than or equal to 4

Specify the order of the Lagrange-interpolation-filter polynomial as 1, 2, 3, or 4.

#### Dependencies

To enable this parameter, set **Filter specification method** to **Polynomial order**.

**Allow arbitrary frame length for fixed-size input signals** — Allow arbitrary frame length for fixed-size input signals

on (default) | off

Specify whether fixed-size input signals (whose size does not change during simulation) can have an arbitrary frame length, where the frame length does not have to be a multiple of the decimation

factor. The block uses this parameter setting only for fixed-size input signals and ignores this parameter if the input has a variable-size.

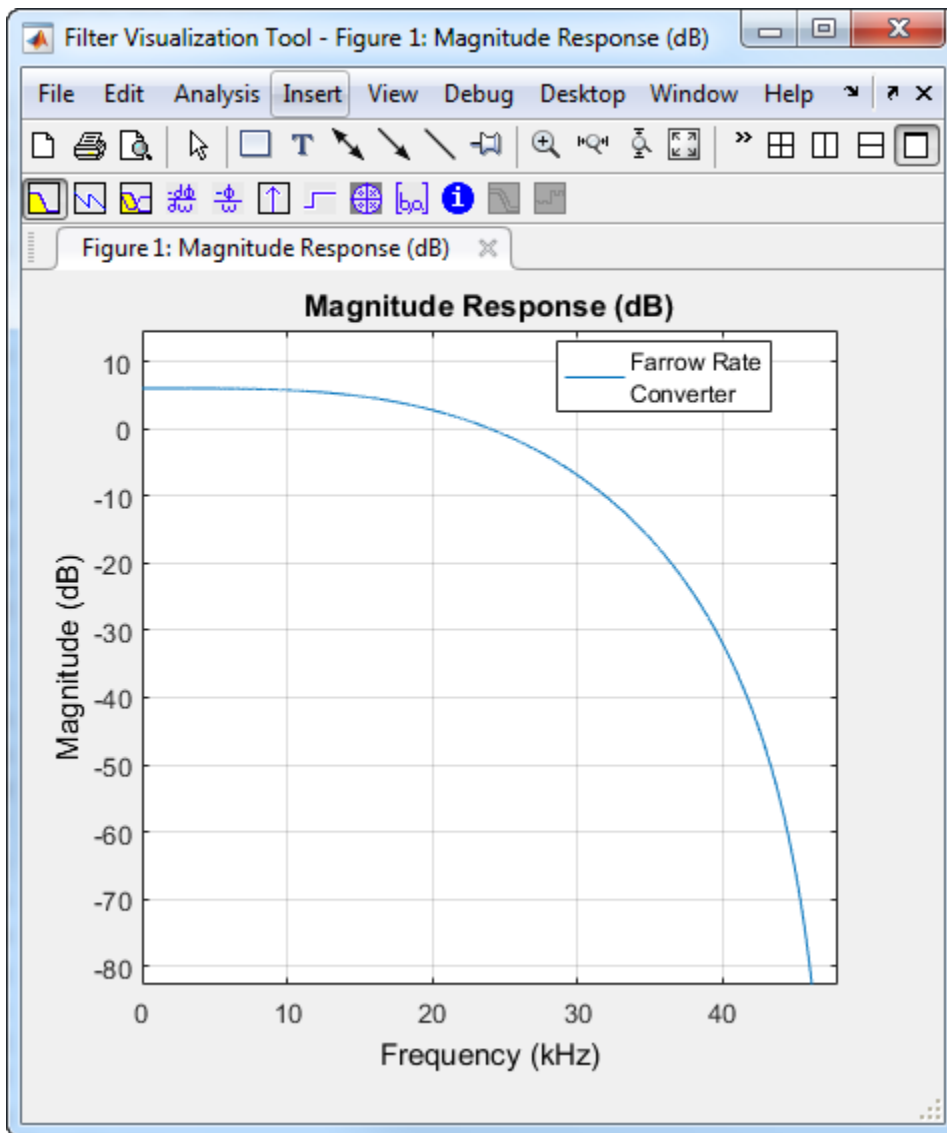
When the input signal is a variable-size signal, the signal can have arbitrary frame length, that is, the frame length does not have to be a multiple of the decimation factor.

For fixed-size input signals, if you:

- Select the **Allow arbitrary frame length for fixed-size input signals** parameter, the frame length of the signal does not have to be a multiple of the decimation factor. If the input is not a multiple of the decimation factor, then the output is generally a variable-size signal. Therefore, to support arbitrary input size, the block must also support variable-size operations, which you can enable by selecting the **Allow arbitrary frame length for fixed-size input signals** parameter.
- Clear the **Allow arbitrary frame length for fixed-size input signals** parameter, the input frame length must be a multiple of the decimation factor.

**View Filter Response** — View filter response  
button

Opens the FVTool and displays the magnitude/phase response of the Farrow Rate Converter. The response is based on the block parameters. FVTool updates when you change the parameters.

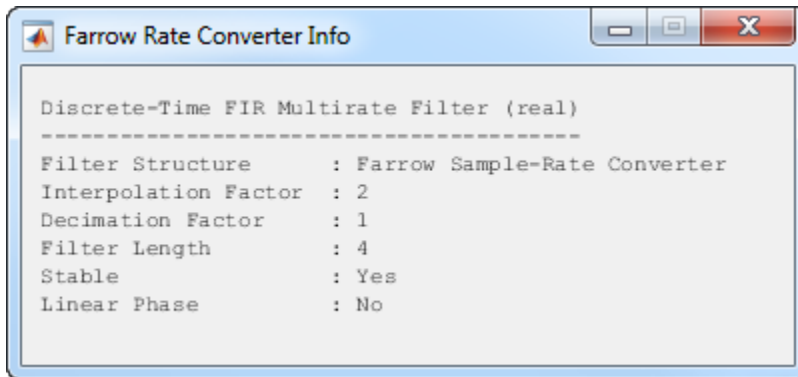


To update the magnitude response while FVTool is running, modify the parameters in the dialog box and click **Apply**.

**View Info** — View info  
button

Click this button for the block to display the following information about the Farrow filter system:

- Filter Structure
- Interpolation Factor
- Decimation Factor
- Filter Length
- Stable
- Linear Phase



This button brings the functionality of the `info` method into the Simulink environment.

**Simulate using** — Type of simulation to run

Code generation (default) | Interpreted execution

Type of simulation to run. You can set this parameter to:

- **Code generation** -- Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than **Interpreted execution**.
- **Interpreted execution** -- Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than **Code generation**.

### Data Types Tab

**Rounding mode** — Rounding mode

Floor (default) | Ceiling | Convergent | Nearest | Round | Zero

Specify the rounding mode for fixed-point operations as one of the following:

- Floor
- Ceiling
- Convergent
- Nearest
- Round
- Simplest
- Zero

For more details, see rounding mode.

**Saturate on integer overflow** — Method of overflow action

off (default) | on

When you select this parameter, the block saturates the result of its fixed-point operation. When you clear this parameter, the block wraps the result of its fixed-point operation. For details on `saturate` and `wrap`, see `overflow mode for fixed-point operations`.

**Coefficients** — Data type of filter coefficients

`fixdt(1,16)` (default) | `fixdt(1,16,0)`

Specify the data type of the filter coefficients as a signed fixed-point object. The default `fixdt(1,16)` corresponds to a signed fixed-point type object with 16-bit coefficients. For maximum precision, the block determines the fraction length based on the coefficient values.

**Fractional Delay** — Data type of fractional delay

`fixdt(0,8)` (default) | `fixdt(0,16)`

Specify the data type of the fractional delay as an unsigned fixed-point object. The default `fixdt(0,8)` corresponds to an unsigned fixed-point data type object with 8-bit word length. For maximum precision, the block computes the fractional length based on the fractional delay values.

**Multiplicand** — Data type of multiplicand

`fixdt(1,16,13)` (default) | `fixdt(1,16,0)`

Specify the multiplicand data type as a signed fixed-point object. The default `fixdt(1,16,13)` corresponds to a signed fixed-point multiplicand data type with 16-bit word length and 13-bit fraction length.

**Output** — Data type of output

Inherit: Same word length as input (default) | Inherit: Same as accumulator |  
`fixdt(1,16)` | `fixdt(1,16,0)`

Specify the word length and fraction length of the output data type as one of the following:

- `Inherit: Same word length as input` (default) — Output word length and fraction lengths are the same as the input.
- `Inherit: Same as accumulator` — Output word length and fraction lengths are the same as the accumulator.
- `fixdt(1,16)` — Signed fixed-point data type with 16-bit word length. For maximum precision, the block computes the fraction length based on the input range and preserves the dynamic range of the input.
- `fixdt(1,16,0)` — Signed fixed-point data type with 16-bit word length and zero fraction length.

**Output Minimum** — Minimum value that the block can output

`[]` (default) | scalar

Specify the minimum value that the block can output. Simulink software uses this minimum value to perform:

- Simulation range checking. See “Specify Signal Ranges” (Simulink).
- Automatic scaling of fixed-point data types.

**Output Maximum** — Maximum value that the block can output

[] (default) | scalar

Specify the maximum value that the block can output. Simulink software uses this maximum value to perform:

- Simulation range checking. See “Specify Signal Ranges” (Simulink).
- Automatic scaling of fixed-point data types.

**Lock data type settings against changes by the fixed-point tools** — Prevent fixed-point tools from overriding data types

off (default) | on

Select this parameter to prevent the fixed-point tools from overriding the data types you specify in the block dialog box.

## Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | double   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no                                      |
| <b>Multidimensional Signals</b> | no                                      |
| <b>Variable-Size Signals</b>    | yes                                     |
| <b>Zero-Crossing Detection</b>  | no                                      |

## Algorithms

Farrow filters implement piecewise polynomial interpolation using Horner’s rule to compute samples from the polynomial. The polynomial coefficients used to fit the input samples correspond to the Lagrange interpolation coefficients.

Once a polynomial is fitted to the input data, the value of the polynomial can be calculated at any point. Therefore, a polynomial filter enables interpolation at arbitrary locations between input samples.

You can use a polynomial of any order to fit to the existing samples. However, since large-order polynomials frequently oscillate, polynomials of order 1, 2, 3, or 4 are used in practice.

The algorithm computes interpolated values at the desired locations by varying only the fractional delay  $\mu$ . This value is the interval between the previous input sample and the current output sample. All filter coefficients remain constant.

- The input samples are filtered using  $M + 1$  FIR filters, where  $M$  is the polynomial order.
- The outputs of these filters are multiplied by the fractional delay,  $\mu$ .
- The output is the sum of the multiplication results.

## Version History

Introduced in R2015b

### R2022b: Support for arbitrary input frame length

Starting in R2022b, this block supports input signals with arbitrary frame lengths when the:

- Input signal is a fixed-size signal (frame length does not change during simulation) and you select the **Allow arbitrary frame length for fixed-size input signals** parameter
- Input signal is a variable-size signal (frame length changes during simulation)

When this block supports an arbitrary frame length input signal, the input frame length does not have to be a multiple of the decimation factor.

### Extended Capabilities

#### C/C++ Code Generation

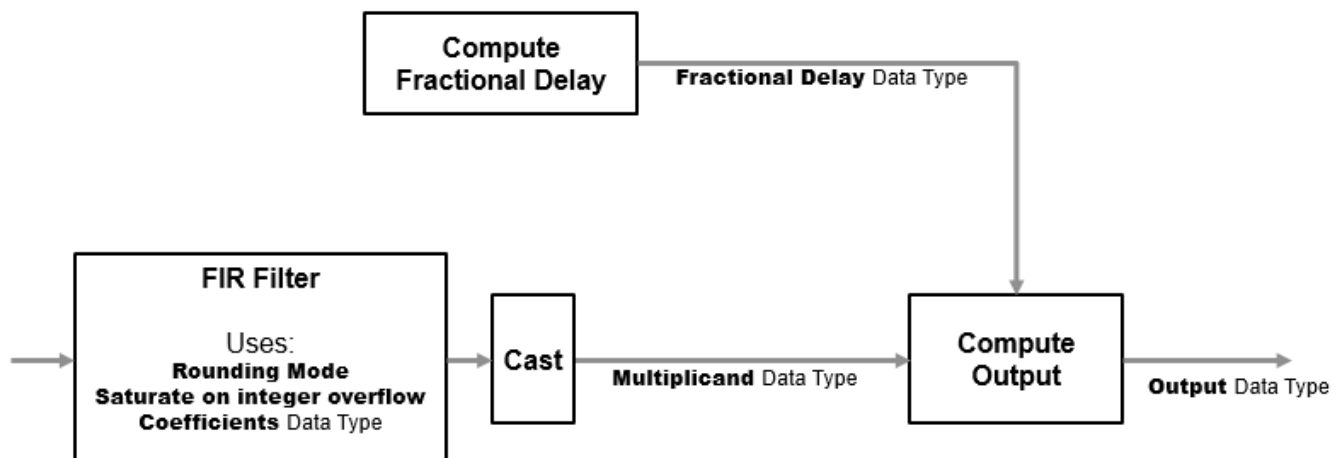
Generate C and C++ code using Simulink® Coder™.

#### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

If the input is fixed point, it must be a signed integer or a signed fixed point value with a power-of-two slope and zero bias.

The diagram shows the data types that the Farrow Rate Converter block uses for fixed-point signals and floating-point signals. You can specify these data types using the parameters on the **Data Types** tab. If the input is floating point, all data types in the filter are the same as the input data type, single or double.



If the input is fixed point, the FIR filter defines internal data types using the **Rounding mode**, **Saturate on integer overflow**, and **Coefficients** parameters. The accumulators and products within the FIR filter use full-precision data types. The block casts the output of the FIR filter to the **Multiplicand** data type.

## **See Also**

### **Objects**

`dsp.FarrowRateConverter`

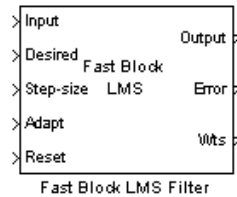
### **Blocks**

Sample-Rate Converter



## Fast Block LMS Filter

Compute output, error, and weights using LMS adaptive algorithm



## Library

Filtering / Adaptive Filters

dspadpt3

## Description

The Fast Block LMS Filter block implements an adaptive least mean-square (LMS) filter, where the adaptation of the filter weights occurs once for every block of data samples. The block estimates the filter weights, or coefficients, needed to convert the input signal into the desired signal. Connect the signal you want to filter to the Input port. The input signal can be a scalar or a column vector. Connect the signal you want to model to the Desired port. The desired signal must have the same data type, complexity, and dimensions as the input signal. The Output port outputs the filtered input signal. The Error port outputs the result of subtracting the output signal from the desired signal.

The block calculates the filter weights using the Block LMS Filter equations. For more information, see Block LMS Filter. The Fast Block LMS Filter block implements the convolution operation involved in the calculations of the filtered output,  $y$ , and the weight update function in the frequency domain using the FFT algorithm used in the Overlap-Save FFT Filter block. See Overlap-Save FFT Filter (Obsolete) for more information.

Use the **Filter length** parameter to specify the length of the filter weights vector.

The **Block size** parameter determines how many samples of the input signal are acquired before the filter weights are updated. The input frame length must be a multiple of the **Block size** parameter.

The **Step-size ( $\mu$ )** parameter corresponds to  $\mu$  in the equations. You can either specify a step-size using the input port, Step-size, or enter a value in the Block Parameters: Block LMS Filter dialog box.

Use the **Leakage factor (0 to 1)** parameter to specify the leakage factor,  $0 < 1 - \mu\alpha \leq 1$ , in the leaky LMS algorithm shown below.

$$\mathbf{w}(k) = (1 - \mu\alpha)\mathbf{w}(k - 1) - f(\mathbf{u}(n), e(n), \mu)$$

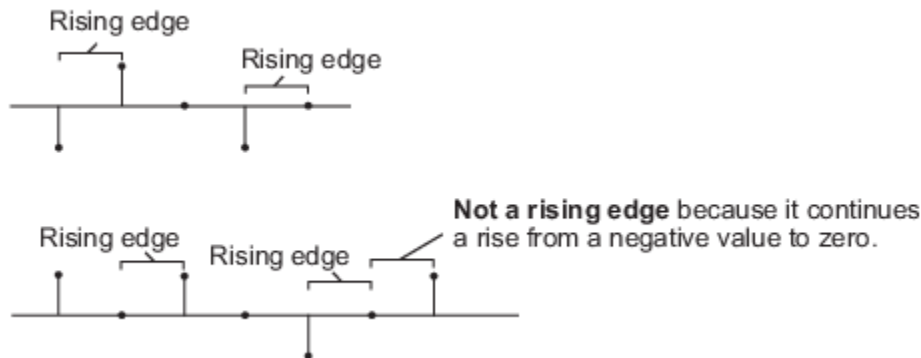
Enter the initial filter weights,  $\mathbf{w}(0)$ , as a vector or a scalar in the **Initial value of filter weights** text box. When you enter a scalar, the block uses the scalar value to create a vector of filter weights. This vector has length equal to the filter length and all of its values are equal to the scalar value.

When you select the **Adapt port** check box, an Adapt port appears on the block. When the input to this port is nonzero, the block continuously updates the filter weights. When the input to this port is zero, the filter weights remain at their current values.

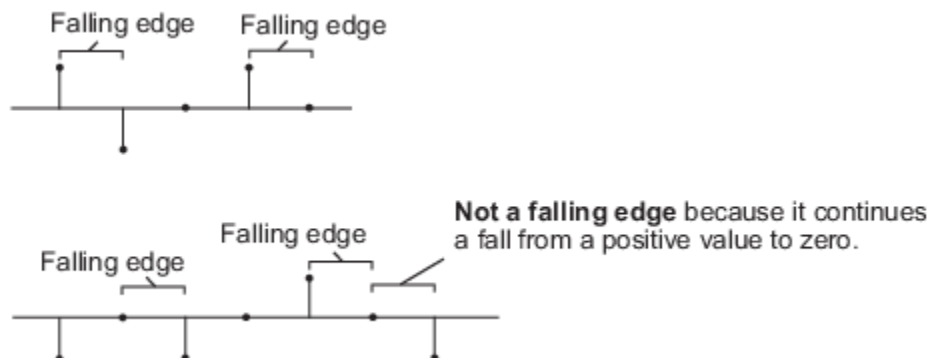
When you want to reset the value of the filter weights to their initial values, use the **Reset input** parameter. The block resets the filter weights whenever a reset event is detected at the Reset port. The reset signal rate must be the same rate as the data signal input.

From the **Reset input** list, select **None** to disable the Reset port. To enable the Reset port, select one of the following from the **Reset input** list:

- **Rising edge** — Triggers a reset operation when the Reset input does one of the following:
  - Rises from a negative value to a positive value or zero
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure)



- **Falling edge** — Triggers a reset operation when the Reset input does one of the following:
  - Falls from a positive value to a negative value or zero
  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero (see the following figure)



- **Either edge** — Triggers a reset operation when the Reset input is a **Rising edge** or **Falling edge** (as described above)
- **Non-zero sample** — Triggers a reset operation at each sample time that the Reset input is not zero

Select the **Output filter weights** check box to create a Wts port on the block. For each iteration, the block outputs the current updated filter weights from this port.

## Parameters

### Filter length

Enter the length of the FIR filter weights vector. The sum of the **Block size** and the **Filter length** must be a power of 2.

### Block size

Enter the number of samples to acquire before the filter weights are updated. The number of rows in the input must be an integer multiple of the **Block size**. The sum of the **Block size** and the **Filter length** must be a power of 2.

### Specify step-size via

Select Dialog to enter a value for mu, or select Input port to specify mu using the Step-size input port.

### Step-size (mu)

Enter the step-size. Tunable (Simulink).

### Leakage factor (0 to 1)

Enter the leakage factor,  $0 < 1 - \mu\alpha \leq 1$ . Tunable (Simulink).

### Initial value of filter weights

Specify the initial values of the FIR filter weights.

### Adapt port

Select this check box to enable the Adapt input port.

### Reset input

Select this check box to enable the Reset input port.

### Output filter weights

Select this check box to export the filter weights from the Wts port.

## References

Hayes, M.H. *Statistical Digital Signal Processing and Modeling*. New York: John Wiley & Sons, 1996.

## Supported Data Types

| Port      | Supported Data Types   |
|-----------|--|
| Input     | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> </ul> |
| Desired   | <ul style="list-style-type: none"> <li>• Must be the same as Input</li> </ul>  |
| Step-size | <ul style="list-style-type: none"> <li>• Must be the same as Input</li> </ul>  |

| Port   | Supported Data Types  |
|--------|---|
| Adapt  | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul> |
| Reset  | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul> |
| Output | <ul style="list-style-type: none"> <li>• Same as Input</li> </ul>   |
| Error  | <ul style="list-style-type: none"> <li>• Same as Input</li> </ul>   |
| Wts    | <ul style="list-style-type: none"> <li>• Same as Input</li> </ul>   |

## See Also

|                                   |                    |
|-----------------------------------|--------------------|
| Block LMS Filter                  | DSP System Toolbox |
| Kalman Adaptive Filter (Obsolete) | DSP System Toolbox |
| LMS Filter                        | DSP System Toolbox |
| RLS Filter                        | DSP System Toolbox |

See “Noise Cancellation in Simulink Using Normalized LMS Adaptive Filter” for related information.

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Generated code relies on the memcopy or memset function (string.h) under certain conditions.

# FFT

Fast Fourier transform (FFT) of input



**Libraries:**  
DSP System Toolbox / Transforms

## Description

The FFT block computes the fast Fourier transform (FFT) across the first dimension of an  $N$ -D input array,  $u$ . The block uses one of two possible FFT implementations. You can select an implementation based on the FFTW library or an implementation based on a collection of Radix-2 algorithms. To allow the block to choose the implementation, you can select **Auto**. For more information about the FFT implementations, see “Algorithms” on page 2-571.

For user-specified FFT lengths not equal to  $P$ , zero padding or truncating, or modulo-length data wrapping occurs before the FFT operation. For an FFT with  $P \leq M$ :

```
y = fft(u,M) % P ≤ M
```

Wrapping:

```
y(:,L) = fft(datawrap(u(:,L),M)) % P > M; L = 1, ..., N
```

Truncating:

```
y(:,L) = fft(u,M) % P > M; L = 1, ..., N
```

---

**Tip** When the input length,  $P$ , is greater than the FFT length,  $M$ , you may see magnitude increases in your FFT output. These magnitude increases occur because the FFT block uses modulo- $M$  data wrapping to preserve all available input samples.

To avoid such magnitude increases, you can truncate the length of your input sample,  $P$ , to the FFT length,  $M$ . To do so, place a Pad block before the FFT block in your model.

---

## Ports

### Input

**Port\_1** — Input signal  
vector | matrix |  $N$ -D array

Input signal for computing the FFT. The block computes the FFT along the first dimension of the  $N$ -D input signal.

For more information on how the block computes the FFT, see “Description” on page 2-565 and “Algorithms” on page 2-571.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fixed point

## Output

### Port 1 — FFT of input

vector | matrix |  $N$ -D array

The FFT, computed across the first dimension of an  $N$ -D input array. When the output of the block has an integer or fixed-point data type, it is always signed.

The  $k$ th entry of the  $L$ th output channel,  $y(k,L)$ , equals the  $k$ th point of the  $M$ -point discrete Fourier transform (DFT) of the  $L$ th input channel:

$$y(k, L) = \sum_{p=1}^P u(p, L) e^{-j2\pi(p-1)(k-1)/M} \quad k = 1, \dots, M$$

For more information on how the block computes the FFT, see “Description” on page 2-565 and “Algorithms” on page 2-571.

Data Types: single | double | int8 | int16 | int32 | fixed point

## Parameters

### Main

#### FFT implementation — FFT implementation

Auto (default) | Radix-2 | FFTW

Set this parameter to FFTW to support an arbitrary length input signal. The block restricts generated code with FFTW implementation to host computers capable of running MATLAB.

Set this parameter to Radix-2 for bit-reversed processing, fixed or floating-point data, or portable C-code generation using the Simulink Coder. The dimension  $M$  of the  $M$ -by- $N$  input matrix, must be a power of two. To work with other input sizes, use the Pad block to pad or truncate these dimensions to powers of two, or if possible choose the FFTW implementation. For more information about the algorithms used by the Radix-2 mode, see “Radix-2 Implementation” on page 2-571.

Set this parameter to Auto to let the block choose the FFT implementation. For floating-point inputs with non-power-of-two transform lengths, the FFTW algorithm is automatically chosen. Otherwise a Radix-2 algorithm is automatically chosen. For non-power-of-two transform lengths, the block restricts generated code to MATLAB host computers.

#### Output in bit-reversed order — Output in bit-reversed order

off (default) | on

Designate the order of the output channel elements relative to the ordering of the input elements. When you select this check box, the output channel elements appear in bit-reversed order relative to the input ordering. If you clear this check box, the output channel elements appear in linear order relative to the input ordering.

---

**Note** The FFT block calculates its output in bit-reversed order. Linearly ordering the FFT block output requires an extra bit-reversal operation. In many situations, you can increase the speed of the FFT block by selecting the **Output in bit-reversed order** check box.

---

For more information ordering of the output, see “Linear and Bit-Reversed Output Order”.

#### Dependencies

To enable this parameter, set **FFT implementation** to Auto or Radix-2.

#### Divide output by FFT length — Divide output by FFT length

off (default) | on

When you select this parameter, the block divides the output of the FFT by the FFT length. This option is useful when you want the output of the FFT to stay in the same amplitude range as its input. This is particularly useful when working with fixed-point data types.

#### Inherit FFT length from input dimensions — Inherit FFT length from input dimensions

on (default) | off

Select to inherit the FFT length from the input dimensions. When you select this check box, the input length must be a power of two.

#### Dependencies

When you do not select this check box, the **FFT length** parameter becomes available to specify the length.

#### FFT length — FFT length

64 (default) | integer

Specify FFT length as an integer greater than or equal to two.

When you set the **FFT implementation** parameter to Radix-2, or when you check the **Output in bit-reversed order** check box, this value must be a power of two.

#### Dependencies

To enable this parameter, clear the **Inherit FFT length from input dimensions** check box.

#### Wrap input data when FFT length is shorter than input length — Wrap or truncate input

on (default) | off

Choose to wrap or truncate the input, depending on the FFT length. If you select this parameter, modulo-length data wrapping occurs before the FFT operation when the FFT length is shorter than the input length. If you clear this check box, truncation of the input data to the FFT length occurs before the FFT operation.

#### Dependencies

To enable this parameter, clear the **Inherit FFT length from input dimensions** check box.

#### Data Types

#### Rounding mode — Rounding method

Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Select the rounding mode for fixed-point operations.

#### Limitations

The sine table values do not obey this parameter; instead, they always round to **Nearest**.

The **Rounding mode** parameter has no effect on numeric results when all these conditions are met:

- **Product output** data type is `Inherit: Inherit via internal rule`.
- **Accumulator** data type is `Inherit: Inherit via internal rule`.

With these data type settings, the block operates in full-precision mode.

#### Saturate on integer overflow — Saturate on integer overflow

`off (default) | on`

When you select this parameter, the block saturates the result of its fixed-point operation. When you clear this parameter, the block wraps the result of its fixed-point operation. For details on **saturate** and **wrap**, see **overflow mode for fixed-point operations**.

#### Limitations

The **Saturate on integer overflow** parameter has no effect on numeric results when all these conditions are met:

- **Product output** data type is `Inherit: Inherit via internal rule`.
- **Accumulator** data type is `Inherit: Inherit via internal rule`.


With these data type settings, the block operates in full-precision mode.

#### Sine table — Data type of sine table values

`Inherit: Same word length as input (default) | fixdt(1,16)`

Choose how to specify the word length of the values of the sine table. The fraction length of the sine table values always equals the word length minus one. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Same word length as input`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Sine table** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) for more information.

#### Limitations

The sine table values do not obey the **Rounding mode** and **Saturate on integer overflow** parameters; instead, they are always saturated and rounded to **Nearest**.


#### Product output — Product output data type

`Inherit: Inherit via internal rule (default) | Inherit: Same as input | fixdt(1,16,0)`



Specify the product output data type. See “Fixed Point” on page 2-573 and “Multiplication Data Types” for illustrations depicting the use of the product output data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Product output** parameter.


See “Specify Data Types Using Data Type Assistant” (Simulink) for more information.

### Accumulator — Accumulator data type

`Inherit: Inherit via internal rule (default) | Inherit: Same as input | Inherit: Same as product output | fixdt(1,16,0)`

Specify the accumulator data type. See “Fixed Point” on page 2-573 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Accumulator** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) for more information.

### Output — Output data type

`Inherit: Inherit via internal rule (default) | Inherit: Same as input | fixdt(1,16,0)`

Specify the output data type. See “Fixed Point” on page 2-573 for illustrations depicting the use of the output data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`.

When you select `Inherit: Inherit via internal rule`, the block calculates the output word length and fraction length automatically. The equations that the block uses to calculate the ideal output word length and fraction length depend on the setting of the **Divide output by FFT length** check box.


- When you select the **Divide output by FFT length** check box, the ideal output word and fraction lengths are the same as the input word and fraction lengths.
- When you clear the **Divide output by FFT length** check box, the block computes the ideal output word and fraction lengths according to the following equations:

$$WL_{idealoutput} = WL_{input} + \text{floor}(\log_2(FFTlength - 1)) + 1$$

$$FL_{idealoutput} = FL_{input}$$

Using these ideal results, the internal rule then selects word lengths and fraction lengths that are appropriate for your hardware. For more information, see “Inherit via Internal Rule”.

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Output** parameter.

See “Control Data Types of Signals” (Simulink) for more information.

**Output Minimum** — Minimum value block should output  
[] (default) | scalar

Specify the minimum value that the block should output. The default value is [] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Specify Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

**Output Maximum** — Maximum value block should output  
[] (default) | scalar

Specify the maximum value that the block should output. The default value is [] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Specify Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

**Lock data type settings against changes by the fixed-point tools** — Prevent fixed-point tools from overriding data types  
off (default) | on

Select this parameter to prevent the fixed-point tools from overriding the data types you specify in the block dialog box.

## Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | double   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no                                      |
| <b>Multidimensional Signals</b> | yes                                     |
| <b>Variable-Size Signals</b>    | yes <sup>a</sup>                        |
| <b>Zero-Crossing Detection</b>  | no                                      |

<sup>a</sup> Variable-size signals are only supported when the Inherit FFT length from input dimensions checkbox is selected.

## Algorithms

### FFTW Implementation

The FFTW implementation provides an optimized FFT calculation including support for power-of-two and non-power-of-two transform lengths in both simulation and code generation. Generated code using the FFTW implementation can only run on computers capable of running MATLAB. The input data type must be floating-point.

### Radix-2 Implementation

The Radix-2 implementation supports bit-reversed processing, fixed or floating-point data, and allows the block to provide portable C-code generation using the Simulink Coder. The dimension  $M$  of the  $M$ -by- $N$  input matrix must be a power of two. To work with other input sizes, use the Pad block to pad or truncate these dimensions to powers of two, or if possible choose the FFTW implementation.

With Radix-2 selected, the block implements one or more of the following algorithms:

- Butterfly operation
- Double-signal algorithm
- Half-length algorithm
- Radix-2 decimation-in-time (DIT) algorithm
- Radix-2 decimation-in-frequency (DIF) algorithm

### Radix-2 Algorithms for Real and Complex Signals

| Complexity of Input | Output Ordering | Algorithms Used for FFT Computation   |
|---------------------|-----------------|---|
| Complex             | Linear          | Bit-reversed operation and radix-2 DIT  |
| Complex             | Bit-reversed    | Radix-2 DIF   |
| Real                | Linear          | Bit-reversed operation and radix-2 DIT in conjunction with the half-length and double-signal algorithms |
| Real                | Bit-reversed    | Radix-2 DIF in conjunction with the half-length and double-signal algorithms                            |

The efficiency of the FFT algorithm can be enhanced for real input signals by forming complex-valued sequences from the real-valued sequences prior to the computation of the DFT. When there are  $2N+1$  real input channels, the FFT block forms these complex-valued sequences by applying the double-signal algorithm to the first  $2N$  input channels, and the half-length algorithm to the last odd-numbered channel.

For real input signals with fixed-point data types, different numerical results might appear in the output of the last odd-numbered channel, even when all input channels are identical. This numerical difference results from differences in the double-signal algorithm and the half-length algorithm.

You can eliminate this numerical difference in two ways:

- Using full precision arithmetic for fixed-point input signals
- Changing the input data type to floating point

For more information on the double-signal algorithm, see [2], “Efficient Computation of the DFT of Two Real Sequences” on page 475. For more information on the half-length algorithm, see [2], “Efficient Computation of the DFT of a 2N-Point Real Sequence” on page 476.

### Radix-2 Optimization for the Table of Trigonometric Values

In certain situations, the block’s Radix-2 algorithm computes all the possible trigonometric values of the twiddle factor

$$e^{j\frac{2\pi k}{K}}$$

where  $K$  is the greater value of either  $M$  or  $N$  and  $k = 0, \dots, K - 1$ . The block stores these values in a table and retrieves them during simulation. The number of table entries for fixed-point and floating-point is summarized in the following table:

| Number of Table Entries for N-Point FFT |        |
|---|--------|
| floating-point                          | $3N/4$ |
| fixed-point                             | $N$    |

## Version History

Introduced before R2006a

## References

- [1] Orfanidis, S. J. *Introduction to Signal Processing*. Upper Saddle River, NJ: Prentice Hall, 1996, p. 497.
- [2] Proakis, John G. and Dimitris G. Manolakis. *Digital Signal Processing*, 3rd ed. Upper Saddle River, NJ: Prentice Hall, 1996.
- [3] FFTW (<https://www.fftw.org>)
- [4] Frigo, M. and S. G. Johnson, “FFTW: An Adaptive Software Architecture for the FFT,” *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Vol. 3, 1998, pp. 1381-1384.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Usage notes and limitations:

- When the following conditions apply, the executable generated from this block relies on prebuilt dynamic library files (.dll files) included with MATLAB:
  - **FFT implementation** is set to FFTW.
  - **Inherit FFT length from input dimensions** is cleared, and **FFT length** is set to a value that is not a power of two.

Use the packNGo function to package the code generated from this block and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see “How To Run a Generated Executable Outside MATLAB”.

- When the FFT length is a power of two, you can generate standalone C and C++ code from this block.

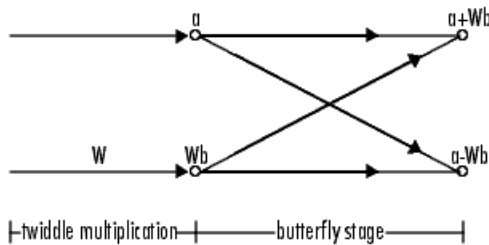
**Fixed-Point Conversion**

Design and simulate fixed-point systems using Fixed-Point Designer™.

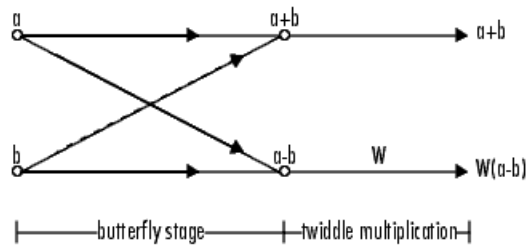
The following diagrams show the data types used in the FFT block for fixed-point signals. You can set the **Sine table**, **Accumulator**, **Product output**, and **Output** data types displayed in the diagrams in the FFT dialog box as discussed in “Parameters” on page 2-566.

Inputs to the FFT block are first cast to the output data type and stored in the output buffer. Each butterfly stage then processes signals in the accumulator data type, with the final output of the butterfly being cast back into the output data type. The block multiplies in a twiddle factor before each butterfly stage in a decimation-in-time FFT and after each butterfly stage in a decimation-in-frequency FFT.

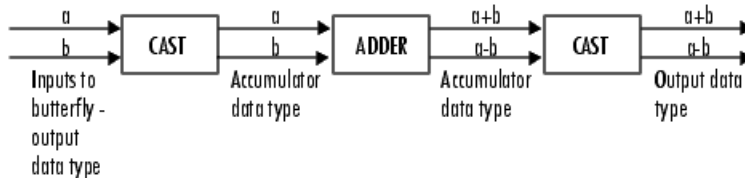
**Decimation-in-time IFFT**



**Decimation-in-frequency IFFT**



**Butterfly stage data types**



**Twiddle multiplication data types**



The output of the multiplier appears in the accumulator data type because both of the inputs to the multiplier are complex. For details on the complex multiplication performed, see “Multiplication Data Types”.

---

**Note** When the block input is fixed point, all internal data types are signed fixed point.

---

## **See Also**

### **Objects**

`dsp.FFT` | `dsp.IFFT`

### **Functions**

`fft` | `ifft` | `bitrevorder`

### **Blocks**

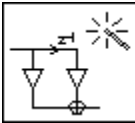
DCT | IFFT | Pad

### **Topics**

“Linear and Bit-Reversed Output Order”

# Filter Realization Wizard

Construct filter realizations using digital filter blocks or Sum, Gain, and Delay blocks



## Library

Filtering / Filter Designs

dspfdesign

## Description

---

**Note** Use this block to implement fixed-point or floating-point digital filters using Sum, Gain, and Delay blocks or digital filter blocks from the DSP System Toolbox library. You can either design a filter by using the block parameters, or import the coefficients of a filter you have designed elsewhere.

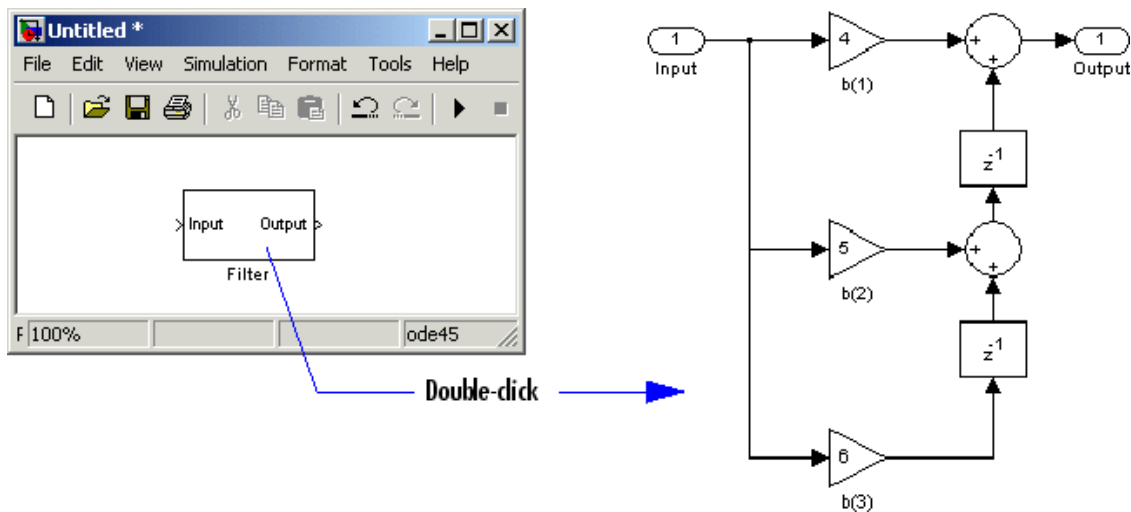
The following blocks also implement digital filters, but serve slightly different purposes:

- Discrete FIR Filter and Biquad Filter— Use to implement floating-point or fixed-point filters that you have already designed
- Digital Filter Design — Use to design, analyze, and then implement floating-point filters.

---

The Filter Realization Wizard is a tool for automatically implementing a digital filter. You must specify a filter, its structure, and the data types for the inputs, outputs, and computations. The filter can support double-precision, single-precision, or fixed-point data types.

The Filter Realization Wizard can implement a digital filter in one of two ways. It can use digital filter blocks from the DSP System Toolbox library, or it can create a subsystem block that implements the specified filter using Sum, Gain, and Delay blocks. If the Filter Realization Wizard creates a block, double-click the block to open the dialog box. If it creates a subsystem, double-click the subsystem block to see the filter implementation as shown in the figure below.



For more information about filter implementation, see “Specify the Filter Implementation” on page 2-577.

The parameters of the Filter Realization Wizard are a part of a larger app, the Filter Designer (`filterDesigner`). You can use filter designer to design and analyze your filter, and then use the Filter Realization Wizard parameters to implement the filter in your models.

### Specify the Filter and Data Types

To specify a purely double-precision filter, you can either design a filter using the **Design Filter** panel, or import a filter using the **Import Filter** panel. In the **Import Filter** panel, you can specify the coefficients directly or specify the workspace variables which store the coefficients.

You can also specify a fixed-point filter or a single-precision filter by using the **Set Quantization Parameters** panel.

---

**Note** *Running* a model containing implementations of fixed-point filters requires the Fixed-Point Designer product, but you can still edit models containing such filter implementations without it. See the Fixed-Point Designer documentation for more information.

---

See the following topics to learn how to use the panels to specify your filter:

- For more information on the **Design Filter** panel, see `filterDesigner`.
- For more information on the **Import Filter** panel, see “Importing a Filter Design”.
- For more information on the **Set Quantization Parameters** panel, see “Access the Quantization Features of Filter Designer”.

To open a panel, click the appropriate button in the lower-left corner of filter designer.

### Supported Filter Structures

The Filter Realization Wizard supports the following structures:

- Direct form I



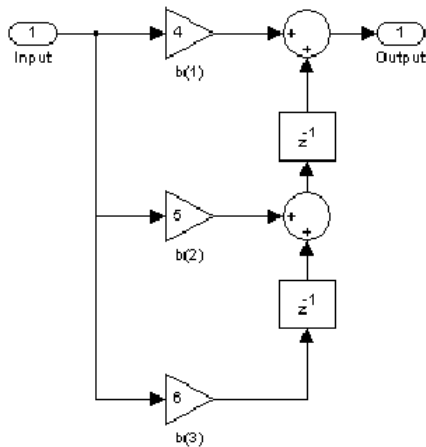
- Direct form I, second-order sections
- Direct form I transposed
- Direct form I transposed, second-order sections
- Direct form II
- Direct form II, second-order sections
- Direct form II transposed
- Direct form II transposed, second-order sections
- Direct form FIR
- Direct form FIR transposed
- Direct form symmetric FIR
- Direct form antisymmetric FIR
- Lattice all-pass
- Lattice AR
- Lattice ARMA
- Lattice MA for maximum phase
- Lattice MA for minimum phase
- Cascade
- Parallel

### Specify the Filter Implementation

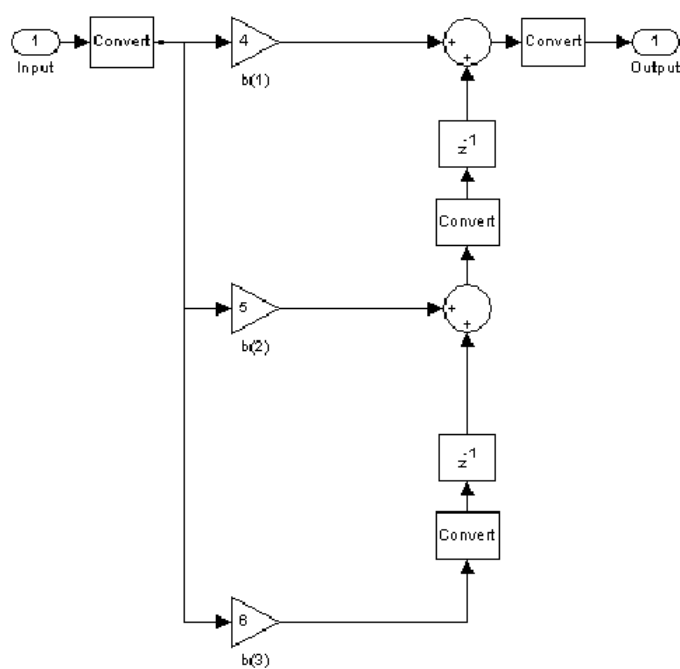
You can determine how the Filter Realization Wizard models the specified filter. In the **Realize Model** panel, select the **Build model using basic elements** check box. When you select this check box and click on the **Realize Model** button, the Filter Realization Wizard creates a subsystem block that implements your filter using Sum, Gain, and Delay blocks. When you clear this check box, the Filter Realization Wizard uses a digital filter block to implement your filter. The **Build model using basic elements** check box is available only when your filter can be implemented using a digital filter block available in the DSP System Toolbox library.

The Filter Realization Wizard can generate a subsystem that represents either a double-precision or fixed-point filter. You must install the Fixed-Point Designer product to simulate a fixed-point filter. You can still edit the blocks used to implement the filter without installing the Fixed-Point Designer product.

Double-precision filter implemented with Sum, Gain, and Delay blocks



Fixed-point filter implemented with Sum, Gain, Delay, and Conversion blocks



## Implementations of Double-Precision and Fixed-Point Filters

### Command Line Alternative to Realize Model Button

You can enter `realizemdl(sysobj)` in the MATLAB command prompt to generate an architectural model of the filter System object, `sysobj`, in a Simulink subsystem block using individual sum, gain, and delay blocks, according to user-defined specifications. For more information, see `realizemdl`.

## Parameters

---

**Note** The following parameters for the Filter Realization Wizard are in the **Realize Model** pane of the filter designer app. To open different panels of filter designer, click the different buttons at the lower-left corner. For more information about relevant panels, see “Specify the Filter and Data Types” on page 2-576.

---

### Block Name

Enter the name of the new filter block.

### Destination

Specify where the new filter block should be created. This can be in a new model or in the current (most recently selected) model.

### User Defined

Specify the name of the target subsystem in which the Filter Realization Wizard should create the new filter block.

**Overwrite generated block “Filter” block**

When selected, the block overwrites any filter block in the current model with the name specified in the **Block Name** parameter. This parameter is enabled when the **Destination** parameter is set to Current.

**Build model using basic elements**

Select this check box to implement your filter using Sum, Gain, and Delay blocks. Clear this check box to implement your filter using digital filter blocks from the DSP System Toolbox library. This parameter is available only when your filter can be modeled using an available digital filter block.

**Optimize for zero gains**

Select this check box to remove zero-gain paths from the filter structure. For an example, see “Optimize the Filter Structure”.

**Optimize for unity gains**

Select this check box to substitute gains equal to 1 with a wire (short circuit). For an example, see “Optimize the Filter Structure”.

**Optimize for negative gains**

Select this check box to substitute gains equal to -1 with a wire (short circuit), and change the corresponding sums to subtractions. For an example, see “Optimize the Filter Structure”.

**Optimize delay chains**

Select this check box to substitute any delay chains made up of  $n$  unit delays with a single delay by  $n$ . For an example, see “Optimize the Filter Structure”.

**Optimize for unity scale values**

Select this check box to remove all scale value multiplications by 1 from the filter structure.

**Input processing**

Specify how the generated filter block or subsystem block processes the input. Depending on the type of filter you are designing, one or both of the following options may be available:

- **Columns as channels (frame based)** — When you select this option, the block treats each column of the input as a separate channel.
- **Elements as channels (sample based)** — When you select this option, the block treats each element of the input as a separate channel.

For more information about sample- and frame-based processing, see “Sample- and Frame-Based Concepts”.

**Rate options**

For multirate filters, specify how the block should process the input. You can set this parameter to one of the following options:

- **Enforce single-rate processing** — When you select this option, the block maintains the input sample rate.
- **Allow multirate processing** — When you select this option, the block adjusts the rate at the output to accommodate an increased or reduced number of samples.

**Realize Model**

Click to create a filter block according to the settings you’ve specified. When the **Build model using basic elements** check box is selected, the filter is implemented as a subsystem block consisting of Sum, Gain, and Delay blocks. To see the filter implementation, double-click the subsystem block in your model.

---

**Note** For more information about relevant parameters in other panels of filter designer, see “Specify the Filter and Data Types” on page 2-576.

---

## Supported Data Types

- Double-precision floating point
- Single-precision floating point — Supported only when you install Fixed-Point Designer.
- Fixed point (signed and unsigned) — Supported only when you install Fixed-Point Designer and Fixed-Point Designer.

## Version History

Introduced before R2006a

## References

- [1] Oppenheim, A. V. and R. W. Schaffer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989.
- [2] Proakis, J. and D. Manolakis. *Digital Signal Processing*. 3rd ed. Englewood Cliffs, NJ: Prentice-Hall, 1996.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

## See Also

### Functions

`filter` | `realizemdl`

### Blocks

Discrete FIR Filter | Biquad Filter | Digital Filter Design

### Topics

“Filter Design”

“Filter Analysis”

“Select a Filter Design Block”

# Filter Visualizer

Display magnitude and phase response of time-varying digital filters

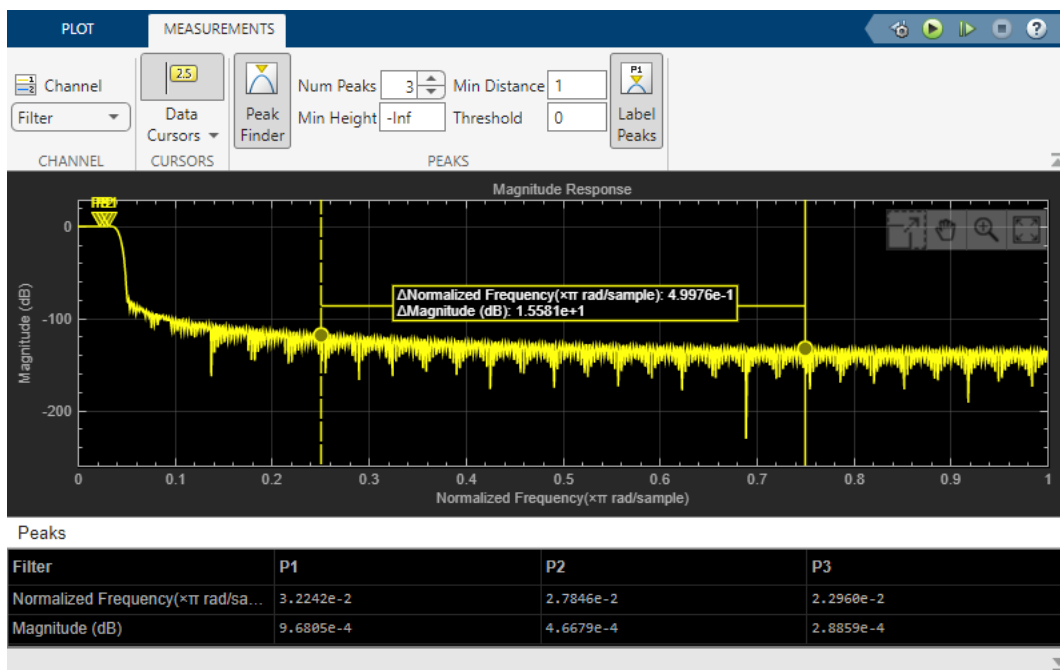


**Libraries:**  
DSP System Toolbox / Sinks

## Description

The Filter Visualizer block displays the magnitude response and phase response of time-varying digital filters or time-varying filter coefficients. You can visualize the frequency response of up to 20 filters at a time using this visualizer.

You can use the visualizer to configure the plot settings, find the peak values, enable cursor measurements, and even copy the scope display to the clipboard. For more information, see “Configure Filter Visualizer”. To access the filter visualizer programmatically, use the `dsp.DynamicFilterVisualizer` object.



## Examples

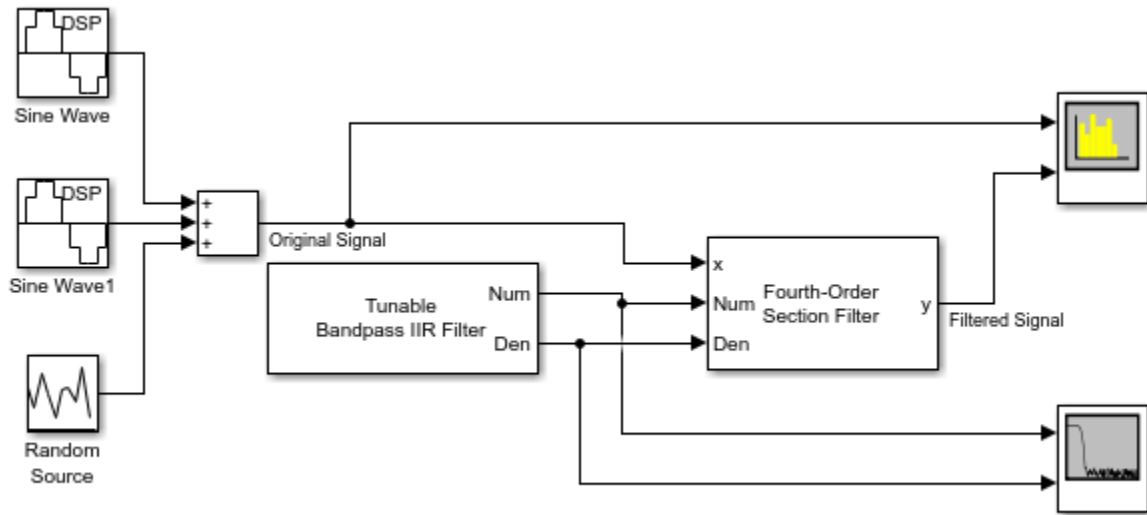
### Filter Noisy Signal Using Fourth-Order Section (FOS) Filter in Simulink

Filter a noisy sinusoidal signal using the Fourth-Order Section Filter block. The numerator and denominator coefficients of the fourth-order section filter are obtained using the Tunable Bandpass IIR Filter block. You can tune the frequency specifications of this filter during simulation.

### Open and Run the Model

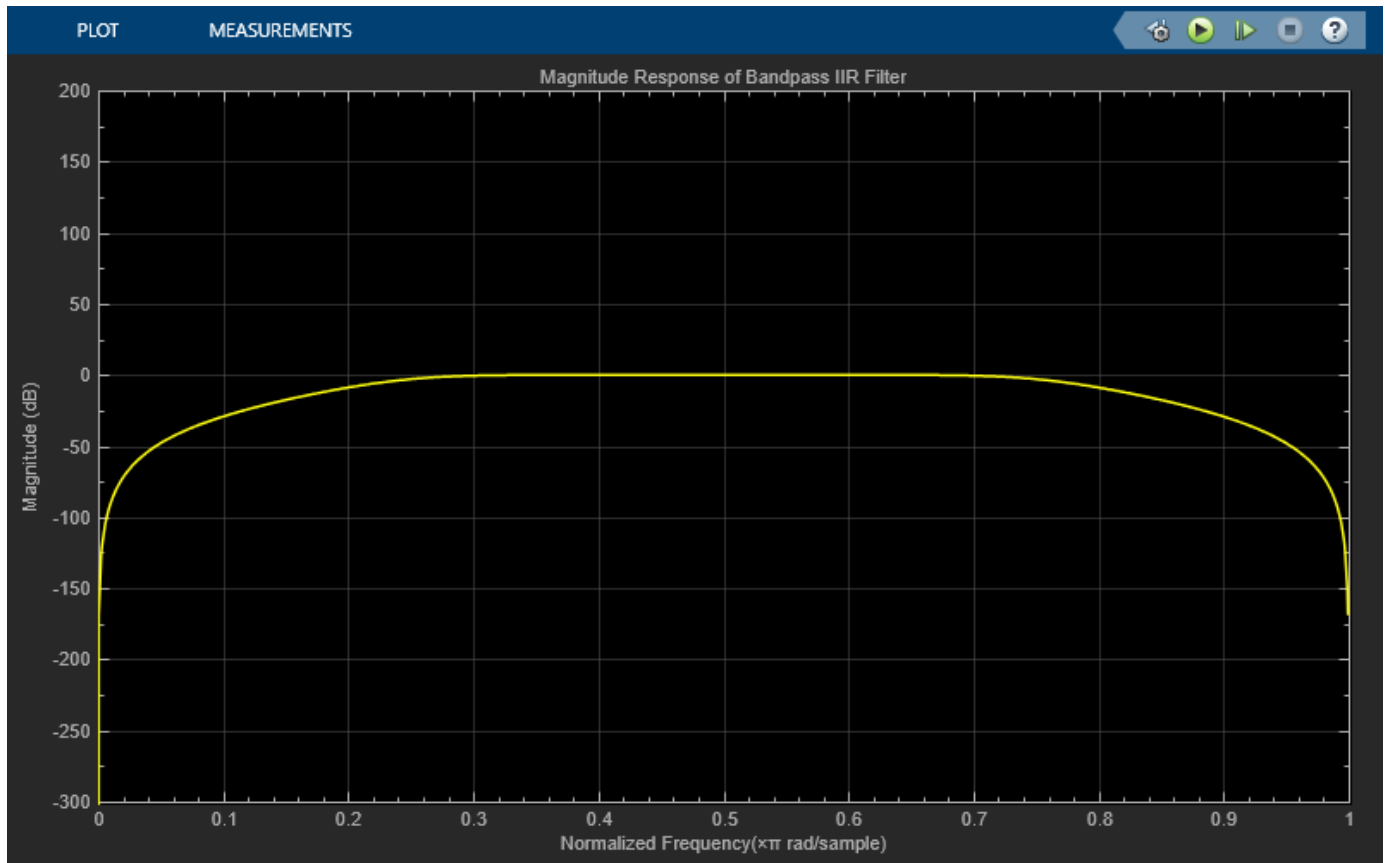
The input signal is a sum of two sine waves with the frequencies of 100 Hz and 350 Hz. The sample rate is 1000 Hz and the number of samples in each frame is 1024. Add zero-mean white Gaussian noise with a variance of  $1e-4$  to the sum of the sine waves.

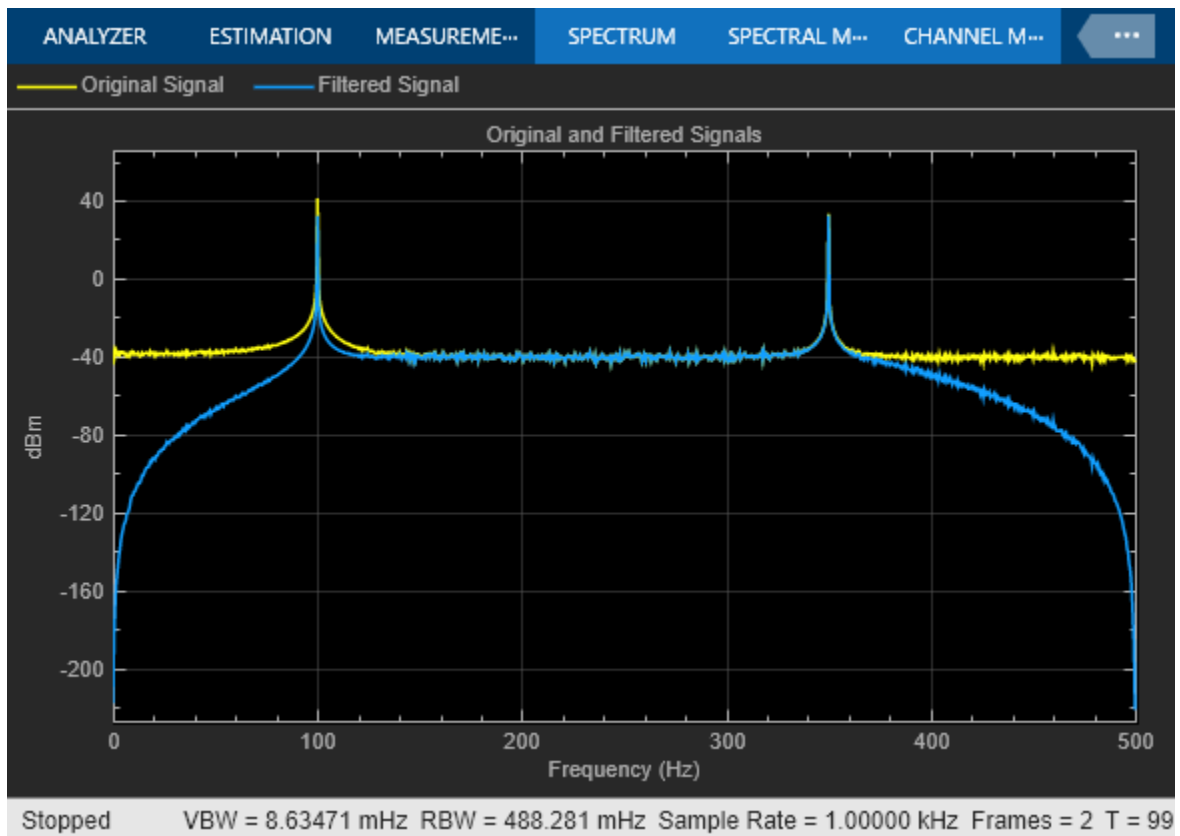
The Tunable Bandpass IIR Filter block designs a sixth order bandpass IIR filter with the first and second 3-dB cutoff frequencies at  $0.25 \pi$  rad/sample and  $0.75 \pi$  rad/sample, respectively. Visualize the frequency response of the filter using Filter Visualizer.



Copyright 2021-2023 The MathWorks, Inc.

Filter the noisy sinusoidal signal using the Fourth-Order Section Filter block. Visualize the original sinusoidal signal and the filtered signal using the Spectrum Analyzer.



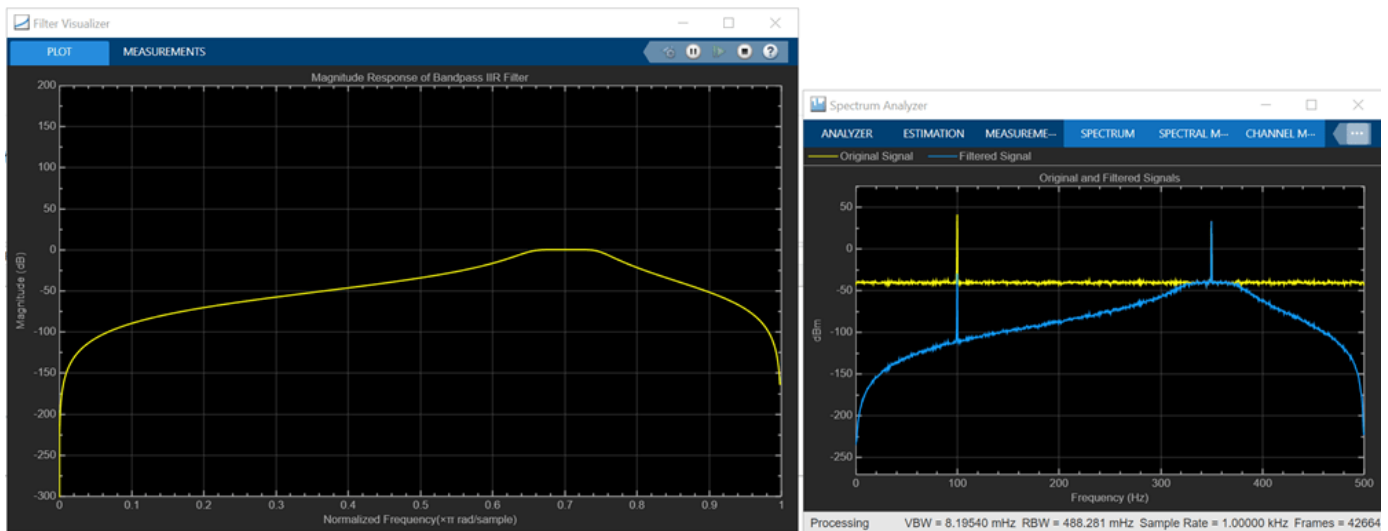


### Tuning Frequency Specification of FOS Filter

During simulation, you can tune the frequency specifications of the FOS filter by tuning the frequency parameters in the Tunable Bandpass IIR Filter block. The filter response updates accordingly.

Change the first 3-dB cutoff frequency to  $0.65 \pi$  rad/sample in the Tunable Bandpass IIR Filter block. The first tone of the sinusoidal signal is attenuated as it no longer falls in the passband frequency of the filter.





## Ports

### Input

**Num** — Numerator coefficients

column vector | row vector |  $L$ -by-3 matrix |  $L$ -by-5 matrix

Specify the numerator coefficients of the filter as a row vector, column vector,  $L$ -by-3 matrix corresponding to a cascade of second-order sections, or an  $L$ -by-5 matrix corresponding to a cascade of fourth-order sections, where  $L$  is the number of filter sections.

If you set **Num Filters** to a value greater than 1, then the block displays a **Num** port for each filter.

When you set **Filter Type** to FIR, inputs on all **Num** ports must be column or row vectors.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | Boolean | fixed point | enumerated

**Den** — Denominator coefficients

column vector | row vector |  $L$ -by-3 matrix |  $L$ -by-5 matrix

Specify the denominator coefficients of the filter as a row vector, column vector,  $L$ -by-3 matrix corresponding to a cascade of second-order sections, or an  $L$ -by-5 matrix corresponding to a cascade of fourth-order sections, where  $L$  is the number of filter sections.

If you set **Num Filters** to a value greater than 1, then the block displays a **Den** port for each filter.

### Dependency

To enable this port, set **Filter Type** to IIR.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | Boolean | fixed point | enumerated

## Parameters

### Plot Tab

#### Configuration

**Num Filters** — Number of filters to display  
1 (default) | positive integer in the range [1 20]

Specify the number of filters to display as a positive integer in the range [1 20]. The value of this parameter determines the number of input ports on the block.

#### Programmatic Use

**Block Parameter:** NumFilters  
**Type:** character vector or string scalar  
**Values:** scalar in the range [1 20]

**Filter Type** — Type of filter response  
FIR (default) | IIR

Specify the filter response type as one of these options:

- FIR -- The block displays one input port per filter. You input the numerator coefficients of the filter(s) through the port(s).
- IIR -- The block displays two input ports per filter. You input the numerator and the denominator coefficients of the filter(s) through the ports.


#### Programmatic Use

**Block Parameter:** FilterType  
**Type:** character vector or string scalar  
**Values:** 'FIR', 'IIR'

**Legend** — Display signal legend  
off (default) | on

Select this option to display the legend on the plot. The legend contains the filter names in the format `Filter:Number`. If you are visualizing only one filter, the legend displays `Filter`. If you are visualizing the magnitude and phase responses of multiple filters, the visualizer displays a separate legend for each response in the format `mag(Filter:Number)` and `phase(Filter:Number)`, respectively. If you are visualizing the responses of only one filter, the legend displays `mag(Filter)` and `phase(Filter)`.

You can control which filters are visible in the scope using the legend. Click a filter name in the legend to hide the frequency response of that filter in the scope. To display the filter response, click the filter name again. To display only one filter response, right-click the name of that filter, which hides all other filter responses. To show all filter responses, press **Esc**. This setting is equivalent to

changing the **Visible** parameter in the **Plot** tab > **Settings** (  ) > **Color and Styling**.

**Tunable:** Yes

**Programmatic Use****Block Parameter:** ShowLegend**Type:**boolean**Magnitude and Phase** — Plot magnitude and phase response of filter  
button

Click the button to plot the magnitude and phase response of the filter on two separate axes. Click the button again to plot only the magnitude response.

**Tunable:** Yes**Programmatic Use****Block Parameter:** PlotAsMagnitudePhase**Type:**boolean**Configuration > Filter Visualizer Settings ( )****FFT Length** — FFT length

2048 (default) | positive integer

Specify the FFT length that the filter visualizer uses to compute the spectral estimates as a positive integer.

**Tunable:** Yes**Programmatic Use****Block Parameter:** FFTLength**Type:** character vector or string scalar**Values:** scalar**Normalized Frequency** — Display the frequency response in normalized units

on (default) | off

Select this parameter to display the frequency response of the filter in normalized frequency units.

When you select this parameter, the frequency range is [0,1] and the frequency axis (x-axis) units switch to Normalized Frequency ( $\times\pi$  rad/sample).

When you clear this parameter, the frequency range is [0  $F_s/2$ ] and the frequency axis (x-axis) units are in Hz.

**Tunable:** Yes**Programmatic Use****Block Parameter:** NormalizedFrequency**Type:**boolean**Sample Rate (Hz)** — Sample rate in Hz

44100 (default) | positive scalar

Enter the sample rate as a positive scalar in Hz.

**Tunable:** Yes

**Dependencies**

To enable this parameter, clear the **Normalized Frequency** parameter.

**Programmatic Use**

**Block Parameter:** SampleRate

**Type:** character vector or string scalar

**Values:** scalar

**Range** — Frequency range

[0, 1] (default) | two-element numeric vector

Specify the frequency range as a two-element numeric vector that is monotonically increasing and of the form [*fmin*, *fmax*].

When you select the **Normalized Frequency** parameter, *fmax* is in normalized frequency units and must be a positive scalar that is less than or equal to 1. When you clear the **Normalized Frequency** parameter, *fmax* is in Hz and must be less than or equal to  $F_s/2$ , where  $F_s$  is the value you specify in the **Sample Rate (Hz)** parameter.

**Tunable:** Yes

**Programmatic Use**

**Block Parameter:** FrequencyRange

**Type:** character vector or string

**Value:** two-element numeric vector

**Display Unit** — Magnitude units

Magnitude (dB) (default) | Magnitude | Magnitude squared

Specify the magnitude units in the display as one of these:

- Magnitude
- Magnitude (dB)
- Magnitude squared

**Tunable:** Yes

**Programmatic Use**

**Block Parameter:** MagnitudeDisplay

**Type:** character vector or string scalar

**Values:** 'Magnitude (dB)', 'Magnitude', 'Magnitude squared'

**Plot Type** — Type of plot

Line (default) | Stairs | Stem

Specify the type of plot to use in the filter visualizer window as one of these:

- **Line** -- The filter visualizer connects each point on the magnitude and phase response plot with a line.
- **Stairs** -- The filter visualizer displays the filter response (magnitude, phase, or both) as a stair-step graph. A stair-step graph is made up of only horizontal lines and vertical lines. Each

horizontal line represents the filter response over a frequency value and is connected to two vertical lines. Each vertical line represents a change in values occurring at a frequency.

- **Stem** -- The filter visualizer displays the frequency response as circles with vertical lines extending down to the x-axis at each of the frequency values.

**Tunable:** Yes

**Programmatic Use**

**Block Parameter:** PlotType

**Type:** character vector or string scalar

**Values:** 'Stem', 'Line', 'Stairs'

**Font Size** — Font size of labels

Small (default) | Medium | Large | Extra Large

Specify the font size of all labels in the display as Small, Medium, Large, and Extra Large.

**Tunable:** Yes

**Programmatic Use**

**Block Parameter:** -

**Type:** character vector or string scalar

**Title** — Display title

character vector | string

Specify the display title as a character vector or a string.

**Tunable:** Yes

**Programmatic Use**

**Block Parameter:** Title

**Type:** character vector or string

**X-Scale** — Scale of frequency axis

Linear (default) | Log

Specify the scale of the frequency axis as 'Linear' or 'Log'.

**Tunable:** Yes

**Programmatic Use**

**Block Parameter:** XScale

**Type:** character vector or string

**Values:** 'Linear', 'Log'

**Y-Limits** — y-axis limits

[-25 25] | [ymin ymax]

Specify the y-axis limits of the magnitude display in the Filter Visualizer as a two-element numeric vector of the form [ymin ymax]. The units of the y-axis limits depend on the **Display Unit** parameter.

**Tunable:** Yes

**Programmatic Use****Block Parameter:** YLimits**Type:** character vector or string**Value:** two-element numeric vector**Show Grid** — Show internal grid lines

on (default) | off

Select this check box to show the grid in the Filter Visualizer display.

**Tunable:** Yes**Programmatic Use****Block Parameter:** ShowGrid**Type:** logical**Preserve colors for copy to clipboard** — Preserve colors in clipboard copy

off (default) | on

When you select this parameter, the Filter Visualizer preserves the colors in the display that you copy to the clipboard using the **Copy Display** and the **Print** options in the **Share** section on the **Plot** tab.

**Tunable:** Yes**Background** — Window background

black (default) | color picker

Specify the color that you want to apply to the background of the scope figure.

**Tunable:** Yes**Axes** — Axes background color

black (default) | color picker

Specify the color that you want to apply to the background of the axes.

**Tunable:** Yes**Labels** — Color of labels

gray (default) | color picker

Specify the color of the labels, grid, and the channel names in the legend.

**Tunable:** Yes**Line** — Filter for line property settings

filter names

Select the filter for which you want to modify the visibility, line color, style, width, and marker properties.

**Tunable:** Yes**Visible** — Filter visibility

on (default) | off

Specify whether the selected filter is visible. If you clear this check box, the frequency response line disappears in the display.

You can control this parameter directly from the legend. Click a filter name in the legend to hide the frequency response of the filter in the scope. To show the filter response, click the filter name again. To show only one filter response, right-click the filter name, which hides all other filter responses. To show all filter responses, press **Esc**.

**Tunable:** Yes

#### **Programmatic Use**

**Block Parameter:** Visible

**Type:** logical

**Style** — Line style

- (default) | : | -. | -- | None

Specify the line style for the selected filter response.

**Tunable:** Yes

**Width** — Line width

1.5 (default) | 0.5 | 1 | 2 | ...

Specify the line width for the selected filter response.

**Tunable:** Yes

**Marker** — Data point markers

None (default) | + | x | ...

Specify markers for the selected filter response to show at its data points. This parameter is similar to the 'Marker' property for plots. You can choose any of the marker symbols from the drop-down list.

**Tunable:** Yes

**Color** — Line color

yellow (default) | color picker

Specify the line color for the selected filter response.

**Tunable:** Yes

#### **Share**

**Copy Display** — Send display to clipboard

button

Click this button to copy the scope display to the clipboard. You can preserve the color of the display by selecting the **Preserve colors for copy to clipboard** parameter.

**Print** — Print display

button

Click this button to save the scope display as an image or a PDF or to print the display.

## Measurements Tab

### Channel

**Channel** — Filter for which to obtain measurements

`Filter (default) | filter name`

Choose the filter response for which you need to obtain measurements. The filter names are in the format `Filter:Number`. If you are visualizing only one filter, the legend displays `Filter`. If you are visualizing the magnitude and phase responses of multiple filters, the visualizer displays a separate name for each response in the format `mag(Filter:Number)` and `phase(Filter:Number)`, respectively. If you are visualizing the responses of only one filter, the legend becomes `mag(Filter)` and `phase(Filter)`.

**Tunable:** Yes

### Dependency

To enable this parameter, pass some data through the scope.

### Cursors

**Data Cursors** — Enable cursor measurements

`button`

Click the **Data Cursors** button to enable data cursor measurements. Each cursor tracks a vertical line along the frequency response. The scope displays the difference between frequency( $x$ ) values and frequency response ( $y$ ) values of the filter at the two cursors in the box between the cursors.

**Tunable:** Yes

**Snap to data** — Snap cursors to data

`off (default) | on`

Select this parameter to position the cursors on the signal data points.

**Tunable:** Yes

**Lock cursor spacing** — Lock cursor spacing

`off (default) | on`

Select this parameter to lock the frequency difference between the two cursors.

**Tunable:** Yes

### Peaks

**Peak Finder** — Enable peak finder measurements

`button`

Select **Peak Finder** to enable peak finder measurements. An arrow appears on the plot at each maxima and a Peaks panel appears at the bottom of the scope window.

**Tunable:** Yes

**Num Peaks** — Maximum number of peaks to show

`3 (default) | positive integer less than 100`



Specify the maximum number of peaks to show as a positive integer less than 100.

**Tunable:** Yes

**Min Height** — Level above which scope detects peaks

-Inf (default) | real scalar value

Specify the level above which the scope detects peaks as a real scalar.

**Tunable:** Yes

**Min Distance** — Minimum number of samples between adjacent peaks

1 (default) | positive integer

Specify the minimum number of samples between adjacent peaks as a positive integer.

**Tunable:** Yes

**Threshold** — Minimum difference in height of peak and its neighboring samples

0 (default) | nonnegative scalar

Specify the minimum difference in the height of the peak and its neighboring samples as a nonnegative scalar.

**Tunable:** Yes

**Label Peaks** — Label peaks

button

Select **Label Peaks** to label the peaks. The scope displays the labels (**P1**, **P2**, ...) above the arrows in the plot.

**Tunable:** Yes

### Property Inspector Only

**Filter Names** — Input filter names

[] (default) | character vector | string | array of strings or character vectors.

Input filter names, specified as a character vector, string, or array of either. The names appear in the legend, **Filter Visualizer Settings**, and **Measurements** panels. If you do not specify filter names, the block displays the filter names as Filter: 1, Filter: 2, etc.

### Dependency

To see filter names, select **Legend** in the **Plot** tab.

### Programmatic Use

**Block Parameter:** FilterNames


**Type:** cell array of character vectors or string array

**Maximize Axes** — Maximize size of plots

Auto (default) | Off | On

- Auto — If you do not specify **Title** and y-label, maximize all plots by default.

- On — Maximize all plots and hide **Title** and y-label.
- Off — Do not maximize plots.

Point anywhere in the Filter Visualizer to see the maximize axes button .

**Tunable:** Yes

#### Programmatic Use

**Block Parameter:** MaximizeAxes

**Type:** character vector or string scalar

**Axes Scaling** — Y-axis scaling mode

Auto (default) | Manual | Updates | OnceAtStop

- Auto — The filter visualizer scales the axes as needed to fit the data, both during and after simulation.
- Manual — The filter visualizer does not scale the axes automatically.
- Updates — The filter visualizer scales the axes limits once after completing the number of visual updates specified in the **Number of Updates** text box (100 by default). Scaling occurs only once during each run.
- OnceAtStop — The filter visualizer scales the axes when the simulation stops.

**Tunable:** Yes

#### Programmatic Use

**Block Parameter:** AxesScaling

**Type:** character vector or string scalar

**Number of Updates** — Number of updates before scaling

100 (default) | integer

Set this property to delay auto scaling the y-axis.

**Tunable:** Yes

#### Dependency

To enable this property, set **Axes Scaling** to Updates.

#### Programmatic Use

**Block Parameter:** AxesScalingNumUpdates

**Type:** character vector or string scalar

**Values:** scalar

## Block Characteristics

|                           |  |
|---------------------------|--|
| <b>Data Types</b>         | Boolean   double   enumerated   fixed point   integer   single |
| <b>Direct Feedthrough</b> | no   |

|                                 |    |
|---------------------------------|----|
| <b>Multidimensional Signals</b> | no |
| <b>Variable-Size Signals</b>    | no |
| <b>Zero-Crossing Detection</b>  | no |

## More About

### Cursor Measurements

Measure signal values using vertical waveform cursors that track along the signal.

When you click the **Data Cursors** button in the **Measurements** tab of the Filter Visualizer, the display shows vertical cursors on the filter response curve that you choose in the **Channel** section. Each cursor tracks a vertical line along the curve. The scope displays the difference between  $x$ - and  $y$ -values of the curve at the two cursors in the box between the cursors.

To enable cursor measurements, you must select the **Data Cursors** button in the **Measurements** tab. The cursors appear only when the Filter Visualizer has at least one frequency response in its display.

You can use the mouse to move the vertical cursors to the left or to the right.

On the **Measurements** tab, click the **Data Cursors** drop-down list and you can select:

- **Snap to Data** — To position the cursors on frequency response data points.
- **Lock Cursor Spacing** — To lock the frequency difference between the two cursors.

For modifying the cursor measurements programmatically, see the `dsp.DynamicFilterVisualizer` object. For more information on cursor measurements, see the `CursorMeasurementsConfiguration` object.

### Peak Finder Measurements

Compute and display peak values on the scope display.

When you click on the **Peak Finder** button in the **Measurements** tab of the Filter Visualizer, an arrow appears on the plot at each maxima and a **Peaks** panel appears at the bottom of the scope window. The Filter Visualizer computes peaks from the portion of the filter response that is currently on display in the scope, and the **Peaks** panel shows the peak values and the frequencies at which these peaks occur.

The **Peaks** section in the **Measurements** tab allows you to specify the number of peaks you want the scope to display, the minimum height above which you want the scope to detect peaks, the minimum distance between peaks, and label the peaks.

The Filter Visualizer algorithm defines a peak as a local maximum with lower values present on either side of the peak. It does not consider end points as peaks. For more information on the algorithm, see the `findpeaks` function.

For modifying the peak finder measurements programmatically, use the `dsp.DynamicFilterVisualizer` object. For more information on peak finder measurements, see

the `PeakFinderConfiguration` object. For more information on these settings in the UI, see “Peaks” on page 2-0 .

## **Version History**

**Introduced in R2023a**

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

This block can be used for simulation visibility in systems that generate code, but is not included in the generated code.

## **See Also**

### **Objects**

`dsp.DynamicFilterVisualizer`

### **Blocks**

Array Plot

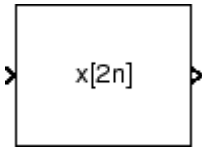
### **Topics**

“Configure Filter Visualizer”

“Configure Array Plot”

# FIR Decimation

Perform polyphase FIR decimation



### Libraries:

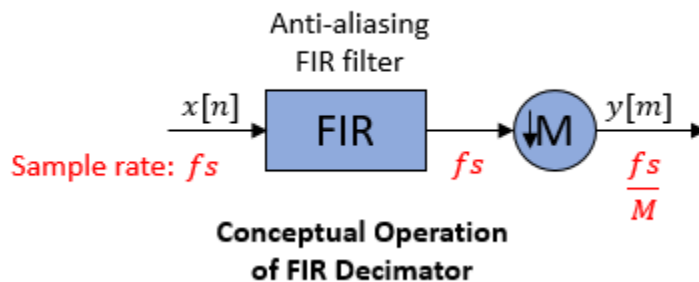
DSP System Toolbox / Filtering / Multirate Filters  
 DSP System Toolbox HDL Support / Filtering

## Description

The FIR Decimation block performs an efficient polyphase decimation using an integer downsampling factor  $M$  along the first dimension.

Conceptually, the FIR decimator (as shown in the schematic) consists of an anti-aliasing FIR filter followed by a downsampler. To design an FIR anti-aliasing filter, use the `designMultirateFIR` function.

The FIR filter filters the data in each channel of the input using a direct-form FIR filter. The downsampler that follows downsamples each channel of filtered data by taking every  $M$ -th sample and discarding the  $M - 1$  samples that follow.  $M$  is the value of the decimation factor that you specify. The resulting discrete-time signal has a sample rate that is  $1/M$  times the original sample rate.



Note that the actual block algorithm implements a direct-form FIR polyphase structure, an efficient equivalent of the combined system depicted in the diagram. For more details, see “Algorithms” on page 2-609.

Under specific conditions, this block also supports SIMD code generation. For details, see “Code Generation” on page 2-612.

## Ports

### Input

**In** — Data input  
 vector | matrix

Specify the data input as a vector or a matrix of size  $P$ -by- $Q$ . The columns in the input signal represent  $Q$  independent channels.

The block supports variable-size input signals (frame length changes during simulation) when you set **Input processing** to `Columns as channels (frame based)` and **Rate options** to `Enforce single-rate processing`. When the block accepts variable-size input signals, they can be of arbitrary frame length, that is, the input frame length does not have to be a multiple of the decimation factor. When you specify fixed-size signals, the frame length can be arbitrary under certain conditions. For more details, see “Frame-Based Processing” on page 2-607 and “Sample-Based Processing” on page 2-608.

This port is unnamed until you set **Coefficient source** to `Input port`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

**Num** — Numerator coefficients  
vector

Specify the numerator coefficients of the FIR filter as a vector.

The transfer function  $H(z)$  of the FIR filter is given by:

You can generate the FIR filter coefficient vector,  $\mathbf{b} = [b_0, b_1, \dots, b_N]$ , using one of the DSP System Toolbox filter design functions such as `designMultirateFIR`, `firnyquist`, `firhalfband`, `firgr`, or `firceqrip`.

To act as an effective anti-aliasing filter, the coefficients usually correspond to a lowpass filter with a normalized cutoff frequency no greater than  $1/M$ , where  $M$  is the decimation factor. To design such a filter, use the `designMultirateFIR` function.

Coefficient values obtained through **Num** are tunable, that is, they can change during simulation, while their properties must remain constant.

The data type of the **Num** input must match the data type of the **In** input.

### Dependencies

The **Num** input port appears when you set **Coefficient source** as `Input port`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

### Output

**Out** — Decimator output  
vector | matrix

Output of the FIR Decimator block, returned as a vector or a matrix.

When you set **Rate options** to:

- `Enforce single-rate processing` -- The block maintains the input sample rate and decimates the signal by decreasing the output frame size by a factor of  $M$ .

The output has an upper bound size of  $\text{ceil}(P/M)$ -by- $Q$  for an input of size  $P$ -by- $Q$ .

- `Allow multirate processing` -- The block decimates the signal such that the output sample rate is  $M$  times slower than the input sample rate.

The output frame size is the same as the input frame size.

For more details, see “Frame-Based Processing” on page 2-607 and “Sample-Based Processing” on page 2-608.

This port is unnamed until you set **Coefficient source** to **Input port**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

## Parameters

### Coefficient source

**Coefficient source** — FIR filter coefficient source

**Auto** (default) | **Dialog parameters** | **Input port** | **Filter object**

Specify the FIR filter coefficient source as one of the following:

- **Dialog parameters** -- Specify the filter coefficients through the **FIR filter coefficients** parameter in the block dialog box.
- **Input port** -- Specify the filter coefficients through the **Num** input port.
- **Filter object** -- Specify the filter using a `dsp.FIRDecimator` System object.
- **Auto** -- When you select **Auto**, the block designs an FIR decimator using the decimation factor that you specify in **Decimation factor**. The `designMultirateFIR` function designs the filter and returns the coefficients used by the block.

For more information on the filter design, see Orfanidis [2].

### Main Tab

**FIR filter coefficients** — Lowpass FIR filter coefficients

`designMultirateFIR(1,2)` (default) | `vector`

Specify the lowpass FIR filter coefficients, in descending powers of  $z$ , as a vector. By default, `designMultirateFIR(1,2)` computes the filter coefficients.

The transfer function  $H(z)$  of the FIR filter is given by:

You can generate the FIR filter coefficient vector,  $\mathbf{b} = [b_0, b_1, \dots, b_N]$ , using one of the DSP System Toolbox filter design functions such as `designMultirateFIR`, `firnyquist`, `firhalfband`, `firgr`, or `firceqrip`.

To act as an effective anti-aliasing filter, the coefficients usually correspond to a lowpass filter with a normalized cutoff frequency no greater than  $1/M$ , where  $M$  is the decimation factor. To design such a filter, use the `designMultirateFIR` function.

The block internally initializes all filter states to zero.

### Dependencies

This parameter appears only when you set the **Coefficient source** to **Dialog parameters**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**Decimation factor** — Decimation factor  
2 (default) | positive scalar

Specify the integer factor  $M$ . The block decreases the sample rate of the input sequence by this factor.

**Dependencies**

This parameter appears only when you set the **Coefficient source** to **Dialog parameters, Input port**, or **Auto**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**Filter structure** — FIR filter structure  
`Direct form` (default) | `Direct form transposed`

Specify the FIR filter structure as either `Direct form` or `Direct form transposed`.

**Dependencies**

This parameter appears only when you set the **Coefficient source** to **Dialog parameters, Input port**, or **Auto**.

**Filter object** — Filter object  
`dsp.FIRDecimator`

Specify the name of the multirate filter object that you want the block to implement. You must specify the filter as a `dsp.FIRDecimator` System object.

You can define the System object directly in the block dialog box. Alternatively, you can define the object in a MATLAB workspace variable and specify the variable in the block dialog box.

For information on creating System objects, see “Define Basic System Objects”.

**Dependencies**

This parameter appears only when you set the **Coefficient source** to **Filter object**.

**Input processing** — Method to process input signals  
`Columns as channels (frame based)` (default) | `Elements as channels (sample based)`

Specify how the block should process the input. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

**Rate options** — Method by which block decimates input  
`Enforce single-rate processing` (default) | `Allow multirate processing`



Specify the method by which the block should decimate the input. You can select one of the following options:

- **Enforce single-rate processing** — When you select this option, the block maintains the input sample rate and decimates the signal by decreasing the output frame size by a factor of  $M$ . To select this option, you must set the **Input processing** parameter to `Columns as channels (frame based)`.

When you set the **Rate options** parameter to `Enforce single-rate processing`, you can use the FIR Decimation block inside triggered subsystems.

- **Allow multirate processing** — When you select this option, the block decimates the signal such that the output sample rate is  $M$  times slower than the input sample rate.

**Allow arbitrary frame length for fixed-size input signals** — Allow arbitrary frame length for fixed-size input signals

off (default) | on

Specify whether fixed-size input signals (whose size does not change during simulation) can have an arbitrary frame length, where the frame length does not have to be a multiple of the decimation factor. The block uses this parameter setting only for fixed-size input signals and ignores this parameter if the input has a variable-size.

When the input signal is a variable-size signal, the signal can have arbitrary frame length, that is, the frame length does not have to be a multiple of the decimation factor.

For fixed-size input signals, if you:

- Select the **Allow arbitrary frame length for fixed-size input signals** parameter, the frame length of the signal does not have to be a multiple of the decimation factor. If the input is not a multiple of the decimation factor, then the output is generally a variable-size signal. Therefore, to support arbitrary input size, the block must also support variable-size operations, which you can enable by selecting the **Allow arbitrary frame length for fixed-size input signals** parameter.
- Clear the **Allow arbitrary frame length for fixed-size input signals** parameter, the input frame length must be a multiple of the decimation factor.

### Dependency

To enable this parameter, set **Input processing** to `Columns as channels (frame based)` and **Rate options** to `Enforce single-rate processing`.

**Output buffer initial conditions** — Initial conditions

0 (default) | scalar | matrix

When you set the FIR Decimation block to the frame-based processing mode, the block can exhibit one-frame latency. In the case of one-frame latency, this parameter specifies the output of the block until the first filtered input sample is available. Specify this parameter as a scalar value to be applied to all signal channels, or as a matrix containing one value for each channel.

Cases of one-frame latency can occur when the input frame size is greater than one, and you set the **Input processing** and **Rate options** parameters of the FIR Decimation block as follows:

- **Input processing** set to `Columns as channels (frame based)`

- **Rate options** set to Allow multirate processing

For more information on latency in the FIR Decimation block, see “Latency” on page 2-608.

#### Dependencies

This parameter appears only when you configure the block to perform multirate processing by setting **Rate options** to Allow multirate processing.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**View Filter Response** — View filter response  
button

Click on this button to open the Filter Visualization Tool (fvtool) and display the filter response of the filter defined in the block dialog box.

#### Data Types Tab

**Rounding mode** — Rounding mode

Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Select the rounding mode for fixed-point operations. The default is Floor. The filter coefficients do not obey this parameter and always round to Nearest.

---

**Note** The **Rounding mode** and **Saturate on integer overflow** settings have no effect on numerical results when all the following conditions exist:

- **Product output** is Inherit: Inherit via internal rule
- **Accumulator** is Inherit: Inherit via internal rule
- **Output** is Inherit: Same as accumulator

With these data type settings, the block is effectively operating in the full-precision mode.

---

**Saturate on integer overflow** — Saturate on integer overflow

off (default) | on

When you select this parameter, the block saturates the result of its fixed-point operation. When you clear this parameter, the block wraps the result of its fixed-point operation. For details on saturate and wrap, see overflow mode for fixed-point operations.

---

**Note** The **Rounding mode** and **Saturate on integer overflow** parameters have no effect on numeric results when all these conditions are met:

- **Product output** data type is Inherit: Inherit via internal rule.
- **Accumulator** data type is Inherit: Inherit via internal rule.

With these data type settings, the block operates in full-precision mode.

---


**Coefficients Data Type** — Coefficients data type

Inherit: Same word length as input (default) | `fixdt(1,16)` | `fixdt(1,16,0)`

Specify the coefficients data type. See “Fixed Point” on page 2-615 and “Multiplication Data Types” for illustrations depicting the use of the coefficients data type in this block.

You can set this parameter to one of the following:

- Inherit: Same word length as input
- `fixdt(1,16,0)` or `fixdt(1,16)` -- Specify a data type object.

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Coefficients** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in *Simulink User's Guide* (Simulink) for more information.

**Dependencies**

This parameter appears only when you set **Coefficient source** to `Dialog` parameters, `Filter` object, or `Auto`.

When **Coefficient source** is set to `Filter` object, **Coefficients** parameter is automatically set to `Same word length as input`.

**Coefficients Minimum** — Minimum value of filter coefficients

`[]` (default) | scalar

Specify the minimum value of the filter coefficients. The default value is `[]` (unspecified). Simulink software uses this value to perform automatic scaling of fixed-point data types.

**Dependencies**

This parameter appears only when you set **Coefficient source** to `Dialog` parameters or `Auto`.

**Coefficients Maximum** — Maximum value of filter coefficients

`[]` (default) | scalar

Specify the maximum value of the filter coefficients. The default value is `[]` (unspecified). Simulink software uses this value to perform automatic scaling of fixed-point data types.

**Dependencies**

This parameter appears only when you set **Coefficient source** to `Dialog` parameters or `Auto`.

**Product output Data Type** — Product output data type

Inherit: Inherit via internal rule (default) | Inherit: Same as input | `fixdt(1,16,0)`


Specify the product output data type. See “Fixed Point” on page 2-615 and “Multiplication Data Types” for illustrations depicting the use of the product output data type in this block.

You can set this parameter to one of the following:

- Inherit: Inherit via internal rule

For more information on this rule, see “Inherit via Internal Rule”.

- Inherit: Same as input
- `fixdt(1,16,0)` -- Specify a data type object.

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Product output** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in *Simulink User's Guide* (Simulink) for more information.

### Dependencies

When **Coefficient source** is set to `Filter` object, **Product output** parameter is automatically set to `Full` precision.

### Accumulator Data Type — Accumulator data type

Inherit: `Inherit via internal rule (default)` | Inherit: `Same as input` | Inherit: `Same as product output` | `fixdt(1,16,0)`


Specify the accumulator data type. See “Fixed Point” on page 2-615 for illustrations depicting the use of the accumulator data type in this block.

You can set this parameter to one of the following:

- Inherit: `Inherit via internal rule`.

For more information on this rule, see “Inherit via Internal Rule”.

- Inherit: `Same as input`
- Inherit: `Same as product output`
- `fixdt(1,16,0)` -- Specify a data type object.

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Accumulator** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in *Simulink User's Guide* (Simulink) for more information.

### Dependencies

When **Coefficient source** is set to `Filter` object, **Accumulator** parameter is automatically set to `Full` precision.

### Output Data Type — Output data type


Inherit: `Same as accumulator (default)` | Inherit: `Same as input` | Inherit: `Same as product output` | `fixdt(1,16,0)`

Specify the output data type. See “Fixed Point” on page 2-615 for illustrations depicting the use of the output data type in this block.

You can set it to one of the following:

- Inherit: `Same as accumulator`

- Inherit: Same as input
- Inherit: Same as product output
- `fixdt(1,16,0)` -- Specify a data type object.

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Output** parameter.

See “Control Data Types of Signals” (Simulink) in *Simulink User's Guide* (Simulink) for more information.

### Dependencies

When **Coefficient source** is set to `Filter` object, **Output** parameter is automatically set to `Same as accumulator`.

**Output Minimum** — Minimum value of block output

`[]` (default) | scalar

Specify the minimum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Specify Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

### Dependencies

This parameter appears only when you set **Coefficient source** to `Dialog` parameters, `Input port`, or `Auto`.

**Output Maximum** — Maximum value of block output

`[]` (default) | scalar

Specify the maximum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Specify Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

### Dependencies

This parameter appears only when you set **Coefficient source** to `Dialog` parameters, `Input port`, or `Auto`.

**Lock data type settings against changes by the fixed-point tools** — Prevent fixed-point tools from overriding data types

`off` (default) | `on`

Select this parameter to prevent the fixed-point tools from overriding the data types you specify in the block dialog box.

## Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | double   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no                                      |
| <b>Multidimensional Signals</b> | no                                      |
| <b>Variable-Size Signals</b>    | yes                                     |
| <b>Zero-Crossing Detection</b>  | no                                      |

## More About

### Polyphase Subfilters

A polyphase implementation of an FIR decimator *splits* the lowpass FIR filter impulse response into  $M$  different subfilters, where  $M$  is the downsampling or decimation factor. For more details on the polyphase implementation, see “Algorithms” on page 2-609.

Let  $h(n)$  denote the FIR filter impulse response of length  $N+1$  and  $x(n)$  the input signal. Decimating the filter output by a factor of  $M$  is equivalent to the downsampled convolution:

$$y(n) = \sum_{l=0}^N h(l)x(nM - l)$$

The key to the efficiency of polyphase filtering is that specific input values are only multiplied by select values of the impulse response in the downsampled convolution. For example, letting  $M = 2$ , the input values  $x(0), x(2), x(4), \dots$  are combined only with the filter coefficients  $h(0), h(2), h(4), \dots$ , and the input values  $x(1), x(3), x(5), \dots$  are combined only with the filter coefficients  $h(1), h(3), h(5), \dots$ . By splitting the filter coefficients into two polyphase subfilters, no unnecessary computations are performed in the convolution. The outputs of the convolutions with the polyphase subfilters are interleaved and summed to yield the filter output.

The following code demonstrates how to construct the two polyphase subfilters for the default order 35 filter.

```
M = 2;
Num = fir1(35,0.4);
FiltLength = length(Num);
Num = flipud(Num(:));

if (rem(FiltLength, M) ~= 0)
    nzeros = M - rem(FiltLength, M);
    Num = [zeros(nzeros,1); Num]; % Appending zeros
end

len = length(Num);
nrows = len / M;
PolyphaseFilt = flipud(reshape(Num, M, nrows).');
```

The columns of `PolyphaseFilt` are subfilters containing the two *phases* of the filter in `Num`. For a general downsampling factor of  $M$ , there are  $M$  phases and therefore  $M$  subfilters.

## Frame-Based Processing

When you set the **Input processing** parameter to **Columns as channels (frame based)**, the block resamples each column of the input over time. In this mode, the block can perform either single-rate or multirate processing. You can use the **Rate options** parameter to specify how the block resamples the input.

- When you set the **Rate options** parameter to **Enforce single-rate processing**, the input and output of the block have the same sample rate. To decimate the output while maintaining the input sample rate, the block resamples the data in each column of the input such that the frame length of the output has an upper bound size of  $\text{ceil}(P/M)$ , where  $P$  is the input frame length and  $M$  is the decimation factor specified in the **Decimation factor** parameter.

In this mode, if you input a fixed-size signal (frame length does not change during simulation) and select the **Allow arbitrary frame length for fixed-size input signals** parameter, the input frame length can be arbitrary and does not have to be a multiple of the decimation factor. If you clear the **Allow arbitrary frame length for fixed-size input signals** parameter, the input frame length must be a multiple of the decimation factor.

In this mode, if you input a variable-size signal (frame length changes during simulation), the **Allow arbitrary frame length for fixed-size input signals** appears on the block dialog box but does not have any impact on the input frame length. You can input a variable-size signal of any frame length even if you do not select the **Allow arbitrary frame length for fixed-size input signals** parameter.

This table summarizes the support for arbitrary input frame length when you set **Input processing** to **Columns as channels (frame based)** and **Rate options** to **Enforce single-rate processing**.

| Input Signal               | Block Support for this Signal | Support for Arbitrary Input Frame Length   | Input Size   | Output Size                                     |
|----------------------------|-------------------------------|--|--------------|---|
| Fixed-size input signal    | Yes                           | When you select <b>Allow arbitrary frame length for fixed-size input signals</b> | $P$ -by- $Q$ | Upper bound size of $\text{ceil}(P/M)$ -by- $Q$ |
| Variable-size input signal | Yes                           | Always   | $P$ -by- $Q$ | Upper bound size of $\text{ceil}(P/M)$ -by- $Q$ |

For an example that shows single-rate FIR decimation, see “FIR Decimation Using Single-Rate Processing”.

- When you set the **Rate options** parameter to **Allow multirate processing**, the input and output of the FIR Decimation block are of the same size, but the sample rate of the output is  $M$  times slower than that of the input. In this mode, the block treats a  $P$ -by- $Q$  matrix input as  $Q$  independent channels. The block decimates each column of the input over time by keeping the frame size constant, and making the output frame period ( $T_{fo}$ )  $M$  times longer than the input frame period ( $T_{fo} = M \cdot T_{fi}$ ).

In this mode, the block accepts only fixed-size signals and these signals can have arbitrary frame length.

This table summarizes the support for arbitrary input frame length when you set **Input processing** to Columns as channels (frame based) and **Rate options** to Allow multirate processing.

| Input Signal               | Block Support for this Signal | Support for Arbitrary Input Frame Length | Input Size     | Output Size    |
|----------------------------|-------------------------------|--|----------------|----------------|
| Fixed-size input signal    | Yes                           | Always                                   | $P$ -by- $Q$   | $P$ -by- $Q$   |
| Variable-size input signal | No                            | Not applicable                           | Not applicable | Not applicable |

See “FIR Decimation Using Multirate Frame-Based Processing” for an example that uses the FIR Decimation block in this mode.

### Sample-Based Processing

When you set the **Input processing** parameter to Elements as channels (sample based), the block treats a  $P$ -by- $Q$  matrix input as  $P*Q$  independent channels, and decimates each channel over time. The output sample period ( $T_{so}$ ) is  $M$  times longer than the input sample period ( $T_{so} = M*T_{si}$ ), and the input and output sizes are identical.

In this mode, the block accepts only fixed-size signals and these signals can have an arbitrary frame length.

This table summarizes the support for arbitrary input frame length when you set **Input processing** to Elements as channels (sample based). **Rate options** is automatically set to Allow multirate processing.

| Input Signal               | Block Support for this Signal | Support for Arbitrary Input Frame Length | Input Size     | Output Size    |
|----------------------------|-------------------------------|--|----------------|----------------|
| Fixed-size input signal    | Yes                           | Always                                   | $P$ -by- $Q$   | $P$ -by- $Q$   |
| Variable-size input signal | No                            | Not applicable                           | Not applicable | Not applicable |

### Latency

When you use the FIR Decimation block in the sample-based processing mode, the block always has zero-tasking latency. Zero-tasking latency means that the block propagates the first filtered input sample (received at time  $t = 0$ ) as the first output sample. That first output sample is then followed by filtered input samples  $M+1$ ,  $2M+1$ , and so on.

When you use the FIR Decimation block in the frame-based processing mode with a frame size greater than one, the block may exhibit *one-frame latency*. Cases of one-frame latency can occur when the input frame size is greater than one and you set the **Input processing** and **Rate options** parameters of the FIR Decimation block as follows:

- **Input processing** = Columns as channels (frame based)
- **Rate options** = Allow multirate processing



In cases of one-frame latency, you can define the value of the first  $P$  output rows by setting the **Output buffer initial conditions** parameter. The default value of the **Output buffer initial conditions** parameter is  $\emptyset$ . However, you can enter a matrix containing one value for each channel of the input, or a scalar value to be applied to all channels. The first filtered input sample (first filtered row of the input matrix) appears in the output as sample  $P+1$ . That sample is then followed by filtered input samples  $M+1$ ,  $2M+1$ , and so on.

---

**Note** For more information on latency and the Simulink tasking modes, see “Excess Algorithmic Delay (Tasking Latency)” and “Time-Based Scheduling and Code Generation” (Simulink Coder).

---

## Algorithms

The FIR decimation filter is implemented efficiently using a polyphase structure. For more details on polyphase filters, see Polyphase Subfilters on page 2-606.

To derive the polyphase structure, start with the transfer function of the FIR filter:

$$H(z) = b_0 + b_1z^{-1} + \dots + b_Nz^{-N}$$

$N+1$  is the length of the FIR filter.

You can rearrange this equation as follows:

$$H(z) = \begin{aligned} & (b_0 + b_Mz^{-M} + b_{2M}z^{-2M} + \dots + b_{N-M+1}z^{-(N-M+1)}) + \\ & z^{-1}(b_1 + b_{M+1}z^{-M} + b_{2M+1}z^{-2M} + \dots + b_{N-M+2}z^{-(N-M+1)}) + \\ & \quad \vdots \\ & z^{-(M-1)}(b_{M-1} + b_{2M-1}z^{-M} + b_{3M-1}z^{-2M} + \dots + b_Nz^{-(N-M+1)}) \end{aligned}$$

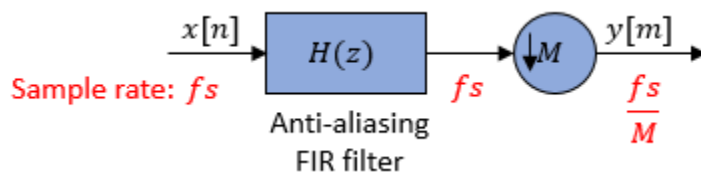
$M$  is the number of polyphase components, and its value equals the decimation factor that you specify.

You can write this equation as:

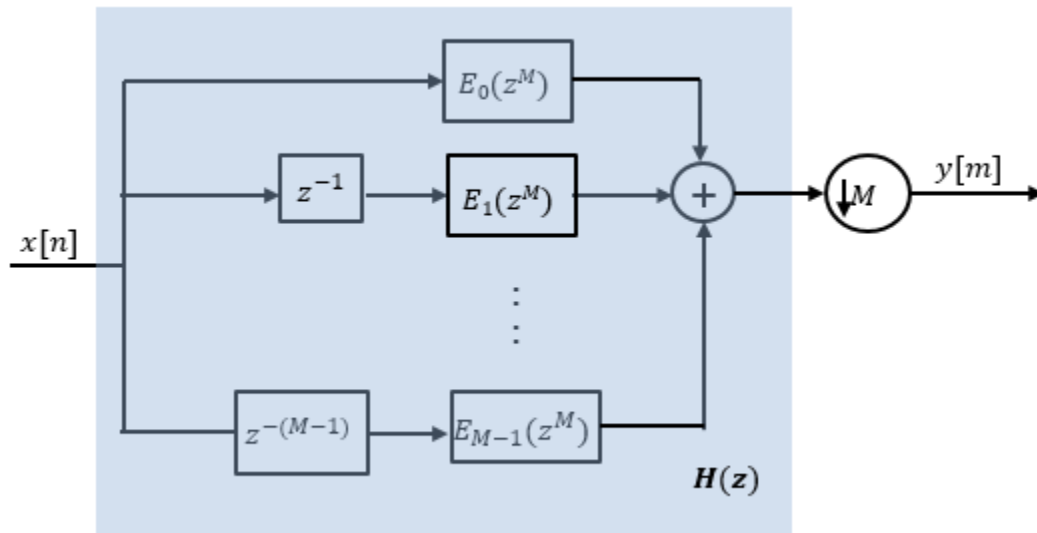
$$H(z) = E_0(z^M) + z^{-1}E_1(z^M) + \dots + z^{-(M-1)}E_{M-1}(z^M)$$

$E_0(z^M)$ ,  $E_1(z^M)$ , ...,  $E_{M-1}(z^M)$  are the polyphase components of the FIR filter  $H(z)$ .

Conceptually, the FIR decimation filter contains a lowpass FIR filter followed by a downsampler.



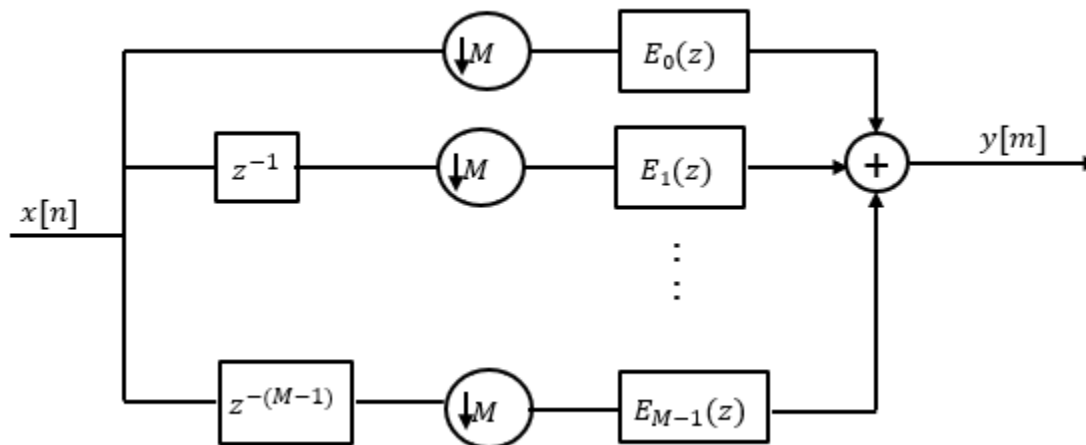
Replace  $H(z)$  with its polyphase representation.



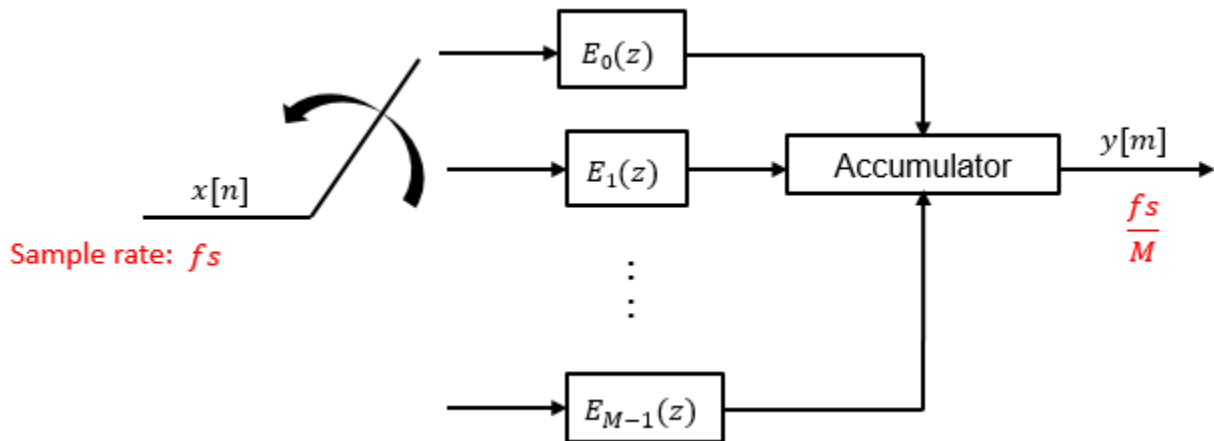
Here is the multirate noble identity for decimation.



Applying the noble identity for decimation moves the downsampling operation to before the filtering operation. This move enables you to filter the signal at a lower rate.



You can replace the delays and the decimation factor at the input with a commutator switch. The switch starts on the first branch 0 and moves in the counterclockwise direction as shown in this diagram. The accumulator at the output receives the processed input samples from each branch of the polyphase structure and accumulates these processed samples until the switch goes to branch 0. When the switch goes to branch 0, the accumulator outputs the accumulated value.



When the first input sample is delivered, the switch feeds this input to the branch 0 and the decimator computes the first output value. As more input samples come in, the switch moves in the counter clockwise direction through branches  $M-1$ ,  $M-2$ , and all the way up to branch 0, delivering one sample at a time to each branch. When the switch comes to branch 0, the decimator outputs the next set of output values. This process continues as data keeps coming in. Every time the switch comes to the branch 0, the decimator outputs  $y[m]$ . The decimator effectively outputs one sample for every  $M$  samples it receives. Hence the sample rate at the output of the FIR decimation filter is  $f_s/M$ .

## Version History

Introduced before R2006a

### R2022b: Support for arbitrary input frame length

Starting in R2022b, this block supports input signals with arbitrary frame lengths when the:

- Input signal is a fixed-size signal (frame length does not change during simulation) and the block allows for multirate processing
- Input signal is a fixed-size signal, the block enforces single-rate processing, and you select the **Allow arbitrary frame length for fixed-size input signals** parameter
- Input signal is a variable-size signal (frame length changes during simulation)

When this block supports an arbitrary frame length input signal, the input frame length does not have to be a multiple of the decimation factor.

For more details, see “Frame-Based Processing” on page 2-607 and “Sample-Based Processing” on page 2-608.

### R2023a: Generate SIMD code for FIR Decimation block on all Intel platforms

In R2023a, if you have Embedded Coder, you can generate SIMD code for the FIR Decimation block on all Intel platforms by using the model configuration parameter **Leverage target hardware instruction set extensions**. Previously, you had to use a code replacement library to generate SIMD code. For more details, see “Code Generation” on page 2-612.

## References

- [1] Fliege, N. J. *Multirate Digital Signal Processing: Multirate Systems, Filter Banks, Wavelets*. West Sussex, England: John Wiley & Sons, 1994.
- [2] Orfanidis, Sophocles J. *Introduction to Signal Processing*. Upper Saddle River, NJ: Prentice-Hall, 1996.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Generated code relies on the `memcpy` or `memset` function (`string.h`) under certain conditions.

If you have an Embedded Coder® license, you can generate SIMD code for the FIR Decimation block.

### Generate SIMD code using Intel AVX2 technology

The FIR Decimation block supports SIMD code generation using Intel AVX2 technology through a code replacement library under these conditions:

- **Filter structure** is set to `Direct` form.
- **Input processing** is set to `Columns as channels (frame based)`.
- **Rate options** is set to `Enforce single-rate processing`.
- Input signal is real-valued with real filter coefficients.
- Input signal is complex-valued with real or complex filter coefficients.
- Input signal has a data type of `single` or `double`.

For more information, see “SIMD Code Generation”.

### Generate SIMD code on all Intel® platforms

You can generate SIMD code for the FIR Decimation block on all Intel platforms by using the model configuration parameter **Leverage target hardware instruction set extensions**. Previously, you had to use a code replacement library to generate SIMD code.

You can generate SIMD code using the **Leverage target hardware instruction set extensions** parameter under these conditions:

- You set **Filter structure** to `Direct` form for the FIR Decimation block
- You set **Input processing** to `Columns as channels (frame based)`
- Input signal is real-valued with real filter coefficients
- Data type of the input signal is `single` or `double`

In addition, configure your model appropriately. In the **Modeling** tab of the Simulink model window, click **Model Settings** and configure these parameters under **Code Generation**.

- In the **Optimization** pane:
  - Provide a specific instruction set in the **Leverage target hardware instruction set extensions** parameter.

- Select the **Optimize reductions** parameter.
- Under **Optimization levels**, set **Level** to Maximum and **Priority** to Maximize execution speed.
- In the **Interface** pane, under **Software environment**, clear **non-finite numbers**.

For computationally intensive operations on supported blocks, SIMD intrinsics can significantly improve the performance of the generated code on Intel platforms. For more details, see “Optimize Code for Reduction Operations by Using SIMD” (Simulink Coder).

The SIMD technology significantly improves the performance of the generated code.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

---

**Note** For a FIR decimation filter with hardware-friendly control signals and simulation of HDL latency in Simulink, or for complex data with complex coefficients, use the FIR Decimator block instead of this block.

---

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Coder supports **Coefficient source** options **Dialog parameters**, **Filter object**, or **Auto**. Programmable coefficients are not supported.

### Frame-Based Input Support

HDL Coder supports the use of vector inputs to FIR Decimation blocks, where each element of the vector represents a sample in time. You can use an input vector of up to 512 samples. The frame-based implementation supports fixed-point input and output data types, and uses full-precision internal data types. The output is a column vector of reduced size, corresponding to your decimation factor. You can use real input signals with real coefficients, complex input signals with real coefficients, or real input signals with complex coefficients.

- 1 Connect a column vector signal to the FIR Decimation block input port.
- 2 Specify **Input processing** as Columns as channels (frame based).
- 3 Set **Rate options** to Enforce single-rate processing.
- 4 Right-click the block and open **HDL Code > HDL Block Properties**. Set the **Architecture** to Frame Based. The block implements a parallel HDL architecture. See “Frame-Based Architecture” (HDL Coder).

### Block Optimizations

To reduce area or increase speed, the FIR Decimator block supports block-level optimizations.

Right-click on the block or the subsystem to open the corresponding **HDL Properties** dialog box and set optimization properties.

|                        |   |
|------------------------|---|
| Serial Architectures   | <p>To use block-level optimizations to reduce hardware resources, set <b>Architecture</b> to Fully Serial or Partly Serial. See “HDL Filter Architectures” (HDL Coder).</p> <p>When you specify <b>SerialPartition</b> for a FIR Decimator block, set <b>Filter structure</b> to Direct form. The Direct form transposed structure is not supported with serial architectures. Accumulator reuse is not supported for FIR Decimation filters.</p>   |
| Distributed Arithmetic | <p>To minimize multipliers by replacing them with LUTs and shift registers, use a distributed arithmetic (DA) filter implementation. See “Distributed Arithmetic for HDL Filters” (HDL Coder).</p> <p>When you select the Distributed Arithmetic (DA) architecture and use the <b>DALUTPartition</b> and <b>DARadix</b> distributed arithmetic properties, set <b>Filter structure</b> to Direct form. The Direct form transposed structure is not supported with distributed arithmetic.</p> |
| Pipelining             | <p>To improve clock speed, use <b>AddPipelineRegisters</b> to use a pipelined adder tree rather than the default linear adder. This option is supported for Direct form architecture. You can also specify the number of pipeline stages before and after the multipliers. See “HDL Filter Architectures” (HDL Coder).</p>  |

#### HDL Filter Properties

|                                 |   |
|---------------------------------|---|
| <b>AddPipelineRegisters</b>     | Insert a pipeline register between stages of computation in a filter. See also AddPipelineRegisters (HDL Coder).  |
| <b>CoeffMultipliers</b>         | Specify the use of canonical signed digit (CSD) optimization to decrease filter area by replacing coefficient multipliers with shift-and-add logic. When you choose a fully parallel filter implementation, you can set <b>CoeffMultipliers</b> to <code>csd</code> or <code>factored-csd</code> . The default is <code>multipliers</code> , which retains multipliers in the HDL. See also CoeffMultipliers (HDL Coder). |
| <b>DALUTPartition</b>           | Specify distributed arithmetic partial-product LUT partitions as a vector of the sizes of each partition. The sum of all vector elements must be equal to the filter length. The maximum size for a partition is 12 taps. Set <b>DALUTPartition</b> to a scalar value equal to the filter length to generate DA code without LUT partitions. See also DALUTPartition (HDL Coder).   |
| <b>DARadix</b>                  | Specify how many distributed arithmetic bit sums are computed in parallel. A DA radix of 8 ( $2^3$ ) generates a DA implementation that computes three sums at a time. The default value is $2^1$ , which generates a fully serial DA implementation. See also DARadix (HDL Coder).   |
| <b>MultiplierInputPipeline</b>  | Specify the number of pipeline stages to add at filter multiplier inputs. See also MultiplierInputPipeline (HDL Coder).   |
| <b>MultiplierOutputPipeline</b> | Specify the number of pipeline stages to add at filter multiplier outputs. See also MultiplierOutputPipeline (HDL Coder).   |
| <b>SerialPartition</b>          | Specify partitions for partly serial or cascade-serial filter implementations as a vector of the lengths of each partition. For a fully serial implementation, set this parameter to the length of the filter. See also SerialPartition (HDL Coder).  |

### HDL Block Properties

|                                  |  |
|----------------------------------|--|
| <b>ConstrainedOutputPipeline</b> | Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder). |
| <b>InputPipeline</b>             | Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).                 |
| <b>OutputPipeline</b>            | Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).               |

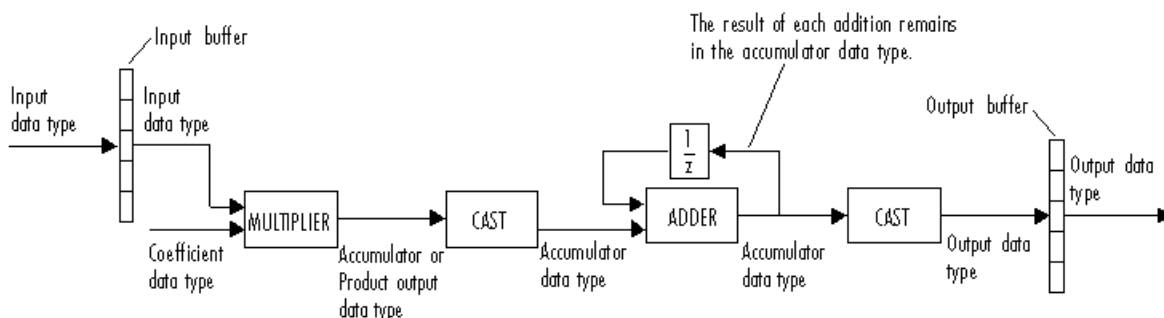
### Restrictions

- You must set **Initial conditions** to zero. HDL code generation is not supported for nonzero initial states.
- When you select **Dialog parameters**, the following fixed-point options are not supported for HDL code generation:
  - Slope and Bias scaling
- **CoeffMultipliers** options are supported only when using a fully parallel architecture. When you select a serial architecture, **CoeffMultipliers** is hidden from the HDL Block Properties dialog box.
- Programmable coefficients are not supported.
- Frame-based input filters are not supported for:
  - Resettable and enabled subsystems
  - Complex input signals with complex coefficients. You can use either complex input signals and real coefficients, or complex coefficients and real input signals.
  - Sharing and streaming optimizations

### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

The following diagram shows the data types used within the FIR Decimation block for fixed-point signals.



This diagram shows that data is stored in the input buffer with the same data type and scaling as the input. The block stores filtered data and any initial conditions in the output buffer using the output data type and scaling that you set in the block dialog box.

When at least one of the inputs to the multiplier is real, the output of the multiplier is in the product output data type. When both inputs to the multiplier are complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication performed by this block, see “Multiplication Data Types”.

---

**Note** When the block input is fixed point, all internal data types are signed fixed-point values.

---

## See Also

### Functions

`firgr` | `firceqrip` | `firhalfband` | `firnyquist`

### Objects

`dsp.FIRDecimator` | `dsp.CICCompensationDecimator` | `dsp.FIRHalfbandDecimator`

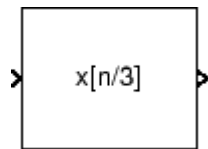
### Blocks

Variable FIR Decimation | FIR Interpolation | FIR Rate Conversion | FIR Halfband Interpolator | FIR Halfband Decimator | IIR Halfband Interpolator | IIR Halfband Decimator | CIC Compensation Interpolator | CIC Compensation Decimator | Downsample | CIC Decimation | Digital Up-Converter | Digital Down-Converter



# FIR Interpolation

Perform polyphase FIR interpolation



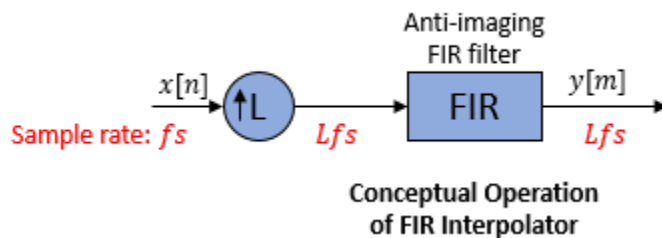
**Libraries:**

DSP System Toolbox / Filtering / Multirate Filters  
 DSP System Toolbox HDL Support / Filtering

## Description

The FIR Interpolation block performs an efficient polyphase interpolation using an integer upsampling factor  $L$  along the first dimension.

Conceptually, the FIR interpolator (as shown in the schematic) consists of an upsampler followed by an FIR anti-imaging filter, which is usually an approximation of an ideal band-limited interpolation filter. To design an FIR anti-imaging filter, use the `designMultirateFIR` function. The upsampler upsamples each channel of the input to a higher rate by inserting  $L-1$  zeros between samples. The FIR filter that follows filters each channel of the upsampled data. The resulting discrete-time signal has a sample rate that is  $L$  times the original sample rate.



However, the actual block algorithm implements a direct-form FIR polyphase structure, an efficient equivalent of the combined system depicted in the diagram. For more details, see “Algorithms” on page 2-626.

You can use the FIR Interpolation block inside triggered subsystems when you set the **Rate options** parameter to `Enforce single-rate processing`.

Under specific conditions, this block also supports SIMD code generation. For more details, see “Code Generation” on page 2-629.

## Ports

### Input

**In** — Data input  
 vector | matrix

Specify the data input as a vector or a matrix.

When you set **Input processing** to `Columns as channels (frame based)` and **Rate options** to `Enforce single-rate processing`, the input can be a variable-size signal. That is, the frame size (number of rows) and the number of channels (columns) of the signal can change during simulation.

This port is unnamed until you set **Coefficient source** to `Input port`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

**Num** — Numerator coefficients  
vector

Specify the numerator coefficients of the FIR filter as a vector.

The transfer function  $H(z)$  of the FIR filter is given by:

You can generate the FIR filter coefficient vector,  $\mathbf{b} = [b_0, b_1, \dots, b_N]$ , using one of the DSP System Toolbox filter design functions such as `designMultirateFIR`, `firnyquist`, `firhalfband`, `firgr` or `firceqrip`.

To act as an effective anti-imaging filter, the coefficients usually correspond to a lowpass filter with a normalized cutoff frequency no greater than the reciprocal of the interpolation factor. To design such a filter, use the `designMultirateFIR` function.

Coefficient values are tunable. That is, their values can change during simulation while their properties such as size, data type, and complexity cannot change.

The data type of the **Num** input must match the data type of the **In** input.

### Dependencies

The **Num** input port appears when you set **Coefficient source** as `Input port`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

### Output

**Out** — Interpolator output  
vector | matrix

Output of the FIR Interpolator block, returned as a vector or a matrix.

When **Rate options** is set to:

- `Enforce single-rate processing` — When you select this option, the block maintains the input sample rate, and interpolates the signal by increasing the output frame size by a factor of  $L$ .
- `Allow multirate processing` — When you select this option, the block interpolates the signal such that the output sample rate is  $L$  times faster than the input sample rate.

When the input is a variable-size signal, the output is also a variable-size signal.

This port is unnamed until you set **Coefficient source** to `Input port`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

## Parameters

### Coefficient source

**Coefficient source** — FIR filter coefficient source

**Auto** (default) | **Dialog parameters** | **Input port** | **Filter object**

Specify the FIR filter coefficient source as one of the following:

- **Dialog parameters** -- Specify the filter coefficients through the **FIR filter coefficients** parameter in the block dialog box.
- **Input port** -- Specify the filter coefficients through the **Num** input port.
- **Filter object** -- Specify the filter using a `dsp.FIRInterpolator` System object.
- **Auto** -- When you select **Auto**, the block designs an FIR interpolator using the interpolation factor you specify in **Interpolation factor**. The `designMultirateFIR` function designs the filter and returns the coefficients used by the block.

For more information on the filter design, see Orfanidis [2].

### Main Tab

**FIR filter coefficients** — Lowpass FIR filter coefficients

`designMultirateFIR(3,1)` (default) | vector

Specify the numerator coefficients of the FIR filter transfer function  $H(z)$ .

$$H(z) = b_0 + b_1z^{-1} + \dots + b_Nz^{-N}$$

You can generate the FIR filter coefficient vector,  $\mathbf{b} = [b_0, b_1, \dots, b_N]$ , using one of the DSP System Toolbox filter design functions such as `designMultirateFIR`, `firnyquist`, `firhalfband`, `firgr` or `firceqrip`.

To act as an effective anti-imaging filter, the coefficients usually correspond to a lowpass filter with a normalized cutoff frequency no greater than the reciprocal of the interpolation factor. To design such a filter, use the `designMultirateFIR` function.

The block internally initializes all filter states to zero.

### Dependencies

This parameter appears only when you set the **Coefficient source** to **Dialog parameters**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**Interpolation factor** — Interpolation factor

3 (default) | positive scalar

Specify the integer factor  $L$ . The block increases the sample rate of the input sequence by this factor.

### Dependencies

This parameter appears only when you set the **Coefficient source** to **Dialog parameters**, **Input port**, or **Auto**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**Filter object** — Filter object  
`dsp.FIRInterpolator`

Specify the name of the multirate filter object that you want the block to implement. You must specify the filter as a `dsp.FIRInterpolator` System object.

You can define the System object directly in the block dialog box. Alternatively, you can define the object in a MATLAB workspace variable and specify the variable in the block dialog box.

For information on creating System objects, see “Define Basic System Objects”.

### Dependencies

This parameter appears only when you set the **Coefficient source** to **Filter object**.

**Input processing** — Method to process input signals  
`Columns as channels (frame based)` (default) | `Elements as channels (sample based)`

Specify how the block should process the input. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

**Rate options** — Method by which block interpolates input  
`Enforce single-rate processing` (default) | `Allow multirate processing`

Specify the method by which the block should interpolate the input. You can select one of the following options:

- `Enforce single-rate processing` — When you select this option, the block maintains the input sample rate, and interpolates the signal by increasing the output frame size by a factor of  $L$ . To select this option, you must set the **Input processing** parameter to `Columns as channels (frame based)`.
- `Allow multirate processing` — When you select this option, the block interpolates the signal such that the output sample rate is  $L$  times faster than the input sample rate.

**Output buffer initial conditions** — Initial conditions  
`0` (default) | `scalar` | `matrix`

When you set the **Rate options** parameter to `Allow multirate processing` and run your models in Simulink `MultiTasking` mode, the block exhibits latency. The amount of latency for multirate, multitasking operation depends on how you set the **Input processing** parameter.

| Input processing                    | Latency                               |
|-------------------------------------|---------------------------------------|
| Elements as channels (sample based) | $L$ samples                           |
| Columns as channels (frame based)   | $L$ frames ( $K_i$ samples per frame) |

When the block exhibits latency, the default initial condition is zero. Alternatively, you can use the **Output buffer initial conditions** parameter to specify a matrix of initial conditions containing one value for each channel or a scalar initial condition that the block applies to all channels. The block divides the **Output buffer initial conditions** by the **Interpolation factor** and outputs the scaled initial conditions until the first filtered input sample becomes available.

Output buffer initial conditions are stored in the output data type and scaling.

See “Latency” on page 2-626 for more information about latency in the FIR Interpolation block.

### Dependencies

This parameter appears only when you configure the block to perform multirate processing by setting **Rate options** to Allow multirate processing.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**View Filter Response** — View filter response  
button

Click on this button to open the Filter Visualization Tool (fvtool) and display the filter response of the filter defined in the block dialog box.

### Data Types Tab

**Rounding mode** — Rounding mode

Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Select the rounding mode for fixed-point operations. The default is Floor. The filter coefficients do not obey this parameter and always round to Nearest.

---

**Note** The **Rounding mode** and **Saturate on integer overflow** settings have no effect on numerical results when all the following conditions exist:

- **Product output** is Inherit: Inherit via internal rule
- **Accumulator** is Inherit: Inherit via internal rule
- **Output** is Inherit: Same as accumulator

With these data type settings, the block is effectively operating in the full-precision mode.

---

**Saturate on integer overflow** — Saturate on integer overflow  
off (default) | on

When you select this parameter, the block saturates the result of its fixed-point operation. When you clear this parameter, the block wraps the result of its fixed-point operation. For details on `saturate` and `wrap`, see `overflow mode for fixed-point operations`.

---

**Note** The **Rounding mode** and **Saturate on integer overflow** parameters have no effect on numeric results when all these conditions are met:

- **Product output** data type is `Inherit`: `Inherit via internal rule`.
- **Accumulator** data type is `Inherit`: `Inherit via internal rule`.

With these data type settings, the block operates in the full-precision mode.

---


### **Coefficients Data Type** — Coefficients data type

`Inherit`: Same word length as input (default) | `fixdt(1,16)` | `fixdt(1,16,0)`

Specify the coefficients data type. See “Fixed Point” on page 2-632 and “Multiplication Data Types” for illustrations depicting the use of the coefficients data type in this block.

You can set this parameter to one of the following:

- `Inherit`: Same word length as input
- `fixdt(1,16,0)` or `fixdt(1,16)` -- Specify a data type object.

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Coefficients** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in *Simulink User's Guide* (Simulink) for more information.

### **Dependencies**

This parameter appears only when you set **Coefficient source** to `Dialog` parameters, `Filter` object, or `Auto`.

When **Coefficient source** is set to `Filter` object, **Coefficients** parameter is automatically set to `Same word length as input`.

### **Coefficients Minimum** — Minimum value of filter coefficients

`[]` (default) | scalar

Specify the minimum value of the filter coefficients. The default value is `[]` (unspecified). Simulink software uses this value to perform automatic scaling of fixed-point data types.

### **Dependencies**

This parameter appears only when you set **Coefficient source** to `Dialog` parameters or `Auto`.

### **Coefficients Maximum** — Maximum value of filter coefficients

`[]` (default) | scalar

Specify the maximum value of the filter coefficients. The default value is `[]` (unspecified). Simulink software uses this value to perform automatic scaling of fixed-point data types.

### Dependencies

This parameter appears only when you set **Coefficient source** to `Dialog` parameters or `Auto`.


#### **Product output Data Type** — Product output data type

Inherit: `Inherit via internal rule (default)` | `Inherit: Same as input` | `fixdt(1,16,0)`

Specify the product output data type. See “Fixed Point” on page 2-632 and “Multiplication Data Types” for illustrations depicting the use of the product output data type in this block.

You can set this parameter to one of the following:

- `Inherit: Inherit via internal rule`  
For more information on this rule, see “Inherit via Internal Rule”.
- `Inherit: Same as input`
- `fixdt(1,16,0)` -- Specify a data type object.

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Product output** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in *Simulink User's Guide* (Simulink) for more information.

### Dependencies

When **Coefficient source** is set to `Filter` object, **Product output** parameter is automatically set to `Full` precision.


#### **Accumulator Data Type** — Accumulator data type

Inherit: `Inherit via internal rule (default)` | `Inherit: Same as input` | `Inherit: Same as product output` | `fixdt(1,16,0)`

Specify the accumulator data type. See “Fixed Point” on page 2-632 for illustrations depicting the use of the accumulator data type in this block.

You can set this parameter to one of the following:

- `Inherit: Inherit via internal rule`.  
For more information on this rule, see “Inherit via Internal Rule”.
- `Inherit: Same as input`
- `Inherit: Same as product output`
- `fixdt(1,16,0)` -- Specify a data type object.

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Accumulator** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in *Simulink User's Guide* (Simulink) for more information.

### Dependencies

When **Coefficient source** is set to `Filter` object, **Accumulator** parameter is automatically set to `Full` precision.


### Output Data Type — Output data type

Inherit: Same as accumulator (default) | Inherit: Same as input | Inherit: Same as product output | `fixdt(1,16,0)`

Specify the output data type. See “Fixed Point” on page 2-632 for illustrations depicting the use of the output data type in this block.

You can set it to one of the following:

- Inherit: Same as accumulator
- Inherit: Same as input
- Inherit: Same as product output
- `fixdt(1,16,0)` -- Specify a data type object.

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Output** parameter.

See “Control Data Types of Signals” (Simulink) in *Simulink User's Guide* (Simulink) for more information.

### Dependencies

When **Coefficient source** is set to `Filter` object, **Output** parameter is automatically set to `Same as accumulator`.

### Output Minimum — Minimum value of block output

[] (default) | scalar

Specify the minimum value that the block should output. The default value is [] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Specify Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

### Dependencies

This parameter appears only when you set **Coefficient source** to `Dialog` parameters, `Input port`, or `Auto`.

### Output Maximum — Maximum value of block output

[] (default) | scalar

Specify the maximum value that the block should output. The default value is [] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Specify Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types



## Dependencies

This parameter appears only when you set **Coefficient source** to Dialog parameters, Input port, or Auto.

**Lock data type settings against changes by the fixed-point tools** — Prevent fixed-point tools from overriding data types  
off (default) | on

Select this parameter to prevent the fixed-point tools from overriding the data types you specify in the block dialog box.

## Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | double   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no                                      |
| <b>Multidimensional Signals</b> | no                                      |
| <b>Variable-Size Signals</b>    | yes                                     |
| <b>Zero-Crossing Detection</b>  | no                                      |

## More About

### Frame-Based Processing

When you set the **Input processing** parameter to Columns as channels (frame based), the block resamples each column of the input over time. In this mode, the block can perform either single-rate or multirate processing. You can use the **Rate options** parameter to specify how the block resamples the input:

- When you set the **Rate options** parameter to Enforce single-rate processing, the input and output of the block have the same sample rate. To interpolate the output while maintaining the input sample rate, the block resamples the data in each column of the input such that the frame size of the output ( $K_o$ ) is  $L$  times larger than that of the input ( $K_o = K_i * L$ ).

For an example of single-rate FIR Interpolation, see “FIR Interpolation Using Single-Rate Processing”.

- When you set the **Rate options** parameter to Allow multirate processing, the input and output of the FIR Interpolation block are the same size. However, the sample rate of the output is  $L$  times faster than that of the input. In this mode, the block treats a  $K_i$ -by- $N$  matrix input as  $N$  independent channels. The block interpolates each column of the input over time by keeping the frame size constant ( $K_i = K_o$ ), while making the output frame period ( $T_{fo}$ )  $L$  times shorter than the input frame period ( $T_{fo} = T_{fi}/L$ ).

See “FIR Interpolation Using Multirate Frame-Based Processing” for an example that uses the FIR Interpolation block in this mode.

## Sample-Based Processing

When you set the **Input processing** parameter to `Elements as channels (sample based)`, the block treats a  $P$ -by- $Q$  matrix input as  $P*Q$  independent channels, and interpolates each channel over time. The output sample period ( $T_{so}$ ) is  $L$  times shorter than the input sample period ( $T_{so} = T_{si}/L$ ), while the input and output sizes remain identical.

## Latency

When you run your models in the Simulink `SingleTasking` mode or set the **Input processing** parameter to `Columns as channels (frame based)` and the **Rate options** parameter to `Enforce single-rate processing`, the FIR Interpolation block always has zero-tasking latency. Zero-tasking latency means that the block propagates the first filtered input sample (received at time  $t=0$ ) as the first output sample. That first output sample is then followed by  $L-1$  interpolated values, the second filtered input sample, and so on.

The only time the FIR Interpolation block exhibits latency is when you set the **Rate options** parameter set to `Allow multirate processing` and run your models in the Simulink `MultiTasking` mode. The amount of latency for a multirate, multitasking operation depends on how you set the **Input processing** parameter.

| Input processing                    | Latency                               |
|-------------------------------------|---------------------------------------|
| Elements as channels (sample based) | $L$ samples                           |
| Columns as channels (frame based)   | $L$ frames ( $K_i$ samples per frame) |

When the block exhibits latency, the default initial condition is zero. Alternatively, you can use the **Output buffer initial conditions** parameter to specify a matrix of initial conditions containing one value for each channel or a scalar initial condition that the block applies to all channels. The block scales the **Output buffer initial conditions** by the **Interpolation factor** and outputs the scaled initial conditions until the first filtered input sample becomes available.

When the block is in the sample-based processing mode, the block outputs the scaled initial conditions at the start of each channel, followed immediately by the first filtered input sample, then  $L-1$  interpolated values, and so on.

When the block is in the frame-based processing mode and using the default initial condition of zero, the first  $K_i*L$  output rows contain zeros, where  $K_i$  is the input frame size. The first filtered input sample (first filtered row of the input matrix) appears in the output as sample  $K_i*L+1$ . That value is then followed by  $L-1$  interpolated values, the second filtered input sample, and so on.

---

**Note** For more information on latency and the Simulink tasking modes, see “Excess Algorithmic Delay (Tasking Latency)” and “Time-Based Scheduling and Code Generation” (Simulink Coder).

---

## Algorithms

The FIR interpolation filter is implemented efficiently using a polyphase structure.

To derive the polyphase structure, start with the transfer function of the FIR filter:

$$H(z) = b_0 + b_1z^{-1} + \dots + b_Nz^{-N}$$

$N+1$  is the length of the FIR filter.

You can rearrange this equation as follows:

$$H(z) = (b_0 + b_L z^{-L} + b_{2L} z^{-2L} + \dots + b_{N-L+1} z^{-(N-L+1)}) + z^{-1}(b_1 + b_{L+1} z^{-L} + b_{2L+1} z^{-2L} + \dots + b_{N-L+2} z^{-(N-L+1)}) + \dots + z^{-(L-1)}(b_{L-1} + b_{2L-1} z^{-L} + b_{3L-1} z^{-2L} + \dots + b_N z^{-(N-L+1)})$$

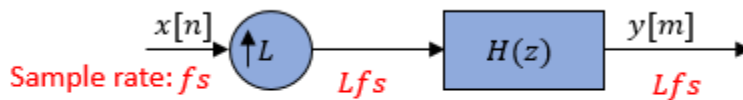
$L$  is the number of polyphase components, and its value equals the interpolation factor that you specify.

You can write this equation as:

$$H(z) = E_0(z^L) + z^{-1}E_1(z^L) + \dots + z^{-(L-1)}E_{L-1}(z^L)$$

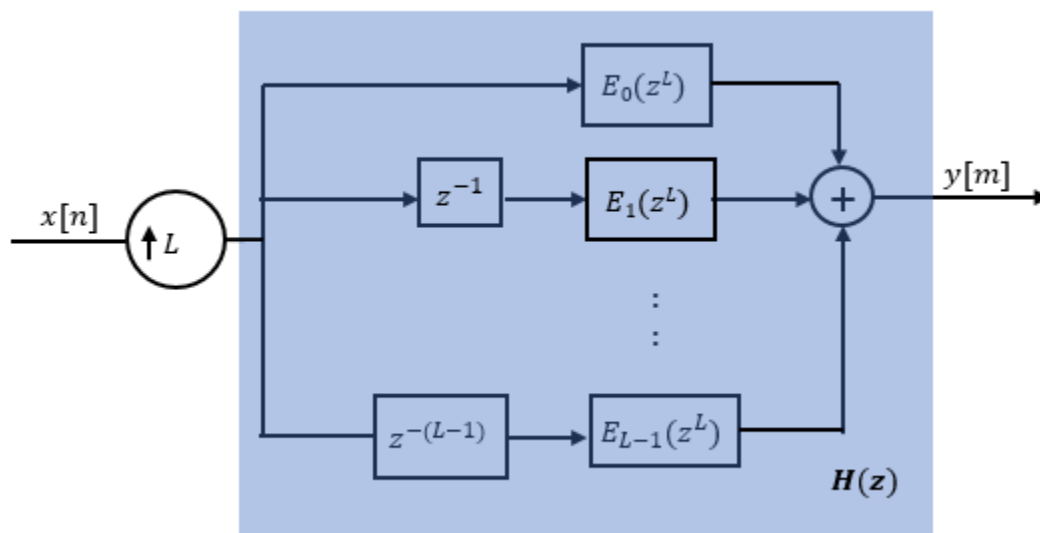
$E_0(z^L), E_1(z^L), \dots, E_{L-1}(z^L)$  are polyphase components of the FIR filter  $H(z)$ .

Conceptually, the FIR interpolation filter contains an upsampler followed by an FIR lowpass filter  $H(z)$ .

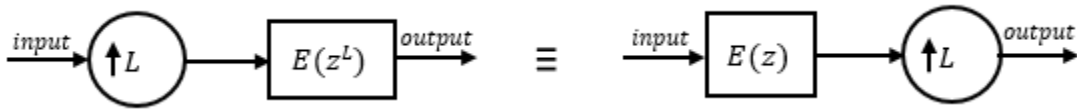


**Conceptual Operation of FIR Interpolator**

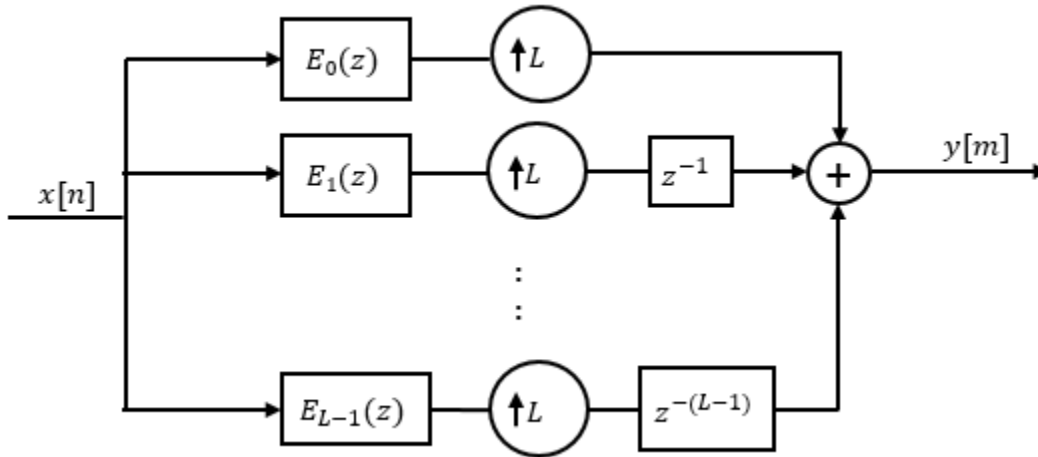
Replace  $H(z)$  with its polyphase representation.



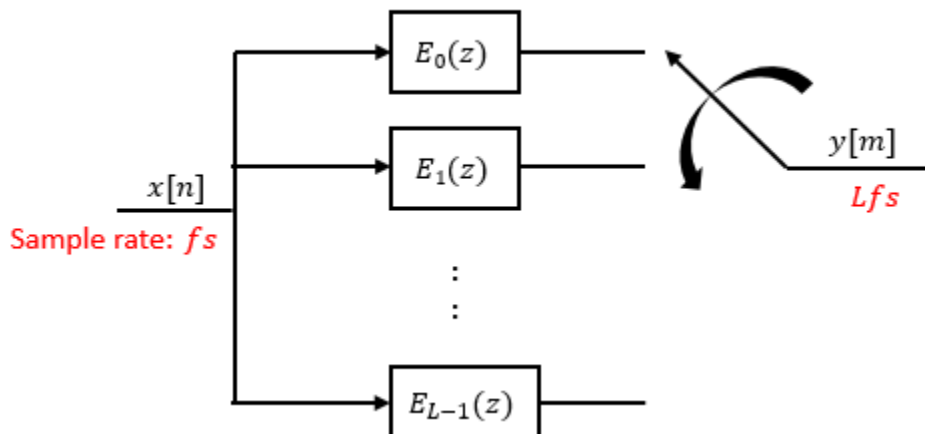
Here is the multirate noble identity for interpolation.



Applying the noble identity for interpolation moves the upsampling operation to after the filtering operation. This move enables you to filter the signal at a lower rate.



You can replace the upsampling operator, delay block, and adder with a commutator switch. The switch starts on the first branch 0 and moves in the counterclockwise direction, each time receiving one sample from each branch. The interpolator effectively outputs  $L$  samples for every one input sample it receives. Hence the sample rate at the output of the FIR interpolation filter is  $Lfs$ .



## Version History

Introduced before R2006a

**R2023a: Generate SIMD code for FIR Interpolation block on all Intel platforms**

In R2023a, if you have Embedded Coder license, you can generate SIMD code for the FIR Interpolation block on all Intel platforms by using the model configuration parameter **Leverage target hardware instruction set extensions**. Previously, you had to use a code replacement library to generate SIMD code. For more details, see “Code Generation” on page 2-629.

## References

- [1] Fliege, N. J. *Multirate Digital Signal Processing: Multirate Systems, Filter Banks, Wavelets*. West Sussex, England: John Wiley & Sons, 1994.
- [2] Orfanidis, Sophocles J. *Introduction to Signal Processing*. Upper Saddle River, NJ: Prentice-Hall, 1996.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Generated code relies on the `memcpy` or `memset` function (`string.h`) under certain conditions.

If you have an Embedded Coder license, you can generate SIMD code for the FIR Interpolation block.

### Generate SIMD code using Intel AVX2 technology

The FIR Interpolation block supports SIMD code generation using Intel AVX2 technology through a code replacement library under these conditions:

- **Input processing** is set to `Columns as channels (frame based)`.
- **Rate options** is set to `Enforce single-rate processing`.
- Input signal is real-valued with real filter coefficients.
- Input signal is complex-valued with real or complex filter coefficients.
- Input signal has a data type of `single` or `double`.

For more information, see “SIMD Code Generation”.

### Generate SIMD code on all Intel platforms

You can generate SIMD code for the FIR Interpolation block on all Intel platforms by using the model configuration parameter **Leverage target hardware instruction set extensions**. Previously, you had to use a code replacement library to generate SIMD code.

You can generate SIMD code using the **Leverage target hardware instruction set extensions** parameter under these conditions:

- You set **Input processing** to `Columns as channels (frame based)`
- Input signal is real-valued with real filter coefficients
- Data type of the input signal is `single` or `double`

In addition, configure your model appropriately. In the **Modeling** tab of the Simulink model window, click **Model Settings** and configure these parameters under **Code Generation**.

- In the **Optimization** pane:
  - Provide a specific instruction set in the **Leverage target hardware instruction set extensions** parameter.
  - Select the **Optimize reductions** parameter.
  - Under **Optimization levels**, set **Level** to Maximum and **Priority** to Maximize execution speed.
- In the **Interface** pane, under **Software environment**, clear **non-finite numbers**.

For computationally intensive operations on supported blocks, SIMD intrinsics can significantly improve the performance of the generated code on Intel platforms. For more details, see “Optimize Code for Reduction Operations by Using SIMD” (Simulink Coder).

The SIMD technology significantly improves the performance of the generated code.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

---

**Note** For an HDL-optimized filter architecture with hardware-friendly control signals, use the FIR Interpolator block. The DSP HDL Toolbox block simulates the latency of the HDL algorithm in Simulink.

---

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Coder supports **Coefficient source** options **Dialog parameters**, **Filter object**, or **Auto**.

### Block Optimizations

To reduce area or increase speed, the FIR Decimator block supports block-level optimizations.

Right-click on the block or the subsystem to open the corresponding **HDL Properties** dialog box and set optimization properties.

|                        |  |
|------------------------|--|
| Serial Architectures   | When you select Fully Serial architecture, the SerialPartition property is set on the FIR Interpolation Block.   |
| Distributed Arithmetic | Distributed Arithmetic properties <b>DALUTPartition</b> and <b>DARadix</b> are supported for the Distributed Arithmetic (DA) architecture with a default FIR filter structure. structures.   |
| Pipelining             | When you use <b>AddPipelineRegisters</b> , registers are placed based on the filter structure. The pipeline register placement determines the latency.<br><br>A pipeline register is added between levels of a tree-based adder, for a latency of $\text{ceil}(\log_2(PL)) - 1$ , where PL is polyphase filter length. |

### HDL Filter Properties

|                             |  |
|-----------------------------|--|
| <b>AddPipelineRegisters</b> | Insert a pipeline register between stages of computation in a filter. See also AddPipelineRegisters (HDL Coder). |
|-----------------------------|--|

|                                 |  |
|---------------------------------|--|
| <b>CoeffMultipliers</b>         | Specify the use of canonical signed digit (CSD) optimization to decrease filter area by replacing coefficient multipliers with shift-and-add logic. When you choose a fully parallel filter implementation, you can set <b>CoeffMultipliers</b> to <code>csd</code> or <code>factored-csd</code> . The default is <code>multipliers</code> , which retains multipliers in the HDL. See also <code>CoeffMultipliers</code> (HDL Coder). |
| <b>DALUTPartition</b>           | Specify distributed arithmetic partial-product LUT partitions as a vector of the sizes of each partition. The sum of all vector elements must be equal to the filter length. The maximum size for a partition is 12 taps. Set <b>DALUTPartition</b> to a scalar value equal to the filter length to generate DA code without LUT partitions. See also <code>DALUTPartition</code> (HDL Coder).   |
| <b>DARadix</b>                  | Specify how many distributed arithmetic bit sums are computed in parallel. A DA radix of 8 ( $2^3$ ) generates a DA implementation that computes three sums at a time. The default value is $2^1$ , which generates a fully serial DA implementation. See also <code>DARadix</code> (HDL Coder).   |
| <b>MultiplierInputPipeline</b>  | Specify the number of pipeline stages to add at filter multiplier inputs. See also <code>MultiplierInputPipeline</code> (HDL Coder).   |
| <b>MultiplierOutputPipeline</b> | Specify the number of pipeline stages to add at filter multiplier outputs. See also <code>MultiplierOutputPipeline</code> (HDL Coder).   |
| <b>SerialPartition</b>          | Specify partitions for partly serial or cascade-serial filter implementations as a vector of the lengths of each partition. For a fully serial implementation, set this parameter to the length of the filter. See also <code>SerialPartition</code> (HDL Coder).  |

#### HDL Block Properties

|                                  |   |
|----------------------------------|---|
| <b>ConstrainedOutputPipeline</b> | Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ <code>ConstrainedOutputPipeline</code> ” (HDL Coder). |
| <b>InputPipeline</b>             | Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “ <code>InputPipeline</code> ” (HDL Coder).                 |
| <b>OutputPipeline</b>            | Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “ <code>OutputPipeline</code> ” (HDL Coder).               |

#### Restrictions

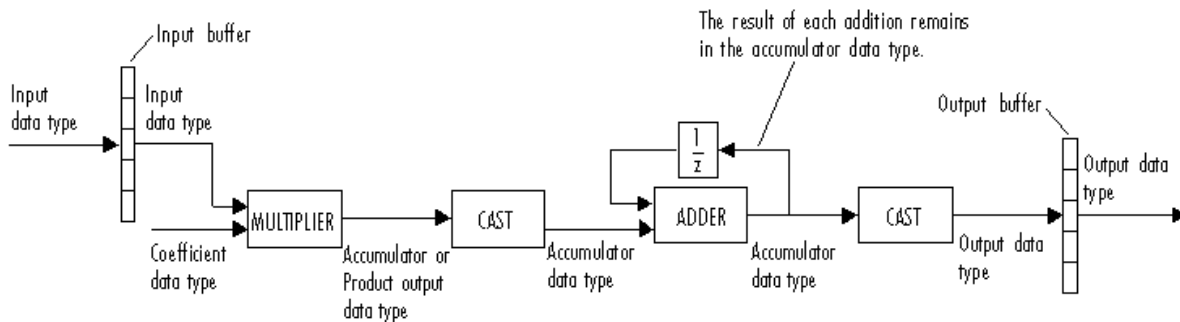
- You must set **Initial conditions** to zero. HDL code generation is not supported for nonzero initial states.
- Vector and frame inputs are not supported for HDL code generation.
- When you select **Dialog parameters**, the following fixed-point options are not supported for HDL code generation:
  - **Coefficients:** Slope and Bias scaling

- **CoeffMultipliers** options are supported only when using a fully parallel architecture. When you select a serial architecture, **CoeffMultipliers** is hidden from the HDL Block Properties dialog box.

### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

The following diagram shows the data types used within the FIR Interpolation block for fixed-point signals.



This diagram shows that input data is stored in the input buffer with the same data type and scaling as the input. The block stores filtered data and any initial conditions in the output buffer using the output data type and scaling that you set in the block dialog box.

When at least one of the inputs to the multiplier is real, the output of the multiplier is in the product output data type. When both inputs to the multiplier are complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication performed by this block, see “Multiplication Data Types”.

---

**Note** When the block input is fixed point, all internal data types are signed fixed point.

---

## See Also

### Functions

`firgr` | `firceqrip` | `firhalfband` | `firnyquist`

### Objects

`dsp.FIRInterpolator` | `dsp.FIRHalfbandInterpolator` | `dsp.CICCompensationInterpolator`

### Blocks

Variable FIR Interpolation | Upsample | FIR Decimation | FIR Rate Conversion | FIR Halfband Interpolator | FIR Halfband Decimator | IIR Halfband Interpolator | IIR Halfband Decimator | CIC Compensation Interpolator | CIC Compensation Decimator | Upsample | CIC Interpolation | Digital Down-Converter | Digital Up-Converter



# FIR Halfband Decimator

Decimate signal using polyphase FIR halfband filter



## Libraries:

DSP System Toolbox / Filtering / Multirate Filters

## Description

The FIR Halfband Decimator block performs polyphase decimation of the input signal by a factor of 2. The block uses an FIR equiripple design or a Kaiser window design to construct the halfband filters. The implementation takes advantage of the zero-valued coefficients of the FIR halfband filter, making one of the polyphase branches a delay. You can use the block to implement the analysis portion of a two-band filter bank to separate a signal into lowpass and highpass subbands. For more information, see “Algorithms” on page 2-639.

The block supports fixed-point operations and ARM® Cortex® code generation. For more information on ARM Cortex code generation, see “Code Generation for ARM Cortex-M and ARM Cortex-A Processors”.

## Ports

### Input

**Input** — Input signal  
column vector | matrix

Specify the input signal as a column vector or a matrix of size  $P$ -by- $Q$ . If the input signal is a matrix, the block treats each column of the matrix as an independent channel. The number of rows in the input signal must be a multiple of 2.

This block supports variable-size input signals.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fixed point

### Output

**LP** — Lowpass subband of decimator output  
column vector | matrix

Lowpass subband of the decimator output, returned as a column vector or a matrix of size  $P/2$ -by- $Q$ . As the filter is a halfband filter, the downsampling factor is always 2.

When the output is fixed-point, it is signed only.

This port is unnamed until you select the **Output highpass subband** parameter.

Data Types: single | double | int8 | int16 | int32 | int64 | fixed point

**HP** — Highpass subband of decimator output  
column vector | matrix

Highpass subband of the decimator output, returned as a column vector or a matrix of size  $P/2$ -by- $Q$ . As the filter is a halfband filter, the downsampling factor is always 2.

When the output is fixed-point, it is signed only.

#### Dependency

To enable this port, select the **Output highpass subband** parameter.

Data Types: single | double | int8 | int16 | int32 | int64 | fixed point

## Parameters

### Main Tab

**Filter specification** — Filter design parameters

Transition width and stopband attenuation (default) | Filter order and transition width | Filter order and stopband attenuation | Coefficients

Select the parameters that the block uses to design the FIR halfband filter.

- Transition width and stopband attenuation (default) — Design the filter using **Transition width (Hz)** and **Stopband attenuation (dB)**. This design is the minimum-order design.
- Filter order and transition width — Design the filter using **Filter order** and **Transition width (Hz)**.
- Filter order and stopband attenuation — Design the filter using **Filter order** and **Stopband attenuation (dB)**.
- Coefficients — Specify the filter coefficients directly through the **Numerator** parameter.

**Filter order** — Filter order

52 (default) | even positive integer

Specify the filter order as an even positive integer.

#### Dependencies

To enable this parameter, set **Filter specification** to Filter order and transition width or Filter order and stopband attenuation.

**Transition width (Hz)** — Transition width

4.1e3 (default) | positive real scalar

Specify the transition width as a real positive scalar in Hz. The transition width must be less than  $1/2$  the sample rate of the input signal.

**Dependencies**

To enable this parameter, set **Filter specification** to Filter order and transition width or Transition width and stopband attenuation.

**Stopband attenuation (dB)** — Stopband attenuation

80 (default) | positive real scalar

Specify the stopband attenuation as a real positive scalar in dB.

**Dependencies**

To enable this parameter, set **Filter specification** to Filter order and stopband attenuation or Transition width and stopband attenuation.

**Numerator** — FIR halfband filter coefficients

`firhalfband('minorder',0.407,1e-4)` (default) | row vector

Specify the FIR halfband filter coefficients directly as a row vector. The coefficients must comply with the FIR halfband impulse response format. If  $(\text{length}(\text{Numerator}) - 1)/2$  is even, where  $(\text{length}(\text{Numerator}) - 1)$  is the filter order, every other coefficient starting with the first coefficient must be 0 except the center coefficient which must be 0.5. If  $(\text{length}(\text{Numerator}) - 1)/2$  is odd, the sequence of alternating zeros with 0.5 at the center starts at the second coefficient.

**Dependencies**

To enable this parameter, set **Filter specification** to Coefficients.

**Design method** — Filter design method

Auto (default) | Equiripple | Kaiser

Specify the filter design method as one of the following:

- **Auto** -- The algorithm automatically chooses the filter design method depending on the filter design parameters. The algorithm uses the equiripple or the Kaiser window method to design the filter.

If the design constraints are very tight, such as very high stopband attenuation or very narrow transition width, then the algorithm automatically chooses the Kaiser method, as this method is optimal for designing filters with very tight specifications. However, if the design constraints are not tight, then the algorithm chooses the equiripple method.

When you set the **Design method** parameter to **Auto**, you can determine the method used by the algorithm by examining the passband and stopband ripple characteristics of the designed filter. If the block used the equiripple method, the passband and stopband ripples of the designed filter have a constant amplitude in the frequency response. If the filter design method the block chooses in the **Auto** mode is not suitable for your application, manually specify the **Design method** as **Equiripple** or **Kaiser**.

- **Equiripple** -- The algorithm uses the equiripple method.
- **Kaiser** -- The algorithm uses the Kaiser window method.

**Dependencies**

To enable this parameter, set **Filter specification** to Filter order and stopband attenuation, Filter order and transition width, or Transition width and stopband attenuation.

**Output highpass subband** — Output highpass subband

off (default) | on

When you select this check box, the block acts as an analysis filter bank and analyzes the input signal into highpass and lowpass subbands. When you clear this check box, the block acts as an FIR halfband decimator.

**Inherit sample rate from input** — Inherit sample rate from input signal

off (default) | on

When you select this check box, the block inherits its sample rate from the input signal. When you clear this check box, you specify the sample rate in **Input sample rate (Hz)**.

**Input sample rate (Hz)** — Sample rate of input signal

44100 (default) | positive real scalar

Specify the sample rate of the input signal as a positive scalar in Hz.

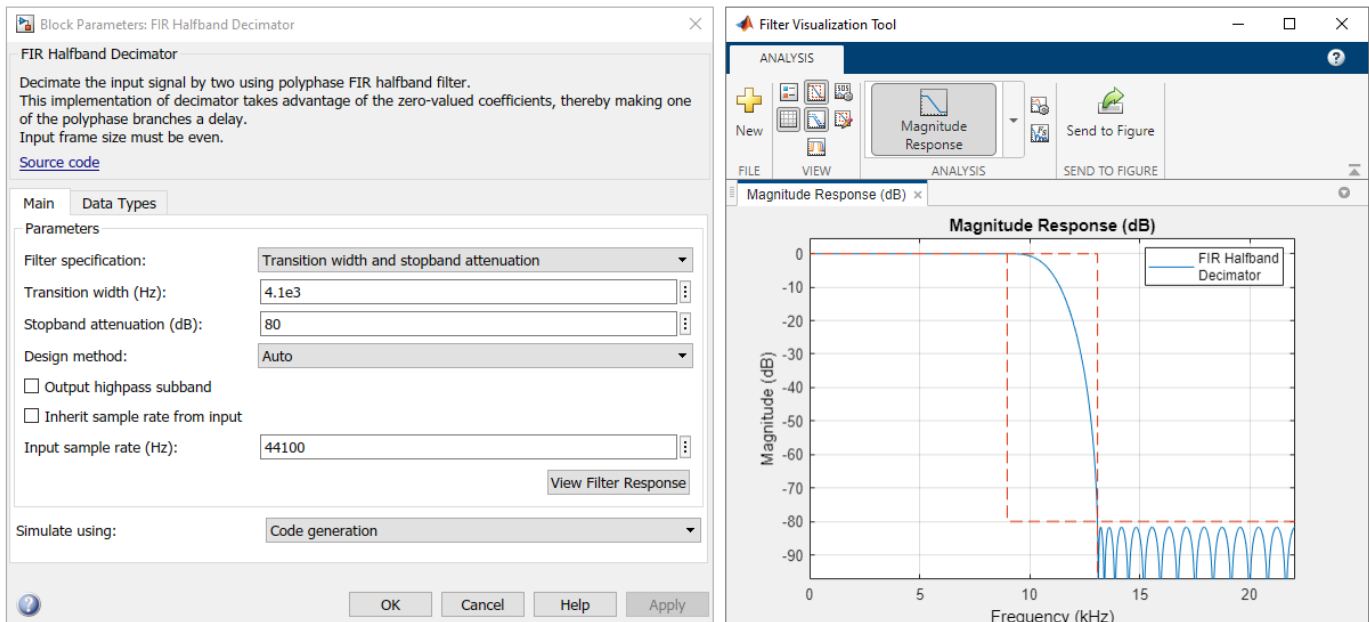
**Dependencies**

To enable this parameter, clear the **Inherit sample rate from input** parameter.

**View Filter Response** — View Filter Response

button

Click this button to open the Filter Visualization Tool (FVTool) and display the magnitude and phase response of the FIR Halfband Decimator. The response is based on the values you specify in the block parameters dialog box. Changes made to these parameters update FVTool.



To update the magnitude response while FVTool is running, modify the dialog box parameters and click **Apply**.

**Simulate using** — Simulate using

Code generation (default) | Interpreted execution

Specify the type of simulation to run. You can set this parameter to:

- Code generation (default)

Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than Interpreted execution.

- Interpreted execution

Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than Code generation.

## Data Types Tab

**Rounding mode** — Rounding mode for output fixed-point operations

Floor (default) | Ceiling | Nearest | Round | Simplest | Zero


Select the rounding mode for output fixed-point operations. The default is Floor.

**Coefficients** — Word and fraction lengths of coefficients

fixdt(1,16) (default) | fixdt(1,16,0)

Specify the fixed-point data type of the coefficients as one of the following:

- `fixdt(1,16)` (default) — Signed fixed-point data type of word length 16 with binary point scaling. The block determines the fraction length automatically from the coefficient values in such a way that the coefficients occupy maximum representable range without overflowing.
- `fixdt(1,16,0)` — Signed fixed-point data type of word length 16 and fraction length 0. You can change the fraction length to any other integer value.
- `<data type expression>` — Specify the coefficients data type by using an expression that evaluates to a data type object. For example, `numericType(fixdt([ ],18,15))`. Specify the sign mode of this data type as `[ ]` or `true`.
- Refresh Data Type — Refresh to the default data type.

Click the **Show data type assistant** button  to display the data type assistant, which helps you set the coefficients data type.

## Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | double   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no                                      |
| <b>Multidimensional Signals</b> | no                                      |
| <b>Variable-Size Signals</b>    | yes                                     |
| <b>Zero-Crossing Detection</b>  | no                                      |

## More About

### Halfband Filters

An ideal lowpass halfband filter is given by

$$h(n) = \frac{1}{2\pi} \int_{-\pi/2}^{\pi/2} e^{j\omega n} d\omega = \frac{\sin(\frac{\pi}{2}n)}{\pi n}.$$

An ideal filter is not realizable because the impulse response is noncausal and not absolutely summable. However, the impulse response of an ideal lowpass filter possesses some important properties that are required in a realizable approximation. The impulse response of an ideal lowpass halfband filter is:

- Equal to 0 for all even-indexed samples.
- Equal to 1/2 at  $n=0$  as shown by L'Hôpital's rule on the continuous-valued equivalent of the discrete-time impulse response

The ideal highpass halfband filter is given by

$$g(n) = \frac{1}{2\pi} \int_{-\pi}^{-\pi/2} e^{j\omega n} d\omega + \frac{1}{2\pi} \int_{\pi/2}^{\pi} e^{j\omega n} d\omega.$$

Evaluating the preceding integral gives the following impulse response

$$g(n) = \frac{\sin(\pi n)}{\pi n} - \frac{\sin(\frac{\pi}{2}n)}{\pi n}.$$

The impulse response of an ideal highpass halfband filter is:

- Equal to 0 for all even-indexed samples
- Equal to 1/2 at  $n=0$

The FIR halfband decimator uses a causal FIR approximation to the ideal halfband response, which is based on minimizing the  $\ell^\infty$  norm of the error (minimax). See “Algorithms” on page 2-639 for more information.

### Kaiser Window

The coefficients of a Kaiser window are computed from this equation:

$$w(n) = \frac{I_0\left(\beta\sqrt{1 - \left(\frac{n - N/2}{N/2}\right)^2}\right)}{I_0(\beta)}, \quad 0 \leq n \leq N,$$

where  $I_0$  is the zeroth-order modified Bessel function of the first kind.

To obtain a Kaiser window that represents an FIR filter with stopband attenuation of  $\alpha$  dB, use this  $\beta$ .

$$\beta = \begin{cases} 0.1102(\alpha - 8.7), & \alpha > 50 \\ 0.5842(\alpha - 21)^{0.4} + 0.07886(\alpha - 21), & 50 \geq \alpha \geq 21 \\ 0, & \alpha < 21 \end{cases}$$

The filter order  $n$  is given by:

$$n = \frac{\alpha - 7.95}{2.285(\Delta\omega)}$$

where  $\Delta\omega$  is the transition width.

## Algorithms

### Filter Design Method

The FIR halfband decimator algorithm uses the equiripple or the Kaiser window method to design the FIR halfband filter. When the design constraints are tight, such as very high stopband attenuation or very narrow transition width, use the Kaiser window method. When the design constraints are not tight, use the equiripple method. If you are not sure of which method to use, set the design method to **Auto**. In this mode, the algorithm automatically chooses a design method that optimally meets the specified filter constraints.

### Halfband Equiripple Design

In the equiripple method, the algorithm uses a minimax (minimize the maximum error) FIR design to design a fullband linear phase filter with the desired specifications. The algorithm upsamples a fullband filter to replace the even-indexed samples of the filter with zeros and creates a halfband filter. It then sets the filter tap corresponding to the group delay of the filter in samples to 1/2. This yields a causal linear-phase FIR filter approximation to the ideal halfband filter defined in “Halfband

Filters” on page 2-638. See [1] for a description of this filter design method using the Remez exchange algorithm. Since you can design a filter using this approximation method with a constant ripple both in the passband and stopband, the filter is also known as the equiripple filter.

### Kaiser Window Design

In the Kaiser window method, the algorithm first truncates the ideal halfband filter defined in “Halfband Filters” on page 2-638, then it applies a Kaiser window defined in “Kaiser Window” on page 2-639. This yields a causal linear-phase FIR filter approximation to the ideal halfband filter.

For more information on designing FIR halfband filters, see “FIR Halfband Filter Design”.

### Polyphase Implementation with Halfband Filters

The FIR halfband decimator uses an efficient polyphase implementation for halfband filters when you filter the input signal. The chief advantage of the polyphase implementation is that you can downsample the signal prior to filtering. This allows you to filter at the lower sampling rate.

Splitting a filter’s impulse response  $h(n)$  into two polyphase components results in an even polyphase component with  $z$ -transform of

$$H_0(z) = \sum_n h(2n)z^{-n},$$

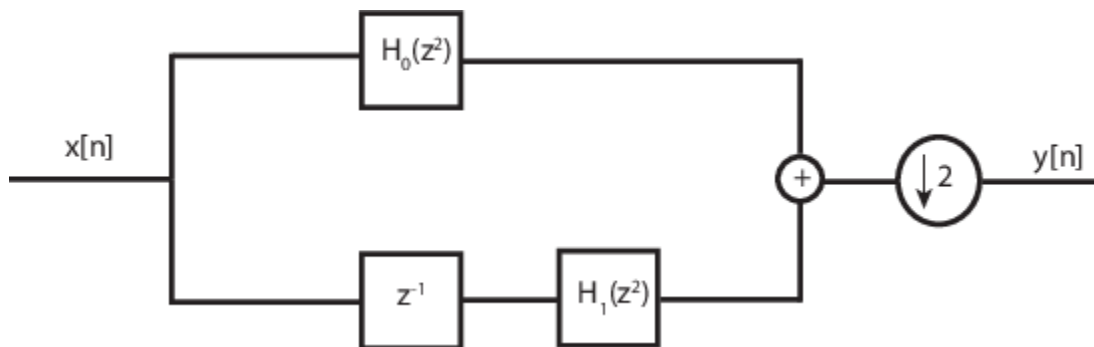
and an odd polyphase component with  $z$ -transform of

$$H_1(z) = \sum_n h(2n + 1)z^{-n}.$$

The  $z$ -transform of the filter can be written in terms of the even and odd polyphase components as

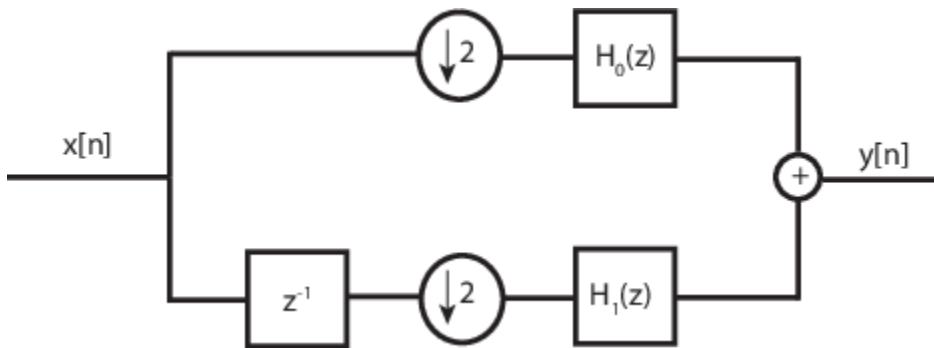
$$H(z) = H_0(z^2) + z^{-1}H_1(z^2).$$

You can represent filtering the input signal and then downsampling it by 2 using this figure.

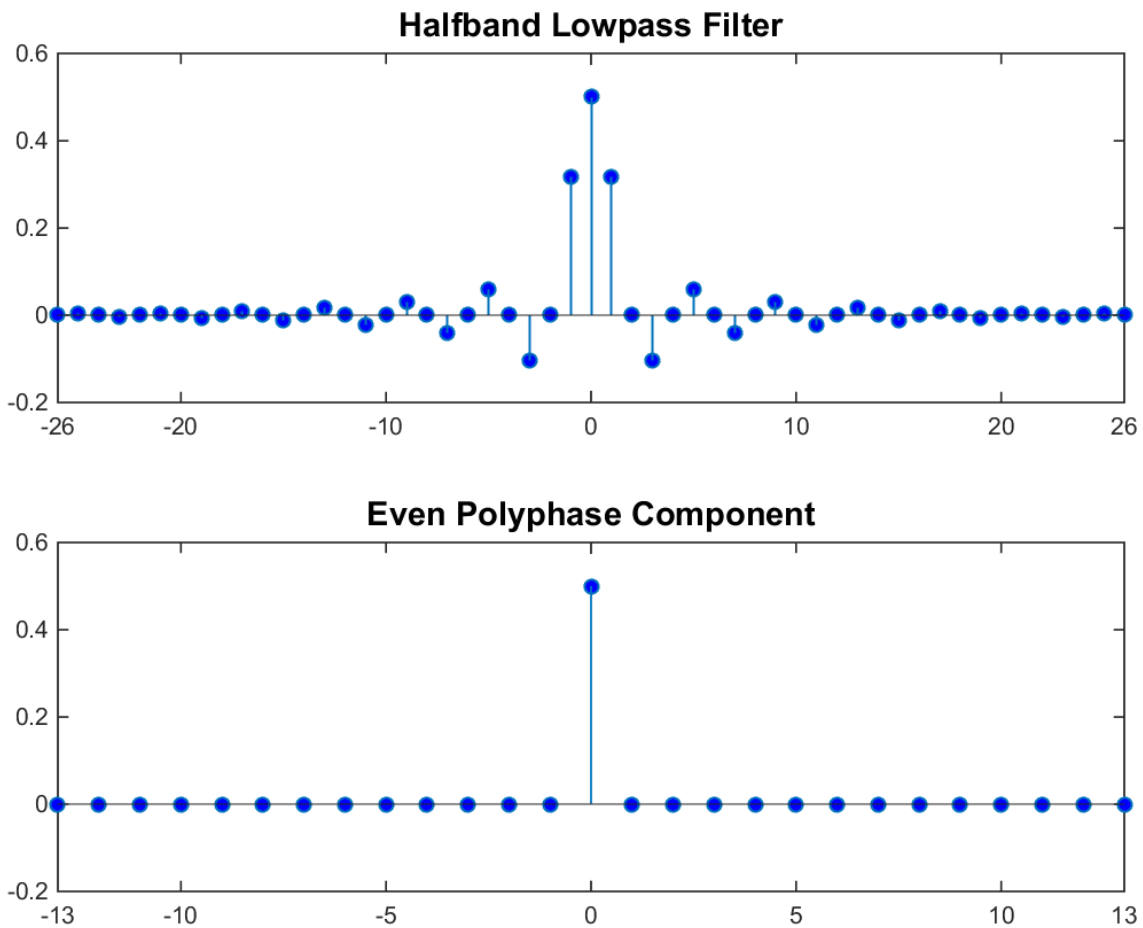


Using the multirate noble identity for downsampling, you can move the downsampling operation before the filtering. This allows you to filter at the lower rate.



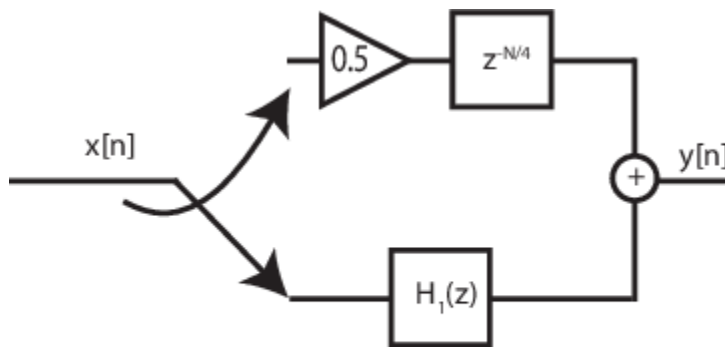


For a halfband filter, the only nonzero coefficient in the even polyphase component is the coefficient corresponding to  $z^0$ . Implementing the halfband filter as a causal FIR filter shifts the nonzero coefficient to approximately  $z^{-N/4}$  where  $N$  is the number of filter taps. This process is illustrated in the following figure.



The top plot shows a halfband filter of order 52. The bottom plot shows the even polyphase component. Both of these filters are noncausal. Delaying the even polyphase component by 13 samples creates a causal FIR filter.

To efficiently implement the halfband decimator, the algorithm replaces the delay block and downsampling operator with a commutator switch. This is illustrated in the following figure where one polyphase component is replaced by a gain and delay.



The commutator switch takes input samples from a single branch and supplies every other sample to one of the two polyphase components for filtering. This halves the sampling rate of the input signal. Which polyphase component reduces to a simple delay depends on whether the half order of the filter is even or odd. This is because the delay required to make the even polyphase component causal can be odd or even depending on the filter half order.

To confirm this behavior, run the following code in the MATLAB command prompt and inspect the polyphase components of the following filters.

```
filterspec = "Filter order and stopband attenuation";
halfOrderEven = dsp.FIRHalfbandDecimator(Specification=filterspec,...
    FilterOrder=64,StopbandAttenuation=80,DesignMethod="Auto");
halfOrderOdd = dsp.FIRHalfbandDecimator(Specification=filterspec,...
    FilterOrder=54,StopbandAttenuation=80,DesignMethod="Auto");
polyphase(halfOrderEven)
polyphase(halfOrderOdd)
```

To summarize, the FIR halfband decimator:

- Decimates the input prior to filtering and filters the even and odd polyphase components of the input separately with the even and odd polyphase components of the filter.
- Exploits the fact that one filter polyphase component is a simple delay for a halfband filter.

## Version History

Introduced in R2015b

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

## See Also

### Objects

`dsp.FIRHalfbandInterpolator` | `dsp.FIRHalfbandDecimator`

### Blocks

FIR Halfband Interpolator | IIR Halfband Decimator | CIC Compensation Decimator

# FIR Halfband Interpolator

Interpolate signal using polyphase FIR half band filter



## Libraries:

DSP System Toolbox / Filtering / Multirate Filters

## Description

The FIR Halfband Interpolator block performs interpolation of the input signal by a factor of two. The block uses an FIR equiripple design or a Kaiser window design to construct the halfband filters. The block uses an efficient polyphase implementation to filter the input. The implementation takes advantage of the zero-valued coefficients of the FIR halfband filter, making one of the polyphase branches a delay. For more information, see “Algorithms” on page 2-650. You can also use this block to implement the synthesis portion of a two-band filter bank to synthesize a signal from lowpass and highpass subbands.

The input signal can be a real- or complex-valued column vector or a matrix. If the input signal is a matrix, the block treats each column of the matrix as an independent channel. The block supports fixed-point operations and ARM Cortex code generation. For more information on ARM Cortex code generation, see “Code Generation for ARM Cortex-M and ARM Cortex-A Processors”.

The block also supports SIMD code generation. For details, see “Code Generation” on page 2-654.

## Ports

### Input

#### LP — Data input

column vector | matrix

Specify the data input as a vector or a matrix. If the input signal is a matrix, the block treats each column of the matrix as an independent channel.

When you select the **Input highpass subband** parameter, this block acts as a halfband synthesis filter bank. The input at this port is then the lowpass subband output of a halfband analysis filter bank.

This port is unnamed until you select the **Input highpass subband** parameter.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fixed point

#### HP — Second input to the synthesis filter bank

column vector | matrix

Specify the second input to the synthesis filter bank as a column vector or a matrix. This signal is the highpass subband output of a halfband analysis filter bank. If the input signal is a matrix, the block treats each column of the matrix as an independent channel.

The size, data type, and complexity of both the inputs must be the same.

### Dependency

To enable this port, select the **Input highpass subband** parameter.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `fixed point`

### Output

**Output** — Output of FIR halfband interpolator  
column vector | matrix

Output of the FIR halfband interpolator, returned as a column vector or a matrix. The number of rows in the interpolator output is twice the number of rows in the input signal.

When the output is fixed-point, it is signed only.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `fixed point`

## Parameters

### Main Tab

**Filter specification** — Filter design parameters

Transition width and stopband attenuation (default) | Filter order and transition width | Filter order and stopband attenuation | Coefficients

Select the parameters that the block uses to design the FIR halfband filter.

- Transition width and stopband attenuation (default) — Design the filter using **Transition width (Hz)** and **Stopband attenuation (dB)**. This design is the minimum order design.
- Filter order and transition width — Design the filter using **Filter order** and **Transition width (Hz)**.
- Filter order and stopband attenuation — Design the filter using **Filter order** and **Stopband attenuation (dB)**.
- Coefficients — Specify the filter coefficients directly through the **Numerator** parameter.

**Transition width (Hz)** — Transition width in Hz

4.1e3 | positive real scalar

Specify the transition width as a real positive scalar in Hz. The transition width must be less than 1/2 the sample rate of the input signal.

### Dependencies

To enable this parameter, set **Filter specification** to Filter order and transition width or Transition width and stopband attenuation.

**Filter order** — Filter order

52 | even positive integer

Specify the filter order as an even positive integer.

#### Dependencies

To enable this parameter, set **Filter specification** to Filter order and transition width or Filter order and stopband attenuation.

#### Stopband attenuation (dB) — Stopband attenuation

80 (default) | positive real scalar

Specify the stopband attenuation as a real positive scalar in dB.

#### Dependencies

To enable this parameter, set **Filter specification** to Filter order and stopband attenuation or Transition width and stopband attenuation.

#### Numerator — FIR halfband filter coefficients

`2*firhalfband("minorder",0.407,1e-4)` (default) | row vector

Specify the FIR halfband filter coefficients directly as a row vector. The coefficients must comply with the FIR halfband impulse response format. If  $(\text{length}(\text{Numerator}) - 1)/2$  is even, where  $(\text{length}(\text{Numerator}) - 1)$  is the filter order, every other coefficient starting with the first coefficient must be 0 except the center coefficient which must be 1.0. If  $(\text{length}(\text{Numerator}) - 1)/2$  is odd, the sequence of alternating zeros with 1.0 at the center starts at the second coefficient.

#### Dependencies

To enable this parameter, set **Filter specification** to Coefficients.

#### Design method — Filter design method

Auto (default) | Equiripple | Kaiser

Specify the filter design method as one of the following:

- **Auto** -- The algorithm automatically chooses the filter design method depending on the filter design parameters. The algorithm uses the equiripple or the Kaiser window method to design the filter.

If the design constraints are very tight, such as very high stopband attenuation or very narrow transition width, then the algorithm automatically chooses the Kaiser method, as this method is optimal for designing filters with very tight specifications. However, if the design constraints are not tight, then the algorithm chooses the equiripple method.

When you set the **Design method** parameter to **Auto**, you can determine the method used by the algorithm by examining the passband and stopband ripple characteristics of the designed filter. If the block used the equiripple method, the passband and stopband ripples of the designed filter have a constant amplitude in the frequency response. If the filter design method the block chooses in the **Auto** mode is not suitable for your application, manually specify the **Design method** as **Equiripple** or **Kaiser**.

- **Equiripple** -- The algorithm uses the equiripple method.
- **Kaiser** -- The algorithm uses the Kaiser window method.

**Dependencies**

To enable this parameter, set **Filter specification** to Filter order and stopband attenuation, Filter order and transition width, or Transition width and stopband attenuation.

**Input highpass subband** — Input highpass subband

off (default) | on

When you select this check box, the block acts as a synthesis filter bank. The block accepts two inputs to synthesize: lowpass and highpass subbands. When you clear this check box, the block acts as an IIR half band interpolator and accepts a single vector or matrix as input. By default, this check box is cleared.

**Inherit sample rate from input** — Inherit sample rate from input signal

off (default) | on

When you select this check box, the block inherits its sample rate from the input signal. When you clear this check box, you specify the sample rate in **Input Sample Rate (Hz)**.

**Input sample rate (Hz)** — Sample rate of input signal

44100 (default) | positive real scalar

Specify the sample rate of the input signal as a scalar in Hz.

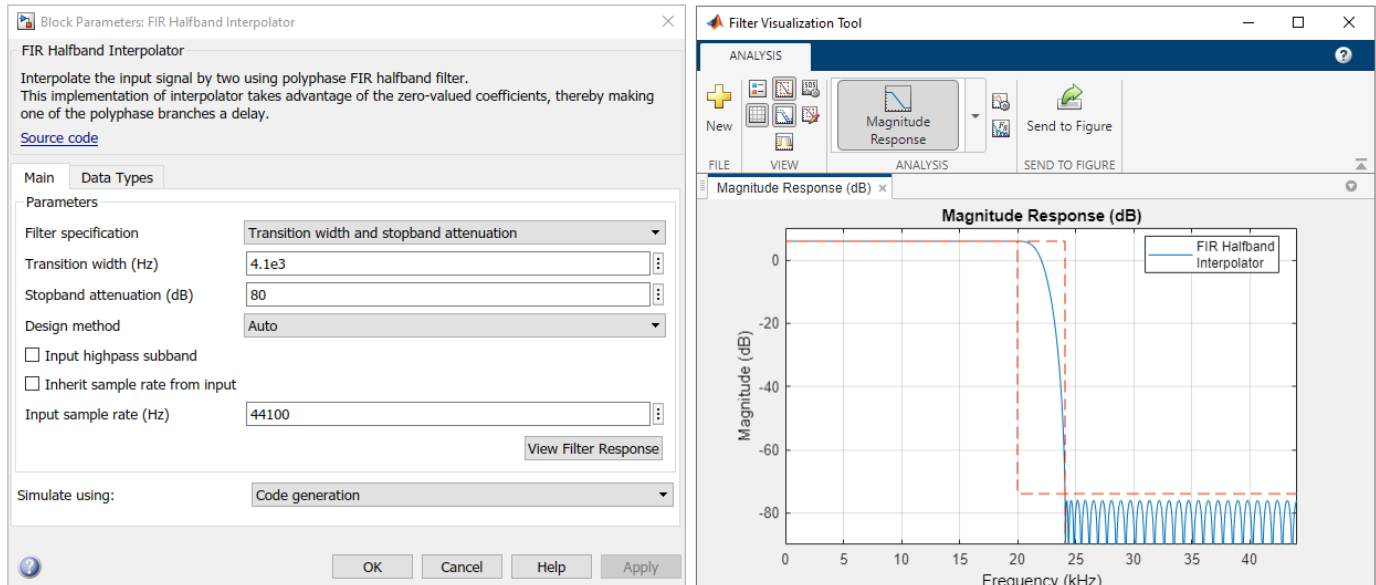
**Dependencies**

To enable this parameter, clear the **Inherit sample rate from input** parameter.

**View Filter Response** — View Filter Response

button

Click this button to open the Filter Visualization Tool (FVTool) and display the magnitude and phase response of the FIR Halfband Interpolator. The response is based on the values you specify in the block parameters dialog box. Changes made to these parameters update FVTool.



To update the magnitude response while FVTool is running, modify the dialog box parameters and click **Apply**.

**Simulate using** — Simulate using

Code generation (default) | Interpreted execution

Specify the type of simulation to run. You can set this parameter to:

- Code generation (default)

Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than Interpreted execution.

- Interpreted execution

Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than Code generation.

### Data Types Tab

**Rounding mode** — Rounding mode for output fixed-point operations

Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Specify the rounding method for the output fixed-point operations. The rounding methods are Ceiling, Convergent, Floor, Nearest, Round, Simplest, and Zero. The default is Floor.


**Coefficients** — Word and fraction lengths of coefficients

fixdt(1,16) (default) | fixdt(1,16,0)

Specify the fixed-point data type of the coefficients as one of the following:



- `fixdt(1,16)` (default) — Signed fixed-point data type of word length 16 with binary point scaling. The block determines the fraction length automatically from the coefficient values in such a way that the coefficients occupy maximum representable range without overflowing.
- `fixdt(1,16,0)` — Signed fixed-point data type of word length 16 and fraction length, 0. You can change the fraction length to any other integer value.
- `<data type expression>` — Specify the coefficients data type by using an expression that evaluates to a data type object. For example, `numericType(fixdt([ ],18,15))`. Specify the sign mode of this data type as `[ ]` or `true`.
- Refresh Data Type — Refresh to the default data type.

Click the **Show data type assistant** button  to display the data type assistant, which helps you set the coefficients data type.

## Block Characteristics

|                                 |                               |
|---------------------------------|-------------------------------|
| <b>Data Types</b>               | double   fixed point   single |
| <b>Direct Feedthrough</b>       | no                            |
| <b>Multidimensional Signals</b> | no                            |
| <b>Variable-Size Signals</b>    | yes                           |
| <b>Zero-Crossing Detection</b>  | no                            |

## More About

### Halfband Filters

An ideal lowpass halfband filter is given by

$$h(n) = \frac{1}{2\pi} \int_{-\pi/2}^{\pi/2} e^{j\omega n} d\omega = \frac{\sin(\frac{\pi}{2}n)}{\pi n}.$$

An ideal filter is not realizable because the impulse response is noncausal and not absolutely summable. However, the impulse response of an ideal lowpass filter possesses some important properties that are required in a realizable approximation. The impulse response of an ideal lowpass halfband filter is:

- Equal to 0 for all even-indexed samples
- Equal to 1/2 at  $n=0$  as shown by L'Hôpital's rule on the continuous-valued equivalent of the discrete-time impulse response

The ideal highpass halfband filter is given by

$$g(n) = \frac{1}{2\pi} \int_{-\pi}^{-\pi/2} e^{j\omega n} d\omega + \frac{1}{2\pi} \int_{\pi/2}^{\pi} e^{j\omega n} d\omega.$$

Evaluating the preceding integral gives the following impulse response

$$g(n) = \frac{\sin(\pi n)}{\pi n} - \frac{\sin(\frac{\pi}{2}n)}{\pi n}.$$

The impulse response of an ideal highpass halfband filter is:

- Equal to 0 for all even-indexed samples
- Equal to 1/2 at  $n=0$

The FIR halfband interpolator uses a causal FIR approximation to the ideal halfband response, which is based on minimizing the  $\ell^\infty$  norm of the error (minimax). See “Algorithms” on page 2-650 for more information.

### Kaiser Window

The coefficients of a Kaiser window are computed from this equation:

$$w(n) = \frac{I_0\left(\beta\sqrt{1 - \left(\frac{n - N/2}{N/2}\right)^2}\right)}{I_0(\beta)}, \quad 0 \leq n \leq N,$$

where  $I_0$  is the zeroth-order modified Bessel function of the first kind.

To obtain a Kaiser window that represents an FIR filter with stopband attenuation of  $\alpha$  dB, use this  $\beta$ .

$$\beta = \begin{cases} 0.1102(\alpha - 8.7), & \alpha > 50 \\ 0.5842(\alpha - 21)^{0.4} + 0.07886(\alpha - 21), & 50 \geq \alpha \geq 21 \\ 0, & \alpha < 21 \end{cases}$$

The filter order  $n$  is given by:

$$n = \frac{\alpha - 7.95}{2.285(\Delta\omega)}$$

where  $\Delta\omega$  is the transition width.

## Algorithms

### Filter Design Method

The FIR halfband interpolator algorithm uses the equiripple or the Kaiser window method to design the FIR halfband filter. When the design constraints are tight, such as very high stopband attenuation or very narrow transition width, use the Kaiser window method. When the design constraints are not tight, use the equiripple method. If you are not sure of which method to use, set the design method to **Auto**. In this mode, the algorithm automatically chooses a design method that optimally meets the specified filter constraints.

### Halfband Equiripple Design

In the equiripple method, the algorithm uses a minimax (minimize the maximum error) FIR design to design a fullband linear phase filter with the desired specifications. The algorithm upsamples a fullband filter to replace the even-indexed samples of the filter with zeros and creates a halfband filter. It then sets the filter tap corresponding to the group delay of the filter in samples to 1/2. This yields a causal linear-phase FIR filter approximation to the ideal halfband filter defined in “Halfband

Filters” on page 2-649. See [1] for a description of this filter design method using the Remez exchange algorithm. Since you can design a filter using this approximation method with a constant ripple both in the passband and stopband, the filter is also known as the equiripple filter.

### Kaiser Window Design

In the Kaiser window method, the algorithm first truncates the ideal halfband filter defined in “Halfband Filters” on page 2-649, then it applies a Kaiser window defined in “Kaiser Window” on page 2-650. This yields a causal linear-phase FIR filter approximation to the ideal halfband filter.

The coefficients of the designed halfband interpolation filter are scaled by the interpolation factor, two, to preserve the output power of the signal.

For more information on designing FIR halfband filters, see “FIR Halfband Filter Design”.

### Polyphase Implementation with Halfband Filters

The FIR halfband interpolator uses an efficient polyphase implementation for halfband filters when you filter the input signal. You can use a polyphase implementation to move the upsampling operation after filtering. This allows you to filter at the lower sampling rate.

Splitting a filter’s impulse response  $h(n)$  into two polyphase components results in an even polyphase component with  $z$ -transform of

$$H_0(z) = \sum_n h(2n)z^{-n},$$

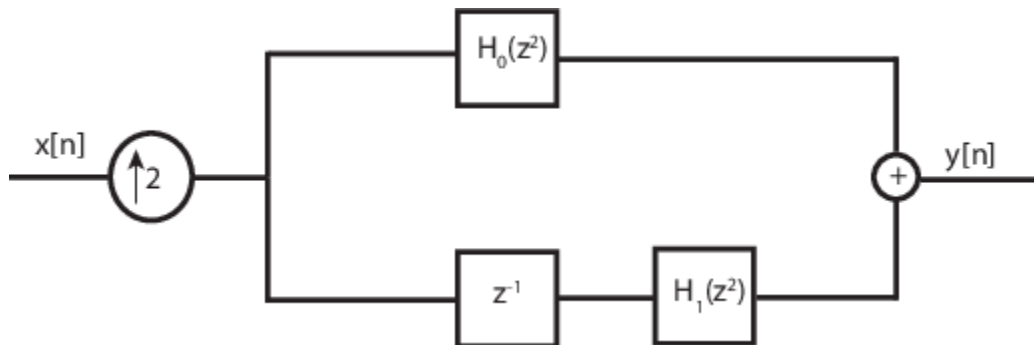
and an odd polyphase component with  $z$ -transform of

$$H_1(z) = \sum_n h(2n + 1)z^{-n}.$$

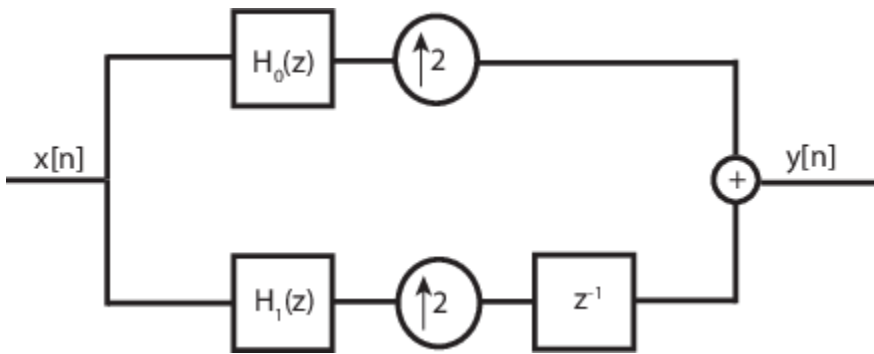
The  $z$ -transform of the filter can be written in terms of the even and odd polyphase components as

$$H(z) = H_0(z^2) + z^{-1}H_1(z^2).$$

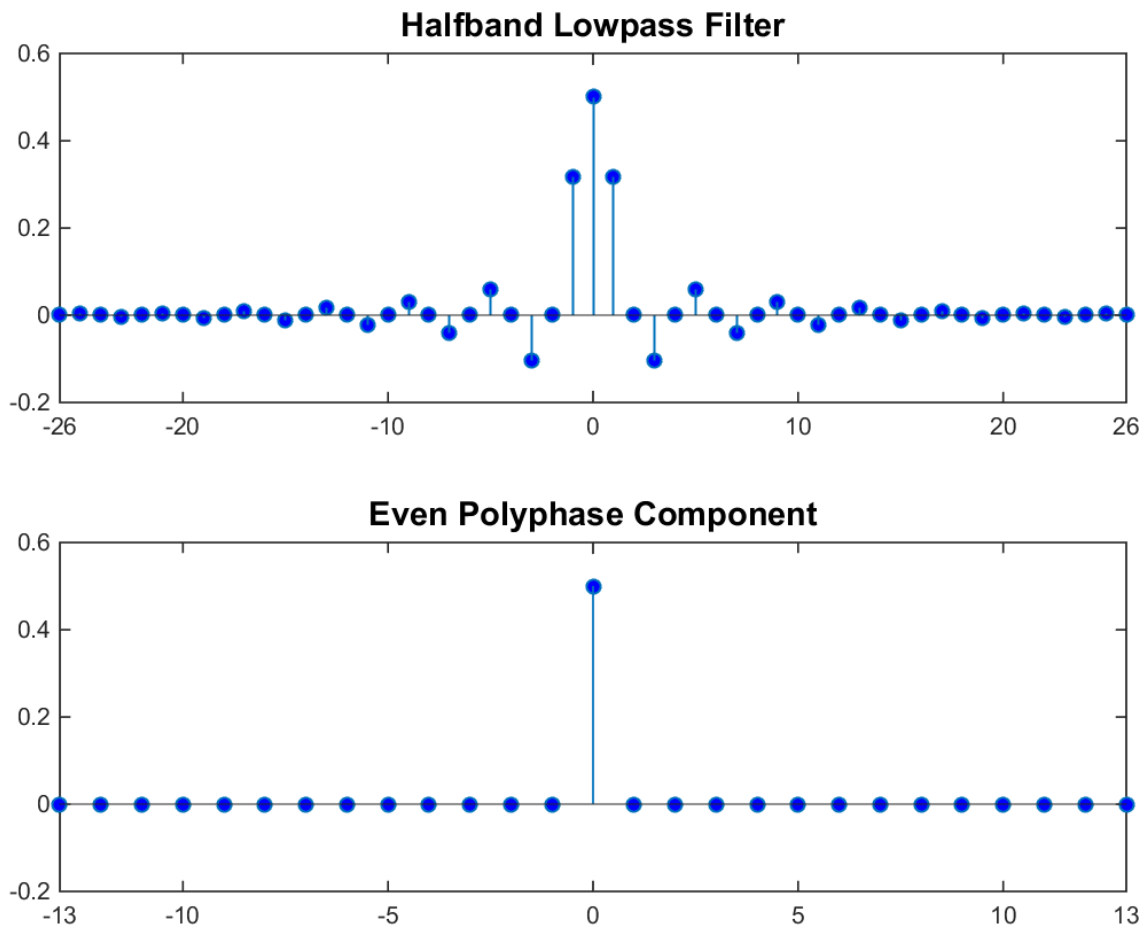
You can represent the upsampling by 2 and then filtering the signal using this figure.



Using the multirate noble identity for upsampling, you can move the upsampling operation after the filtering. This enables you to filter at the lower rate.

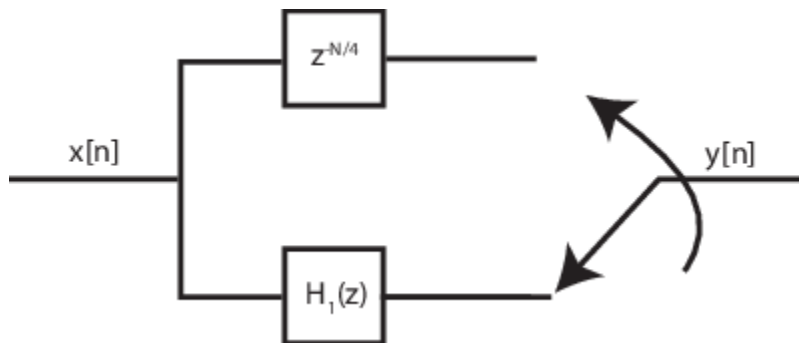


For a halfband filter, the only nonzero coefficient in the even polyphase component is the coefficient corresponding to  $z^0$ . Implementing the halfband filter as a causal FIR filter shifts the nonzero coefficient to approximately  $z^{-N/4}$ , where  $N$  is the number of filter taps. This process is shown in the following figure.



The top plot shows a halfband filter of order 52. The bottom plot shows the even polyphase component. Both filters are noncausal. Delaying the even polyphase component by 13 samples creates a causal FIR filter.

To efficiently implement the halfband interpolator, the algorithm replaces the upsampling operator, delay block, and adder with a commutator switch. This is shown in the following figure, where one polyphase component is replaced by a delay.



The commutator switch takes input samples from the two branches alternately, one sample at a time. This doubles the sampling rate of the input signal. The polyphase component that reduces to a simple delay depends on whether the half order of the filter is even or odd. This is because the delay required to make the even polyphase component causal can be odd or even, depending on the filter half order.

To confirm this behavior, run the following code in the MATLAB command prompt and inspect the polyphase components of the following filters.

```
filterspec = "Filter order and stopband attenuation";
halfOrderEven = dsp.FIRHalfbandInterpolator(Specification=filterspec,...
    FilterOrder=64,StopbandAttenuation=80);
halfOrderOdd = dsp.FIRHalfbandInterpolator(Specification=filterspec,...
    FilterOrder=54,StopbandAttenuation=80);
polyphase(halfOrderEven)
polyphase(halfOrderOdd)
```

One of the polyphase components has a single nonzero coefficient indicating that it is a simple delay. To preserve the output power of the signal, the coefficients are scaled by the interpolation factor, two. To see this scaling, compare the polyphase components of a halfband interpolator with the coefficients of a halfband decimator.

```
hfirinterp = dsp.FIRHalfbandInterpolator;
hfirdecim = dsp.FIRHalfbandDecimator;
polyphase(hfirdecim)
polyphase(hfirinterp)
```

To summarize, the FIR halfband interpolator:

- Filters the input before upsampling with the even and odd polyphase components of the filter.
- Exploits the fact that one filter polyphase component is a simple delay for a halfband filter.

## Version History

Introduced in R2015b

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

The FIR Halfband Interpolator block supports SIMD code generation using Intel AVX2 technology when the data type of the input signal is `single` or `double`.

The SIMD technology significantly improves the performance of the generated code.

### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

## See Also

### Objects

`dsp.FIRHalfbandInterpolator` | `dsp.FIRHalfbandDecimator`

### Blocks

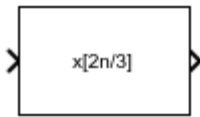
FIR Halfband Decimator | IIR Halfband Interpolator | CIC Compensation Interpolator

### Topics

“FIR Halfband Filter Design”

# FIR Rate Conversion

Perform polyphase FIR sample rate conversion



## Libraries:

DSP System Toolbox / Filtering / Multirate Filters

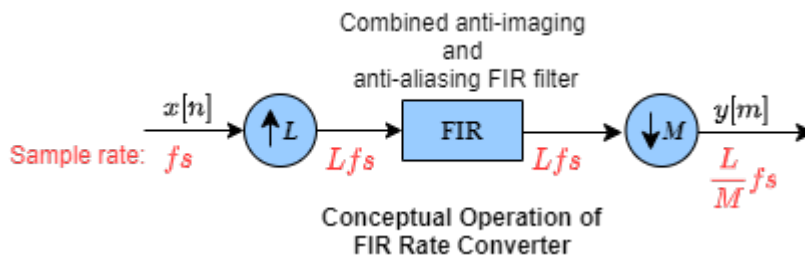
## Description

The FIR Rate Conversion block performs an efficient polyphase sample rate conversion using a rational factor  $L/M$  along the first dimension. The block treats each column of the input as a separate channel and resamples the data in each channel independently over time.

Conceptually, the rate converter combines an FIR interpolator followed by an FIR decimator. The following schematic contains an upsampler, a combined anti-imaging and anti-aliasing FIR filter, and a downsampler. To design an FIR filter which acts as a combined anti-imaging and anti-aliasing FIR filter, use the `designMultirateFIR` function.

The rate converter does the following:

- 1 Upsamples the input to a higher rate by inserting  $L-1$  zeros between input samples.
- 2 Passes the upsampled data through an FIR filter.
- 3 Downsamples the filtered data to a lower rate by discarding  $M-1$  consecutive samples following each sample that the block retains.



Note that the actual block algorithm implements a polyphase structure, an efficient equivalent of the combined system depicted in the diagram. For more details, see “Algorithms” on page 2-667.

## Ports

### Input

**Port\_1(In1)** — Input signal  
vector | matrix

Specify the input signal as a vector or a matrix of size  $P$ -by- $Q$ . The input columns represent the  $Q$  independent channels.

The block supports variable-size input signals (frame length changes during simulation) when you set **Rate options** to `Enforce single-rate processing`. When the block accepts variable-size input

signals, they can be of arbitrary frame length, that is, the input frame length does not have to be a multiple of the decimation factor. For fixed-size signals, the frame length can be arbitrary under certain conditions. For more details, see “Frame-Based Processing” on page 2-666.

When the block input is fixed point, all internal data types are signed fixed point.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fixed point`

## Output

**Port\_1(Out1)** — Rate converted signal  
vector | matrix

Rate converted signal, returned as a vector or a matrix.

When you set the **Rate options** parameter to:

- **Enforce single-rate processing** -- The block maintains the input sample rate and resamples the signal by changing the output frame size by a factor of  $L/M$ .

The output has an upper bound size of  $\text{ceil}(LP/M)$ -by- $Q$  for an input of size  $P$ -by- $Q$ .

- **Allow multirate processing** -- The block decimates the signal such that the output signal sample rate is  $L/M$  times the input sample rate.

The output frame size is the same as the input frame size,  $F_o = (L/M) \times F_i$

All blocks connected to the output operate at  $F_o$ , and all blocks connected to the input operate at  $F_i$ .

For more details, see “Frame-Based Processing” on page 2-666.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fixed point`

## Parameters

**Coefficient source** — Mode of operation  
`Auto` (default) | `Dialog parameters` | `Filter object`

The FIR Rate Conversion block can operate in three different modes. Select the mode in the **Coefficient source** group box.

- **Dialog parameters** — Enter information about the filter, such as FIR filter coefficients in the block dialog box.
- **Filter object** — Specify the filter using a `dsp.FIRRateConverter` System object.
- **Auto** (default) — The block determines the filter coefficients.

The settings in the FIR Rate Conversion block dialog box change based on the mode selected.

### Main Tab

**Interpolation factor** — Interpolation factor  
3 (default) | positive integer



Specify the interpolation factor,  $L$ , as a positive integer. The block upsamples the signal by this value before filtering it.

#### Dependencies

To enable this parameter, set **Coefficient source** to either **Dialog parameters** or **Auto**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**FIR filter coefficients** — FIR filter coefficients  
`designMultirateFIR(3,2)` (default) | `vector`

Specify the FIR filter coefficients in descending powers of  $z$ . By default, the block uses the `designMultirateFIR(3,2)` function to compute the filter coefficients.

#### Dependencies

To enable this parameter, set **Coefficient source** to **Dialog parameters**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**Decimation factor** — Decimation factor  
 2 (default) | `positive integer`

Specify the decimation factor,  $M$ , as a positive integer. The block downsamples the signal by this value after filtering it.

#### Dependencies

To enable this parameter, set **Coefficient source** to either **Dialog parameters** or **Auto**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**Rate options** — Enforce single-rate or allow multirate processing  
`Enforce single-rate processing` (default) | `Allow multirate processing`

Specify whether to enforce single-rate processing or allow multirate processing.

- **Enforce single-rate processing** -- The output frame size  $P_o$  is  $L/M$  times the input frame size  $P_i$ , where  $L$  is the interpolation factor and  $M$  is the decimation factor.

$$P_o = (L/M) \times P_i$$

The output signal rate in Simulink equals the input signal rate.

$$F_o = F_i$$

- **Allow multirate processing** -- The output frame size equals the input frame size.

$$P_o = P_i$$

The output signal rate in Simulink is  $L/M$  times the input signal rate.

$$F_o = (L/M) \times F_i$$

All blocks connected to the output operate at  $F_o$ , and all blocks connected to the input operate at  $F_i$ .

### Filter object — Filter object

FRC (default) | `dsp.FIRRateConverter` System object

Specify the multirate filter object that you want the block to implement. The specified filter object must be a `dsp.FIRRateConverter` System object.

You can define the System object in the block mask or in a MATLAB workspace variable.

For information on creating System objects, see “Define Basic System Objects”.

### Dependencies

This parameter appears when **Coefficient source** is set to **Filter object**.

### Allow arbitrary frame length for fixed-size input signals — Allow arbitrary frame length for fixed-size input signals

off (default) | on

Specify whether fixed-size input signals (whose size does not change during simulation) can have an arbitrary frame length, where the frame length does not have to be a multiple of the decimation factor. The block uses this parameter setting only for fixed-size input signals and ignores this parameter if the input has a variable-size.

When the input signal is a variable-size signal, the signal can have arbitrary frame length, that is, the frame length does not have to be a multiple of the decimation factor.

For fixed-size input signals, if you:

- Select the **Allow arbitrary frame length for fixed-size input signals** parameter, the frame length of the signal does not have to be a multiple of the decimation factor. If the input is not a multiple of the decimation factor, then the output is generally a variable-size signal. Therefore, to support arbitrary input size, the block must also support variable-size operations, which you can enable by selecting the **Allow arbitrary frame length for fixed-size input signals** parameter.
- Clear the **Allow arbitrary frame length for fixed-size input signals** parameter, the input frame length must be a multiple of the decimation factor.

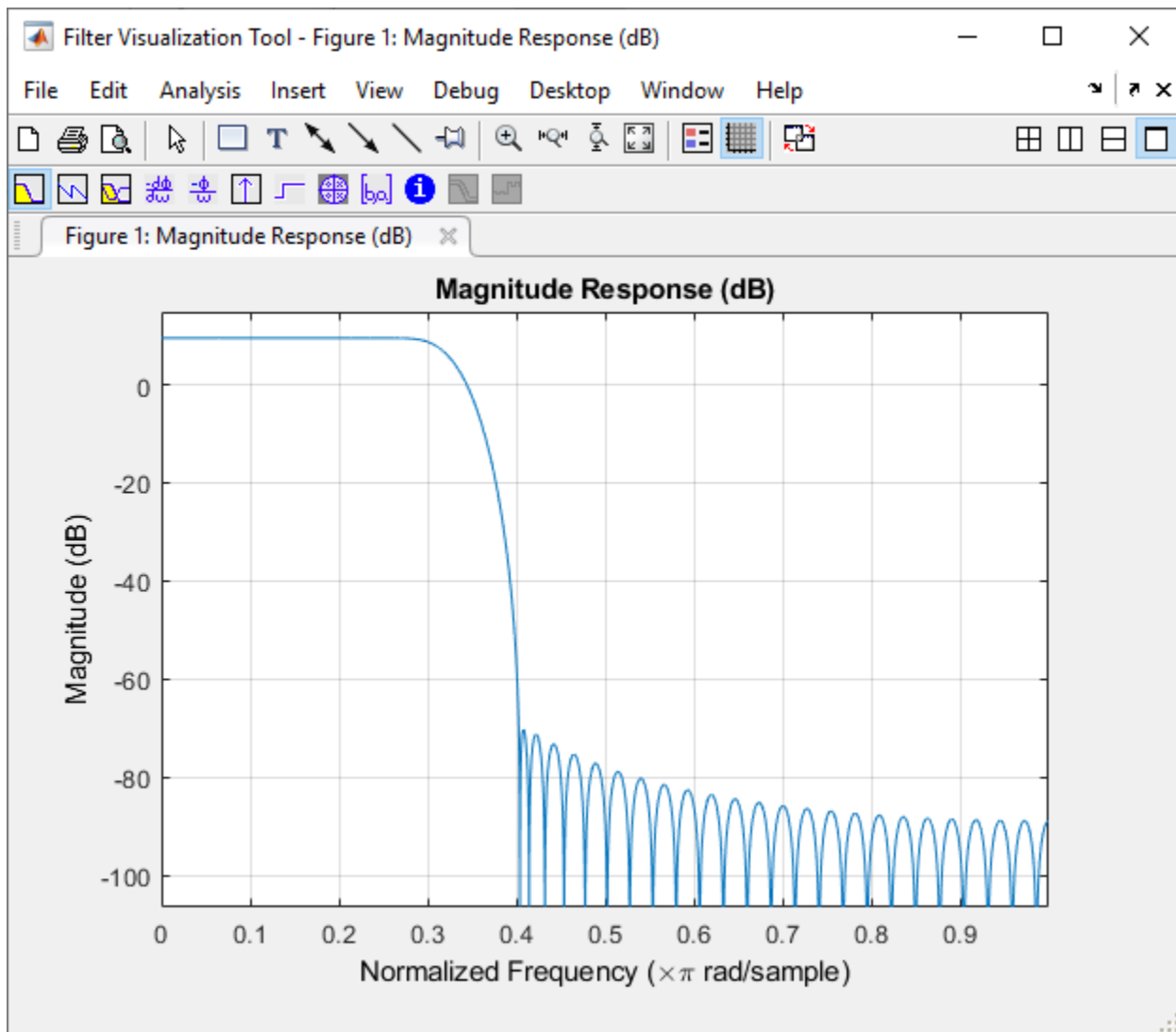
### Dependency

To enable this parameter, set **Rate options** to `Enforce single-rate processing`.

### View Filter Response — Visualize filter response

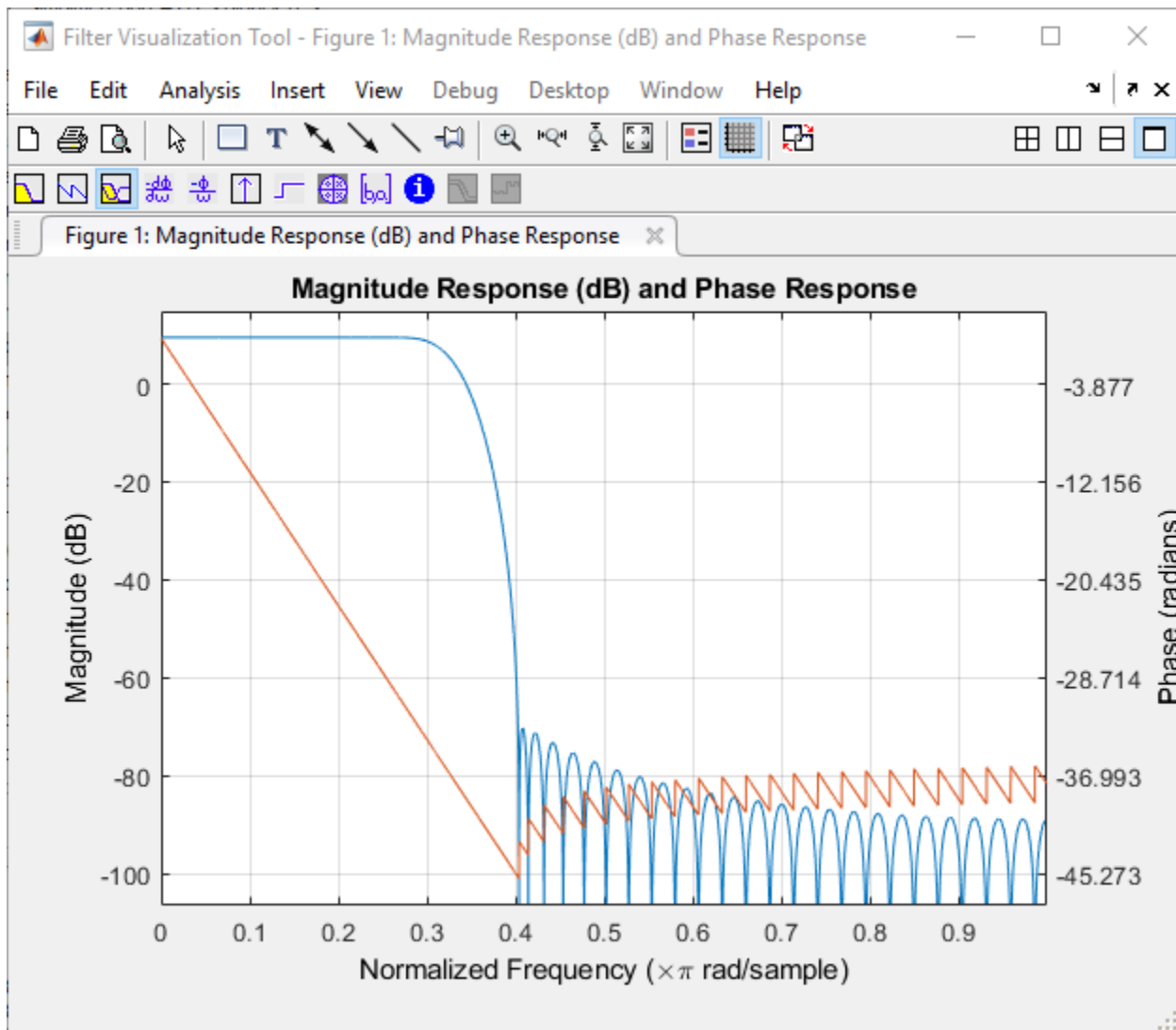
button

Select this parameter to open the Filter Visualization Tool, `fvtool`, and display the magnitude response of the FIR filter. The response is based on the parameters selected in the block dialog box. Changes made to these parameters update `fvtool`.



To update the magnitude response while `fvtool` is running, modify the block parameters and click **Apply**.

To view the magnitude response and phase response simultaneously, click the **Magnitude and Phase responses** button on the toolbar.



### Data Types Tab

When **Coefficient source** is set to **Filter object**, the fixed-point settings of the filter object specified on the **Main tab** are displayed on the **Data Types tab**. You cannot change these settings directly on the block dialog box. To change the fixed-point settings, you must edit the filter object.

For more information on System objects, see the “What Are System Objects?”.

When **Coefficient source** is set to **Auto**, the block chooses the filter coefficients automatically. For more information on the filter design algorithm that the block uses, see “Specify FIR Filter Coefficients” on page 2-665.

### Rounding mode — Rounding method

Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Specify the rounding mode for fixed-point operations as:

- Floor
- Ceiling
- Convergent
- Nearest
- Round
- Simplest
- Zero

For more details, see Rounding Modes.

The filter coefficients do not obey this parameter and always round to Nearest.

---

**Note** The **Rounding mode** and **Saturate on integer overflow** parameters have no effect on numerical results when all these conditions are met:

- **Product output** data type is Inherit: Inherit via internal rule.
- **Accumulator** data type is Inherit: Inherit via internal rule.
- **Output** data type is Inherit: Same as accumulator.

With these data-type settings, the block operates in a full-precision mode.

---

#### Dependencies

To enable this parameter, set **Coefficient source** to either **Dialog parameters** or **Auto**.

**Saturate on integer overflow** — Overflow handling method  
off (default) | on

Select this parameter to saturate the result of the fixed-point operation. Clear this parameter to wrap the result of the fixed-point operation. For details on *saturate* and *wrap*, see Overflow Handling for fixed-point operations.

---

**Note** The **Rounding mode** and **Saturate on integer overflow** parameters have no effect on numeric results when all these conditions are met:

- **Product output** data type is Inherit: Inherit via internal rule.
- **Accumulator** data type is Inherit: Inherit via internal rule.

With these data-type settings, the block operates in a full-precision mode.

---


#### Dependencies

This parameter is editable only when **Coefficient source** is set to either **Dialog parameters** or **Auto**.

**Coefficients** — Coefficients data type  
Inherit: Same word length as input (default) | fixdt(1,16) | fixdt(1,16,0)

**Coefficients** specifies the data type of the filter coefficients.

- **Inherit: Same word length as input** -- The block inherits the word length of the coefficients from the fixed-point input. The fraction length is determined based on the coefficient values in order to obtain the best possible precision.
- **fixdt(1,16)** -- The coefficients data type is a signed, binary-point, scaled, fixed-point data type with a word length of 16 bits.
- **fixdt(1,16,0)** -- The coefficients data type is a signed, binary-point, scaled, fixed-point data type with a word length of 16 bits and a fraction length of 0.

Alternatively, you can set the **Coefficients** data type by using the **Data Type Assistant**. To use the assistant, click the **Show data type assistant** button .

For more information on the data type assistant, see “Specify Data Types Using Data Type Assistant” (Simulink).

For a diagrammatic representation of how this block uses the filter coefficients data type, see “Fixed Point” on page 2-670.

### Dependencies

This parameter is editable only when **Coefficient source** is set to either **Dialog parameters** or **Auto**.

**Coefficients Minimum** — Minimum value of filter coefficients

[ ] (default) | scalar

Specify the minimum value of the filter coefficients. Simulink uses this minimum value to perform automatic scaling of fixed-point data types.

**Coefficients Maximum** — Maximum value of filter coefficients

[ ] (default) | scalar


Specify the maximum value of the filter coefficients. Simulink uses this maximum value to perform automatic scaling of fixed-point data types.

**Product output** — Product output data type

Inherit: *Inherit via internal rule* (default) | Inherit: Same as input |  
fixdt(1,16,0)

**Product output** specifies the data type of the output of a product operation in the FIR Rate Conversion block.

- **Inherit: Inherit via internal rule** — The block inherits the product output data type based on an internal rule. For more information on this rule, see “Inherit via Internal Rule”.
- **Inherit: Same as input** — The block specifies the product output data type to be the same as the input data type.
- **fixdt(1,16,0)** — The block specifies a signed, binary-point, scaled, fixed-point data type with a word length of 16 bits and a fraction length of 0.

Alternatively, you can set the **Product output** data type by using the **Data Type Assistant**. To use the assistant, click the **Show data type assistant** button .

For more information on the data type assistant, see “Specify Data Types Using Data Type Assistant” (Simulink).

For a diagrammatic representation of how this block uses the product output data type, see “Fixed Point” on page 2-670.

### Dependencies


This parameter is editable only when **Coefficient source** is set to either **Dialog parameters** or **Auto**.

#### Accumulator — Accumulator data type

Inherit: `Inherit via internal rule (default) | Inherit: Same as input | Inherit: Same as product output | fixdt(1,16,0)`

**Accumulator** specifies the data type of the output of an accumulation operation in the FIR Rate Conversion block. For illustrations on how this block uses the accumulator data type, see “Fixed Point” on page 2-670.

- **Inherit: Inherit via internal rule** — The block inherits the accumulator data type based on an internal rule. For more information on this rule, see “Inherit via Internal Rule”.
- **Inherit: Same as input** — The block specifies the accumulator data type to be the same as the input data type.
- **Inherit: Same as product output** — The block specifies the accumulator data type to be the same as the product output data type.
- **fixdt(1,16,0)** — The block specifies a signed, binary-point, scaled, fixed-point data type with a word length of 16 bits and a fraction length of 0.

Alternatively, you can set the **Accumulator** data type by using the **Data Type Assistant**. To use the assistant, click the **Show data type assistant** button .

For more information on the data type assistant, see “Specify Data Types Using Data Type Assistant” (Simulink).

### Dependencies


This parameter is editable only when **Coefficient source** is set to either **Dialog parameters** or **Auto**.

#### Output — Output data type

Inherit: `Same as accumulator (default) | Inherit: Same as input | Inherit: Same as product output | fixdt(1,16,0)`

**Output** specifies the data type of the output of the FIR Rate Conversion block.

- **Inherit: Same as input** — The block specifies the output data type to be the same as the input data type.
- **Inherit: Same as product output** — The block specifies the output data type to be the same as the product output data type.
- **Inherit: Same as accumulator** — The block specifies the output data type to be the same as the accumulator data type.
- **fixdt(1,16,0)** — The block specifies a signed, binary-point, scaled, fixed-point data type with a word length of 16 bits and a fraction length of 0.

Alternatively, you can set the **Output** data type by using the **Data Type Assistant**. To use the assistant, click the **Show data type assistant** button .

For more information on the data type assistant, see “Specify Data Types Using Data Type Assistant” (Simulink).

For a diagrammatic representation of how this block uses the output data type, see “Fixed Point” on page 2-670.

### Dependencies

This parameter is editable only when **Coefficient source** is set to either **Dialog parameters** or **Auto**.

**Output Minimum** — Minimum value the block can output

[ ] (default) | scalar

Specify the minimum value the block can output. Simulink uses this minimum value to perform:

- Simulation range checking. For more information, see “Specify Signal Ranges” (Simulink).
- Automatic scaling of fixed-point data types.

**Output Maximum** — Maximum value block can output

[ ] (default) | scalar

Specify the maximum value the block can output. Simulink uses this maximum value to perform:

- Simulation range checking. For more information, see “Specify Signal Ranges” (Simulink).
- Automatic scaling of fixed-point data types.

**Lock data type settings against changes by the fixed-point tools** — Prevent fixed-point tools from overriding data types

off (default) | on

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block dialog box.

### Dependencies

This parameter appears only when **Coefficient source** is set to either **Dialog parameters** or **Auto**.

## Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | double   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no                                      |
| <b>Multidimensional Signals</b> | no                                      |
| <b>Variable-Size Signals</b>    | yes                                     |
| <b>Zero-Crossing Detection</b>  | no                                      |



## More About

### Specify FIR Filter Coefficients

When you select the **Dialog parameters** option, you use the **FIR filter coefficients** parameter to specify the numerator coefficients of the FIR filter transfer function  $H(z)$ .

$$H(z) = b_0 + b_1z^{-1} + \dots + b_Nz^{-N}$$

You can generate the FIR filter coefficient vector,  $\mathbf{b} = [b_0, b_1, \dots, b_N]$ , using one of the DSP System Toolbox filter design functions such as `designMultirateFIR`, `firnyquist`, `firhalfband`, `firgr`, or `firceqrip`.

The coefficient vector you specify must have a length greater than the interpolation factor ( $N + 1 > L$ ). The FIR filter must be a lowpass filter with a normalized cutoff frequency no greater than  $\min(1/L, 1/M)$ . The block internally initializes all filter states to zero.

When you select the **Auto** mode, the block designs an FIR multirate filter with the decimation factor specified in **Decimation factor** and interpolation factor specified in **Interpolation factor**. The `designMultirateFIR` function designs the filter and returns the coefficients used by the block. For more information on the filter design algorithm, see Orfanidis [1].

### Specify Resampling Rate

This section applies only to the single-rate processing mode when the **Rate options** parameter is set to **Enforce single-rate processing**.

You specify the resampling rate of the FIR Rate Conversion block using the **Decimation factor** and **Interpolation factor** parameters. For a  $P_i$ -by- $Q$  matrix input, the **Decimation factor**  $M$  and the **Interpolation factor**  $L$  must satisfy these requirements:

- $M$  and  $L$  must be relatively prime integers. That is, the ratio  $M/L$  cannot be reduced to a ratio of smaller integers.
- $\frac{M}{L} = \frac{P_i}{P_o}$ , where  $P_i$  and  $P_o$  are the integer frame sizes of the input and output, respectively.

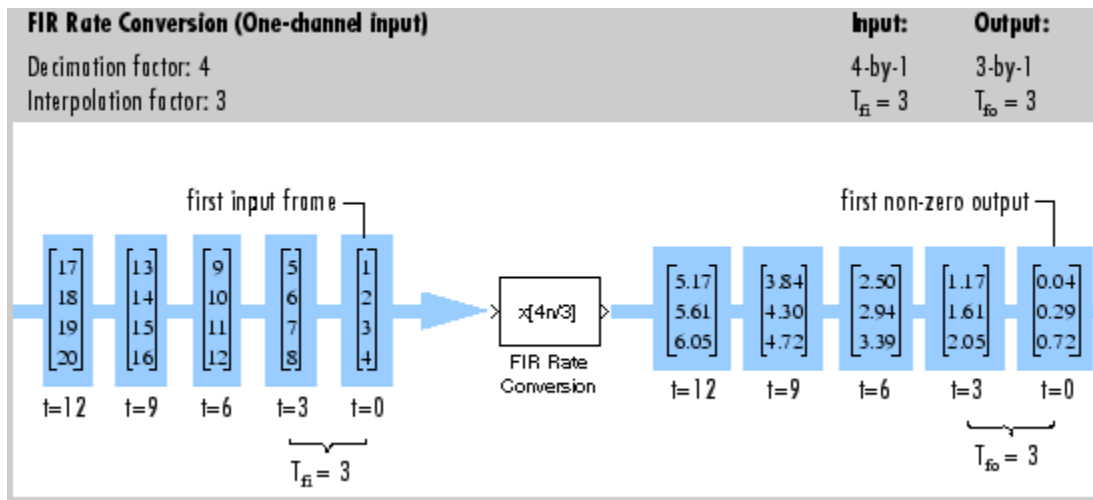
You can satisfy the second requirement by setting the **Decimation factor**,  $M$ , to equal the input frame size  $P_i$ . When you do so, the output frame size  $P_o$  equals the **Interpolation factor**  $L$ .

By changing the frame size in this way, the block is able to hold the frame period constant ( $T_{\hat{f}} = T_{f_0}$ ) and achieve the desired conversion of the sample period, such that

$$T_{so} = \frac{M}{L} \times T_{si}$$

where  $T_{so}$  is the output sample period.

This figure shows how the FIR Rate Conversion block converts a 4-by-1 input with a sample period of  $3/4$  to a 3-by-1 output with a sample period of 1. The frame period ( $T_f$ ) of 3 remains constant.



### Frame-Based Processing

The FIR Rate Conversion block resamples each column of the input over time. In this mode, the block can perform either single-rate or multirate processing. You can use the **Rate options** parameter to specify how the block resamples the input.

- When you set the **Rate options** parameter to Enforce single-rate processing, the input and output of the block have the same sample rate. To resample the output while maintaining the input sample rate, the block resamples the data in each column of the input such that the frame length of the output has an upper bound size of  $\text{ceil}(LP/M)$ , where  $L$  is the interpolation factor specified in the **Interpolation factor** parameter,  $P$  is the input frame length, and  $M$  is the decimation factor specified in the **Decimation factor** parameter.

In this mode, if you input a fixed-size signal (frame length does not change during simulation) and select the **Allow arbitrary frame length for fixed-size input signals** parameter, the input frame length can be arbitrary and does not have to be a multiple of the decimation factor. If you clear the **Allow arbitrary frame length for fixed-size input signals** parameter, the input frame length must be a multiple of the decimation factor.

In this mode, if you input a variable-size signal (frame length changes during simulation), the **Allow arbitrary frame length for fixed-size input signals** appears on the block dialog box but does not have any impact on the input frame length. You can input a variable-size signal of any frame length even if you do not select the **Allow arbitrary frame length for fixed-size input signals** parameter.

This table summarizes the support for arbitrary input frame length when you set **Rate options** to Enforce single-rate processing.

| Input Signal               | Block Support for this Signal | Support for Arbitrary Input Frame Length   | Input Size   | Output Size                                      |
|----------------------------|-------------------------------|--|--------------|--|
| Fixed-size input signal    | Yes                           | When you select <b>Allow arbitrary frame length for fixed-size input signals</b> | $P$ -by- $Q$ | Upper bound size of $\text{ceil}(LP/M)$ -by- $Q$ |
| Variable-size input signal | Yes                           | Always   | $P$ -by- $Q$ | Upper bound size of $\text{ceil}(LP/M)$ -by- $Q$ |

- When you set the **Rate options** parameter to Allow multirate processing, the input and output of the FIR Rate Conversion block are of the same size, but the sample rate of the output is  $L/M$  times that of the input. In this mode, the block treats a  $P$ -by- $Q$  matrix input as  $Q$  independent channels. The block decimates each column of the input over time by keeping the frame size constant, and making the output frame period ( $T_{fo}$ )  $M/L$  times the input frame period ( $T_{fo} = (M/L)*T_{fi}$ ).

In this mode, the block accepts only fixed-size signals and these signals can have an arbitrary frame length.

This table summarizes the support for arbitrary input frame length when you set **Rate options** to Allow multirate processing.

| Input Signal               | Block Support for this Signal | Support for Arbitrary Input Frame Length | Input Size     | Output Size    |
|----------------------------|-------------------------------|--|----------------|----------------|
| Fixed-size input signal    | Yes                           | Always                                   | $P$ -by- $Q$   | $P$ -by- $Q$   |
| Variable-size input signal | No                            | Not applicable                           | Not applicable | Not applicable |

## Algorithms

The FIR rate converter is implemented efficiently using a polyphase structure.

To derive the polyphase structure, start with the transfer function of the FIR filter: This FIR filter is a combined anti-imaging and anti-aliasing filter.

$$H(z) = b_0 + b_1z^{-1} + \dots + b_Nz^{-N}$$

$N+1$  is the length of the FIR filter.

You can rearrange this equation as follows:

$$\begin{aligned}
 H(z) = & (b_0 + b_Lz^{-L} + b_{2L}z^{-2L} + \dots + b_{N-L+1}z^{-(N-L+1)}) + \\
 & z^{-1}(b_1 + b_{L+1}z^{-L} + b_{2L+1}z^{-2L} + \dots + b_{N-L+2}z^{-(N-L+1)}) + \\
 & \vdots \\
 & z^{-(L-1)}(b_{L-1} + b_{2L-1}z^{-L} + b_{3L-1}z^{-2L} + \dots + b_{N-L}z^{-(N-L+1)})
 \end{aligned}$$

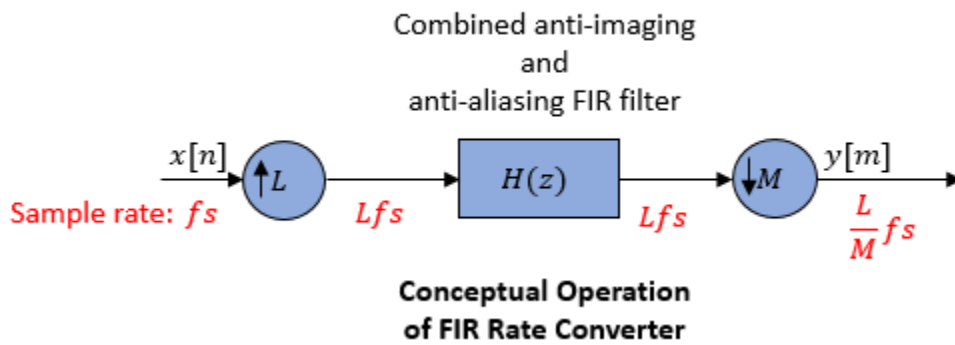
$L$  is the number of polyphase components, and its value equals the interpolation factor that you specify.

You can write this equation as:

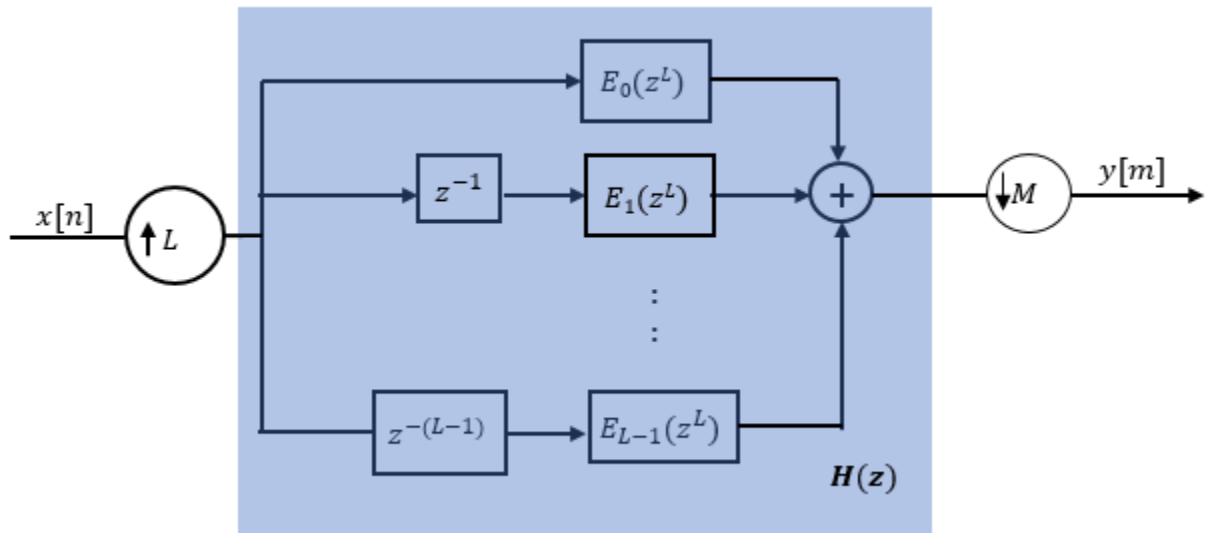
$$H(z) = E_0(z^L) + z^{-1}E_1(z^L) + \dots + z^{-(L-1)}E_{L-1}(z^L)$$

$E_0(z^L), E_1(z^L), \dots, E_{L-1}(z^L)$  are polyphase components of the FIR filter  $H(z)$ .

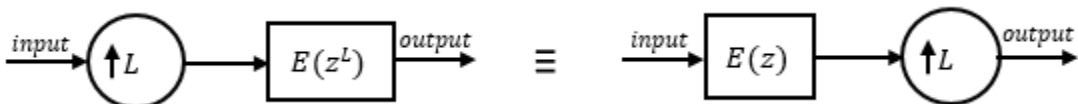
Conceptually, the FIR rate converter contains an upsampler, followed by a combined anti-imaging, anti-aliasing FIR filter  $H(z)$ , which is followed by a downsampler.



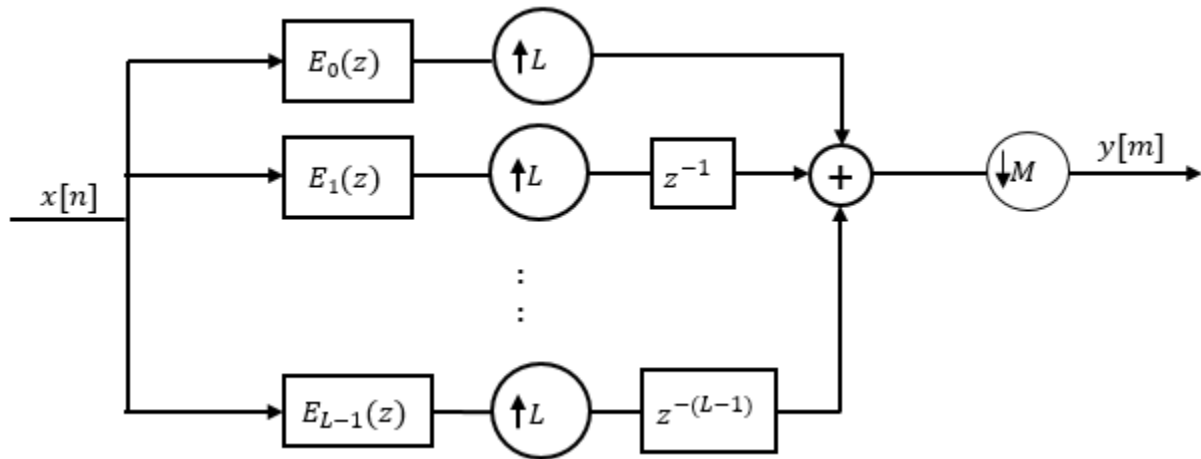
Replace  $H(z)$  with its polyphase representation.



Here is the multirate noble identity for interpolation.



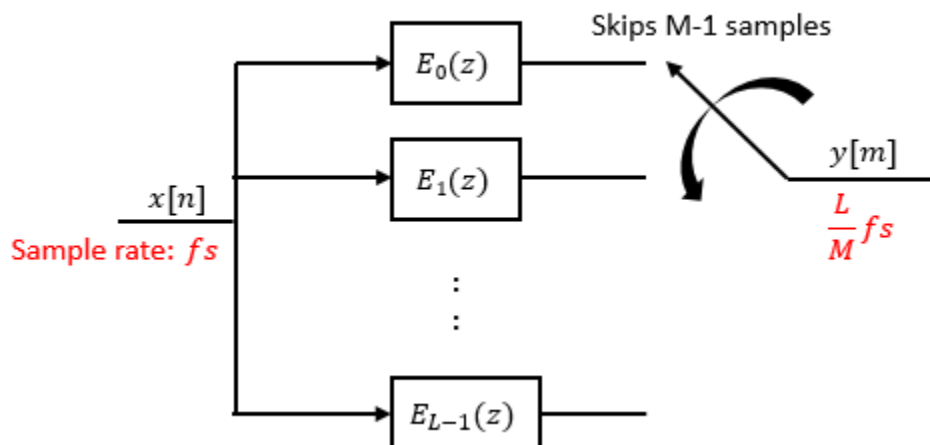
Applying the noble identity for interpolation moves the upsampling operation to after the filtering operation. This move enables you to filter the signal at a lower rate.



You can replace the upsampling operator, delay block, and the adder with a commutator switch. To account for the downsampler that follows, the switch moves in steps of size  $M$ . The switch receives the first sample from branch 0 and moves in the counter clockwise direction, each time skipping  $M-1$  branches.

As an example, consider a rate converter with  $L$  set to 5 and  $M$  set to 3. The polyphase components are  $E_0(z)$ ,  $E_1(z)$ ,  $E_2(z)$ ,  $E_3(z)$ , and  $E_4(z)$ . The switch starts on the first branch 0, skips branches 1 and 2, receives the next sample from branch 3, then skips branches 4 and 0, receives the next sample from branch 2, and so on. The sequence of branches from which the switch receives the data sample is [0, 3, 1, 4, 2, 0, 3, 1, ...].

The rate converter implements the  $L/M$  conversion by first applying the interpolation factor  $L$  to the incoming data, and using the commutator switch at the end to receive only 1 in  $M$  samples, effectively accounting for the downsampling factor  $M$ . Hence, the sample rate at the output of the FIR rate converter is  $Lfs/M$ .



## Version History

Introduced before R2006a

### R2022b: Support for arbitrary input frame length

Starting in R2022b, this block supports input signals with arbitrary frame lengths when the:

- Input signal is a fixed-size signal (frame length does not change during simulation) and the block allows for multirate processing
- Input signal is a fixed-size signal, the block enforces single-rate processing, and you select the **Allow arbitrary frame length for fixed-size input signals** parameter
- Input signal is a variable-size signal (frame length changes during simulation)

When this block supports an arbitrary frame length input signal, the input frame length does not have to be a multiple of the decimation factor.

For more details, see “Frame-Based Processing” on page 2-666.

## References

- [1] Orfanidis, Sophocles J. *Introduction to Signal Processing*. Upper Saddle River, NJ: Prentice-Hall, 1996.

## Extended Capabilities

### C/C++ Code Generation

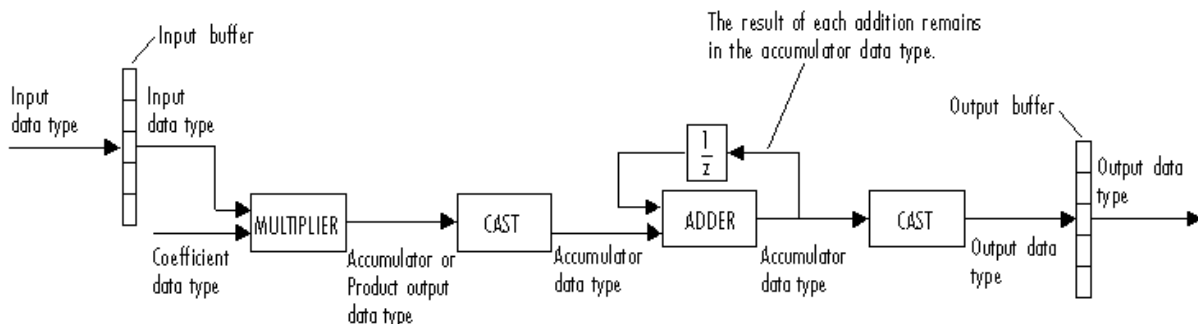
Generate C and C++ code using Simulink® Coder™.

Generated code relies on the memcpy or memset function (string.h) under certain conditions.

### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

This diagram shows the data types used within the FIR Rate Conversion block for fixed-point signals.



You can set the coefficient, product output, accumulator, and output data types in the block dialog box. The diagram shows that input data is stored in the input buffer in the same data type and scaling as the input. Filtered data resides in the output buffer in the output data type and scaling that you set in the block dialog. The block stores any initial conditions in the output buffer using the output data type and scaling that you set in the block dialog box.

The output of the multiplier is in the product output data type when at least one of the inputs to the multiplier is real. When both the inputs to the multiplier are complex, the result of the multiplication is in the accumulator data type. For details on how the complex multiplication is performed in the block, see “Multiplication Data Types”.

---

**Note** When the block input is fixed point, all internal data types are signed fixed point.

---

## See Also

### Functions

`firceqrip` | `firgr` | `firhalfband` | `firnyquist`

### Objects

`dsp.FIRRateConverter` | `dsp.FIRInterpolator` | `dsp.FIRDecimator` |  
`dsp.FIRHalfbandInterpolator` | `dsp.FIRHalfbandDecimator` |  
`dsp.CICCompensationInterpolator` | `dsp.CICCompensationDecimator`

### Blocks

CIC Compensation Decimator | CIC Compensation Interpolator | IIR Halfband Decimator | IIR Halfband Interpolator | FIR Halfband Decimator | FIR Halfband Interpolator | CIC Interpolation | CIC Decimation | FIR Interpolation | FIR Decimation | Upsample | Downsample

### Topics

“Convert Sample and Frame Rates in Simulink Using Rate Conversion Blocks”  
“Multirate and Multistage Filters”

# Flip

Flip input vertically or horizontally



**Libraries:**

DSP System Toolbox / Signal Management / Indexing

## Description

The Flip block vertically or horizontally reverses the  $M$ -by- $N$  input matrix,  $u$ . The block determines the dimension to flip along using the **Flip along** parameter.

This block supports Simulink virtual buses.

## Ports

### Input

**Input** — Input signal  
matrix

Input signal, specified as a matrix of size  $M$ -by- $N$ .

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated

### Output

**Output** — Flipped output  
matrix

Flipped output, returned as a matrix of size  $M$ -by- $N$ . The output always has the same dimensionality as the input.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated

## Parameters

**Flip along** — Specify dimension to flip along

Columns (default) | Rows

Specify the dimension along which to flip the input as **Columns** or **Rows**. When you set this parameter to **Columns**, the block flips the input vertically. When you set this parameter to **Rows**, the block flips the input horizontally.

When you set this parameter to **Columns**, the block flips the input *vertically* so the first row of the input becomes the last row of the output.

`y = flipud(u)` % Equivalent MATLAB code



When flipping the input vertically, the block treats length- $M$  unoriented vector inputs as  $M$ -by-1 column vectors.

When you set this parameter to **Rows**, the block flips the input *horizontally* so the first column of the input becomes the last column of the output.

```
y = fliplr(u) % Equivalent MATLAB code
```

When flipping the input horizontally, the block treats length- $N$  unoriented vector inputs as 1-by- $N$  row vectors.

## Block Characteristics

|                                 |  |
|---------------------------------|--|
| <b>Data Types</b>               | Boolean   double   enumerated   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no   |
| <b>Multidimensional Signals</b> | no   |
| <b>Variable-Size Signals</b>    | no   |
| <b>Zero-Crossing Detection</b>  | no   |

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### Fixed-Point Conversion

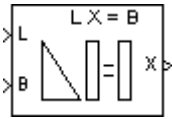
Design and simulate fixed-point systems using Fixed-Point Designer™.

## See Also

Selector | Variable Selector | flipud | fliplr

## Forward Substitution

Solve  $LX = B$  for  $X$  when  $L$  is lower triangular matrix



### Libraries:

DSP System Toolbox / Math Functions / Matrices and Linear Algebra / Linear System Solvers

### Description

The Forward Substitution block solves the linear system  $LX = B$  by simple forward substitution of variables, where:

- $L$  is the lower triangular  $M$ -by- $M$  matrix input to the **L** port.
- $B$  is the  $M$ -by- $N$  matrix input to the **B** port.
- $X$  is the  $M$ -by- $N$  output matrix and is the solution of the system of equations.

The block does not check the rank of the inputs. The block only uses the elements in the *lower triangle* of input  $L$  and ignores the upper elements. When you select **Input L is unit-lower triangular**, the block assumes the elements on the diagonal of  $L$  are 1s. This is useful when matrix  $L$  is the result of another operation, such as an LDL decomposition, that uses the diagonal elements to represent the  $D$  matrix.

### Ports

#### Input

**L** — Input lower triangular matrix  
 $M$ -by- $M$  matrix

Specify the input lower triangular  $M$ -by- $M$  matrix  $L$  through this port. Inputs  $L$  and  $B$  must have the same number of rows.

When the input  $L$  is fixed-point, it is signed only.

Data Types: single | double | int8 | int16 | int32 | fixed point

**B** —  $B$  matrix  
 $M$ -by- $N$  matrix |  $M$ -by-1 vector

Specify the  $B$  matrix in the  $LX = B$  equation as an  $M$ -by- $N$  matrix or a  $M$ -by-1 vector.

When you specify an  $M$ -by-1 vector, the block treats the length- $M$  vector input at port **B** as an  $M$ -by-1 matrix. Inputs  $L$  and  $B$  must have the same number of rows.

When the input  $B$  is fixed-point, it is signed only.

Data Types: single | double | int8 | int16 | int32 | fixed point

## Output

**X** — Solution of system of equations

*M*-by-*N* matrix | *M*-by-1 matrix

The block returns the output as an *M*-by-*N* matrix or an *M*-by-1 matrix. The size of *X* matrix is same as the size of the *B* matrix.

When the inputs are fixed-point, the output matrix *X* is signed only fixed-point.

Data Types: single | double | int8 | int16 | int32 | fixed point

## Parameters

### Main Tab

**Input L is unit-lower triangular** — Diagonal elements of input matrix have a value of 1

off (default) | on

Select this check box only when all elements on the diagonal of *L* have a value of 1. When you do so, the block optimizes its behavior by skipping an unnecessary division operation.

Do not select this check box if any elements on the diagonal of *L* do not have a value of 1. When you clear the **Input L is unit-lower triangular** check box, the block always performs the division operation.

**Diagonal of complex input L is real** — Diagonal elements of complex input are real

off (default) | on

Select to optimize simulation speed when the diagonal elements of complex input *L* are real.

---

**Note** When *L* is a complex fixed-point signal, you must select either **Input L is unit-lower triangular** or **Diagonal of complex input L is real**. In these cases, the block ignores any imaginary part of the diagonal of *L*.

---

### Dependencies

This parameter is visible only when **Input L is unit-upper triangular** is not selected.

### Data Types Tab

---

**Note** Floating-point inheritance takes precedence over the data type settings defined on this pane. When inputs are floating point, the block ignores these settings, and all internal data types are floating point.

---

**Rounding mode** — Method of rounding operation

Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Specify the rounding mode for fixed-point operations as one of the following:

- Floor
- Ceiling
- Convergent
- Nearest
- Round
- Simplest
- Zero

For more details, see rounding mode.

### **Saturate on integer overflow** — Method of overflow action

off (default) | on

When you select this parameter, the block saturates the result of its fixed-point operation. When you clear this parameter, the block wraps the result of its fixed-point operation. For details on `saturate` and `wrap`, see overflow mode for fixed-point operations.

### **Product output** — Product output data type

Inherit: `Inherit via internal rule (default)` | Inherit: `Same as first input` | `fixdt(1,16,0)`

**Product output** specifies the data type of the output of a product operation in the Backward Substitution block. For more information on the product output data type, see “Fixed Point” on page 2-678 and “Multiplication Data Types”.

- `Inherit: Inherit via internal rule` — The block inherits the product output data type based on an internal rule. For more information on this rule, see “Inherit via Internal Rule”.
- `Inherit: Same as first input` — The block specifies the product output data type to be the same as the first input data type.
- `fixdt(1,16,0)` — The block specifies a signed, binary-point, scaled, fixed-point data type with a word length of 16 bits and a fraction length of 0.

Alternatively, you can set the **Product output** data type by using the **Data Type Assistant**. To use

the assistant, click the **Show data type assistant** button .

For more information on the data type assistant, see “Specify Data Types Using Data Type Assistant” (Simulink) in *Simulink User's Guide* (Simulink).


### **Accumulator** — Accumulator data type

Inherit: `Inherit via internal rule (default)` | Inherit: `Same as product output` | Inherit: `Same as first input` | `fixdt(1,16,0)`

**Accumulator** specifies the data type of the output of an accumulation operation in the Backward Substitution block. For illustrations on how to use the accumulator data type in this block, see the “Fixed Point” on page 2-678 section.

- `Inherit: Inherit via internal rule` — The block inherits the accumulator data type based on an internal rule. For more information on this rule, see “Inherit via Internal Rule”.

- **Inherit: Same as product output** — The block specifies the accumulator data type to be the same as the product output data type.
- **Inherit: Same as first input** — The block specifies the accumulator data type to be the same as the first input data type.
- **fixdt(1,16,0)** — The block specifies a signed, binary-point scaled, fixed-point data type with a word length of 16 bits and a fraction length of 0.

Alternatively, you can set the **Accumulator** data type by using the **Data Type Assistant**. To use the assistant, click the **Show data type assistant** button .


For more information on the data type assistant, see “Specify Data Types Using Data Type Assistant” (Simulink) in *Simulink User's Guide* (Simulink).

#### **Output** — Output data type

**Inherit: Same as first input** (default) | **fixdt(1,16,0)**

**Output** specifies the data type of the output of the Backward Substitution block. For more information on the output data type, see the “Fixed Point” on page 2-678 section.

- **Inherit: Same as first input** — The block specifies the output data type to be the same as the first input data type.
- **fixdt(1,16,0)** — The block specifies a signed, binary-point scaled, fixed-point data type with a word length of 16 bits and a fraction length of 0.

Alternatively, you can set the **Output** data type by using the **Data Type Assistant**. To use the assistant, click the **Show data type assistant** button .

For more information on the data type assistant, see “Specify Data Types Using Data Type Assistant” (Simulink) in *Simulink User's Guide* (Simulink).

#### **Output Minimum** — Minimum value the block can output

**[]** (default) | **scalar**

Specify the minimum value the block can output. Simulink software uses this minimum value to perform:

- Simulation range checking. See “Specify Signal Ranges” (Simulink).
- Automatic scaling of fixed-point data types.

#### **Output Maximum** — Maximum value block can output

**[]** (default) | **scalar**

Specify the maximum value the block can output. Simulink software uses this maximum value to perform:

- Simulation range checking. See “Specify Signal Ranges” (Simulink).
- Automatic scaling of fixed-point data types.

#### **Lock data type settings against changes by the fixed-point tools** — Prevent fixed-point tools from overriding data types

**off** (default) | **on**

Select this parameter to prevent the fixed-point tools from overriding the data types you specify in the block dialog box.

## Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | double   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no                                      |
| <b>Multidimensional Signals</b> | no                                      |
| <b>Variable-Size Signals</b>    | no                                      |
| <b>Zero-Crossing Detection</b>  | no                                      |

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Generated code relies on the memcpy or memset function (string.h) under certain conditions.

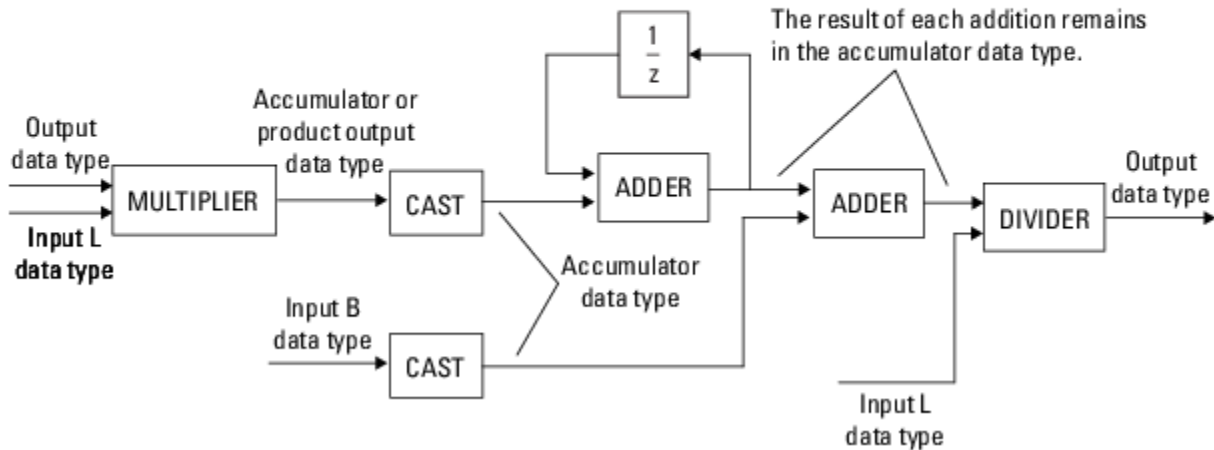
### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

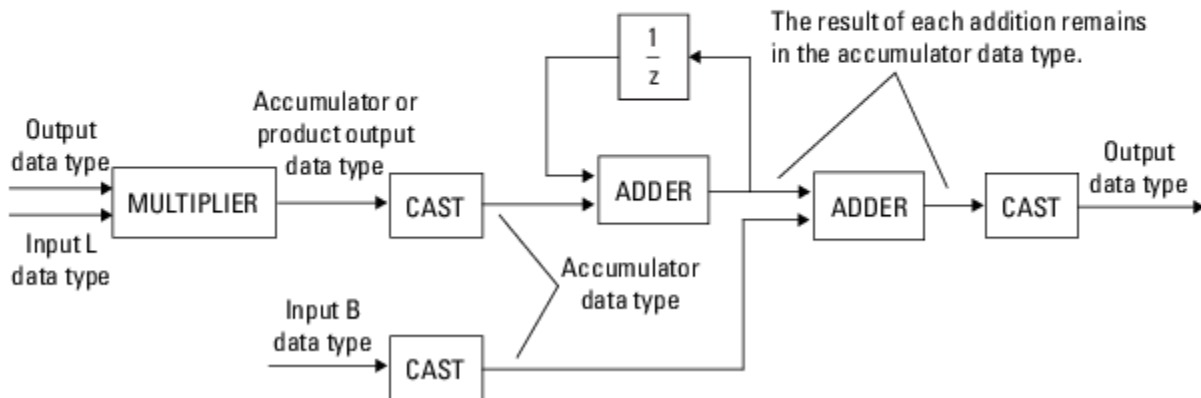
If the input is fixed point, it must be a signed integer or a signed fixed point value with a power-of-two slope and zero bias.

The following diagrams show the data types used within the Forward Substitution block for fixed-point signals.

When input L is not unit-lower triangular:



When input L is unit-lower triangular:



You can set the product output, accumulator, and output data types in the block dialog box.

The output of the multiplier is in the product output data type when the input is real. When the input is complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication performed, see "Multiplication Data Types".

## See Also

### Blocks

Cholesky Solver | Backward Substitution | LDL Solver | Levinson-Durbin | LU Solver | QR Solver

**Topics**

“Linear System Solvers”



# Fourth-Order Section Filter

Implement cascade of fourth-order section filters



## Libraries:

DSP System Toolbox / Filtering / Filter Implementations

## Description

The Fourth-Order Section Filter block implements a cascade of fourth-order section filters in Simulink. You can specify the numerator and denominator coefficients of the filter in the block parameters dialog box or through input ports.

## Ports

### Input

**x** — Input signal  
vector | matrix

Input signal, specified as a vector or a matrix. The input can be a variable size signal. That is, the frame size of the signal can change during simulation but the number of channels cannot.

This port is unnamed until you set the **Coefficient source** parameter to Input ports.

Data Types: single | double

**Num** — Numerator coefficients  
 $P$ -by-5 matrix

Specify the numerator coefficients of the fourth-order section filter as a  $P$ -by-5 matrix, where  $P$  is the number of filter sections. For more details on this input port, see **Numerator coefficients of filter**.

While the simulation is running, you can change the number of sections (rows) in the numerator coefficients. You can also change the coefficient values.

### Dependencies

This port appears only when you set the **Coefficient source** parameter to Input ports.

Data Types: single | double

**Den** — Denominator coefficients  
 $P$ -by-5 matrix |  $P$ -by-4 matrix

Specify the denominator coefficients of the fourth-order section filter as a  $P$ -by-5 matrix or a  $P$ -by-4 matrix, where  $P$  is the number of filter sections. For more details on this input port, see **Denominator coefficients of filter**.

While the simulation is running, you can change the number of sections (rows) in the denominator coefficients. You can also change the coefficient values.

### Dependencies

This port appears only when you set the **Coefficient source** parameter to `Input ports`.

Data Types: `single` | `double`

### Output

**y** — Filtered output  
vector | matrix

Filtered output, returned as a vector or a matrix. The output has the same size and data type as the input. The output signal is complex if either the input signal, numerator coefficients, or the denominator coefficients are complex.

This port is unnamed until you set the **Coefficient source** parameter to `Input ports`.

Data Types: `single` | `double`

### Parameters

**Coefficient source** — Filter coefficient source

Dialog parameters (default) | `Input ports` | `Filter object`

Specify the filter coefficient source as one of the following:

- `Dialog parameters` -- Specify the filter coefficients through the **Numerator coefficients of filter** and the **Denominator coefficients of filter** parameters in the block dialog box.
- `Input ports` -- Specify the filter coefficients through the **Num** and **Den** input ports.
- `Filter object` -- Specify the filter coefficients using a `dsp.FourthOrderSectionFilter` object.

To view the filter response, set this parameter to `Dialog parameters` or `Filter object`, and then click the **View Filter Response** button.

**Numerator coefficients of filter** — Numerator coefficients

[0.07795634 0 -0.15591268 0 0.07795634; 0.06188520 0 -0.12377039 0 0.06188520]  
(default) |  $P$ -by-5 matrix

Specify the numerator coefficients **b** of the fourth-order section filter as a  $P$ -by-5 matrix, where  $P$  is the number of filter sections.

$$\mathbf{b} = \begin{bmatrix} b_{01} & b_{11} & b_{21} & b_{31} & b_{41} \\ b_{02} & b_{12} & b_{22} & b_{32} & b_{42} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ b_{0P} & b_{1P} & b_{2P} & b_{3P} & b_{4P} \end{bmatrix}$$

In the transfer function form, the fourth-order section filter can be represented using the following equation:

$$H(z) = \prod_{k=1}^P H_k(z) = \prod_{k=1}^P \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2} + b_{3k}z^{-3} + b_{4k}z^{-4}}{a_{0k} + a_{1k}z^{-1} + a_{2k}z^{-2} + a_{3k}z^{-3} + a_{4k}z^{-4}}$$

where,

- **a** -- Denominator coefficients matrix. For more details on how to specify this matrix, see **Denominator coefficients of filter**.
- *k* -- Row index.

You cannot change the size of this parameter during simulation, but you can change its value.

**Tunable:** Yes

**Dependencies**

To enable this parameter, set **Coefficient source** to Dialog parameters.

Data Types: single | double

**Denominator coefficients of filter** — Denominator coefficients

[1 0 1.32091343 0 0.63273879; 1 0 1.04859957 0 0.29614035] (default) | *P*-by-5 matrix | *P*-by-4 matrix

Specify the denominator coefficients **a** of the fourth-order section filter as a *P*-by-5 matrix or a *P*-by-4 matrix, where *P* is the number of filter sections.

$$\mathbf{a} = \begin{bmatrix} a_{01} & a_{11} & a_{21} & a_{31} & a_{41} \\ a_{02} & a_{12} & a_{22} & a_{32} & a_{42} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{0P} & a_{1P} & a_{2P} & a_{3P} & a_{4P} \end{bmatrix}$$

The block algorithm assumes that the value of the leading coefficients is always 1. If the denominator is of size *P*-by-4, the block algorithm places 1s in the first column to make the denominator size *P*-by-5. If the denominator is of size *P*-by-5 and the elements in the first column do not equal 1, the algorithm ignores the values in the first column and appends them with 1s.

In the transfer function form, the fourth-order section filter can be represented using the following equation:

$$H(z) = \prod_{k=1}^P H_k(z) = \prod_{k=1}^P \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2} + b_{3k}z^{-3} + b_{4k}z^{-4}}{a_{0k} + a_{1k}z^{-1} + a_{2k}z^{-2} + a_{3k}z^{-3} + a_{4k}z^{-4}}$$

where,

- **b** -- Numerator coefficients matrix. For more details on how to specify this matrix, see **Numerator coefficients of filter**.
- *k* -- Row index.

You cannot change the size of this parameter during simulation, but you can change its value.

**Tunable:** Yes

**Dependencies**

To enable this parameter, set **Coefficient source** to Dialog parameters.

Data Types: single | double

**Filter** — Filter object`dsp.FourthOrderSectionFilter` object

Specify the filter coefficients using a `dsp.FourthOrderSectionFilter` object. You can enter this object directly in the **Filter** parameter. Alternatively, you can create this object in the MATLAB workspace or model workspace and specify the filter variable in the **Filter** parameter. When you specify the filter as a variable and you simulate the model, you cannot change the complexity, data type, and size of the filter coefficients in the workspace between simulations.

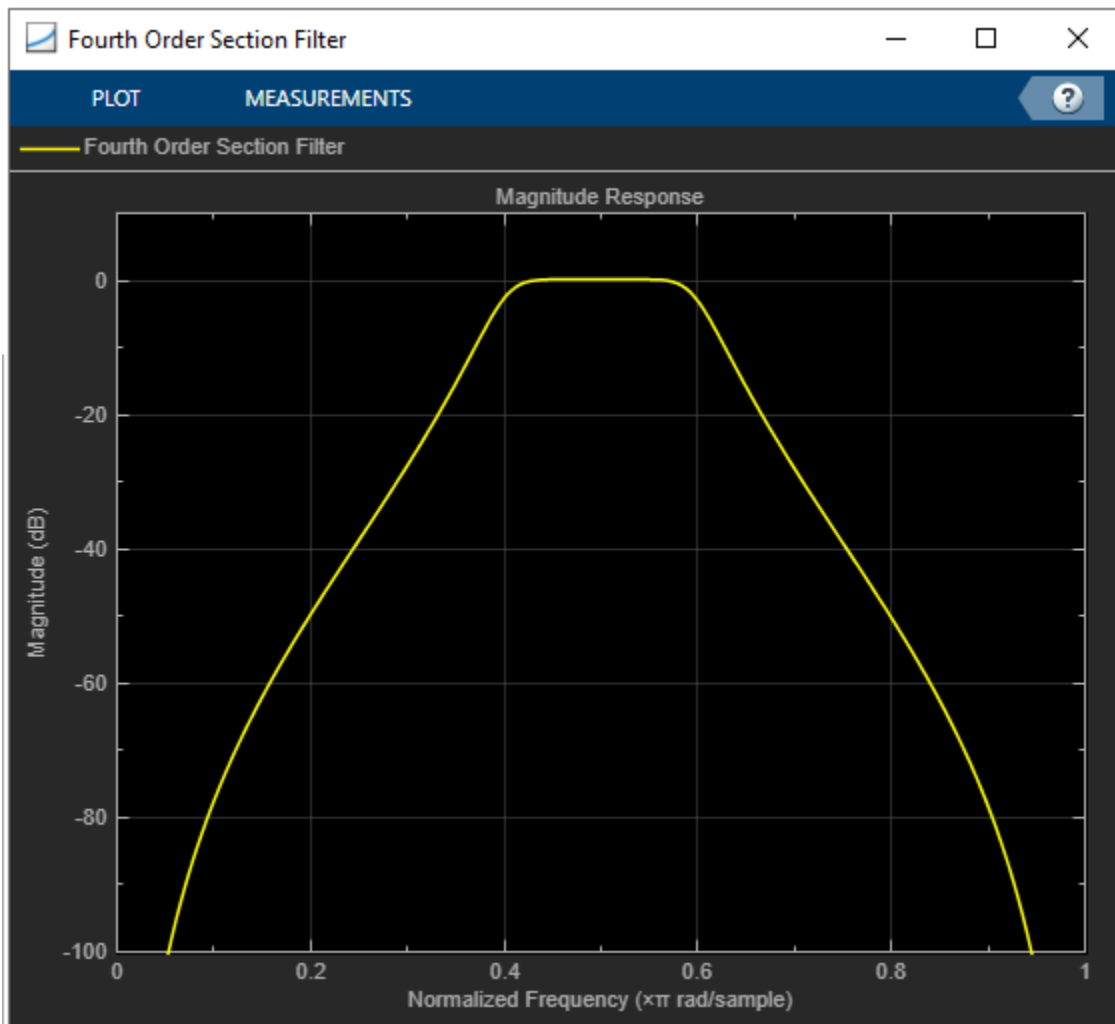
**Dependencies**

To enable this parameter, set **Coefficient source** to `Filter` object.

**View Filter Response** — Visualize frequency response`gui` button

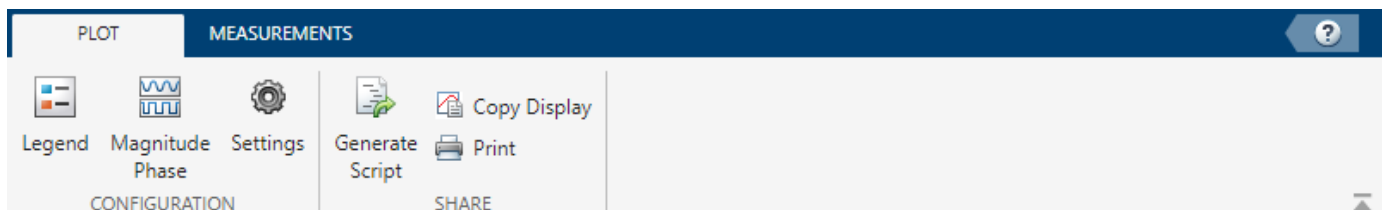
Click this button to open the Dynamic Filter Visualizer window and display the magnitude and phase response of the fourth-order section filter. The response is based on the coefficients that you specify in the block dialog box. If you set **Coefficient source** to `Input ports` and click **Apply**, you cannot view the magnitude response using this button. To view the response in the visualizer, you must specify the filter coefficients directly in the block dialog box or through the `dsp.FourthOrderSectionFilter` object.

To update the filter response while the visualizer is running, modify the coefficients in the block dialog box and click **Apply**.

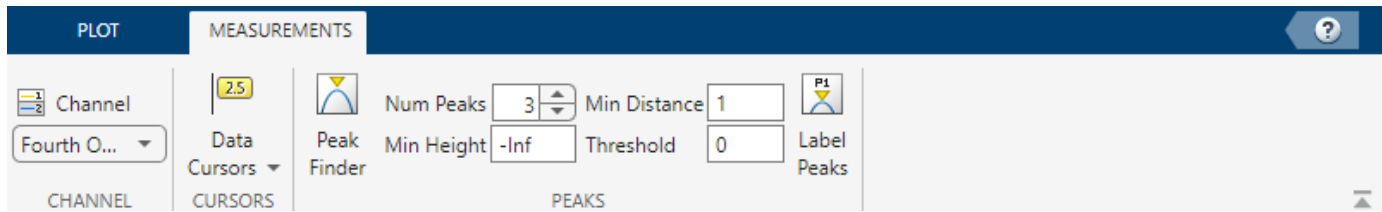


You can configure the plot settings and the frequency response measurements from the interface of the visualizer.

On the **Plot** tab, you can enable the legend, specify to plot both magnitude and phase responses of the filter, modify the plot settings, generate a script to re-create the plot, and even save or share the settings.



On the **Measurements** tab, you can enable data cursors, and display the peak values of the filter response.



For more details on the Dynamic Filter Visualizer interface and its tools, see “Configure Filter Visualizer”.

**Simulate using** — Type of simulation to run

Interpreted execution (default) | Code generation

Type of simulation to run. You can set this parameter to:

- **Interpreted execution:** Simulate model using the MATLAB interpreter. This option shortens startup time.
- **Code generation:** Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations as long as the model does not change. This option requires additional startup time but provides faster subsequent simulations.

## Block Characteristics

|                                 |                           |
|---------------------------------|---------------------------|
| <b>Data Types</b>               | double   integer   single |
| <b>Direct Feedthrough</b>       | no                        |
| <b>Multidimensional Signals</b> | no                        |
| <b>Variable-Size Signals</b>    | yes                       |
| <b>Zero-Crossing Detection</b>  | no                        |

## Version History

Introduced in R2022a

### R2022b: Enhancements to Fourth-Order Section Filter block

Starting in R2022b, when you pass numerator and denominator coefficients through the input ports of the Fourth-Order Section Filter block, you can change the number of sections (rows) in the filter coefficients while the simulation is running.

You can also specify the coefficients in the block dialog box using a `dsp.FourthOrderSectionFilter` object. Set **Coefficient source** to `Filter` object and specify a `dsp.FourthOrderSectionFilter` object directly in the **Filter** parameter. Alternatively, you can create this object in the MATLAB workspace or model workspace and specify the filter variable in the **Filter** parameter.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

## See Also

### Objects

`dsp.SOSFilter` | `dsp.FourthOrderSectionFilter`

### Blocks

Biquad Filter

## Frame Conversion

(Removed) Specify sampling mode of output signal

---

**Note** The Frame Conversion block has been removed.

---

### Library

Signal Management / Signal Attributes

dspsigattribs

### Description

The Frame Conversion block passes the input through to the output and sets the output sampling mode to the value of the **Sampling mode of output signal** parameter, which can be either **Frame-based** or **Sample-based**. The output sampling mode can also be inherited from the signal at the Ref (reference) input port, which you make visible by selecting the **Inherit output sampling mode from <Ref> input port** check box.

The Frame Conversion block does not make any changes to the input signal other than the sampling mode. In particular, the block does not rebuffer or resize 2-D inputs. Because 1-D vectors cannot be frame based, when the input is a length- $M$  1-D vector and the block is in **Frame-based** mode, the output is a frame-based  $M$ -by-1 matrix — that is, a single channel.

### Parameters

#### Inherit output sampling mode from <Ref> input port

Select to enable the Ref port from which the block inherits the output sampling mode.

#### Sampling mode of output signal

Specify the sampling mode of the output signal, **Frame-based** or **Sample-based**.

### Supported Data Types

| Port | Supported Data Types   |
|------|--|
| In   | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed only)</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> <li>• Enumerated</li> </ul> |



| Port   | Supported Data Types   |
|--------|--|
| Ref    | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed only)</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> <li>• Enumerated</li> </ul> |
| Output | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed only)</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> <li>• Enumerated</li> </ul> |

## See Also

|                         |                    |
|-------------------------|--------------------|
| Buffer                  | DSP System Toolbox |
| Check Signal Attributes | DSP System Toolbox |
| Convert 1-D to 2-D      | DSP System Toolbox |
| Convert 2-D to 1-D      | DSP System Toolbox |
| Inherit Complexity      | DSP System Toolbox |
| Unbuffer                | DSP System Toolbox |
| Probe                   | Simulink           |
| Reshape                 | Simulink           |
| Signal Specification    | Simulink           |

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

**HDL Architecture**

This block has one default HDL architecture.

**HDL Block Properties**

|                                  |  |
|----------------------------------|--|
| <b>ConstrainedOutputPipeline</b> | Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see "ConstrainedOutputPipeline" (HDL Coder). |
| <b>InputPipeline</b>             | Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "InputPipeline" (HDL Coder).                 |
| <b>OutputPipeline</b>            | Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "OutputPipeline" (HDL Coder).               |

**Complex Data Support**

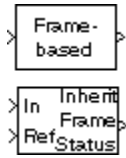
This block supports code generation for complex signals.

**Fixed-Point Conversion**

Design and simulate fixed-point systems using Fixed-Point Designer™.

## Frame Status Conversion (Obsolete)

Specify frame status of output as sample based or frame based



### Library

dspobslib

### Description

---

**Note** The Frame Status Conversion block has been removed.

---

The Frame Status Conversion block passes the input through to the output, and sets the output frame status to the **Output signal** parameter, which can be either **Frame-based** or **Sample-based**. The output frame status can also be inherited from the signal at the **Ref** (reference) input port, which is made visible by selecting the **Inherit output frame status from Ref input port** check box.

When the **Output signal** parameter setting or the inherited signal's frame status differs from the input frame status, the block changes the input frame status accordingly, but does not otherwise alter the signal. In particular, the block does not rebuffer or resize 2-D inputs. Because 1-D vectors cannot be frame based, when the input is a length- $M$  1-D vector, and the **Output signal** parameter is set to **Frame-based**, the output is a frame-based  $M$ -by-1 matrix (that is, a single channel).

When the **Output signal** parameter or the inherited signal's frame status matches the input frame status, the block passes the input through to the output unaltered.

### Parameters

#### Inherit output frame status from Ref input port

When selected, enables the **Ref** input port from which the block inherits the output frame status.

#### Output signal

The output frame status, **Frame-based** or **Sample-based**.

## Supported Data Types

| Port   | Supported Data Types   |
|--------|--|
| In     | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed only)</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul> |
| Ref    | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed only)</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul> |
| Output | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed only)</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul> |

## See Also

Check Signal Attributes

Convert 1-D to 2-D

Convert 2-D to 1-D

Inherit Complexity

DSP System Toolbox

DSP System Toolbox

DSP System Toolbox

DSP System Toolbox

## Version History

Introduced in R2008b

# From Audio Device

Read audio data from computer's audio device



## Compatibility

---

**Note** The From Audio Device block will be removed in a future release. Existing instances of the block continue to run. For new models, use the Audio Device Reader block from Audio Toolbox instead.

---

## Library

Sources

dspsrcs4

## Description

The From Audio Device block reads audio data from an audio device in real time. This block is not supported for use with the Simulink Model block.

Use the **Device** parameter to specify the device from which to acquire audio. This parameter is automatically populated based on the audio devices installed on your system. If you plug or unplug an audio device from your system, type `clear mex` at the MATLAB command prompt to update this list.

Use the **Number of channels** parameter to specify the number of audio channels in the signal. For example:

- Enter 2 if the audio source is two channels (stereo).
- Enter 1 if the audio source is single channel (mono).
- Enter 6 if you are working with a 5.1 speaker system.

The block's output is an  $M$ -by- $N$  matrix, where  $M$  is the number of consecutive samples and  $N$  is the number of audio channels.

Use the **Sample rate (Hz)** parameter to specify the number of samples per second in the signal. If the audio data is processed in uncompressed pulse code modulation (PCM) format, it should typically be sampled at one of the standard audio device rates: 8000, 11025, 22050, 44100, or 48000 Hz.

The range of supported audio device sample rates and data type formats, depend on both the sound card and the API which is chosen for the sound card.

Use the **Device data type** parameter to specify the data type of the audio data that the device is placing in the buffer. You can choose:

- 8-bit integer
- 16-bit integer
- 24-bit integer
- 32-bit float
- Determine from output data type

If you choose **Determine from output data type**, the following table summarizes the block's behavior.

| Output Data Type   | Device Data Type      |
|--|-----------------------|
| Double-precision floating point or single-precision floating point | 32-bit floating point |
| 32-bit integer   | 24-bit integer        |
| 16-bit integer   | 16-bit integer        |
| 8-bit integer  | 8-bit integer         |

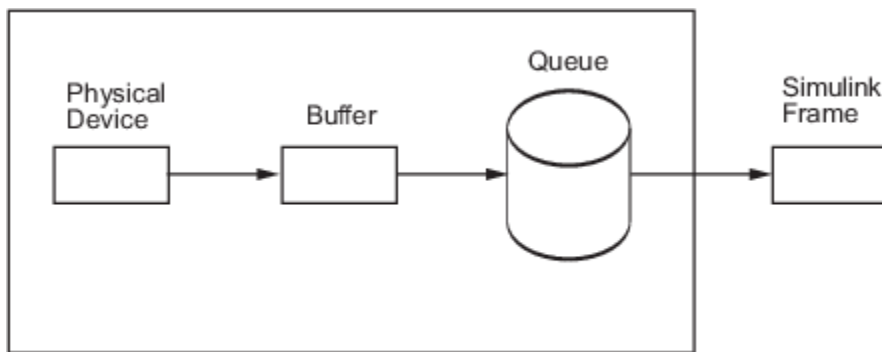
If you choose **Determine from output data type** and the device does not support a data type, the block uses the next lowest precision data type supported by the device.

Use the **Frame size (samples)** parameter to specify the number of samples in the block's output. Use the **Output data type** parameter to specify the data type of audio data output by the block.

The generated code for this block relies on prebuilt .dll files. You can run this code outside the MATLAB environment, or redeploy it, but be sure to account for these extra .dll files when doing so. The packNGo function creates a single zip file containing all of the pieces required to run or rebuild this code. See packNGo for more information.

### Buffering

The From Audio Device block buffers the data from the audio device using the process illustrated by the following figure.



From Audio Device Block

- 1 At the start of the simulation, the audio device begins writing the input data to a buffer. This data has the data type specified by the **Device data type** parameter.
- 2 When the buffer is full, the From Audio Device block writes the contents of the buffer to the queue. Specify the size of this queue using the **Queue duration (seconds)** parameter.

- 3 As the audio device appends audio data to the bottom of the queue, the From Audio Device block pulls data from the top of the queue to fill the Simulink frame. This data has the data type specified by the **Output data type** parameter.

Select the **Automatically determine buffer size** check box to allow the block to calculate a conservative buffer size using the following equation:

$$size = 2^{\lceil \log_2 \frac{sr}{10} \rceil}$$

In this equation, *size* is the buffer size, and *sr* is the sample rate. If you clear this check box, the **Buffer size (samples)** parameter appears on the block. Use this parameter to specify the buffer size in samples.

When the simulation throughput rate is lower than the hardware throughput rate, the queue, which is initially empty, fills up. If the queue is full, the block drops the incoming data from the audio device. You can monitor dropped samples using the optional **Overrun** output port. When the simulation throughput rate is higher than the hardware throughput rate, the From Audio Device block waits for new samples to become available.

### Channel Mapping

The term Channel Mapping refers to a 1-to-1 mapping that associates channels on the selected audio device to channels of the data. When you record audio, channel mapping allows you to specify which channel of the audio data directs input to a specific channel of audio. You can specify channel mapping as a vector of audio channel indices corresponding to each channel of data being read. The default value in the **Device Input Channels** parameter is 1:MAXINPUTCHANNELS. If you do not select the default mapping, you must specify the **Device Input Channels** parameter in the dialog box.

Example: The selected input audio device contains 8 channels. You want to read data from only channels 2, 4, 6, and redirect the data as follows:

- Audio Device channel 2 to first data channel
- Audio Device channel 4 to second data channel
- Audio Device channel 6 to third data channel

Thus you would specify the **Device Input Channels** as [2 4 6].

### Troubleshooting

#### Not Keeping Up in Real Time

When Simulink cannot keep up with an audio device that is operating in real time, the queue fills up and the block begins to lose audio data. Select the **Output number of samples by which the queue was overrun** check box to add an output port indicating when the queue was full. Here are several ways to deal with this situation:

- *Increase the queue duration.*

The **Queue duration (seconds)** parameter specifies the duration of the signal, in seconds, that can be buffered during the simulation. This is the maximum length of time that the block's data demand can lag behind the hardware's data supply.

- *Increase the buffer size.*

The size of the buffer processed in each interrupt from the audio device affects the performance of your model. If the buffer is too small, a large portion of hardware resources are used to write data to the queue. If the buffer is too big, Simulink must wait for the device to fill the buffer before it moves the data to the queue, which introduces latency.

- *Increase the simulation throughput rate.*

Two useful methods for improving simulation throughput rates are increasing the signal frame size and compiling the simulation into native code:

- Increase frame sizes and convert sample-based signals to frame-based signals throughout the model to reduce the amount of block-to-block communication overhead. This can increase throughput rates in many cases. However, larger frame sizes generally result in greater model latency due to initial buffering operations.
- Generate executable code with Simulink Coder code generation software. Native code runs much faster than Simulink and should provide rates adequate for real-time audio processing.

Other ways to improve throughput rates include simplifying the model and running the simulation on a faster PC processor. For other ideas on improving simulation performance, see “Delay and Latency” and “Optimize Performance” (Simulink).

### Running an Executable Outside MATLAB

To run your generated standalone executable application in Shell, you need to set your environment to the following:

| Platform | Command   |
|----------|---|
| Mac      | <pre>setenv DYLD_LIBRARY_PATH \$LD_LIBRARY_PATH: \$MATLABROOT/bin/ maci64 (csh/tcsh)  export DYLD_LIBRARY_PATH \$LD_LIBRARY_PATH: \$MATLABROOT/bin/ maci64 (Bash)</pre> |
| Linux    | <pre>setenv LD_LIBRARY_PATH \$LD_LIBRARY_PATH: \$MATLABROOT/bin/ glnxa64 (csh/tcsh)  export LD_LIBRARY_PATH \$LD_LIBRARY_PATH: \$MATLABROOT/bin/ glnxa64 (Bash)</pre>   |
| Windows  | <pre>set PATH = \$MATLABROOT\bin\win64;%PATH %</pre>  |

### Audio Hardware API

In order to communicate with the audio hardware on a given computer, the To Audio Device and From Audio Device blocks use the open-source PortAudio library. The PortAudio library supports a range of APIs designed to communicate with the audio hardware on a given platform. The following API choices were made when building the PortAudio library for the DSP System Toolbox product:

- Windows®: DirectSound, WDM—KS, ASIO™
- Linux®: OSS, ALSA



- Mac: CoreAudio

For Windows, the default is DirectSound, for Linux, the default is ALSA, and for Mac there is only one choice.

To determine the audio hardware API currently selected, type the following command in the MATLAB command prompt.

```
getpref('dsp', 'portaudioHostApi')
```

The output is a scalar indicating the choice of the API.

- 1 — DirectSound
- 3 — ASIO
- 7 — OSS
- 8 — ALSA
- 11 — WDM-KS

To select a particular API, type the following command in the MATLAB command prompt.

```
setpref('dsp', 'portaudioHostApi', N)
```

where  $N$  is a scalar. Choose  $N$  based on the API choice above.

## Parameters

### Device

Specify the device from which to acquire audio data.

### Use default mapping between Device Input Channels and Data

Select this check box to have the default mapping, where the data from the first channel of audio device is sent to the first channel of the input data, data from second channel of audio device is sent to second channel of data and so on. The maximum number of channels in the input data is determined by the **Number of channels** property.

### Number of channels

Specify the number of audio channels. This parameter is visible when the **Use default mapping between Device Input Channels and Data** check box is enabled.

### Device Input Channels

Specify the channel mapping. This parameter is visible when the **Use default mapping between Device Input Channels and Data** check box is disabled.

### Sample rate (Hz)

Specify the number of samples per second in the signal.

### Device data type

Specify the data type used by the device to acquire audio data.

### Automatically determine buffer size

Select this check box to enable the block to use a conservative buffer size.

### Buffer size (samples)

Specify the size of the buffer that the block uses to communicate with the audio device. This parameter is visible when the **Automatically determine buffer size** check box is cleared.

**Queue duration (seconds)**

Specify the size of the queue in seconds.

**Output number of samples by which the queue was overrun**

Select this check box to output the number of samples lost to queue overrun since the last transfer of a frame from the audio device. You can use this value to debug throughput problems and adjust the queues and buffers in your model. To learn how to improve throughput, see “Troubleshooting” on page 2-695.

**Frame size (samples)**

Specify the number of samples in the block's output signal.

**Output data type**

Select the data type of the block's output.

**Supported Data Types**

| Port    | Supported Data Types  |
|---------|---|
| Output  | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• 32-bit signed integers</li> <li>• 16-bit signed integers</li> <li>• 8-bit unsigned integers</li> </ul> |
| Overrun | 32-bit unsigned integer   |

**See Also**

From Multimedia File  
To Audio Device  
audiorecorder

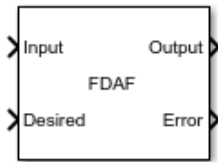
DSP System Toolbox  
DSP System Toolbox  
MATLAB

**Version History**

**Introduced in R2007b**

# Frequency-Domain Adaptive Filter

Compute output, error, and coefficients using frequency domain FIR adaptive filter



## Libraries:

DSP System Toolbox / Filtering / Adaptive Filters

## Description

The Frequency-Domain Adaptive Filter block implements an adaptive finite impulse response (FIR) filter in the frequency domain using the fast block least mean squares (LMS) algorithm. The **Filter length** and the **Block length** parameters specify the filter length and the block length values the algorithm uses. When you select the **Output filter FFT coefficients** check box, the block outputs the discrete Fourier transform of the current filter coefficients. The block offers the constrained and unconstrained versions of the algorithm with partitioned and nonpartitioned modes. For details, see “Algorithms” on page 2-705.

## Ports

### Input

**Input** — Data input  
column vector

The signal to be filtered by the frequency-domain FIR adaptive filter. The data input and the desired signal input must have the same size and data type. The length of the input vector must be divisible by the **Block length** parameter value.

The data input can be a variable-size signal as long as the frame length is a multiple of the **Block length**. You can change the number of elements in the column vector during the model simulation.

Data Types: `single` | `double`

**Desired** — Desired signal  
column vector

The frequency-domain adaptive filter adapts its filter weights to minimize the error, **Error**, and converge the input signal to match the desired signal as closely as possible.

The data input and the desired signal must have the same size and data type. The length of the desired signal vector must be divisible by the **Block length** parameter value.

The desired signal can be a variable-size signal as long as the frame length is a multiple of the **Block length**. You can change the number of elements in the column vector during the model simulation.

Data Types: `single` | `double`

**Mu** — Step size input

real scalar in the range (0,1]

Adaptation step size factor, specified as a real scalar in the range (0,1]. Using a small step size ensures a small steady-state error. However, a small step size decreases the resulting convergence speed of the adaptive filter. Increasing the step size improves the convergence speed, at the cost of increased steady-state mean squared error. When the step size value is 1, the algorithm provides the optimal tradeoff between the convergence speed and the steady-state mean squared error.

**Dependencies**

This port appears when you select the **Specify step size from port** parameter.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`**Leak** — Leakage factor input

real scalar in the range (0,1]

Leakage factor used in leaky adaptive filter, specified as a real scalar in the range (0,1]. When the value is less than 1, the block implements a leaky adaptive algorithm. When the value is 1, the block provides no leakage in the adapting method.

**Dependencies**

This port appears when you select the **Specify leakage factor from port** check box.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`**Avrg** — Averaging factor input

real scalar in the range (0,1]

Averaging factor used to compute the exponentially windowed fast Fourier transform (FFT) input signal powers for the coefficient updates, specified as a real scalar in the range (0,1].

**Dependencies**

This port appears when you select the **Specify averaging factor from port** check box.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`**Offset** — Offset for normalization terms

nonnegative real scalar

Offset for the normalization terms in the coefficient updates, specified as a nonnegative real scalar value. Use this value to avoid division by zero or division by very small numbers if any of the FFT input signal powers become very small.

**Dependencies**

This port appears when you select the **Specify offset from port** check box.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`**Adapt** — Enable filter coefficient updates

nonnegative real scalar

If you input a nonzero scalar value through this port, the block continuously updates its filter coefficients. If you input a zero through this port, the filter coefficients do not update, and their values remain at the current value.

**Dependencies**

This port appears when you select the **Enable adapt port** check box.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

**Reset** — Enable filter states reset

nonnegative real scalar

If you input a nonzero scalar value through this port, the block resets all internal states. If you input a zero through this port, the internal states are not reset.

**Dependencies**

This port appears when you select the **Enable reset port** check box.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

**Output****Output** — Filtered output

column vector

Filtered output, returned as a column vector. The block adapts its filter weights to converge the input signal to match the desired signal as closely as possible. The filter outputs the converged signal.

Data Types: `single` | `double`

**Error** — Difference between output and desired signal

column vector

Difference between the output signal and the desired signal, returned as a column vector. The objective of the adaptive filter is to minimize this error. The block adapts its weights to converge towards optimal filter weights which produce an output signal that matches the desired signal as closely as possible. For more details on how **Error** is computed, see “References” on page 2-707.

Data Types: `single` | `double`

Complex Number Support: Yes

**FFTCoeffs** — Current FFT coefficients of filter

row vector

Current discrete Fourier transform of the filter coefficients, returned as a row vector. For **Constrained FDAF** and **Unconstrained FDAF** algorithms, the length of this vector is equal to the sum of the **Filter length** value and the **Block length** value. This port initially outputs the FFT values of the **Initial time-domain coefficients** parameter. During the model simulation, this port outputs the FFT values of the current filter coefficients.

**Dependencies**

This port appears when you select the **Output filter FFT coefficients** check box.

Data Types: `single` | `double`

Complex Number Support: Yes

## Parameters

**Method** — Method to calculate filter coefficients

Constrained FDAF (default) | Partitioned constrained FDAF | Unconstrained FDAF | Partitioned unconstrained FDAF

Method used to calculate the filter coefficients, specified as::

- **Constrained FDAF** -- Imposes a gradient constraint on the filter tap weights.
- **Partitioned constrained FDAF** -- Partitions the impulse response of the filter to reduce latency.
- **Unconstrained FDAF** -- No gradient constraint is imposed on the filter tap weights.
- **Partitioned unconstrained FDAF** -- Partitions the impulse response of the filter to reduce latency. No gradient constraint is imposed on the filter tap weights.

For more details, see “Algorithms” on page 2-705.

**Filter length** — Length of filter coefficients vector

32 (default) | positive, integer-valued scalar

Length of the FIR filter coefficients vector, specified as a positive, integer-valued scalar.

**Block length** — Block length for coefficient updates

32 (default) | positive, integer-valued scalar

Block length for the coefficients update, specified as a positive, integer-valued scalar. The adaptive filter processes the input data and the desired signal as a block of samples of length set by this parameter. For details on how this data is processed by the filter, see “Algorithms” on page 2-705. The length of the input vector must be divisible by the **Block length** parameter value. The default value of the **Block length** parameter is set to the value of the **Filter length** parameter.

**Specify step size from port** — Flag to specify step size

off (default) | on

When you select this check box, the adaptation step size is input through the **Mu** port. When you clear this check box, the step size is specified on the block dialog through the **Step size** parameter.

**Step size** — Adaptation step size

1 (default) | real scalar in the range (0,1]

Adaptation step size factor, specified as a real scalar in the range (0,1]. Setting the **Step size** parameter to 1 provides the fastest convergence during adaptation.

**Tunable:** Yes

### Dependencies

This parameter appears when you clear the **Specify step size from port** check box.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32

**Specify leakage factor from port** — Flag to specify leakage factor

off (default) | on

When you select this check box, the leakage factor is input through the **Leak** port. When you clear this check box, the leakage factor is specified on the block dialog through the **Leakage factor** parameter.

**LeakageFactor** — Adaptation leakage factor

1 (default) | real scalar in the range (0,1]

Leakage factor used in leaky adaptive filter, specified as a real scalar in the range (0,1]. When the value is less than 1, the block implements a leaky adaptive algorithm. When the value is 1, the block provides no leakage in the adapting method.

**Tunable:** Yes**Dependencies**

This parameter appears when you clear the **Specify leakage factor from port** check box.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32

**Specify averaging factor from port** — Flag to specify averaging factor

off (default) | on

When you select this check box, the averaging factor for signal power is input through the **Avrg** port. When you clear this check box, the averaging factor is specified on the block dialog through the **Averaging factor** parameter.

**Averaging factor** — Averaging factor for signal power

0.9 (default) | real scalar in the range (0,1]

Averaging factor used to compute the exponentially windowed FFT input signal powers for the coefficient updates, specified as a real scalar in the range (0,1].

**Tunable:** Yes**Dependencies**

This parameter appears when you clear the **Specify averaging factor from port** check box.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32

**Specify offset from port** — Flag to specify offset

off (default) | on

When you select this check box, the offset for the normalization terms in the coefficient updates is input through the **Offset** port. When you clear this check box, the offset is specified on the block dialog through the **Offset** parameter.

**Offset** — Offset for normalization terms

0 (default) | nonnegative real scalar

Offset for the normalization terms in the coefficient updates, specified as a nonnegative real scalar value. Use this value to avoid division by zero or division by very small numbers if any of the FFT input signal powers become very small.

**Tunable:** Yes

**Dependencies**

This parameter appears when you clear the **Specify offset from port** check box.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

**Initial FFT input signal power** — Initial FFT input signal power

1 (default) | positive numeric scalar

Initial common value of all of the FFT input signal powers, specified as a positive numeric scalar value.

If you change this value during the simulation, the change takes effect only after a reset event occurs.

**Tunable:** Yes

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

**Initial time-domain coefficients** — Initial time-domain coefficients of filter

0 (default) | scalar | vector

Initial time-domain coefficients of the adaptive filter, specified as a scalar or a vector of length equal to the value you specify in the **Filter length** parameter. The adaptive filter block uses these coefficients to compute the initial frequency-domain filter coefficients.

If you change this value during the simulation, the change takes effect only after a reset event occurs.

**Tunable:** Yes

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

**Enable adapt port** — Flag to enable coefficient update

off (default) | on

When you select this check box, the **Adapt** input port is enabled. If you input a nonzero scalar value through this port, the block continuously updates its filter coefficients. If you input a zero through this port, the filter coefficients do not update and their values remain at the current value.

**Enable reset port** — Flag to reset internal states

off (default) | on

When you select this check box, the **Reset** input port is enabled. If you input a nonzero scalar value through this port, the block resets all internal states. If you input a zero through this port, the internal states are not reset.



**Output filter FFT coefficients** — Flag to output the DFT of the filter coefficients  
off (default) | on

When you select this check box, the **FFTCoeffs** output port is enabled. Through this port, the block outputs the discrete Fourier transform of the current filter coefficients.

**Simulate using** — Type of simulation to run  
Code generation (default) | Interpreted execution

Specify the type of simulation to run as one of the following:

- **Code generation** -- Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than **Interpreted execution**.
- **Interpreted execution** -- Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than **Code generation**.

## Block Characteristics

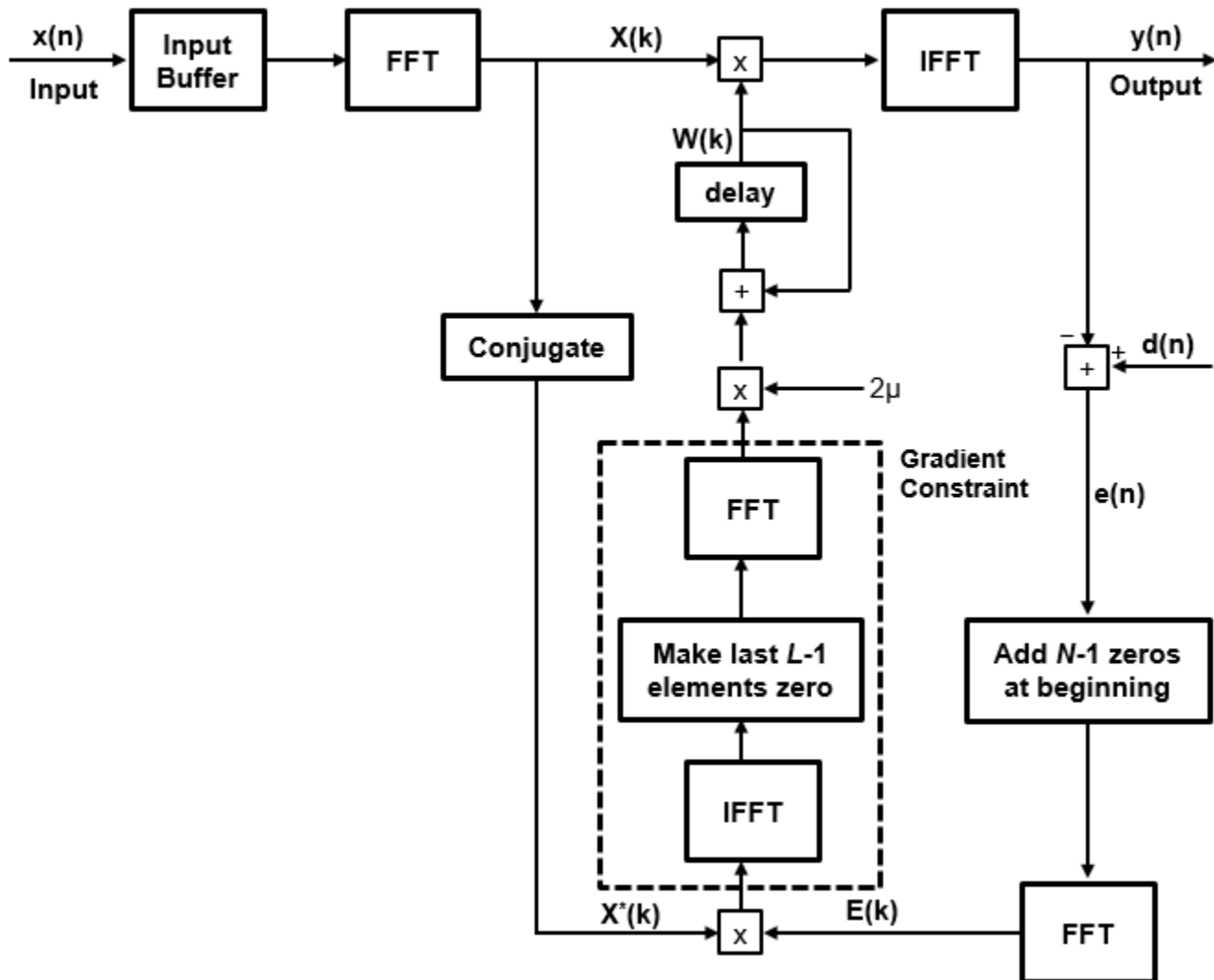
|                                 |                 |
|---------------------------------|-----------------|
| <b>Data Types</b>               | double   single |
| <b>Direct Feedthrough</b>       | no              |
| <b>Multidimensional Signals</b> | no              |
| <b>Variable-Size Signals</b>    | yes             |
| <b>Zero-Crossing Detection</b>  | no              |

## Algorithms

Frequency-domain adaptive filtering consists of three steps - filtering, error estimation, and tap-weight adaptation. This algorithm implements FIR filtering in the frequency domain using the overlap-save or overlap-add method. For more implementation details of these two methods, see the "Algorithms" on page 4-611 section in the `dsp.FrequencyDomainFIRFilter` object page. The error estimation and the tap-weight adaptation are implemented using the fast block LMS algorithm (FBLMS).

### Fast Block LMS Algorithm

The frequency-domain adaptive filter processes input data and the desired signal data as a block of samples using the fast block LMS (FBLMS) algorithm. Here is the block diagram of the frequency-domain adaptive filter using the FBLMS algorithm. The frequency-domain FIR filter in this diagram uses the overlap-save method.



where:

- $N$  -- Filter length
- $L$  -- Block length
- $\mu$  -- Step size parameter
- $x(n)$  -- Input signal
- $X(k)$  -- Transformed input signal in the frequency domain
- $d(n)$  -- Desired signal
- $e(n)$  -- Error between the desired signal and the filter output
- $E(k)$  -- Transformed error signal in the frequency domain
- $W(k)$  -- Tap-weights vector in the frequency domain

For more details on how the error is estimated and the tap-weights are adapted, see [2].

## Constrained and Unconstrained FBLMS Algorithms

The previous diagram is the constrained version. If you remove the gradient constraint portion of the algorithm, you have the unconstrained FBLMS implementation. For details on the convergence behavior of both constrained and unconstrained variations, see [2].

## Partitioned FBLMS Algorithm

The latency of the filter roughly equals the length of the FIR numerator. If the impulse response of the filter is very long, the latency becomes significantly large. The partitioned FBLMS algorithm reduces latency by partitioning the impulse response. The nonpartitioned frequency-domain FIR filtering is faster than the time-domain filtering for long impulse responses, at the cost of increased latency. To mitigate the latency and make the frequency domain filtering even more efficient, the algorithm partitions the impulse response into multiple short blocks and performs overlap-save or overlap-add on each block. The results of the different blocks are then combined to obtain the final output. The latency of this approach is of the order of the block length, rather than the entire impulse response length. This reduced latency comes at the cost of additional computation. For more details on the implementation, see [2].

## Version History

Introduced in R2018a

## References

- [1] Shynk, J.J. "Frequency-Domain and Multirate Adaptive Filtering." *IEEE Signal Processing Magazine*, Vol. 9, No. 1, pp. 14-37, Jan. 1992.
- [2] Farhang-Boroujeny, B., *Adaptive Filters: Theory and Applications*, Chichester, England, Wiley, 1998.
- [3] Stockham, T. G., Jr. "High Speed Convolution and Correlation." *Proceedings of the 1966 Spring Joint Computer Conference, AFIPS*, Vol 28, 1966, pp. 229-233.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

## See Also

### Objects

`dsp.FrequencyDomainFIRFilter` | `dsp.FrequencyDomainAdaptiveFilter` | `dsp.LMSFilter` | `dsp.RLSFilter` | `dsp.AffineProjectionFilter` | `dsp.AdaptiveLatticeFilter` | `dsp.FilteredXLMSFilter` | `dsp.FIRFilter` | `dsp.FastTransversalFilter`

### Blocks

LMS Filter | RLS Filter | LMS Update | Kalman Filter | Fast Block LMS Filter | Block LMS Filter | Frequency-Domain FIR Filter

### Topics

"Overview of Adaptive Filters and Applications"

“Noise Cancellation in Simulink Using Normalized LMS Adaptive Filter”  
“Variable-Size Signal Support DSP System Objects”

# Frequency-Domain FIR Filter

Filter input signal in the frequency domain



## Libraries:

DSP System Toolbox / Filtering / Filter Implementations

## Description

The Frequency-Domain FIR Filter block implements frequency-domain, fast Fourier transform (FFT)-based filtering to filter a streaming input signal. In the time domain, the filtering operation involves a convolution between the input and the impulse response of the finite impulse response (FIR) filter. In the frequency domain, the filtering operation involves the multiplication of the Fourier transform of the input and the Fourier transform of the impulse response. The frequency-domain filtering becomes more efficient than time-domain filtering as the impulse response grows longer. You can specify the filter coefficients directly in the frequency domain by setting **Numerator domain** to Frequency.

This block uses the overlap-save and overlap-add methods to perform the frequency-domain filtering. For filters with a long impulse response length, the latency inherent to these two methods can be significant. To mitigate this latency, the Frequency-Domain FIR Filter block partitions the impulse response into shorter blocks and implements the overlap-save and overlap-add methods on these shorter blocks. To partition the impulse response, select the **Partition numerator to reduce latency** check box. For more details on these two methods and on reducing latency through impulse response partitioning, see “Algorithms” on page 2-715.

## Ports

### Input

**x** — Data Input  
vector | matrix

Data input, specified as a vector or matrix. This block supports variable-size input signals. That is, you can change the input frame size (number of rows) even after calling the algorithm. However, the number of channels (number of columns) must remain constant.

This port is unnamed until you select the **Specify coefficients from input port** parameter.

Data Types: `single` | `double`

**NUM** — Time domain FIR filter coefficients  
row vector

Time domain FIR filter coefficients, specified as a row vector.

### Dependencies

This port appears when you set **Numerator domain** to Time and select the **Specify coefficients from input port** parameter.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

**NUMFFT** — Frequency domain FIR filter coefficients  
row vector

Frequency domain FIR filter coefficients, specified as a row vector or a matrix. When you clear the **Partition numerator to reduce latency** parameter, the coefficients input through this port must be a row vector. The FFT length is equal to the length of the vector input. When you select the **Partition numerator to reduce latency** parameter, **Frequency response** must be a  $2P$ -by- $N$  matrix, where  $P$  is the partition size, and  $N$  is the number of partitions.

### Dependencies

This port appears when you set **Numerator domain** to Frequency and select the **Specify frequency response from input port** parameter.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

### Output Arguments

**y** — Filtered output  
vector | matrix

Filtered output, returned as a vector or matrix. The size, data type, and complexity of the output match those of the input.

This port is unnamed until you select the **Output filter latency** parameter and click **Apply**.

Data Types: `single` | `double`

**latency** — Filter latency  
positive integer

Filter latency, returned as a scalar. This latency is inherent to overlap-add and overlap-save methods and does not include the group delay of the filter. This port appears only when you select the **Output filter latency** check box.

This port is unnamed until you select the **Output filter latency** check box and click **Apply**.

Data Types: `uint32`

## Parameters

**Frequency-domain filter method** — Filtering method in frequency domain  
`Overlap-save (default)` | `Overlap-add`

Filtering method in the frequency domain, specified as either `Overlap-save` or `Overlap-add`. For more details on these two methods, see “Algorithms” on page 2-715

**Numerator domain** — Numerator domain  
`Time (default)` | `Frequency`

Domain of the filter coefficients, specified as one of the following:

- Time -- Specify the time-domain filter coefficients in the **Filter coefficients** parameter or through the **NUM** input port.
- Frequency -- Specify the filter's frequency response in the **Frequency response** parameter or through the **NUMFFT** input port.

**Specify coefficients from input port** — Flag to specify lowpass filter coefficients  
off (default) | on

When you select this check box, the FIR filter coefficients are input through the port, **NUM**. When you clear this check box, the coefficients are specified on the block dialog through the **Filter coefficients** parameter.

To view the filter response, clear this check box, specify the coefficients on the block dialog, and click on the **View Filter Response** button.

#### Dependencies

To enable this parameter, set **Numerator domain** to Time.

**Filter coefficients** — filter coefficients  
fir1(100,0.3) (default) | row vector

FIR filter coefficients, specified as a row vector.

#### Dependencies

To enable this parameter, set **Numerator domain** to Time and clear the **Specify coefficients from input port** parameter.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64  
Complex Number Support: Yes

**Specify frequency response from input port** — Flag to specify frequency-domain filter coefficients  
off (default) | on

When you select this check box, the FIR filter coefficients in the frequency domain are input through the port, **NUMFFT**. When you clear this check box, the coefficients are specified on the block dialog through the **Frequency response** parameter.

To view the filter response, clear this check box, specify the frequency response on the block dialog, and click on the **View Filter Response** button.

#### Dependencies

To enable this parameter, set **Numerator domain** to Frequency.

**Frequency response** — Filter coefficients  
fft(fir1(100,0.3),202) (default) | row vector | matrix

Frequency response of the filter, specified as a row vector or a matrix. When you clear the **Partition numerator to reduce latency** parameter, **Frequency response** must be a row vector. The FFT length is equal to the length of the **Frequency response** vector. When you select the **Partition**

**numerator to reduce latency** parameter, **Frequency response** must be a  $2P$ -by- $N$  matrix, where  $P$  is the partition size, and  $N$  is the number of partitions.

### Dependencies

To enable this parameter, set **Numerator domain** to Frequency and clear the **Specify frequency response from input port** parameter.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`  
Complex Number Support: Yes

**Partition numerator to reduce latency** — Flag to partition the numerator to reduce latency  
off (default) | on

Flag to partition the numerator to reduce latency, specified as one of the following:

- off -- The filter uses the traditional overlap-save or overlap-add method. The latency in this case is FFT length -  $NumLen + 1$ .  $NumLen$  is the length of the numerator vector you specify in the **Filter coefficients** parameter.
- on -- In this mode, the block partitions the numerator into segments of length specified by the **Numerator partition length** parameter. The filter performs overlap-save or overlap-add on each partition, and combines the partial results to form the overall output. The latency is now reduced to the partition length.

**Numerator partition length** — Partition length of numerator  
32 (default) | positive integer

Partition length of the numerator, specified as a positive integer less than or equal to the length of the numerator.

### Dependencies

This parameter applies only when you set **Numerator domain** to Time and select the **Partition numerator to reduce latency** parameter.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**Time-domain numerator length** — Time-domain numerator length  
101 (default) | positive integer-valued scalar

Time-domain numerator length, specified as a positive integer-valued scalar.

### Dependencies

This parameter applies only when you set **Numerator domain** to Frequency and clear the **Partition numerator to reduce latency** check box.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

**Inherit FFT length from numerator length** — Flag to inherit FFT length from the numerator length  
on (default) | off



When you select this check box, the FFT length equals twice the numerator length. When you clear this check box, you specify the FFT length through the **FFT length** parameter.

#### Dependencies

This parameter applies only when you set **Numerator domain** to Time and clear the **Partition numerator to reduce latency** parameter.

**FFT length** — FFT length  
1024 (default) | positive integer

The FFT length you specify must be greater than or equal to the length of the numerator vector you specify in the **Filter coefficients** parameter.

#### Dependencies

This parameter applies when you set **Numerator domain** to Time, clear the **Partition numerator to reduce latency** and the **Inherit FFT length from numerator length** parameters.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**Filter is real** — Flag to specify if filter is real  
true (default) | false

Flag to specify if the filter is real, specified as true or false.

#### Dependencies

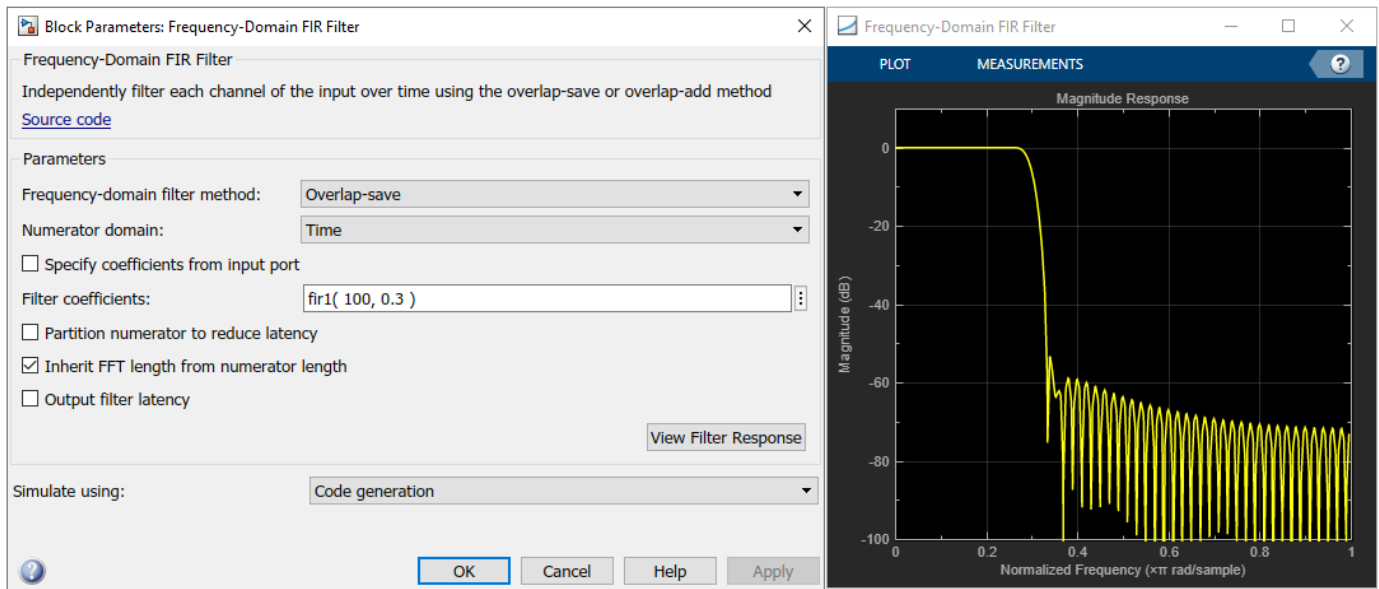
This parameter applies when **Numerator domain** to Frequency.

**Output filter latency** — Flag to output filter latency  
'off' (default) | 'on'

When you select this check box and click **Apply**, the block outputs the filter latency through the **latency** port.

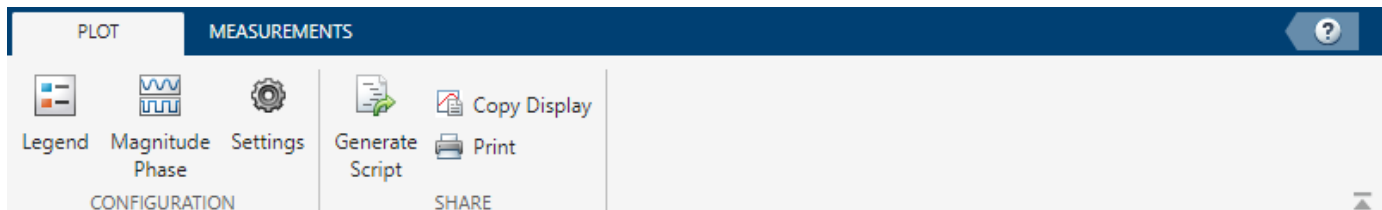
**View Filter Response** — Visualize frequency response  
button

Opens the Dynamic Filter Visualizer window and displays the magnitude and phase response of the frequency domain FIR filter. The response is based on the parameters you select in the block dialog box. To update the response while the visualizer is running, modify the parameters in the dialog box and click **Apply**.

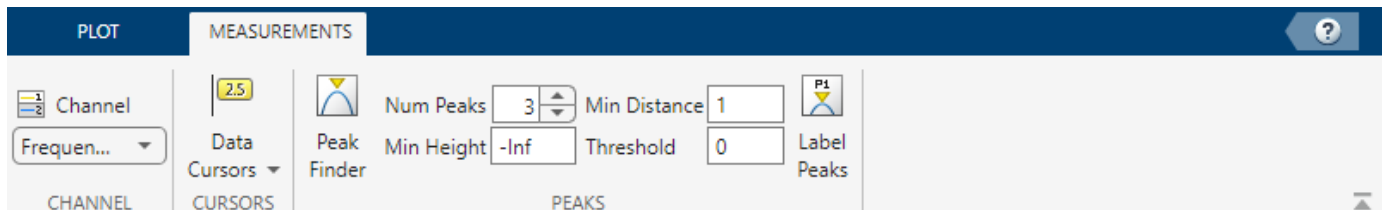


You can configure the plot settings and the frequency response measurements from the interface of the visualizer.

On the **Plot** tab, you can enable the legend, specify to plot both magnitude and phase responses of the filter, modify the plot settings, generate a script to re-create the plot, and even save or share the settings.



On the **Measurements** tab, you can enable data cursors, and display the peak values of the filter response.



For more details on the Dynamic Filter Visualizer interface and its tools, see “Configure Filter Visualizer”.

**Simulate using** — Type of simulation to run  
Code generation (default) | Interpreted execution

Specify the type of simulation to run as one of the following:

- **Code generation** -- Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than **Interpreted execution**.
- **Interpreted execution** -- Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than **Code generation**.

## Block Characteristics

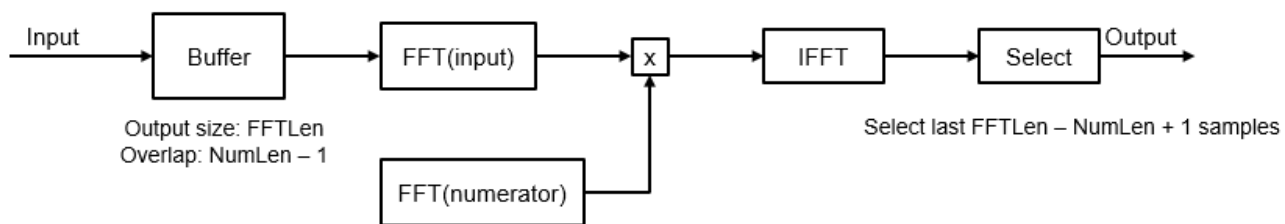
|                                 |                 |
|---------------------------------|-----------------|
| <b>Data Types</b>               | double   single |
| <b>Direct Feedthrough</b>       | no              |
| <b>Multidimensional Signals</b> | no              |
| <b>Variable-Size Signals</b>    | yes             |
| <b>Zero-Crossing Detection</b>  | no              |

## Algorithms

Overlap-save and overlap-add are the two frequency-domain FFT-based filtering methods this algorithm uses.

### Overlap-Save

The overlap-save method is implemented using the following approach:



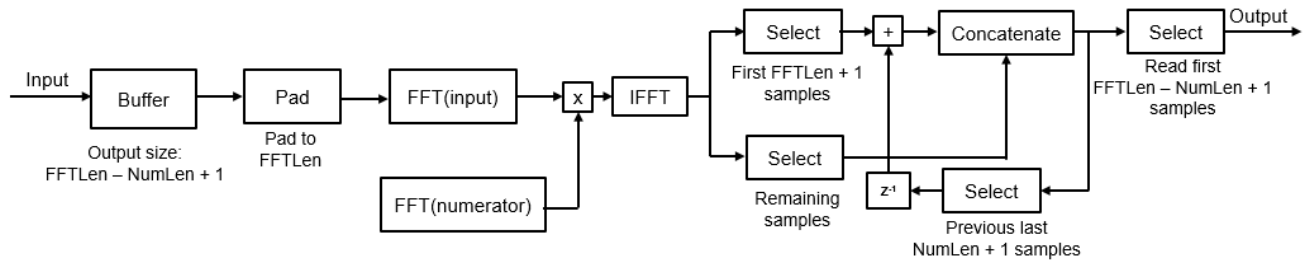
The input stream is partitioned into overlapping blocks of size  $FFTLen$ , with an overlap factor of  $NumLen - 1$  samples.  $FFTLen$  is the FFT length and  $NumLen$  is the length of the FIR filter numerator. The FFT of each block of input samples is computed and multiplied with the length- $FFTLen$  FFT of the FIR numerator. The inverse fast Fourier transform (IFFT) of the result is performed, and the last  $FFTLen - NumLen + 1$  samples are saved. The remaining samples are dropped.

The latency of overlap-save is  $FFTLen - NumLen + 1$ . The first  $FFTLen - NumLen + 1$  samples are equal to zero. The filtered value of the first input sample appears as the  $FFTLen - NumLen + 2$  output sample.

Note that the FFT length must be larger than the numerator length, and is typically set to a value much greater than  $NumLen$ .

## Overlap-Add

The overlap-add method is implemented using the following approach:



The input stream is partitioned into blocks of length  $FFLen - NumLen + 1$ , with no overlap between consecutive blocks. Similar to overlap-save, the FFT of the block is computed, and multiplied by the FFT of the FIR numerator. The IFFT of the result is then computed. The first  $NumLen + 1$  samples are modified by adding the values of the last  $NumLen + 1$  samples from the previous computed IFFT.

The latency of overlap-add is  $FFLen - NumLen + 1$ . The first  $FFLen - NumLen + 1$  samples are equal to zero. The filtered value of the first input sample appears as the  $FFLen - NumLen + 2$  output sample.

## Reduce Latency Through Impulse Response Partitioning

With an FFT length that is twice the length of the FIR numerator, the latency roughly equals the length of the FIR numerator. If the impulse response is very long, the latency becomes significantly large. However, frequency domain FIR filtering is still faster than the time-domain filtering. To mitigate the latency and make the frequency domain filtering even more efficient, the algorithm partitions the impulse response into multiple short blocks and performs overlap-save or overlap-add on each block. The results of the different blocks are then combined to obtain the final output. The latency of this approach is of the order of the block length, rather than the entire impulse response length. This reduced latency comes at the cost of additional computation. For more details, see [1].

## Version History

Introduced in R2017b

## References

- [1] Stockham, T. G., Jr. "High Speed Convolution and Correlation." *Proceedings of the 1966 Spring Joint Computer Conference, AFIPS*, 28 (1966): 229-233.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

## See Also

### Objects

`dsp.FrequencyDomainFIRFilter` | `dsp.FIRFilter` | `dsp.FrequencyDomainAdaptiveFilter`

### Blocks

Variable Bandwidth FIR Filter | FFT | Frequency-Domain Adaptive Filter

## From Multimedia File

Stream from multimedia file



### Libraries:

Audio Toolbox / Sources

DSP System Toolbox / Sources

## Description

The From Multimedia File block reads audio samples, video frames, or both from a multimedia file and generates a signal with one of the following data types and amplitude ranges.

| Output Data Type | Output Amplitude Range                            |
|------------------|---|
| double           | $\pm 1$   |
| single           | $\pm 1$   |
| int16            | $-32768$ to $32767$ ( $-2^{15}$ to $2^{15} - 1$ ) |
| uint8            | 0 to 255  |

The block imports data from the file into a Simulink model.

---

**Note** This block supports code generation for a host computer that has file I/O available. You cannot use this block with Simulink Desktop Real-Time software because that product does not support file I/O. The generated code for this block relies on prebuilt library files. For more details, see “Code Generation” on page 2-727.

---

## Limitations

To support OPUS file format on a Linux machine, you must have `libsndfile 1.0.29` or a later version installed. On Windows and Mac, this file is shipped with MATLAB and is available under `$MATLABROOT/bin/<platform>`.

## Ports

The output ports of the From Multimedia File block change according to the content of the multimedia file. If the file contains only video frames, the **Image** port, intensity **I** port, **R,G,B** ports, or **Y,Cb,Cr** ports appear based on the settings on the block dialog box. If the file contains only audio samples, the **Audio** port appears on the block. If the file contains both audio and video, you can select the data to emit. This section describes these ports in more detail.

### Output

**Image** — Color image or video frame  
array

Color image or video frame, returned as an  $M$ -by- $N$ -by- $P$  array.  $M$  and  $N$  are the number of rows and columns in each image.  $P$  is the number of color channels in each image.

**Dependencies**

To enable this port:

- Set the **File name** parameter to the name of a multimedia file that contains video or image data. If the specified file contains both audio and video data, set the **Multimedia outputs** parameter to `Video only` or `Video and audio`.
- Set the **Output color format** parameter to `RGB` and **Image signal** parameter to `One multidimensional signal`.
- This port is named as **Simulink Image** if the **Output color format** parameter is set to `RGB` and the **Image signal** parameter is set to `Simulink image signal`.

Data Types: `single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Simulink.ImageType`

**I** — Grayscale image or video frame  
matrix

Grayscale image or video frame, returned as an  $M$ -by- $N$  matrix.  $M$  and  $N$  are the number of rows and columns in the grayscale image.

**Dependencies**

To enable this port:

- Set the **File name** parameter to the name of a multimedia file that contains video or image data. If the specified file contains both audio and video data, set the **Multimedia outputs** parameter to `Video only` or `Video and audio`.
- Set the **Output color format** parameter to `Intensity`.

Data Types: `single | double | int8 | int16 | int32 | uint8 | uint16 | uint32`

**R** — Red color component of image or video frame  
matrix

Red color component of the image or video frame, returned as an  $M$ -by- $N$  matrix.  $M$  and  $N$  are the number of rows and columns in the image.

Outputs from the **R**, **G**, or **B** ports must have the same dimensions.

**Dependencies**

To enable this port:

- Set the **File name** parameter to the name of a multimedia file that contains video or image data. If the specified file contains both audio and video data, set the **Multimedia outputs** parameter to `Video only` or `Video and audio`.
- Set the **Output color format** parameter to `RGB` and **Image signal** parameter to `Separate color signals`.

Data Types: `single | double | int8 | int16 | int32 | uint8 | uint16 | uint32`

**G** — Green color component of image or video frame  
matrix

Green color component of the image or video frame, returned as an  $M$ -by- $N$  matrix.  $M$  and  $N$  are the number of rows and columns in the image.

Outputs from the **R**, **G**, or **B** ports must have the same dimensions.

#### Dependencies

To enable this port:

- Set the **File name** parameter to the name of a multimedia file that contains video or image data. If the specified file contains both audio and video data, set the **Multimedia outputs** parameter to **Video only** or **Video and audio**.
- Set the **Output color format** parameter to **RGB** and **Image signal** parameter to **Separate color signals**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

**B** — Blue color component of image or video frame  
matrix

Blue color component of the image or video frame, returned as an  $M$ -by- $N$  matrix.  $M$  and  $N$  are the number of rows and columns in the image.

Outputs from the **R**, **G**, or **B** ports must have the same dimensions.

#### Dependencies

To enable this port:

- Set the **File name** parameter to the name of a multimedia file that contains video or image data. If the specified file contains both audio and video data, set the **Multimedia outputs** parameter to **Video only** or **Video and audio**.
- Set the **Output color format** parameter to **RGB** and **Image signal** parameter to **Separate color signals**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

**Y** — Luma component of image or video frame  
matrix

Luma component of the image or video frame, returned as an  $M$ -by- $N$  matrix.  $M$  and  $N$  are the number of rows and columns in the image.

#### Dependencies

To enable this port:

- Set the **File name** parameter to the name of a multimedia file that contains video or image data. If the specified file contains both audio and video data, set the **Multimedia outputs** parameter to **Video only** or **Video and audio**.
- Set the **Output color format** parameter to **YCbCr 4:2:2**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

**Cb** — Blue-difference chrominance component of image or video frame  
matrix



Blue-difference chrominance component of the image or video frame, returned as an  $M$ -by- $N/2$  matrix.  $M$  and  $N$  are the number of rows and columns in the input image.

#### Dependencies

To enable this port:

- Set the **File name** parameter to the name of a multimedia file that contains video or image data. If the specified file contains both audio and video data, set the **Multimedia outputs** parameter to `Video only` or `Video and audio`.
- Set the **Output color format** parameter to `YCbCr 4:2:2`.

Data Types: `single | double | int8 | int16 | int32 | uint8 | uint16 | uint32`

**Cr** — Red-difference chrominance component of image or video frame matrix

Red-difference chrominance component of the image or video frame, returned as an  $M$ -by- $N/2$  matrix.  $M$  and  $N$  are the number of rows and columns in each input image.

#### Dependencies

To enable this port:

- Set the **File name** parameter to the name of a multimedia file that contains video or image data. If the specified file contains both audio and video data, set the **Multimedia outputs** parameter to `Video only` or `Video and audio`.
- Set the **Output color format** parameter to `YCbCr 4:2:2`.

Data Types: `single | double | int8 | int16 | int32 | uint8 | uint16 | uint32`

**Audio** — Audio signal vector | matrix

Audio signal, returned as an  $M$ -element vector or an  $M$ -by- $N$  matrix.  $M$  is the number of samples in each channel and  $N$  is the number of channels.

#### Dependencies

To enable this port, set the **File name** parameter to the name of a multimedia file that contains audio data. If the specified file contains both audio and video data, set the **Multimedia outputs** parameter to `Audio only` or `Video and audio`.

Data Types: `single | double | int16 | uint8`

**EOF** — End-of-file indicator scalar

End-of-file indicator, returned as a logical scalar for each video frame or audio sample. The end-of-file indicator is 1 when the block outputs the final video frame or audio sample from the file, and is 0 otherwise.

#### Dependencies

To enable this port, select the **Output end-of-file indicator** parameter.

Data Types: `Boolean`

## Parameters

### Main Tab

**File name** — Name of multimedia file

speech\_dft.mp3 (default) | character vector

Specify the name of the multimedia file for the block to read. The block determines the type of file (audio and video, audio only, or video only) and provides the associated parameters.

If the location of the file does not appear on the MATLAB path, use the **Browse** button to specify the full path. If the location of the file appears on the MATLAB path, enter only the file name.

On Windows platforms, this parameter supports URLs that point to MMS (Microsoft Media Server) streams.

**Inherit sample time from file** — Inherit sample time from file

on (default) | off

Specify whether the block inherits the sample time from the multimedia file or uses a specified sample time. When you select this parameter, the block inherits the sample time from the input multimedia file. When you clear this parameter, the block uses the sample time that you specify in the **Desired sample time** parameter.

---

**Tip** Select this parameter when you do not know the intended sample rate of the multimedia file.

---

**Desired sample time** — Block sample time

1/30 (default) | real positive scalar

Specify the block sample time.

### Dependencies

To enable this parameter, clear the **Inherit sample time from file** parameter.

**Number of times to play file** — Number of times to play multimedia file

inf (default) | positive integer

Specify the number of times to play the input multimedia file. The default value of `inf` specifies that the block repeats playing the multimedia file until the simulation stop time.

**Read range** — Range of audio samples

[1 Inf] (default) | two-element vector

Specify the sample range from which to read as a two-element row vector in the form of `[StartSample EndSample]`, where *StartSample* is the sample at which file reading starts, and *EndSample* is the sample at which file reading stops.

**Dependencies**

To enable this parameter, set the **File name** parameter to the name of a multimedia file that contains audio data. If the specified file contains both audio and video data, set the **Multimedia outputs** parameter to `Audio only`.

**Output end-of-file indicator** — End-of-file indicator

`off (default) | on`

Use this parameter to determine whether the output is the last video frame or audio sample in the multimedia file. When you select this parameter, a Boolean output port labeled **EOF** appears on the block. The output from the **EOF** port defaults to 1 when the last video frame or audio sample is output from the block. Otherwise, the output from the **EOF** port defaults to 0.

**Multimedia outputs** — Type of media

`Audio only (default) | Video and audio | Video only`

Specify the type of media to read from a file containing both video and audio data.

**Dependencies**

To enable this parameter, set the **File name** parameter to the name of a multimedia file that contains both audio and video data.

**Samples per audio channel** — Samples per audio channel

`1024 (default) | positive integer`

Specify number of samples per audio channel. This parameter becomes available for files containing audio.

**Dependencies**

To enable this parameter, set the **File name** parameter to the name of a multimedia file that contains audio data. If the specified file contains both audio and video data, set the **Multimedia outputs** parameter to `Audio only`.

**Output color format** — Color format of video

`RGB (default) | Intensity | YCbCr 4:2:2`

Specify the output color format of the input video data. The block supports the RGB, Intensity, and YCbCr 4:2:2 color formats.

If you select RGB, use the **Image signal** parameter to specify how to output a color signal.

**Dependencies**

To enable this parameter, set the **File name** parameter to the name of a multimedia file that contains video data. If the specified file contains both audio and video data, set the **Multimedia outputs** parameter to `Video only` or `Video and audio`.

**Image signal** — Output port configuration for RGB video

One multidimensional signal (default) | Separate color signals | Simulink image signal

Specify how the block outputs an RGB color video. If you select:

- **One multidimensional signal** -- The block outputs an  $M$ -by- $N$ -by- $P$  color video signal through the **Image** port, where  $P$  is the number of color planes.
- **Separate color signals** -- The block outputs the video data using three separate color channels, divided across the **R**, **G**, and **B** ports. Each port outputs one  $M$ -by- $N$  plane of an RGB video stream.
- **Simulink image signal** -- The block outputs a video in `Simulink.ImageType` datatype.

### Dependencies

To enable this parameter:

- Set the **File name** parameter to the name of a multimedia file that contains video data. If the specified file contains both audio and video data, set the **Multimedia outputs** parameter to `Video only` or `Video and audio`.
- Set the **Output color format** parameter to `RGB`.

### Data Types Tab

**Audio output data type** — Audio output data type

double (default) | single | int16 | uint8

Specify the data type for the audio samples output at the **Audio** port.

### Dependencies

To enable this parameter, set the **File name** parameter to the name of a multimedia file that contains audio data. If the specified file contains both audio and video data, set the **Multimedia outputs** parameter to `Audio only` or `Video and audio`.

**Video output data type** — Video output data type

single (default) | double | int8 | uint8 | int16 | uint16 | int32 | uint32 | Inherit from file

Specify the data type of the video frames output at the **R**, **G**, **B**, or **Image** ports..

### Dependencies

To enable this parameter, set the **File name** parameter to the name of a multimedia file that contains video data. If the specified file contains both audio and video data, set the **Multimedia outputs** parameter to `Video only` or `Video and audio`.

## Block Characteristics

|                           |  |
|---------------------------|--|
| <b>Data Types</b>         | double   integer <sup>a</sup>   single |
| <b>Direct Feedthrough</b> | no                                     |

|                                 |    |
|---------------------------------|----|
| <b>Multidimensional Signals</b> | no |
| <b>Variable-Size Signals</b>    | no |
| <b>Zero-Crossing Detection</b>  | no |

a Supports 16- and 32-bit signed and 8-bit unsigned integers.

## More About

### Supported Platforms and File Types

The supported file formats available to you depends on the codecs installed on your system.

#### Windows Platforms Supported File Formats

With the necessary Windows DirectShow codecs installed on your system, the From Multimedia File Block supports many video and audio file formats. This block performs best on platforms with Version 9.0 or later of DirectX® software.

The following table lists the most common file formats.

| <b>Multimedia Types</b> | <b>File Name Extensions</b>   |
|-------------------------|---|
| Image files             | .jpg, .bmp, .png  |
| Video files             | .qt, .mov, .avi, .asf, .asx, .wmv, .mpg, .mpeg, .mp2, .mp4, .m4v                      |
| Audio files             | .wav, .wma, .avi, .aif, .aifc, .aiff, .mp3, .au, .snd, .mp4, .m4a, .flac, .ogg, .opus |

The default for image files is .png, for video files is .avi, and for audio files is .mp3.

Windows 7 and later versions of Windows ship with a limited set of 64-bit video and audio codecs. If the From Multimedia File block cannot work on a compressed multimedia file, save the multimedia file to a file format supported by the block.

If you use Windows, use Windows Media® Player Version 11 or later with this block for best results.

#### Non- Windows Platform Supported File Formats

The following table lists the most common file formats.

| <b>Multimedia Types</b> | <b>File Name Extensions</b>   |
|-------------------------|---|
| Video files             | .avi, .mj2, .mov, .mp4, .m4v  |
| Audio files             | .avi, .mp3, .mp4, .m4a, .wav, .flac, .ogg, .aif, .aifc, .aiff, .au, .opus, .snd |

The default for video files is .avi and for audio files is .mp3.

### Sample Rates

The sample rate that the block uses depends on the audio and video sample rate. While the From Multimedia File block operates at a single rate in Simulink, the underlying audio and video streams can produce different rates. In some cases, when the block outputs both audio and video, it makes a small adjustment to the video rate.

### Sample Time Calculations Used for Video and Audio Files

$$\text{Sample time} = \frac{\text{ceil}(\text{AudioSampleRate}/\text{FPS})}{\text{AudioSampleRate}}.$$

When the audio sample time  $\frac{\text{AudioSampleRate}}{\text{FPS}}$  is noninteger, the equation cannot reduce to  $\frac{1}{\text{FPS}}$ .

In this case, to prevent synchronization problems, the block drops the corresponding video frame when the audio stream leads the video stream by more than  $\frac{1}{\text{FPS}}$ .

In summary, the block outputs one video frame at each Simulink time step. To calculate the number of audio samples to output at each time step, the block divides the audio sample rate by the video frame rate (fps). If the audio sample rate does not divide evenly by the number of video frames per second, the block rounds the number of audio samples up to the nearest whole number. If necessary, the block periodically drops a video frame to maintain synchronization for large files.

### Running Executable Outside MATLAB

To run your generated standalone executable application in Shell, you need to set your environment to the following:

| Platform | Command   |
|----------|---|
| Mac      | <pre>setenv DYLD_LIBRARY_PATH "\$ {DYLD_LIBRARY_PATH}:\$MATLABROOT/bin/ maci64" (csh/tcsh)  export DYLD_LIBRARY_PATH= \$DYLD_LIBRARY_PATH:\$MATLABROOT/bin/ maci64 (Bash)</pre> <p>For more information, see Append library path to "DYLD_LIBRARY_PATH" in MAC.</p> |
| Linux    | <pre>setenv LD_LIBRARY_PATH \$ {LD_LIBRARY_PATH}:\$MATLABROOT/bin/ glnxa64 (csh/tcsh)  export LD_LIBRARY_PATH= \$LD_LIBRARY_PATH:\$MATLABROOT/bin/ glnxa64 (Bash)</pre>   |
| Windows  | <pre>set PATH=%PATH%;%MATLABROOT%\bin\win64</pre>   |

## Version History

Introduced before R2006a

### R2022b: Support for OPUS audio file format

The From Multimedia File block supports audio files in the OPUS file format (.opus).

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Usage notes and limitations:

- Host computer only. Excludes Simulink Desktop Real-Time code generation.
- The executable generated from this block relies on prebuilt dynamic library files (.dll files) included with MATLAB. Use the `packNGo` function to package the code generated from this block and all the relevant files in a compressed ZIP file. Using this ZIP file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see “How To Run a Generated Executable Outside MATLAB”.

## See Also

### Blocks

To Multimedia File

### Topics

“How To Run a Generated Executable Outside MATLAB”

## From Wave Device (Obsolete)

Read audio data from standard audio device in real-time (32-bit Windows operating systems only)



### Library

dspwin32

### Description

---

**Note** The From Wave Device block is still supported but is likely to be obsoleted in a future release. We strongly recommend replacing this block with the From Audio Device block.

---

The From Wave Device block reads audio data from a standard Windows audio device in real-time. It is compatible with most popular Windows hardware, including Sound Blaster cards. (Models that contain both this block and the To Wave Device block require a *duplex-capable* sound card.)

The **Use default audio device** parameter allows the block to detect and use the system's default audio hardware. This option should be selected on systems that have a single sound device installed, or when the default sound device on a multiple-device system is the desired source. In cases when the default sound device is not the desired input source, clear **Use default audio device**, and select the desired device in the **Audio device menu** parameter.

When the audio source contains two channels (stereo), the **Stereo** check box should be selected. When the audio source contains a single channel (mono), the **Stereo** check box should be cleared. For stereo input, the block's output is an  $M$ -by-2 matrix containing one frame ( $M$  consecutive samples) of audio data from each of the two channels. For mono input, the block's output is an  $M$ -by-1 matrix containing one frame ( $M$  consecutive samples) of audio data from the mono input. The frame size,  $M$ , is specified by the **Samples per frame** parameter. For  $M=1$ , the output is sample based; otherwise, the output is frame based.

The audio data is processed in uncompressed pulse code modulation (PCM) format, and should typically be sampled at one of the standard Windows audio device rates: 8000, 11025, 22050, or 44100 Hz. You can select one of these rates from the **Sample rate** parameter. To specify a different rate, select the **User-defined** option and enter a value in the **User-defined sample rate** parameter.

The **Sample Width (bits)** parameter specifies the number of bits used to represent the signal samples read by the audio device. The following settings are available:

- 8 — allocates 8 bits to each sample, allowing a resolution of 256 levels
- 16 — allocates 16 bits to each sample, allowing a resolution of 65536 levels
- 24 — allocates 24 bits to each sample, allowing a resolution of 16777216 levels (only for use with 24-bit audio devices)



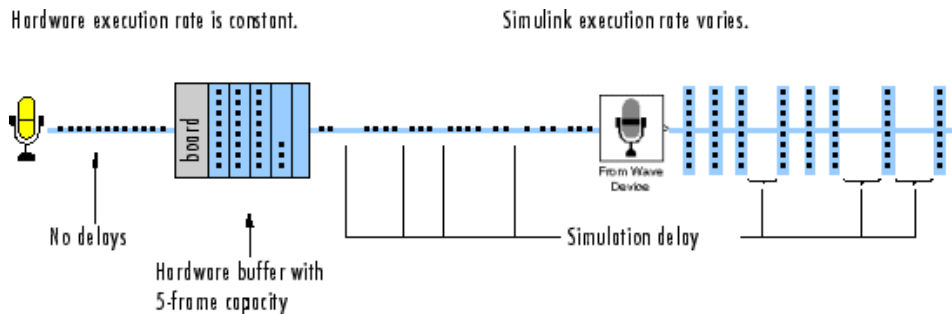
Higher sample width settings require more memory but yield better fidelity. The output from the block is independent of the **Sample width (bits)** setting. The output data type is determined by the **Data type** parameter setting.

## Buffering

Since the audio device accepts real-time audio input, Simulink software must read a continuous stream of data from the device throughout the simulation. Delays in reading data from the audio hardware can result in hardware errors or distortion of the signal. This means that the From Wave Device block must read data from the audio hardware as quickly as the hardware itself acquires the signal. However, the block often *cannot* match the throughput rate of the audio hardware, especially when the simulation is running from within Simulink rather than as generated code. (Simulink operations are generally slower than comparable hardware operations, and execution speed routinely varies during the simulation as the host operating system services other processes.) The block must therefore rely on a buffering strategy to ensure that signal data can be read on schedule without losing samples.

At the start of the simulation, the audio device begins writing the input data to a (hardware) buffer with a capacity of  $T_b$  seconds. The From Wave Device block immediately begins pulling the earliest samples off the buffer (first in, first out) and collecting them in length- $M$  frames for output. As the audio device continues to append inputs to the bottom of the buffer, the From Wave Device block continues to pull inputs off the top of the buffer at the best possible rate.

The following figure shows an audio signal being acquired and output with a frame size of 8 samples. The buffer of the sound board is approaching its five-frame capacity at the instant shown, which means that the hardware is adding samples to the buffer more rapidly than the block is pulling them off. (If the signal sample rate was 8 kHz, this small buffer could hold approximately 0.005 second of data.)



When the simulation throughput rate is higher than the hardware throughput rate, the buffer remains empty throughout the simulation. If necessary, the From Wave Device block simply waits for new samples to become available on the buffer (the block does not interpolate between samples). More typically, the simulation throughput rate is lower than the hardware throughput rate, and the buffer tends to fill over the duration of the simulation.

## Troubleshooting

When the buffer size is too small in relation to the simulation throughput rate, the buffer might fill before the entire length of signal is processed. This usually results in a device error or undesired device output. When this problem occurs, you can choose to either increase the buffer size or the simulation throughput rate:

- *Increase the buffer size*

The **Queue duration** parameter specifies the duration of signal,  $T_b$  (in real-time seconds), that can be buffered in hardware during the simulation. Equivalently, this is the maximum length of time that the block's data acquisition can lag the hardware's data acquisition. The number of frames buffered is approximately

$$\frac{T_b F_s}{M}$$

where  $F_s$  is the sample rate of the signal and  $M$  is the number of samples per frame. The required buffer size for a given signal depends on the signal length, the frame size, and the speed of the simulation. Note that increasing the buffer size might increase model latency.

- *Increase the simulation throughput rate*

Two useful methods for improving simulation throughput rates are increasing the signal frame size and compiling the simulation into native code:

- Increase frame sizes (and convert sample-based signals to frame-based signals) throughout the model to reduce the amount of block-to-block communication overhead. This can drastically increase throughput rates in many cases. However, larger frame sizes generally result in greater model latency due to initial buffering operations.
- Generate executable code with Simulink Coder. Native code runs much faster than Simulink, and should provide rates adequate for real-time audio processing.

More general ways to improve throughput rates include simplifying the model, and running the simulation on a faster PC processor. See “Delay and Latency” and “Optimize Performance” (Simulink) for other ideas on improving simulation performance.

## Parameters

### Sample rate (Hz)

The sample rate of the audio data to be acquired. Select one of the standard Windows rates or the User-defined option.

### User-defined sample rate (Hz)

The (nonstandard) sample rate of the audio data to be acquired.

### Sample width (bits)

The number of bits used to represent each signal sample.

### Stereo

Specifies stereo (two-channel) inputs when selected, mono (one-channel) inputs when cleared. Stereo output is  $M$ -by-2; mono output is  $M$ -by-1.

### Samples per frame

The number of audio samples in each successive output frame,  $M$ . When the value of this parameter is 1, the block outputs a sample-based signal.

### Queue duration (seconds)

The length of signal (in seconds) to buffer to the hardware at the start of the simulation.

**Use default audio device**

Reads audio input from the system's default audio device when selected. Clear to enable the **Audio device ID** parameter and select a device.

**Audio device**

The name of the audio device from which to read the audio output (lists the names of the installed audio device drivers). Select **Use default audio device** when the system has only a single audio card installed.

**Data type**

The data type of the output: double-precision, single-precision, signed 16-bit integer, or unsigned 8-bit integer.

**Supported Data Types**

- Double-precision floating point
- Single-precision floating point
- 16-bit signed integer
- 8-bit unsigned integer

**See Also**

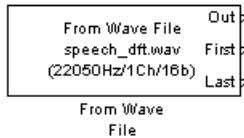
|                           |                    |
|---------------------------|--------------------|
| From Wave File (Obsolete) | DSP System Toolbox |
| To Wave Device (Obsolete) | DSP System Toolbox |
| audiorecorder             | MATLAB             |

**Version History**

**Introduced in R2008b**

## From Wave File (Obsolete)

Read audio data from Microsoft Wave (.wav) file



## Library

dspwin32

## Description

**Note** The From Wave File block is still supported but is likely to be obsoleted in a future release. We strongly recommend replacing this block with the From Multimedia File block.

The From Wave File block streams audio data from a Microsoft® Wave (.wav) file and generates a signal with one of the data types and amplitude ranges in the following table.

**Note** AVI files are the only supported file type for non-Windows platforms.

| Output Data Type | Output Amplitude Range                        |
|------------------|---|
| double           | $\pm 1$                                       |
| single           | $\pm 1$                                       |
| int16            | -32768 to 32767 ( $-2^{15}$ to $2^{15} - 1$ ) |
| uint8            | 0 to 255                                      |

The audio data must be in uncompressed pulse code modulation (PCM) format.

```
y = wavread('filename') % Equivalent MATLAB code
```

The block supports 8-, 16-, 24-, and 32-bit Microsoft Wave (.wav) files.

The **File name** parameter can specify an absolute or relative path to the file. When the file is on the MATLAB path or in the current folder (the folder returned by typing `pwd` at the MATLAB command line), you need only specify the file name. You do not need to specify the .wav extension.

**Note** The From Wave File block does not support .wav file names that contain a + character. To read .wav files that have a + character in the file name, use the From Multimedia File block.

For an audio file containing  $C$  channels, the block's output is an  $M$ -by- $C$  matrix containing one frame ( $M$  consecutive samples) of audio data from each channel. The frame size,  $M$ , is specified by the

**Samples per output frame** parameter. For  $M=1$ , the output is sample based; otherwise, the output is frame based.

The output frame period,  $T_{fo}$ , is

$$T_{fo} = \frac{M}{F_s}$$

where  $F_s$  is the data sample rate in Hz.

To reduce the required number of file accesses, the block acquires  $L$  consecutive samples from the file during each access, where  $L$  is specified by the **Minimum number of samples for each read from file** parameter ( $L \geq M$ ). For  $L < M$ , the block instead acquires  $M$  consecutive samples during each access. Larger values of  $L$  result in fewer file accesses, which reduces run-time overhead.

Use the **Data type** parameter to specify the data type of the block's output. Your choices are `double`, `single`, `uint8`, or `int16`.

Select the **Loop** check box if you want to play the file more than once. Then, enter the number of times to play the file. The number you enter must be a positive integer or `inf`.

Use the **Number of times to play file** parameter to enter the number of times to play the file. The number you enter must be a positive integer or `inf`, to play the file until you stop the simulation.

The **Samples restart** parameter determines whether the samples from the audio file repeat immediately or repeat at the beginning of the next frame output from the output port. When you select `immediately after last sample`, the samples repeat immediately. When you select `at beginning of next frame`, the frame containing the last sample value from the audio file is zero padded until the frame is filled. The block then places the first sample of the audio file in the first position of the next output frame.

Use the **Output start-of-file indicator** parameter to determine when the first audio sample in the file is output from the block. When you select this check box, a Boolean output port labeled SOF appears on the block. The output from the SOF port is 1 when the first audio sample in the file is output from the block. Otherwise, the output from the SOF port is 0.

Use the **Output end-of-file indicator** parameter to determine when the last audio sample in the file is output from the block. When you select this check box, a Boolean output port labeled EOF appears on the block. The output from the EOF port is 1 when the last audio sample in the file is output from the block. Otherwise, the output from the EOF port is 0.

The block icon shows the name, sample rate (in Hz), number of channels (1 or 2), and sample width (in bits) of the data in the specified audio file. All sample rates are supported; the sample width must be either 8, 16, 24, or 32 bits.

## Parameters

### File name

Enter the path and name of the file to read. Paths can be relative or absolute.

---

**Note** The From Wave File block does not support .wav file names that contain a + character. To read .wav files that have a + character in the file name, use the From Multimedia File block.

---

**Samples per output frame**

Enter the number of samples in each output frame,  $M$ . When the value of this parameter is 1, the block outputs a sample-based signal.

**Minimum number of samples for each read from file**

Enter the number of consecutive samples to acquire from the file with each file access,  $L$ .

**Data type**

Select the output data type: double, single, uint8, or int16. The data type setting determines the output's amplitude range.

**Loop**

Select this check box if you want to play the file more than once.

**Number of times to play file**

Enter the number of times you want to play the file.

**Samples restart**

Select immediately after last sample to repeat the audio file immediately. Select at beginning of next frame to place the first sample of the audio file in the first position of the next output frame.

**Output start-of-file indicator**

Use this check box to determine whether the output contains the first audio sample in the file.

**Output end-of-file indicator**

Use this check box to determine whether the output contains the last audio sample in the file.

**Supported Data Types**

- Double-precision floating point
- Single-precision floating point
- 16-bit signed integer
- 8-bit unsigned integer

**See Also**

From Audio Device

Signal From Workspace

To Multimedia File

DSP System Toolbox

DSP System Toolbox

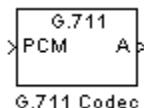
DSP System Toolbox

**Version History**

Introduced in R2010a

## G711 Codec

Quantize narrowband speech input signals



## Library

Quantizers

dspquant2

## Description

The G711 Codec block is a logarithmic scalar quantizer designed for narrowband speech. Narrowband speech is defined as a voice signal with an analog bandwidth of 4 kHz and a Nyquist sampling frequency of 8 kHz. The block quantizes a narrowband speech input signal so that it can be transmitted using only 8-bits. The G711 Codec block has three modes of operation: encoding, decoding, and conversion. You can choose the block's mode of operation by setting the **Mode** parameter.

If, for the **Mode** parameter, you choose **Encode PCM to A-law**, the block assumes that the linear PCM input signal has a dynamic range of 13 bits. Because the block always operates in saturation mode, it assigns any input value above  $2^{12} - 1$  to  $2^{12} - 1$  and any input value below  $-2^{12}$  to  $-2^{12}$ . The block implements an A-law quantizer on the input signal and outputs A-law index values. When you choose **Encode PCM to mu-law**, the block assumes that the linear PCM input signal has a dynamic range of 14 bits. Because the block always operates in saturation mode, it assigns any input value above  $2^{13} - 1$  to  $2^{13} - 1$  and any input value below  $-2^{13}$  to  $-2^{13}$ . The block implements a mu-law quantizer on the input signal and outputs mu-law index values.

If, for the **Mode** parameter, you choose **Decode A-law to PCM**, the block decodes the input A-law index values into quantized output values using an A-law lookup table. When you choose **Decode mu-law to PCM**, the block decodes the input mu-law index values into quantized output values using a mu-law lookup table.

If, for the **Mode** parameter, you choose **Convert A-law to mu-law**, the block converts the input A-law index values to mu-law index values. When you choose **Convert mu-law to A-law**, the block converts the input mu-law index values to A-law index values.

---

**Note** Set the **Mode** parameter to **Convert A-law to mu-law** or **Convert mu-law to A-law** only when the input to the block is A-law or mu-law index values.

---

If, for the **Mode** parameter, you choose **Encode PCM to A-law** or **Encode PCM to mu-law**, the **Overflow diagnostic** parameter appears on the block parameters dialog box. Use this parameter to determine the behavior of the block when overflow occurs. The following options are available:

- **Ignore** — Proceed with the computation and do not issue a warning message.
- **Warning** — Display a warning message in the MATLAB Command Window, and continue the simulation.
- **Error** — Display an error dialog box and terminate the simulation.

---

**Note** Like all diagnostic parameters on the Configuration Parameters dialog box, **Overflow diagnostic** parameter is set to **Ignore** in the code generated for this block by Simulink Coder code generation software.

---

## Parameters

### Mode

- When you choose **Encode PCM to A-law**, the block implements an A-law encoder.
- When you choose **Encode PCM to mu-law**, the block implements a mu-law encoder.
- When you choose **Decode A-law to PCM**, the block decodes the input index values into quantized output values using an A-law lookup table.
- When you choose **Decode mu-law to PCM**, the block decodes the input index values into quantized output values using a mu-law lookup table.
- When you choose **Convert A-law to mu-law**, the block converts the input A-law index values to mu-law index values.
- When you choose **Convert mu-law to A-law**, the block converts the input mu-law index values to A-law index values.

### Overflow diagnostic

Use this parameter to determine the behavior of the block when overflow occurs.

- Select **Ignore** to proceed with the computation without a warning message.
- Select **Warning** to display a warning message in the MATLAB Command Window and continue the simulation.
- Select **Error** to display an error dialog box and terminate the simulation.

This parameter is only visible if, for the **Mode** parameter, you select **Encode PCM to A-law** or **Encode PCM to mu-law**.

## References

ITU-T Recommendation G.711, "Pulse Code Modulation (PCM) of Voice Frequencies," *General Aspects of Digital Transmission Systems; Terminal Equipments*, International Telecommunication Union (ITU), 1993.

## Supported Data Types

| Port | Supported Data Types  |
|------|---|
| PCM  | <ul style="list-style-type: none"> <li>• 16-bit signed integers</li> </ul>  |
| A    | <ul style="list-style-type: none"> <li>• 8-bit unsigned integers</li> </ul> |



| Port | Supported Data Types  |
|------|---|
| mu   | <ul style="list-style-type: none"><li>8-bit unsigned integers</li></ul> |

## See Also

Quantizer

Scalar Quantizer Decoder

Uniform Decoder

Uniform Encoder

Vector Quantizer Decoder

Vector Quantizer Encoder

Simulink

DSP System Toolbox

DSP System Toolbox

DSP System Toolbox

DSP System Toolbox

DSP System Toolbox

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

## Halfband Filter (Obsolete)

(Removed) Design halfband filter



## Compatibility

---

**Note** The Halfband Filter block has been removed from the DSP System Toolbox block library. Existing instances of the Halfband Filter block will continue to operate. To model FIR halfband decimators and interpolators, use the FIR Halfband Decimator and FIR Halfband Interpolator blocks. These blocks replace the functionality of the Halfband Filter block, when **Impulse response** is set to FIR, and **FilterType** is set to Decimator and Interpolator. To model IIR halfband decimators and interpolators, use the IIR Halfband Decimator and IIR Halfband Interpolator blocks. These blocks replace the functionality of the Halfband Filter block, when **Impulse response** is set to IIR, and **FilterType** is set to Decimator and Interpolator.

---

## Library

Filtering / Filter Designs

dspfdesign

## Description

This block brings the filter design capabilities of the `filterBuilder` function to the Simulink environment.

## Dialog Box

See “Halfband Filter Design — Main Pane” on page 5-616 for more information about the parameters of this block. The **Data Types** and **Code Generation** panes are not available for blocks in the DSP System Toolbox Filter Designs library.

Function Block Parameters: Halfband Filter

Halfband Filter

Design a Halfband filter.

View Filter Response

Frequency specifications

Impulse response: FIR

Order mode: Minimum Order:

Response type: Lowpass

Filter Type: Single-rate

Frequency specifications

Frequency units: Normalized (0 to 1) Input Fs: 2

Transition width: .1

Magnitude specifications

Magnitude units: dB

Astop: 80

Algorithm

Design method: Equiripple

► Design options

Filter Implementation

Structure: Direct-form FIR

Use basic elements to enable filter customization

Input processing: Columns as channels (frame based)

Use symbolic names for coefficients

OK Cancel Help Apply

### View filter response

This button opens the Filter Visualization Tool (**FVTool**) from the Signal Processing Toolbox product. You can use the tool to display:

- Magnitude response, phase response, and group delay in the frequency domain.
- Impulse response and step response in the time domain.
- Pole-zero information.

The tool also helps you evaluate filter performance by providing information about filter order, stability, and phase linearity. For more information on FVTool, see the Signal Processing Toolbox documentation.

### Filter Specifications

In this group, you specify your filter type and order.

#### Impulse response

Select either FIR or IIR from the drop-down list. FIR is the default. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly.

---

**Note** The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

---

#### Order mode

Select either Minimum (the default) or Specify from the drop-down list. Selecting Specify enables the **Order** option (see the following sections) so you can enter the filter order.

#### Response type

Specify the filter response as Lowpass (the default) or Highpass.

#### Filter type

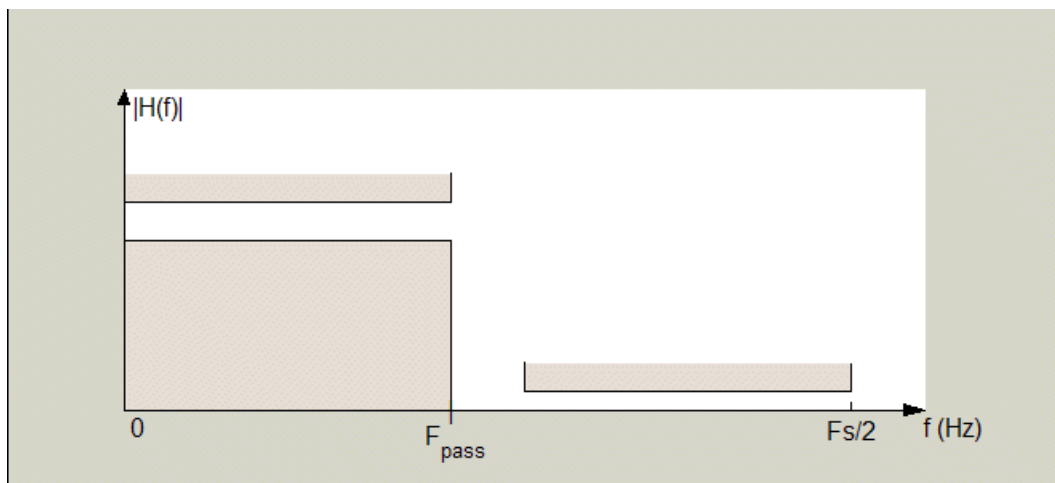
Select Single-rate, Decimator, or Interpolator. By default, the block specifies a single-rate filter.

#### Order

Enter the filter order. This option is enabled only when the **Filter order mode** is set to Specify.

### Frequency Specifications

The parameters in this group allow you to specify your filter response curve. Graphically, the filter specifications for a halfband lowpass filter look similar to those shown in the following figure.



In the figure, the transition region lies between the end of the passband and the start of the stopband. The width is defined explicitly by the value of **Transition width**.

**Frequency constraints**

When **Order mode** is **Specify**, set this parameter to **Unconstrained** or **Transition width**.

**Frequency units**

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select **Normalized (0–1)** to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—Hz, kHz, MHz, or GHz. Selecting one of the unit options enables the **Input Fs** parameter.

**Input Fs**

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

**Transition width**

Specify the width of the transition between the end of the passband and the edge of the stopband. Specify the value in normalized frequency units or the absolute units you select in **Frequency units**.

**Magnitude Specifications**

Parameters in this group specify the filter response in the passbands and stopbands.

**Magnitude constraints**

Specify **Unconstrained** (the default), or select **Stopband attenuation** to constrain the response in the stopband explicitly.

**Magnitude units**

Specify the units for any parameter you provide in magnitude specifications. From the drop-down list, select one of the following options:

- **Linear** — Specify the magnitude in linear units.
- **dB** — Specify the magnitude in decibels (default).

**Astop**

When **Magnitude units** is **Stopband attenuation**, enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels.

**Algorithm**

The parameters in this group allow you to specify the design method and structure of your filter.

**Design Method**

Lists the design methods available for the frequency and magnitude specifications you entered. For FIR halfband filters, the available design options are **equiripple**, and **Kaiser window**. For IIR halfband filters, the available design options are **Butterworth**, **elliptic**, and **IIR quasi-linear phase**.

**Design Options**

The following design options are available for FIR halfband filters when the user specifies an equiripple design:

**Minimum phase**

Select the check box to specify a minimum-phase design.

**Stopband Shape**

Stopband shape lets you specify how the stopband changes with increasing frequency. Choose one of the following options:

- **Flat** — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.
- **Linear** — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.
- **1/f** — Specifies that the stopband attenuation changes exponentially as the frequency increases, where  $f$  is the frequency. Set the power (exponent) for the decay in **Stopband decay**.

**Stopband Decay**

When you set Stopband shape, Stopband decay specifies the amount of decay applied to the stopband. The following conditions apply to Stopband decay based on the value of Stopband Shape:

- When you set **Stopband shape** to **Flat**, **Stopband decay** has no effect on the stopband.
- When you set **Stopband shape** to **Linear**, enter the slope of the stopband in units of dB/rad/s. The block applies that slope to the stopband.
- When you set **Stopband shape** to **1/f**, enter a value for the exponent  $n$  in the relation  $(1/f)^n$  to define the stopband decay. The block applies the  $(1/f)^n$  relation to the stopband to result in an exponentially decreasing stopband attenuation.

**Filter Implementation****Structure**

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter.

**Use basic elements to enable filter customization**

Select this check box to implement the filter as a subsystem of basic Simulink blocks. Clear the check box to implement the filter as a high-level subsystem. By default, this check box is cleared.

The high-level implementation provides better compatibility across various filter structures, especially filters that would contain algebraic loops when constructed using basic elements. On the other hand, using basic elements enables the following optimization parameters:

- **Optimize for zero gains** — Terminate chains that contain Gain blocks with a gain of zero.
- **Optimize for unit gains** — Remove Gain blocks that scale by a factor of one.
- **Optimize for delay chains** — Substitute delay chains made up of  $n$  unit delays with a single delay by  $n$ .
- **Optimize for negative gains** — Use subtraction in Sum blocks instead of negative gains in Gain blocks.

**Optimize for unit-scale values**

Select this check box to scale unit gains between sections in SOS filters. This parameter is available only for SOS filters.

### Input processing

Specify how the block should process the input. The available options may vary depending on the settings of the **Filter Structure** and **Use basic elements for filter customization** parameters. You can set this parameter to one of the following options:

- **Columns as channels (frame based)** — When you select this option, the block treats each column of the input as a separate channel.
- **Elements as channels (sample based)** — When you select this option, the block treats each element of the input as a separate channel.

---

**Note** The **Inherited** (this choice will be removed – see release notes) option will be removed in a future release. See “Frame-Based Processing” in the *DSP System Toolbox Release Notes* for more information.

---

### Rate options

When the **Filter type** parameter specifies a multirate filter, select the rate processing rule for the block from following options:

- **Enforce single-rate processing** — When you select this option, the block maintains the sample rate of the input.
- **Allow multirate processing** — When you select this option, the block adjusts the rate at the output to accommodate an increased or reduced number of samples. To select this option, you must set the **Input processing** parameter to **Elements as channels (sample based)**.

### Use symbolic names for coefficients

Select this check box to enable the specification of coefficients using MATLAB variables. The available coefficient names differ depending on the filter structure. Using symbolic names allows tuning of filter coefficients in generated code. By default, this check box is cleared.

### Supported Data Types

| Port   | Supported Data Types   |
|--------|--|
| Input  | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> </ul> |
| Output | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> </ul> |

### Version History

Introduced in R2006b

## **See Also**

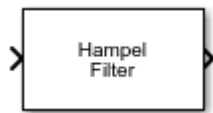
### **Blocks**

FIR Halfband Decimator | FIR Halfband Interpolator | IIR Halfband Decimator | IIR Halfband Interpolator



# Hampel Filter

Filter outliers using Hampel identifier



## Libraries:

DSP System Toolbox / Filtering / Filter Designs

## Description

The Hampel Filter block detects and removes the outliers of the input signal by using the Hampel identifier. The Hampel identifier is a variation of the three-sigma rule of statistics, which is robust against outliers. For each sample of the input signal, the block computes the median of a window composed of the current sample and  $\frac{Len - 1}{2}$  adjacent samples on each side of the current sample.

$Len$  is the window length you specify through the **Window length** parameter. The block also estimates the standard deviation of each sample about its window median by using the median absolute deviation. If a sample differs from the median by more than the threshold multiplied by the standard deviation, the filter replaces the sample with the median. For more information, see “Algorithms” on page 2-748.

## Ports

### Input

**x** — Data input  
vector | matrix

The block accepts multichannel inputs, that is,  $m$ -by- $n$  size inputs, where  $m \geq 1$ , and  $n \geq 1$ .  $m$  is the number of samples in each frame (channel), and  $n$  is the number of channels. The block also accepts variable-size inputs. That is, you can change the size of each input channel during simulation. However, the number of channels cannot change.

This port is unnamed until you select the **Specify threshold from input port** parameter.

Data Types: single | double

**T** — Threshold  
real scalar greater than or equal to 0

Threshold for outlier detection, specified as a real scalar greater than or equal to 0. For information on how this parameter is used to detect the outlier, see “Algorithms” on page 2-748.

### Dependencies

This port appears when you select the **Specify threshold from input port** parameter.

Data Types: single | double

## Output

**y** — Filtered output  
vector | matrix

The size and data type of this output matches the size and data type of the input.

This port is unnamed until you select the **Output outlier status** check box.

Data Types: single | double

**outlier** — Output the outlier status  
vector | matrix

A value of 1 in this output indicates that the corresponding element in the input is an outlier. This output has the same size as the input.

## Dependencies

To enable this port, select the **Output outlier status** check box.

Data Types: Boolean

## Parameters

If a parameter is listed as tunable, then you can change its value during simulation.

**Window length** — Length of sliding window  
11 (default) | positive odd scalar integer

Length of the sliding window, specified as a positive odd scalar integer. The window of finite length slides over the data, and the block computes the median and median absolute deviation of the data in the window.

**Specify threshold from input port** — Flag to specify threshold  
off (default) | on

When you select this check box, the threshold is input through the **T** port. When you clear this check box, the threshold is specified on the block dialog through the **Threshold for outlier detection (standard deviations)** parameter.

**Threshold for outlier detection (standard deviations)** — Threshold  
3 (default) | real scalar greater than or equal to 0

Threshold for outlier detection, specified as a real scalar greater than or equal to 0. For information on how this parameter is used to detect the outlier, see “Algorithms” on page 2-748.

**Tunable:** Yes

## Dependencies

This parameter appears when you clear the **Specify threshold from input port** check box.

**Output outlier status** — Flag to output the outlier status  
off (default) | on

Select this parameter to output a matrix of **boolean** values that has the same size as the input. Each element in this matrix indicates whether the corresponding element in the input is an outlier. A value of 1 indicates an outlier.

**Simulate using** — Type of simulation to run

Interpreted execution (default) | Code generation

Type of simulation to run. You can set this parameter to:

- **Interpreted execution**: Simulate model using the MATLAB interpreter. This option shortens startup time.
- **Code generation**: Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations as long as the model does not change. This option requires additional startup time but provides faster subsequent simulations.

## Block Characteristics

|                                 |                 |
|---------------------------------|-----------------|
| <b>Data Types</b>               | double   single |
| <b>Direct Feedthrough</b>       | no              |
| <b>Multidimensional Signals</b> | no              |
| <b>Variable-Size Signals</b>    | yes             |
| <b>Zero-Crossing Detection</b>  | no              |

## More About

### Hampel Identifier

The Hampel identifier is a variation of the three-sigma rule of statistics that is robust against outliers.

Given a sequence  $x_1, x_2, x_3, \dots, x_n$  and a sliding window of length  $k$ , define point-to-point median and standard-deviation estimates using:

- Local median —  $m_i = \text{median}(x_{i-k}, x_{i-k+1}, x_{i-k+2}, \dots, x_i, \dots, x_{i+k-2}, x_{i+k-1}, x_{i+k})$
- Standard deviation —  $\sigma_i = \kappa \text{median}(|x_{i-k} - m_i|, \dots, |x_{i+k} - m_i|)$ , where  $\kappa = \frac{1}{\sqrt{2} \text{erfc}^{-1}(1/2)} \approx 1.4826$

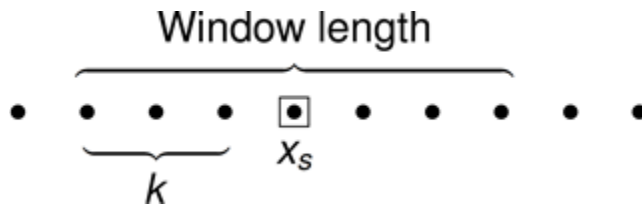
The quantity  $\sigma_i / \kappa$  is known as the median absolute deviation (MAD).

If a sample  $x_i$  is such that

$$|x_i - m_i| > n_\sigma \sigma_i$$

for a given threshold  $n_\sigma$ , then the Hampel identifier declares  $x_i$  an outlier and replaces it with  $m_i$ . If  $n_\sigma$  is 0, then the Hampel filter behaves as a regular median filter.

## Algorithms



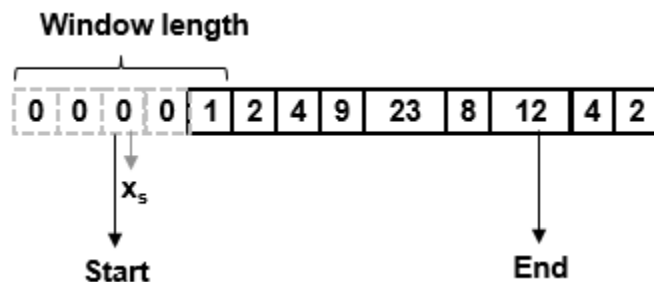
For a given sample of data,  $x_s$ , the algorithm:

- Centers the window of odd length at the current sample.
- Computes the local median,  $m_i$ , and standard deviation,  $\sigma_i$ , over the current window of data.
- Compares the current sample with  $n_\sigma \times \sigma_i$ , where  $n_\sigma$  is the threshold value. If  $|x_s - m_i| > n_\sigma \times \sigma_i$ , the filter identifies the current sample,  $x_s$ , as an outlier and replaces it with the median value,  $m_i$ .

Consider a frame of data that is passed into the Hampel filter.

|   |   |   |   |    |   |    |   |   |
|---|---|---|---|----|---|----|---|---|
| 1 | 2 | 4 | 9 | 23 | 8 | 12 | 4 | 2 |
|---|---|---|---|----|---|----|---|---|

In this example, the Hampel filter slides a window of length 5 ( $Len$ ) over the data. The filter has a threshold value of 2 ( $n_\sigma$ ). To have a complete window at the beginning of the frame, the filter algorithm prepends the frame with  $Len - 1$  zeros. To compute the first sample of the output, the window centers on the  $\left[\frac{Len - 1}{2} + 1\right]^{\text{th}}$  sample in the appended frame, the third zero in this case. The filter computes the median, median absolute deviation, and the standard deviation over the data in the local window.



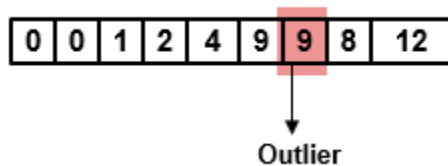
- Current sample:  $x_s = 0$ .
- Window of data:  $win = [0 \ 0 \ 0 \ 0 \ 1]$ .
- Local median:  $m_i = \text{median}([0 \ 0 \ 0 \ 0 \ 1]) = 0$ .
- Median absolute deviation:  $mad_i = \text{median}(|x_{i-k} - m_i|, \dots, |x_{i+k} - m_i|)$ . For this window of data,  $mad = \text{median}(|0 - 0|, \dots, |1 - 0|) = 0$ .

- Standard deviation:  $\sigma_i = \kappa \times mad_i = 0$ , where  $\kappa = \frac{1}{\sqrt{2}\text{erfc}^{-1}(1/2)} \approx 1.4826$ .
- The current sample,  $x_s = 0$ , does not obey the relation for outlier detection.  
 $[|x_s - m_i| = 0] > [(n_\sigma \times \sigma_i) = 0]$

Therefore, the Hampel filter outputs the current input sample,  $x_s = 0$ .

Repeat this procedure for every succeeding sample until the algorithm centers the window on the  $\left[End - \frac{Len - 1}{2}\right]^{\text{th}}$  sample, marked as End. Because the window centered on the last  $\frac{Len - 1}{2}$  samples cannot be full, these samples are processed with the next frame of input data.

Here is the first output frame the Hampel filter generates:



The seventh sample of the appended input frame, 23, is an outlier. The Hampel filter replaces this sample with the median over the local window [4 9 23 8 12].

## Version History

Introduced in R2017a

## References

- [1] Bodenham, Dean. "Adaptive Filtering and Change Detection for Streaming Data." Ph.D. Thesis. Imperial College, London, 2012.
- [2] Liu, Hancong, Sirish Shah, and Wei Jiang. "On-line outlier detection and data cleaning." *Computers and Chemical Engineering*. Vol. 28, March 2004, pp. 1635-1647.
- [3] Suomela, Jukka. Median Filtering Is Equivalent to Sorting, 2014.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

## See Also

### Functions

hampel

### Objects

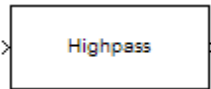
dsp.HampelFilter | dsp.MedianFilter | dsp.MovingAverage

**Blocks**

Median Filter | Moving Average

# Highpass Filter

Design FIR or IIR highpass filter



## Libraries:

DSP System Toolbox / Filtering / Filter Designs

## Description

The Highpass Filter block independently filters each channel of the input signal over time using the given design specifications. You can control whether the block implements an IIR or FIR highpass filter using the **Filter type** parameter. You can specify the passband and stopband edge frequencies in Hz or in normalized frequency units.

This block supports ARM Cortex code generation. Under certain conditions, this block also supports SIMD code generation. For details, see “Code Generation” on page 2-755.

## Ports

### Input

**Port\_1** — Input signal to filter  
column vector | matrix

Input signal, specified as a real- or complex-valued column vector or matrix. If the input signal is a matrix, each column of the matrix is treated as an independent channel. The number of rows in the input signal denotes the channel length.

Data Types: `single` | `double` | `fixed point`

### Output

**Port\_1** — Filtered signal  
vector | matrix

Filtered signal, returned as a vector or matrix. The output has the same size and complexity characteristics as the input. If the output has a fixed-point data type, it is always signed.

Data Types: `single` | `double` | `fixed point`

## Parameters

### Main

**Filter type** — FIR or IIR filter

FIR (default) | IIR

Specify whether the block implements an FIR highpass filter or an IIR highpass filter.

**Design minimum order filter** — Design filter with minimum order

on (default) | off

When you select this check box, the block designs a filter with minimum order. When you clear this check box, you can specify the **Filter order** as a positive integer.

**Filter order** — Order of highpass filter

50 (default) | positive integer

Specify the order of the highpass filter as a positive integer.

#### **Dependencies**

To enable this parameter, clear the **Design minimum order filter** check box.

**Stopband edge frequency (Hz)** — Stopband edge frequency

8e3 (default) | real positive scalar

Specify the stopband edge frequency of the highpass filter as a real positive scalar in Hz or in normalized frequency units.

If you set the **Sample rate mode** parameter to:

- **Specify on dialog or Inherit from input port** -- The value of the stopband edge frequency is in Hz and must be less than the passband edge frequency.
- **Use normalized frequency (0 to 1)** -- The value of the stopband edge frequency is in normalized frequency units. The value must be a positive scalar less than the passband edge frequency.

#### **Dependencies**

To enable this parameter, select the **Design minimum order filter** check box.

**Passband edge frequency (Hz)** — Passband edge frequency

12e3 (default) | real positive scalar

Specify the passband edge frequency of the highpass filter as a real positive scalar in Hz or in normalized frequency units.

If you set the **Sample rate mode** parameter to:

- **Specify on dialog or Inherit from input port** -- The value of the passband edge frequency is in Hz and must be less than half the value of the input sample rate.
- **Use normalized frequency (0 to 1)** -- The value of the passband edge frequency is in normalized frequency units. The value must be a positive scalar less than 1.0.

**Minimum stopband attenuation (dB)** — Minimum stopband attenuation

80 (default) | real positive scalar

Specify the minimum attenuation in the stopband as a real positive scalar in dB.



**Maximum passband ripple (dB)** — Maximum passband ripple

0.1 (default) | real positive scalar

Specify the maximum ripple of the filter response in the passband as a real positive scalar in dB.

**Sample rate mode** — Mode to specify the input sample rate

Specify on dialog (default) | Inherit from input port | Use normalized frequency (0 to 1)

Specify the input sample rate using one of these options:

- **Specify on dialog** -- Specify the input sample rate in the block dialog box using the **Input sample rate (Hz)** parameter.
- **Inherit from input port** -- The block inherits the sample rate from the input signal.
- **Use normalized frequency (0 to 1)** -- Specify the passband edge and the stopband edge frequencies in normalized frequency units (0 to 1).

**Input sample rate (Hz)** — Input sample rate

44100 (default) | scalar

Specify the input sample rate as a scalar in Hz.

**Dependencies**

To enable this parameter, set the **Sample rate mode** parameter to **Specify on dialog**.

**Simulate using** — Type of simulation to run

Interpreted execution (default) | Code generation

Type of simulation to run:

- **Interpreted execution (default)**

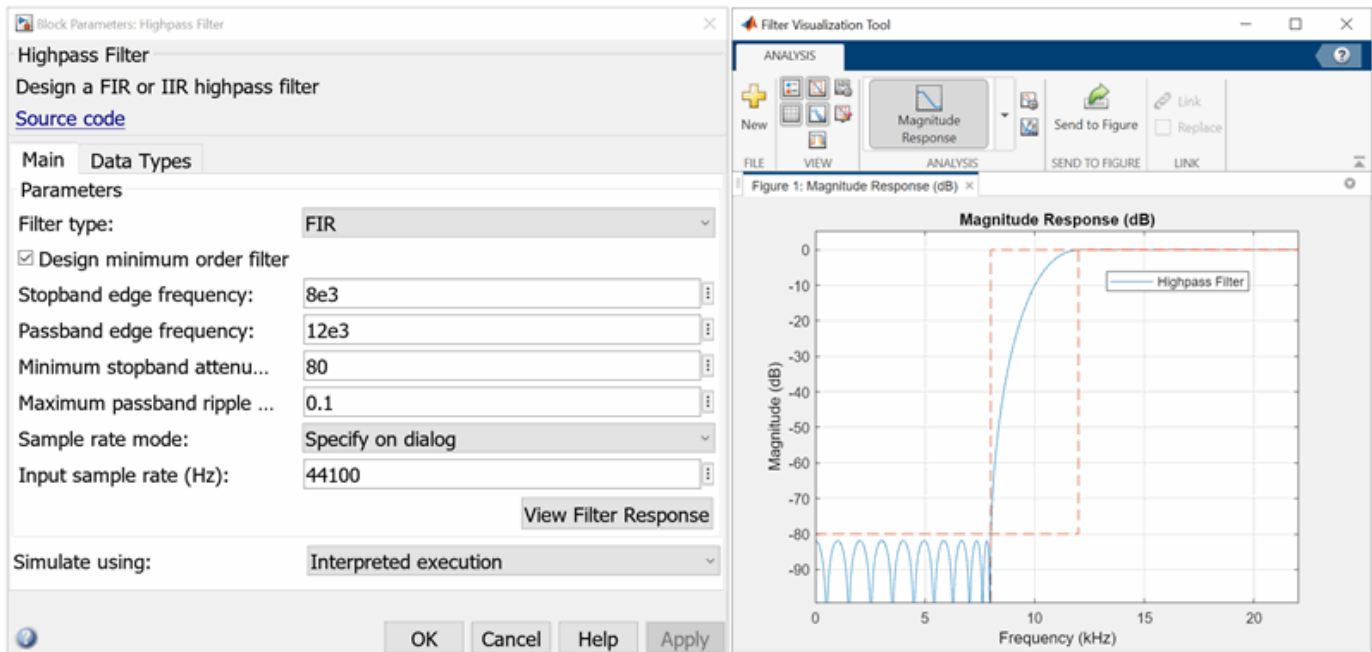
Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than **Code generation**.

- **Code generation**

Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations as long as the model does not change. This option requires additional startup time but provides faster simulation speed than **Interpreted execution**.

**View Filter Response** — Open Filter Visualization Tool button

Opens the Filter Visualization Tool (`fvtool`) and displays the magnitude/phase response of the highpass filter. The response is based on the block dialog box parameters. Changes made to these parameters update FVTool.



To update the magnitude response while FVTool is running, modify the dialog box parameters and click **Apply**.

## Data Types

### Rounding mode — Rounding method

Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero


Rounding method for the output fixed-point operations.

### Coefficients — Coefficient data type

fixdt(1,16) (default) | fixdt(1,16,0) | <data type expression>

Fixed-point data type of the coefficients, specified as one of the following:

- `fixdt(1,16)` — Signed fixed-point data type of word length 16, with binary point scaling. The block determines the fraction length automatically from the coefficient values in such a way that the coefficients occupy maximum representable range without overflowing.
- `fixdt(1,16,0)` — Signed fixed-point data type of word length 16 and fraction length 0. You can change the fraction length to any other integer value.
- `<data type expression>` — Specify the data type using an expression that evaluates to a data type object, for example, numeric type (`fixdt([ ],16,15)`). Specify the sign mode of this data type as `[ ]` or `true`.
- Refresh Data Type — Refresh to the default data type.

Click the **Show data type assistant** button  to display the data type assistant, which helps you set the data type. For more information, see “Specify Data Types Using Data Type Assistant” (Simulink).

## Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | double   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no                                      |
| <b>Multidimensional Signals</b> | no                                      |
| <b>Variable-Size Signals</b>    | yes                                     |
| <b>Zero-Crossing Detection</b>  | no                                      |

## Algorithms

### FIR Highpass Filter

For the minimum order design, the algorithm uses generalized Remez FIR filter design algorithm. For the specified order design, the algorithm uses the constrained equiripple FIR filter design algorithm. The designed filter is then implemented as a linear phase Type-1 filter with a `Direct` form structure.

### IIR Highpass Filter

In the IIR configuration, the algorithm uses the elliptic design method to compute the SOS and scale values required to meet the filter design specifications. The algorithm uses the SOS and scale values to setup a `Direct form I` biquadratic IIR filter, which forms the basis of the IIR version of the highpass filter.

## Version History

### Introduced in R2015b

#### R2023a: Support for normalized frequencies

When you set the **Sample rate mode** parameter to `Use normalized frequency (0 to 1)`, you can specify the passband and stopband edge frequencies in normalized frequency units (0 to 1).

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

The Highpass Filter block supports ARM Cortex code generation. To learn more about ARM Cortex code generation, see “Code Generation for ARM Cortex-M and ARM Cortex-A Processors”.

The Highpass Filter block also supports SIMD code generation using Intel AVX2 technology under these conditions:

- **Filter type** is set to FIR.
- Input signal has a data type of `single` or `double`.

The SIMD technology significantly improves the performance of the generated code.

**Fixed-Point Conversion**

Design and simulate fixed-point systems using Fixed-Point Designer™.

**See Also**

**Blocks**

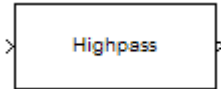
Lowpass Filter

**Objects**

`dsp.HighpassFilter` | `dsp.LowpassFilter`

# Highpass Filter (Obsolete)

Design highpass filter



## Compatibility

---

**Note** The Highpass Filter (Obsolete) block has been replaced by the Highpass Filter block. Existing instances of the Highpass Filter (Obsolete) block will continue to operate. For new models, use the Highpass Filter block.

---

## Library

Filtering / Filter Designs

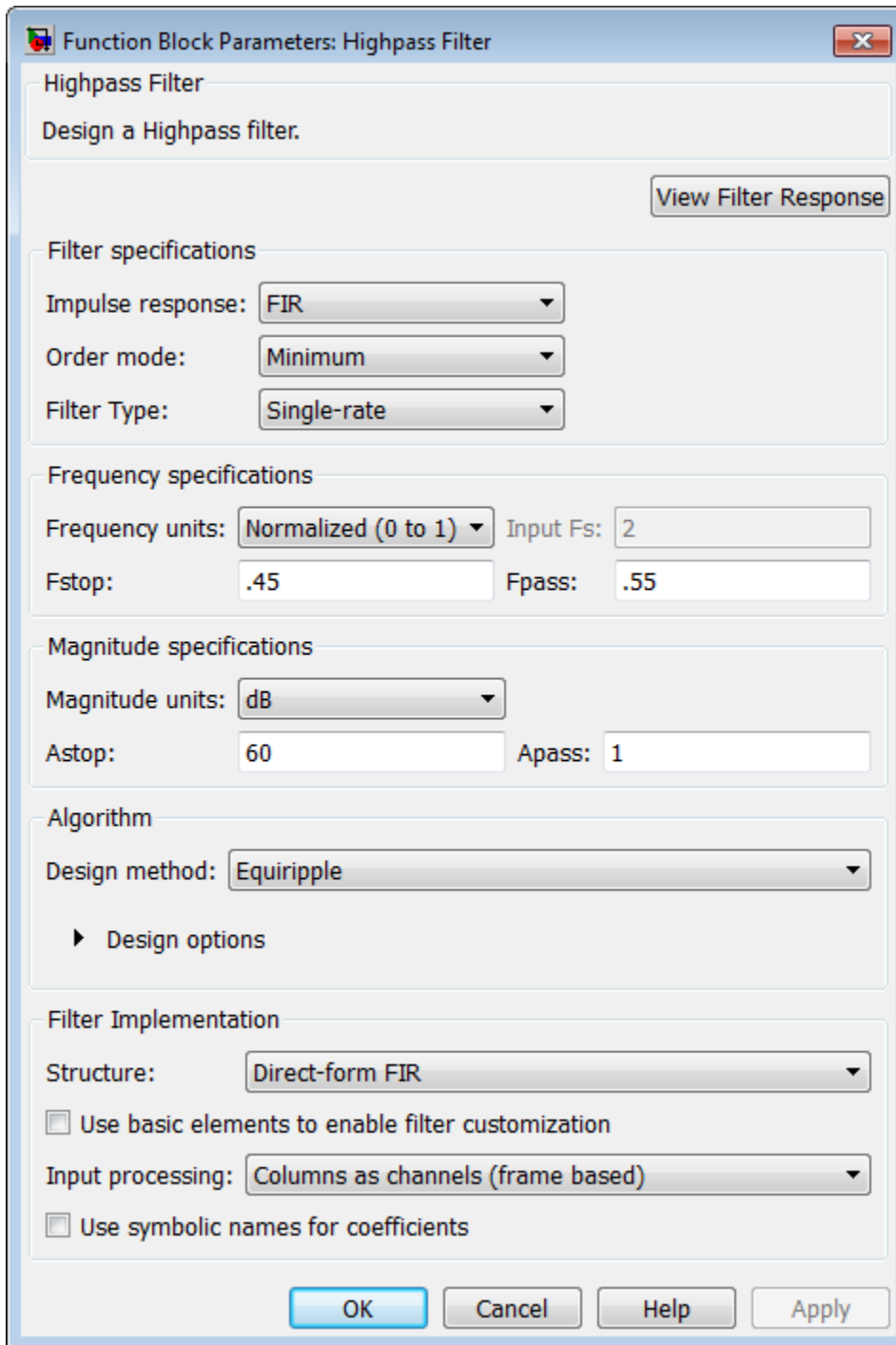
dspfdesign

## Description

This block brings the filter design capabilities of the `filterBuilder` function to the Simulink environment.

## Dialog Box

See “Highpass Filter Design — Main Pane” on page 5-619 for more information about the parameters of this block. The **Data Types** and **Code Generation** panes are not available for blocks in the DSP System Toolbox Filter Designs library.



### View filter response

This button opens the Filter Visualization Tool (**FVTool**) from the Signal Processing Toolbox product. You can use the tool to display:

- Magnitude response, phase response, and group delay in the frequency domain.
- Impulse response and step response in the time domain.

- Pole-zero information.

The tool also helps you evaluate filter performance by providing information about filter order, stability, and phase linearity. For more information on FVTool, see the Signal Processing Toolbox documentation.

### Filter Specifications

In this group, you specify your filter format, such as the impulse response and the filter order.

#### Impulse response

Select either FIR or IIR from the drop-down list. FIR is the default. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly.

---

**Note** The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

---

#### Order mode

Select **Minimum** (the default) or **Specify**. Selecting **Specify** enables the **Order** option so you can enter the filter order. When you set the **Impulse response** to IIR, you can specify different numerator and denominator orders. To specify a different denominator order, you must select the **Denominator order** check box.

#### Order

Enter the filter order. This option is enabled only if you set the **Order mode** to **Specify**.

#### Denominator order

Select this check box to specify a different denominator order. This option is enabled only if you set the **Impulse response** to IIR and the **Order mode** to **Specify**.

#### Filter type

Select **Single-rate**, **Decimator**, **Interpolator**, or **Sample-rate converter**. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, the block specifies a single-rate filter.

- Selecting **Decimator** or **Interpolator** activates the **Decimation Factor** or the **Interpolation Factor** options respectively.
- Selecting **Sample-rate converter** activates both factors.

#### Decimation Factor

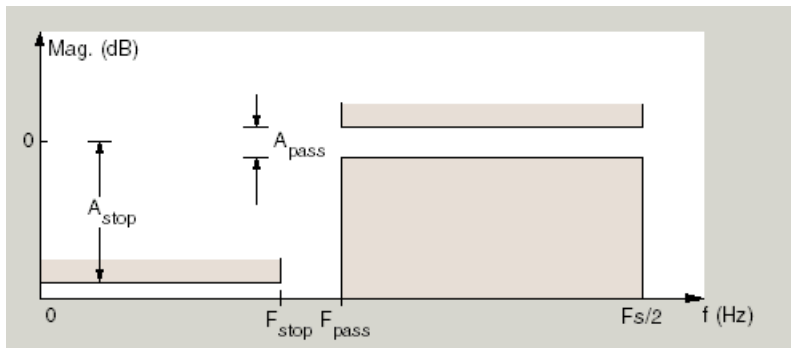
Enter the decimation factor. This option is enabled only if the **Filter type** is set to **Decimator** or **Sample-rate converter**. The default value is 2.

#### Interpolation Factor

Enter the interpolation factor. This option is enabled only if the **Filter type** is set to **Interpolator** or **Sample-rate converter**. The default value is 2.

### Frequency Specifications

The parameters in this group allow you to specify your filter response curve. Graphically, the filter specifications look similar to those shown in the following figure.



In the figure, the region between specification values  $F_{stop}$  and  $F_{pass}$  represents the transition region where the filter response is not constrained.

### Frequency constraints

When **Order mode** is **Specify**, select the filter features that the block uses to define the frequency response characteristics. The list contains the following options, when available for the filter specifications.

- **Stopband edge and passband edge** — Define the filter by specifying the frequencies for the edges for the stop- and passbands.
- **Passband edge** — Define the filter by specifying the edge of the passband.
- **Stopband edge** — Define the filter by specifying the edge of the stopband.
- **Stopband edge and 3 dB point** — For IIR filters, define the filter by specifying the frequency of the 3 dB point in the filter response and the edge of the stopband.
- **3 dB point** — Define the filter response by specifying the location of the 3 dB point. The 3 dB point is the frequency for the point three decibels below the passband value.
- **3 dB point and passband edge** — For IIR filters, define the filter by specifying the frequency of the 3 dB point in the filter response and the edge of the passband.
- **6 dB point** — For FIR filters, define the filter response by specifying the location of the 6 dB point. The 6 dB point is the frequency for the point six decibels below the passband value.

### Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select **Normalized (0–1)** to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—Hz, kHz, MHz, or GHz. Selecting one of the unit options enables the **Input Fs** parameter.

### Input Fs

$F_s$ , specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

### Fstop

Enter the frequency at the edge of the end of the stopband. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.



**Fpass**

Enter the frequency at the edge of the start of the passband. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

**F3dB**

When **Frequency constraints** is 3 dB point, Stopband edge and 3 dB point, or 3 dB point and passband edge, specify the frequency of the 3 dB point. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

**F6dB**

When **Frequency constraints** is 6 dB point, specify the frequency of the 6 dB point. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

**Magnitude Specifications**

Parameters in this group specify the filter response in the passbands and stopbands.

**Magnitude constraints**

This option is only available when you specify the order of your filter design. Depending on the setting of the **Frequency constraints** parameter, some combination of the following options will be available for the **Magnitude constraints** parameter: Unconstrained, Passband ripple, Passband ripple and stopband attenuation or Stopband attenuation.

**Magnitude units**

Specify the units for any parameter you provide in magnitude specifications. From the drop-down list, select one of the following options:

- Linear — Specify the magnitude in linear units.
- dB — Specify the magnitude in decibels (default).
- Squared — Specify the magnitude in squared units.

**Astop**

Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels.

**Apass**

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels.

**Algorithm**

The parameters in this group allow you to specify the design method and structure of your filter.

**Design Method**

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is usually Butterworth, and the default FIR method is Equi ripple.

**Scale SOS filter coefficients to reduce chance of overflow**

Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

## Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

### Density factor

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 20 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

### Phase constraint

Specify the phase constraint of the filter as **Linear**, **Maximum**, or **Minimum**.

### Minimum order

When you select this parameter, the design method determines and design the minimum order filter to meet your specifications. Some filters do not provide this parameter. Select **Any**, **Even**, or **Odd** from the drop-down list to direct the design to be any minimum order, or minimum even order, or minimum odd order.

### Match Exactly

Specifies that the resulting filter design matches either the passband or stopband or both bands when you select **passband** or **stopband** or **both** from the drop-down list.

### Stopband Shape

Stopband shape lets you specify how the stopband changes with increasing frequency. Choose one of the following options:

- **Flat** — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.
- **Linear** — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.
- **1/f** — Specifies that the stopband attenuation changes exponentially as the frequency increases, where  $f$  is the frequency. Set the power (exponent) for the decay in **Stopband decay**.

### Stopband Decay

When you set Stopband shape, Stopband decay specifies the amount of decay applied to the stopband. the following conditions apply to Stopband decay based on the value of Stopband Shape:

- When you set **Stopband shape** to **Flat**, **Stopband decay** has no affect on the stopband.
- When you set **Stopband shape** to **Linear**, enter the slope of the stopband in units of dB/rad/s. `filterbuilder` applies that slope to the stopband.
- When you set **Stopband shape** to **1/f**, enter a value for the exponent  $n$  in the relation  $(1/f)^n$  to define the stopband decay. `filterbuilder` applies the  $(1/f)^n$  relation to the stopband to result in an exponentially decreasing stopband attenuation.

## Filter Implementation

### Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure, and IIR filters use direct-form II filters with SOS.

### Use basic elements to enable filter customization

Select this check box to implement the filter as a subsystem of basic Simulink blocks. Clear the check box to implement the filter as a high-level subsystem. By default, this check box is cleared.

The high-level implementation provides better compatibility across various filter structures, especially filters that would contain algebraic loops when constructed using basic elements. On the other hand, using basic elements enables the following optimization parameters:

- **Optimize for zero gains** — Terminate chains that contain Gain blocks with a gain of zero.
- **Optimize for unit gains** — Remove Gain blocks that scale by a factor of one.
- **Optimize for delay chains** — Substitute delay chains made up of  $n$  unit delays with a single delay by  $n$ .
- **Optimize for negative gains** — Use subtraction in Sum blocks instead of negative gains in Gain blocks.

### Optimize for unit-scale values

Select this check box to scale unit gains between sections in SOS filters. This parameter is available only for SOS filters.

### Input processing

Specify how the block should process the input. The available options may vary depending on the settings of the **Filter Structure** and **Use basic elements for filter customization** parameters. You can set this parameter to one of the following options:

- **Columns as channels (frame based)** — When you select this option, the block treats each column of the input as a separate channel.
- **Elements as channels (sample based)** — When you select this option, the block treats each element of the input as a separate channel.

---

**Note** The Inherited (this choice will be removed – see release notes) option will be removed in a future release. See “Frame-Based Processing” in the *DSP System Toolbox Release Notes* for more information.

---

### Rate options

When the **Filter type** parameter specifies a multirate filter, select the rate processing rule for the block from following options:

- **Enforce single-rate processing** — When you select this option, the block maintains the sample rate of the input.
- **Allow multirate processing** — When you select this option, the block adjusts the rate at the output to accommodate an increased or reduced number of samples. To select this option, you must set the **Input processing** parameter to **Elements as channels (sample based)**.

**Use symbolic names for coefficients**

Select this check box to enable the specification of coefficients using MATLAB variables. The available coefficient names differ depending on the filter structure. Using symbolic names allows tuning of filter coefficients in generated code. By default, this check box is cleared.

**Supported Data Types**

| <b>Port</b> | <b>Supported Data Types</b>   |
|-------------|---|
| Input       | <ul style="list-style-type: none"><li>• Double-precision floating point</li><li>• Single-precision floating point</li></ul> |
| Output      | <ul style="list-style-type: none"><li>• Double-precision floating point</li><li>• Single-precision floating point</li></ul> |

**Version History**

**Introduced in R2006b**

# Hilbert Filter

Design Hilbert filter



## Library

Filtering / Filter Designs

dspfdesign

## Description

This block brings the filter design capabilities of the `filterBuilder` function to the Simulink environment.

This block supports SIMD code generation. For details, see “Code Generation” on page 2-769.

## Dialog Box

See “Hilbert Filter Design — Main Pane” on page 5-623 for more information about the parameters of this block. The **Data Types** and **Code** panes are not available for blocks in the DSP System Toolbox Filter Designs library.

### View filter response

This button opens the Filter Visualization Tool (**FVTool**) from the Signal Processing Toolbox product. You can use the tool to display:

- Magnitude response, phase response, and group delay in the frequency domain.
- Impulse response and step response in the time domain.
- Pole-zero information.

The tool also helps you evaluate filter performance by providing information about filter order, stability, and phase linearity. For more information on FVTool, see the Signal Processing Toolbox documentation.

### Filter Specifications

In this group, you specify your filter format, such as the impulse response and the filter order.

#### Impulse response

Select either FIR or IIR from the drop-down list. FIR is the default. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly.

---

**Note** The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

---

### Filter order mode

Select either **Minimum** (the default) or **Specify** from the drop-down list. Selecting **Specify** enables the **Order** option (see the following sections) so you can enter the filter order.

### Filter type

Select **Single-rate**, **Decimator**, **Interpolator**, or **Sample-rate converter**. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, the block specifies a single-rate filter.

- Selecting **Decimator** or **Interpolator** activates the **Decimation Factor** or the **Interpolation Factor** options respectively.
- Selecting **Sample-rate converter** activates both factors.

### Order

Enter the filter order. This option is enabled only if **Specify** was selected for **Filter order mode**.

### Decimation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to **Decimator** or **Sample-rate converter**. The default value is 2.

### Interpolation Factor

Enter the interpolation factor. This option is enabled only if the **Filter type** is set to **Interpolator** or **Sample-rate converter**. The default value is 2.

### Frequency Specifications

The parameters in this group allow you to specify your filter response curve.

#### Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select **Normalized (0 to 1)** to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—**Hz**, **kHz**, **MHz**, or **GHz**. Selecting one of the unit options enables the **Input sample rate** parameter.

#### Input sample rate

$F_s$ , specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

#### Transition width

Specify the width of the transitions at the ends of the passband. Specify the value in normalized frequency units or the absolute units you select in **Frequency units**.

### Magnitude Specifications

Parameters in this group specify the filter response in the passbands and stopbands.

**Magnitude units**

Specify the units for any parameter you provide in magnitude specifications. From the drop-down list, select one of the following options:

- Linear — Specify the magnitude in linear units.
- dB — Specify the magnitude in decibels (default)
- Squared — Specify the magnitude in squared units.

**Passband ripple**

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels.

**Algorithm**

The parameters in this group allow you to specify the design method and structure of your filter.

**Design Method**

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default FIR method is `Equiripple`.

**Design Options**

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

**Density factor**

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 20 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

**FIR Type**

Specify whether to design a type 3 or a type 4 FIR filter. The filter type is defined as follows:

- Type 3 — FIR filter with even order antisymmetric coefficients
- Type 4 — FIR filter with odd order antisymmetric coefficients

Select either 3 or 4 from the drop-down list.

**Filter Implementation****Structure**

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use `Direct-form FIR`, and IIR filters use `Cascade minimum-multiplier allpass`.

### Use basic elements to enable filter customization

Select this check box to implement the filter as a subsystem of basic Simulink blocks. Clear the check box to implement the filter as a high-level subsystem. By default, this check box is cleared.

The high-level implementation provides better compatibility across various filter structures, especially filters that would contain algebraic loops when constructed using basic elements. On the other hand, using basic elements enables the following optimization parameters:

- **Optimize for zero gains** — Terminate chains that contain Gain blocks with a gain of zero.
- **Optimize for unit gains** — Remove Gain blocks that scale by a factor of one.
- **Optimize for delay chains** — Substitute delay chains made up of  $n$  unit delays with a single delay by  $n$ .
- **Optimize for negative gains** — Use subtraction in Sum blocks instead of negative gains in Gain blocks.

### Input processing

Specify how the block should process the input. The available options may vary depending on the settings of the **Filter Structure** and **Use basic elements for filter customization** parameters. You can set this parameter to one of the following options:

- **Columns as channels (frame based)** — When you select this option, the block treats each column of the input as a separate channel.
- **Elements as channels (sample based)** — When you select this option, the block treats each element of the input as a separate channel.

### Rate options

When the **Filter type** parameter specifies a multirate filter, select the rate processing rule for the block from following options:

- **Enforce single-rate processing** — When you select this option, the block maintains the sample rate of the input.
- **Allow multirate processing** — When you select this option, the block adjusts the rate at the output to accommodate an increased or reduced number of samples. To select this option, you must set the **Input processing** parameter to **Elements as channels (sample based)**.

### Use variable names for coefficients

Select this check box to enable the specification of coefficients using MATLAB variables. The available coefficient names differ depending on the filter structure. Using symbolic names allows tuning of filter coefficients in generated code. By default, this check box is cleared.

## Supported Data Types

| Port   | Supported Data Types   |
|--------|--|
| Input  | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> </ul> |
| Output | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> </ul> |



## Version History

Introduced in R2006b

### Extended Capabilities

#### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

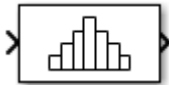
The Hilbert Filter block supports SIMD code generation using Intel AVX2 technology under these conditions:

- **Filter type** is set to Single-rate, Decimator, or Interpolator.
- For **Filter type** that is set to Single-rate, **Structure** is set to Direct-form FIR or Direct-form FIR transposed.
- For **Filter type** that is set to Decimator, **Structure** is set to Direct-form FIR polyphase decimator and **Rate options** is set to Enforce single-rate processing.
- For **Filter type** that is set to Interpolator:
  - **Interpolation Factor** cannot be equal to 1.
  - **Rate options** is set to Enforce single-rate processing.
- **Input processing** is set to Columns as channels (frame based).
- Input signal has a data type of single or double.
- Input port dimensions cannot be equal to [1 1].

The SIMD technology significantly improves the performance of the generated code.

# Histogram

Histogram of input or sequence of inputs



**Libraries:**

DSP System Toolbox / Statistics

## Description

The Histogram block computes the frequency distribution of the input elements along each column or along the entire input. You can specify the dimension using the **Find the histogram over** parameter. The block distributes the input elements based on their value into a number of discrete bins specified by the **Number of bins** parameter. For more details, see “Algorithms” on page 2-775.

When the input data is real, the bin boundaries are cast into the double data type. When the input data is complex, bin boundaries for double-precision inputs are cast into double, and bin boundaries for integer inputs are cast into double and squared. For an example, see “Compute the Histogram of Real and Complex Data”.

To track the frequency distribution of inputs over a period of time, select the **Running histogram** parameter.

When the input is complex, the block sorts the elements by their magnitude.

## Ports

### Input

**In** — Data input

vector | matrix | *N*-D array

The block accepts real-valued or complex-valued multichannel and multidimensional inputs. The input data type must be double precision, single precision, integer, or fixed point, with power-of-two slope and zero bias.

This port is unnamed until you select the **Running histogram** parameter and set the **Reset port** parameter to any option other than **None**.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fixed point | bus

**Rst** — Reset port

scalar

Specify the reset event that causes the block to reset the running histogram. The reset signal and the input data signal must have the same rate.

### Dependencies

To enable this port, select the **Running histogram** parameter, and set the **Reset port** parameter to any option other than **None**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean`

## Output

**Port\_1** — Histogram output  
vector | matrix |  $N$ -D array

Outputs the histogram data as a vector, matrix, or array. The output primarily depends on the settings of the **Running histogram** and **Find the histogram over** parameters.

For example, consider a two-dimensional input signal of size  $M$ -by- $N$ .

When you clear the **Running histogram** parameter, the block computes the frequency distribution in each column of input or along the entire input. When you set **Find the histogram over** to:

- **Each column** — The block computes a histogram for each column of the input. The output is an  $n$ -by- $N$  matrix, where  $n$  is the **Number of bins**. The  $j$ th column of the output matrix contains the histogram for the data in the  $j$ th column of the  $M$ -by- $N$  input matrix.
- **Entire input** — The block computes a histogram for the entire input vector, matrix, or  $N$ -D array. The output is an  $n$ -by-1 vector, where  $n$  is the **Number of bins**.

When you select the **Running histogram** parameter, the block computes the frequency distribution of all inputs over a period of time. In this case, when you set **Find the histogram over** to:

- **Each column** — The block computes a running histogram for each column of the input. The output is an  $n$ -by- $N$  matrix, where  $n$  is the **Number of bins**. The  $j$ th column of the output matrix contains the running histogram for the data in the  $j$ th column of the  $M$ -by- $N$  matrix input.
- **Entire input** — The block computes the running histogram for the entire input vector, matrix, or  $N$ -D array. The output is an  $n$ -by-1 vector, where  $n$  is the **Number of bins**.

The output data type is `uint32` when the input has a data type other than `single` or `double`. The largest number that can be represented by `uint32` is  $2^{32}-1$ . If the range of any input exceeds this value, the block wraps the value back to 0.

Data Types: `single` | `double` | `uint32`

## Parameters

### Main Tab

**Lower limit of histogram** — Lower boundary of lowest-valued bin  
scalar

Specify the lower boundary of the lowest-valued bin as a real scalar. This parameter does not accept NaN or Inf. If the input has a value less than **Lower limit of histogram**, the block places this element in the lowest-valued bin.

**Upper limit of histogram** — Upper boundary of highest-valued bin  
scalar

Specify the upper boundary of the highest-valued bin as a real scalar. This parameter does not accept NaN or Inf. If the input has a value greater than **Upper limit of histogram**, the block places this element in the highest-valued bin.

**Number of bins** — Number of histogram bins  
positive integer

Specify the number of histogram bins as a positive integer.

**Find the histogram over** — Compute histogram over each column or entire input  
Each column (default) | Entire input

Specify whether the histogram is computed over each column or the entire input. The options for this parameter are affected by the setting of the **Running histogram** parameter.

When you clear the **Running histogram** parameter, and you set **Find the histogram over** parameter to:

- Each column — The block computes the histogram over each column.
- Entire input — The block outputs the histogram over the entire input.

When you select the **Running histogram** parameter, and you set **Find the histogram over** parameter to:

- Each column — The block computes the running histogram over each column.
- Entire input — The block outputs the running histogram over the entire input.

**Normalized** — Normalize output  
off (default) | on

When you select this parameter, the block output,  $y$ , is normalized. In mathematical terms,  $\text{sum}(y) = 1$ . This parameter does not apply for fixed-point input signals.

**Running histogram** — Track frequency distribution over time  
off (default) | on

When you select this parameter, the block tracks the frequency distribution of inputs over a period of time.

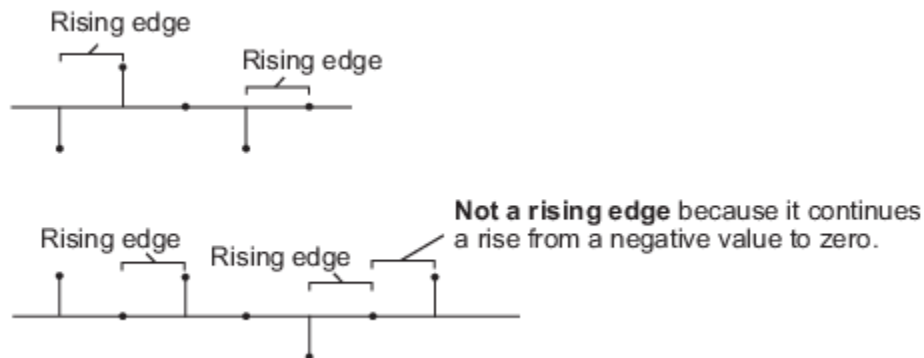
**Reset port** — Reset event  
Non-zero sample (default) | None | Rising edge | Falling edge | Either edge

The block resets the running histogram and empties all the bins whenever a reset event is detected at the optional **Rst** port. The reset sample time must equal the input sample time.

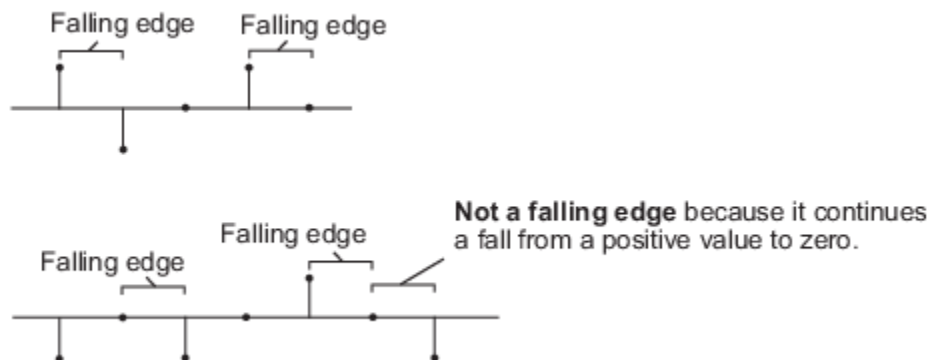
Use this parameter to specify the reset event:

- Non-zero sample — Triggers a reset operation at each sample time when the **Rst** input is not zero.

- None — Disables the **Rst** port.
- Rising edge — Triggers a reset operation when the **Rst** input does one of the following:
  - Rises from a negative value to either a positive value or zero.
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero.



- Falling edge — Triggers a reset operation when the **Rst** input does one of the following:
  - Falls from a positive value to a negative value or zero.
  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero.



- Either edge — Triggers a reset operation when the **Rst** input is a Rising edge or Falling edge.

---

**Note** When running simulations in the Simulink multitasking mode, reset signals have a one-sample latency. When the block detects a reset event, there is a one-sample delay at the reset port rate before the block applies the reset. For more information on latency and the Simulink tasking modes, see “Excess Algorithmic Delay (Tasking Latency)” and “Time-Based Scheduling and Code Generation” (Simulink Coder).

---

### Dependencies

To enable this parameter, select the **Running histogram** parameter.

## Data Types Tab

---

**Note** To use these parameters, the data input must be complex and fixed point. For all other inputs, the parameters on the **Data Types** tab are ignored.

---

### Rounding mode — Method of rounding operation

Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Specify the rounding mode for fixed-point operations. For more details, see rounding mode.

### Saturate on integer overflow — Method of overflow action

off (default) | on

When you select this parameter, the block saturates the result of its fixed-point operation. When you clear this parameter, the block wraps the result of its fixed-point operation. For details on saturate and wrap, see overflow mode for fixed-point operations.


### Product output — Product output data type

Inherit: Same as input (default) | fixdt([],16,0)

The squares of the real and imaginary parts of the complex input are stored in the **Product output** data type.

You can set this parameter to:

- `Inherit: Same as input` — The product output data type is the same as the input data type.
- `fixdt([],16,0)` — The product output data type is an autosigned, binary-point, scaled, fixed-point data type with a word length of 16 bits and a fraction length of 0.

Alternatively, you can set the **Product output** data type by using the **Data Type Assistant**. To use the assistant, click the **Show data type assistant** button .

For more information on the data type assistant, see “Specify Data Types Using Data Type Assistant” (Simulink) in *Simulink User's Guide* (Simulink).


### Accumulator — Accumulator data type

Inherit: Same as input (default) | Inherit: Same as product output | fixdt([],16,0)

The result of the sum of the squares of the real and imaginary parts of the complex input are stored in the **Accumulator** data type.

You can set this parameter to:

- `Inherit: Same as input` — The accumulator data type is the same as the input data type.
- `Inherit: Same as product output` — The accumulator data type is the same as the product output data type.
- `fixdt([],16,0)` — The accumulator data type is an autosigned, binary-point, scaled, fixed-point data type with a word length of 16 bits and a fraction length of 0.

Alternatively, you can set the **Accumulator** data type by using the **Data Type Assistant**. To use the assistant, click the **Show data type assistant** button .

For more information on the data type assistant, see “Specify Data Types Using Data Type Assistant” (Simulink) in *Simulink User's Guide* (Simulink).

**Lock data type settings against changes by the fixed-point tools** — Prevent fixed-point tools from overriding data types  
off (default) | on

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block.

## Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | double   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no                                      |
| <b>Multidimensional Signals</b> | no                                      |
| <b>Variable-Size Signals</b>    | no                                      |
| <b>Zero-Crossing Detection</b>  | no                                      |

## Algorithms

### Histogram

The histogram value for a given bin represents the frequency of occurrence of the input values bracketed by that bin. **Upper limit of histogram** specifies the upper boundary of the highest-valued bin,  $B_M$ . **Lower limit of histogram** specifies the lower boundary of the lowest-valued bin,  $B_m$ . The bins have equal width, given by:

$$\Delta = \frac{B_M - B_m}{n}.$$

The bins are centered at the following locations:

$$B_m + \left(k + \frac{1}{2}\right)\Delta \quad k = 0, 1, 2, \dots, n - 1.$$

$n$  is the number of bins, specified by the **Number of bins** parameter.

Input values that fall on the border between two bins are placed into the lower-valued bin. Each bin includes its upper boundary. For example, a bin of width 4 centered on the value 5 contains the input value 7, but not 3. Input values greater than the **Upper limit of histogram** parameter are placed in the highest-valued bin. Similarly, values less than the **Lower limit of histogram** parameter are placed in the lowest-valued bin.

The block sorts the complex data into bins by magnitude. The magnitude is the sum of the squares of the real and imaginary components of the complex data.

## Version History

Introduced before R2006a

### Extended Capabilities

#### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Generated code relies on the memcpy or memset function (`string.h`) under certain conditions.

#### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

The parameters on the **Data Types** tab are used only for complex fixed-point inputs. Complex inputs are sorted by magnitude, which is the sum of the squares of the real and imaginary components of the input. The results of the squares of the real and imaginary parts are stored in the **Product output** data type. The result of the sum of the squares is stored in the **Accumulator** data type. The parameters on the **Data Types** tab are ignored for all other inputs.

### See Also

#### Functions

histogram

#### Blocks

Sort | Median | Maximum | Minimum



# IDCT

Inverse discrete cosine transform (IDCT) of input



**Libraries:**

DSP System Toolbox / Transforms

## Description

The IDCT block computes the inverse discrete cosine transform (IDCT) of the input signal  $u$ . When the input is an  $M$ -by- $N$  matrix, the block computes the IDCT of each channel in the matrix.

Here is the equivalent MATLAB code.

```
y = idct(u)
```

## Ports

### Input

**Input 1** — Input signal

vector | matrix |  $N$ -D array

Specify the input signal as a vector, matrix, or an  $N$ -D array.

For all  $N$ -D input arrays, the block computes the IDCT across the first dimension. The size of the first dimension (frame size) must be a power of two. To work with other frame sizes, use the Pad block to pad or truncate the frame size to a power-of-two length.

When the input is an  $M$ -by- $N$  matrix, the block treats each input column as an independent channel containing  $M$  consecutive samples.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point | bus

### Output

**Output 1** — IDCT output

vector | matrix |  $N$ -D array

The block outputs the IDCT of the input signal as a vector, matrix, or an  $N$ -D array.

When the block outputs an  $M$ -by- $N$  matrix, the  $l^{\text{th}}$  column contains the length- $M$  IDCT of the corresponding input column.

$$y(m, l) = \sum_{k=1}^M w(k)u(k, l)\cos\frac{\pi(2m-1)(k-1)}{2M}, \quad m = 1, \dots, M$$

where

$$w(k) = \begin{cases} \frac{1}{\sqrt{M}}, & k = 1 \\ \sqrt{\frac{2}{M}}, & 2 \leq k \leq M \end{cases}$$

When the input is fixed-point (signed and unsigned), the output is signed fixed-point.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `fixed point` | `bus`

## Parameters

### Main Tab

**Sine and cosine computation** — Sine and cosine computation

`Table lookup (default)` | `Trigonometric fcn`

Set the block to compute sines and cosines by either looking up sine and cosine values in a speed-optimized table (`Table lookup`), or by making sine and cosine function calls (`Trigonometric fcn`).

To compute sines and cosines, set the parameter to one of these values:

- `Table lookup` -- The block computes and stores the trigonometric values before the simulation starts, and retrieves them during the simulation. When you generate code from the block, the processor running the generated code stores the trigonometric values computed by the block in a speed-optimized table, and retrieves the values during code execution.

The block runs much more quickly, but requires extra memory for storing the precomputed trigonometric values.

- `Trigonometric fcn` -- The block computes sine and cosine values during the simulation. When you generate code from the block, the processor running the generated code computes the sine and cosine values while the code runs.

The block runs more slowly, but does not need extra data memory. For code generation, the block requires a support library to emulate the trigonometric functions, increasing the size of the generated code.

### Data Types Tab

**Rounding mode** — Rounding mode

`Floor (default)` | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

Select the rounding mode for fixed-point operations. The sine table values do not obey this parameter; they always round to `Nearest`.

**Saturate on integer overflow** — Saturate on integer overflow

`off (default)` | `on`

When you select this parameter, the block saturates the result of its fixed-point operation. When you clear this parameter, the block wraps the result of its fixed-point operation. For details on `saturate` and `wrap`, see `overflow mode for fixed-point operations`.

---

**Note** The **Rounding mode** and **Saturate on integer overflow** parameters have no effect on numeric results when these conditions are met:

- **Product output** data type is `Inherit: Inherit via internal rule`.
- **Accumulator** data type is `Inherit: Inherit via internal rule`.

With these data type settings, the block operates in full-precision mode.

---

### Sine table — Sine table

`Inherit: Same word length as input (default) | fixd(1,16)`

Choose how you specify the word length of the values of the sine table. The fraction length of the sine table values always equals the word length minus one. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Same word length as input`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16)`


The sine table values do not obey the **Rounding mode** and **Saturate on integer overflow** parameters. The block always saturates and rounds off the sine table values to `Nearest`.

### Product output — Product output

`Inherit: Inherit via internal rule (default) | Inherit: Same as input | fixd(1,16,0)`

Specify the product output data type. See “Fixed-Point Data Types” on page 2-781 and “Multiplication Data Types” for illustrations depicting the use of the product output data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Product output** parameter.


See “Specify Data Types Using Data Type Assistant” (Simulink) in *Simulink User's Guide* (Simulink) for more information.

### Accumulator — Accumulator

`Inherit: Inherit via internal rule (default) | Inherit: Same as input | Inherit: Same as product output | fixd(1,16,0)`

Specify the accumulator data type. See “Fixed-Point Data Types” on page 2-781 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Accumulator** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in *Simulink User's Guide* (Simulink) for more information.

### Output — Output data type

Inherit: Inherit via internal rule (default) | Inherit: Same as input |  
fixd(1,16,0)

Specify the output data type. See “Fixed-Point Data Types” on page 2-781 for illustrations depicting the use of the output data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`.


When you select `Inherit: Inherit via internal rule`, the block calculates the output word length and fraction length automatically. The internal rule first calculates an ideal output word length and fraction length using the following equations:

$$WL_{idealoutput} = WL_{input} + \text{floor}(\log_2(DCTlength - 1)) + 1$$

$$FL_{idealoutput} = FL_{input}$$

Using these ideal results, the internal rule then selects word lengths and fraction lengths that are appropriate for your hardware. For more information, see “Inherit via Internal Rule”.

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Output** parameter.

See “Control Data Types of Signals” (Simulink) in *Simulink User's Guide* (Simulink) for more information.

### Output Minimum — Output minimum

[] (default) | scalar

Specify the minimum value that the block should output. The default value is [] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Specify Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

### Output Maximum — Output Maximum

[] (default) | scalar

Specify the maximum value that the block should output. The default value is [] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Specify Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

**Lock data type settings against changes by the fixed-point tools** — Prevent fixed-point tools from overriding data types  
off (default) | on

Select this parameter to prevent the fixed-point tools from overriding the data types you specify in the block dialog box.

## Block Characteristics

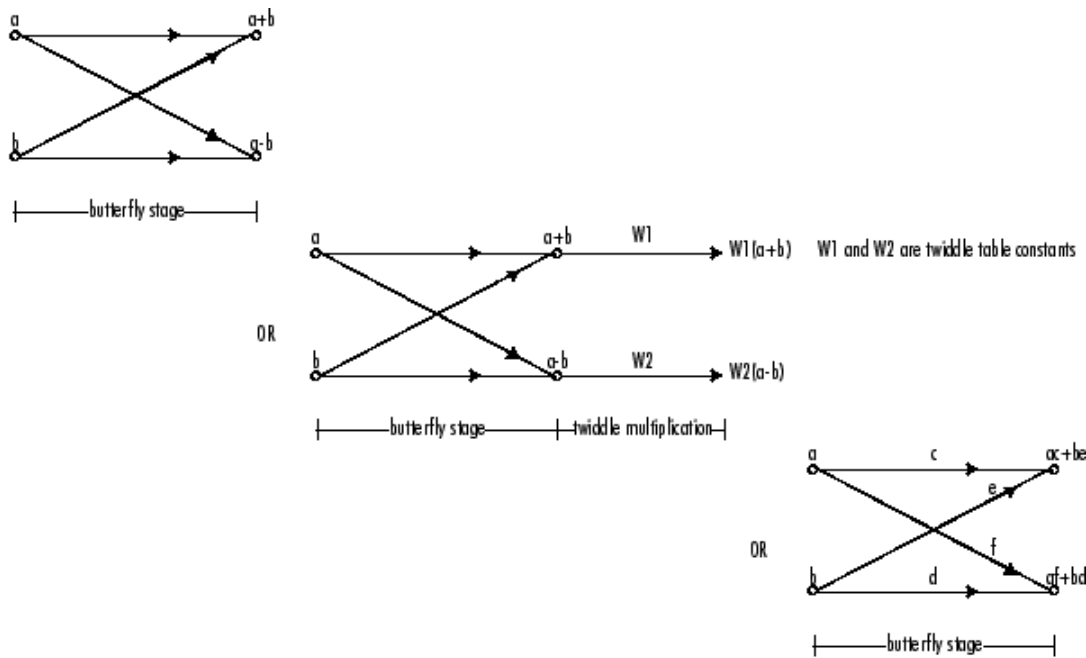
|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | double   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no                                      |
| <b>Multidimensional Signals</b> | yes                                     |
| <b>Variable-Size Signals</b>    | no                                      |
| <b>Zero-Crossing Detection</b>  | no                                      |

## More About

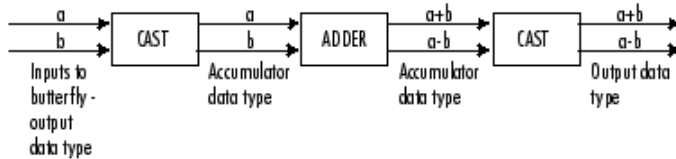
### Fixed-Point Data Types

The following diagrams show the data types used within the IDCT block for fixed-point signals. You can set the sine table, accumulator, product output, and output data types displayed in the diagrams in the block parameters dialog box of the IDCT block.

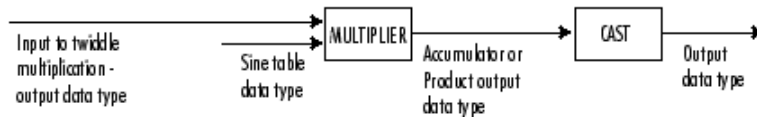
Inputs to the IDCT block are first cast to the output data type and stored in the output buffer. Each butterfly stage processes signals in the accumulator data type, with the final output of the butterfly being cast back into the output data type.



**Butterfly Stage Data Types**



**Twiddle Multiplication Data Types**



The output of the multiplier is in the product output data type when at least one of the inputs to the multiplier is real. When both of the inputs to the multiplier are complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication performed, see “Multiplication Data Types”.

**Note** When the block input is fixed point, all internal data types are signed fixed point.

**Version History**

Introduced before R2006a

**Extended Capabilities**

**C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

Generated code relies on the `memcpy` or `memset` function (`string.h`) under certain conditions.

To comply with the embedded systems coding standards, generated code is non-recursive.

**Fixed-Point Conversion**

Design and simulate fixed-point systems using Fixed-Point Designer™.

**See Also****Functions**

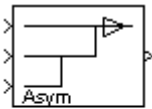
`idct`

**Blocks**

DCT | IFFT

## IDWT

Inverse discrete wavelet transform (IDWT) of input or reconstruct signals from subbands with smaller bandwidths and slower sample rates



**Libraries:**  
DSP System Toolbox / Transforms



### Description

You can configure this block to compute the inverse discrete wavelet transform (IDWT) or reconstruct a signal from subbands with smaller bandwidths and slower sample rates. When the block computes the inverse discrete wavelet transform (IDWT) of the input, the output has the same dimensions as the input. Each column of the output is the IDWT of the corresponding input column. When reconstructing a signal, the block uses a series of highpass and lowpass FIR filters to reconstruct the signal from the input subbands, as illustrated in “Multilevel Filter Banks” (the Asymmetric one). The reconstructed signal has a wider bandwidth and faster sample rate than the input subbands.

You can specify the filter bank highpass and lowpass filters by providing vectors of filter coefficients. You can do so directly on the block dialog box. If you have a Wavelet Toolbox license, you can specify wavelet-based filters by selecting a wavelet from the **Filter** parameter. You must set the filter bank structure to asymmetric or symmetric, and specify the number of levels in the filter bank.

---

**Note** The IDWT block is the same as the Dyadic Synthesis Filter Bank block with different default settings. For more information on block ports and parameters, see the Dyadic Synthesis Filter Bank block reference page.

---

### Block Characteristics

|                                 |                 |
|---------------------------------|-----------------|
| <b>Data Types</b>               | double   single |
| <b>Direct Feedthrough</b>       | no              |
| <b>Multidimensional Signals</b> | no              |
| <b>Variable-Size Signals</b>    | no              |
| <b>Zero-Crossing Detection</b>  | no              |

### Version History

Introduced before R2006a



## References

- [1] Fliege, N. J. *Multirate Digital Signal Processing: Multirate Systems, Filter Banks, Wavelets*. West Sussex, England: John Wiley & Sons, 1994.
- [2] Strang, G. and T. Nguyen. *Wavelets and Filter Banks*. Wellesley, MA: Wellesley-Cambridge Press, 1996.
- [3] Vaidyanathan, P. P. *Multirate Systems and Filter Banks*. Englewood Cliffs, NJ: Prentice Hall, 1993.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Generated code relies on the `memcpy` or `memset` function (`string.h`) under certain conditions.

## See Also

### Objects

`dsp.DyadicSynthesisFilterBank`

### Blocks

Dyadic Synthesis Filter Bank | Two-Channel Synthesis Subband Filter | DWT

### Topics

“Multilevel Filter Banks”

“Calculate Channel Latencies Required for Wavelet Reconstruction”

## IFFT

Inverse fast Fourier transform (IFFT) of input



**Libraries:**  
DSP System Toolbox / Transforms

### Description

The IFFT block computes the inverse fast Fourier transform (IFFT) across the first dimension of an  $N$ -D input array. The block uses one of two possible FFT implementations. You can select an implementation based on the FFTW library or an implementation based on a collection of Radix-2 algorithms. To allow the block to choose the implementation, you can select **Auto**. For more information about the FFT implementations, see “Algorithms” on page 2-792.

When you specify an FFT length not equal to the length of the input vector (or first dimension of the input array), the block implements zero-padding, truncating, or modulo- $M$  (FFT length) data wrapping. This occurs before the IFFT operation. For an IFFT with  $P \leq M$ :

```
y = ifft(u,M) % P ≤ M
```

Wrapping:

```
y(:,L) = ifft(datawrap(u(:,L),M)) % P > M; L = 1, ..., N
```

Truncating:

```
y(:,L) = ifft(u,M) % P > M; L = 1, ..., N
```

---

**Tip** When the input length,  $P$ , is greater than the FFT length,  $M$ , you may see magnitude increases in your IFFT output. These magnitude increases occur because the IFFT block uses modulo- $M$  data wrapping to preserve all available input samples.

To avoid such magnitude increases, you can truncate the length of your input sample,  $P$ , to the FFT length,  $M$ . To do so, place a Pad block before the IFFT block in your model.

---

### Ports

#### Input

**Port\_1** — Input signal  
vector | matrix |  $N$ -D array

Input signal for computing the IFFT. The block computes the IFFT along the first dimension of the  $N$ -D input signal. The input can be floating-point or fixed-point, real, or complex, and conjugate symmetric.

For more information on how the block computes the IFFT, see “Description” on page 2-786 and “Algorithms” on page 2-792.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fixed point`

## Output

### Port 1 — IFFT of input

`vector` | `matrix` | `N-D array`

The IFFT, computed across the first dimension of an  $N$ -D input array. For more information on how the block computes the IFFT, see “Description” on page 2-786 and “Algorithms” on page 2-792.

The  $k$ th entry of the  $L$ th output channel,  $y(k,L)$ , is equal to the  $k$ th point of the  $M$ -point inverse discrete Fourier transform (IDFT) of the  $L$ th input channel:

$$y(k, L) = \frac{1}{M} \sum_{p=1}^P u(p, L) e^{j2\pi(p-1)(k-1)/M} \quad k = 1, \dots, M$$

The output has the same dimensions as the input. If the input signal has a floating-point data type, the data type of the output signal uses the same floating-point data type. Otherwise, the output can be any signed fixed-point data type. The block computes scaled and unscaled versions of the IFFT.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `fixed point`

## Parameters

### Main

#### FFT implementation — FFT implementation

`Auto (default)` | `Radix-2` | `FFTW`

Set this parameter to `FFTW` to support an arbitrary length input signal. The block restricts generated code with `FFTW` implementation to MATLAB host computers.

Set this parameter to `Radix-2` for bit-reversed processing, fixed or floating-point data, or portable C-code generation using the Simulink Coder. The dimension  $M$  of the  $M$ -by- $N$  input matrix, must be a power of two. To work with other input sizes, use the Pad block to pad or truncate these dimensions to powers of two, or if possible choose the `FFTW` implementation. For more information about the algorithms used by the `Radix-2` mode, see “Radix-2 Implementation” on page 2-792.

Set this parameter to `Auto` to let the block choose the FFT implementation. For floating-point inputs with non-power-of-two transform lengths, the `FFTW` algorithm is automatically chosen. Otherwise a `Radix-2` algorithm is automatically chosen. For non-power-of-two transform lengths, the block restricts generated code to MATLAB host computers.

#### Input is in bit-reversed order — Input is in bit-reversed order

`off (default)` | `on`

Select or clear this check box to designate the order of the input channel elements. Select this check box when the input is in bit-reversed order, and clear it when the input is in linear order. The block yields invalid outputs when you do not set this parameter correctly.

You cannot select this check box if you have cleared the **Inherit FFT length from input dimensions** check box, and you are specifying the FFT length using the **FFT length** parameter. Also, it cannot be selected when you set the **FFT implementation** parameter to FFTW.

For more information on ordering of the output, see “Linear and Bit-Reversed Output Order”.

#### Dependencies

To enable this parameter, set **FFT implementation** to Auto or Radix-2.

#### Input is conjugate symmetric — Input is conjugate symmetric

off (default) | on

Select this option when the block inputs conjugate symmetric data and you want real-valued outputs. Selecting this check box optimizes the block's computation method.

The FFT block yields conjugate symmetric output when you input real-valued data. Taking the IFFT of a conjugate symmetric input matrix produces real-valued output. Therefore, if the input to the block is both floating point and conjugate symmetric, and you select this check box, the block produces real-valued outputs.

You cannot select this check box if you have cleared the **Inherit FFT length from input dimensions** check box, and you are specifying the FFT length using the **FFT length** parameter.

If you input conjugate symmetric data to the IFFT block and do not select this check box, the IFFT block outputs a complex-valued signal with small imaginary parts. The block outputs invalid data if you select this option with non conjugate symmetric input data.

#### Divide output by FFT length — Divide output by FFT length

on (default) | off

When you select this check box, the block computes its output according to the IDFT equation, discussed in the Description on page 2-786 section.

When you clear this check box, the block computes the output using a modified version of the IDFT:  $M \cdot y(k, l)$ , which is defined by the following equation:

$$M \cdot y(k, l) = \sum_{p=1}^P u(p, l) e^{j2\pi(p-1)(k-1)/M} \quad k = 1, \dots, M$$

The modified IDFT equation does not include the multiplication factor of  $1/M$ .

#### Inherit FFT length from input dimensions — Inherit FFT length from input dimensions

on (default) | off

Select to inherit the FFT length from the input dimensions. If you do not select this parameter, the **FFT length** parameter becomes available to specify the length. You cannot clear this parameter when you select either the **Input is in bit-reversed order** or the **Input is conjugate symmetric** parameter.

#### FFT length — FFT length

64 (default) | integer

Specify FFT length as an integer greater than or equal to two.

When you set the **FFT implementation** parameter to Radix-2, or when you check the **Output in bit-reversed order** check box, this value must be a power of two.

#### Dependencies

To enable this parameter, clear the **Inherit FFT length from input dimensions** check box.

**Wrap input data when FFT length is shorter than input length** — Wrap or truncate the input

on (default) | off

Choose to wrap or truncate the input, depending on the FFT length. If you select this parameter, modulo-length data wrapping occurs before the FFT operation when the FFT length is shorter than the input length. If you clear this parameter, truncation of the input data to the FFT length occurs before the FFT operation.

#### Dependencies

To enable this parameter, clear the **Inherit FFT length from input dimensions** check box.

#### Data Types

**Rounding mode** — Rounding method

Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Select the rounding mode for fixed-point operations.

#### Limitations

The sine table values do not obey this parameter; instead, they always round to **Nearest**.

The **Rounding mode** parameter has no effect on numeric results when all these conditions are met:

- **Product output** data type is **Inherit: Inherit via internal rule**.
- **Accumulator** data type is **Inherit: Inherit via internal rule**.

With these data type settings, the block operates in full-precision mode.

**Saturate on integer overflow** — Saturate on integer overflow

off (default) | on

When you select this parameter, the block saturates the result of its fixed-point operation. When you clear this parameter, the block wraps the result of its fixed-point operation. For details on **saturate** and **wrap**, see **overflow mode for fixed-point operations**.

#### Limitations

The **Saturate on integer overflow** parameter has no effect on numeric results when all these conditions are met:

- **Product output** data type is `Inherit: Inherit via internal rule`.
- **Accumulator** data type is `Inherit: Inherit via internal rule`.


With these data type settings, the block operates in full-precision mode.

#### **Sine table** — Data type of sine table values

`Inherit: Same word length as input (default) | fixdt(1,16)`

Choose how you specify the word length of the values of the sine table. The fraction length of the sine table values always equals the word length minus one. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Same word length as input`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Sine table** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) for more information.

#### **Limitations**


The sine table values do not obey the **Rounding mode** and **Saturate on integer overflow** parameters; instead, they are always saturated and rounded to Nearest.

#### **Product output** — Product output data type

`Inherit: Inherit via internal rule (default) | Inherit: Same as input | fixdt(1,16,0)`

Specify the product output data type. See “Fixed Point” on page 2-794 and “Multiplication Data Types” for illustrations depicting the use of the product output data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Product output** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) for more information.


#### **Accumulator** — Accumulator data type

`Inherit: Inherit via internal rule (default) | Inherit: Same as input | Inherit: Same as product output | fixdt(1,16,0)`

Specify the accumulator data type. See “Fixed Point” on page 2-794 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Accumulator** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) for more information.

### Output — Output data type

Inherit: `Inherit via internal rule (default) | Inherit: Same as input | fixdt(1,16,0)`

Specify the output data type. See “Fixed Point” on page 2-794 for illustrations depicting the use of the output data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`.

When you select `Inherit: Inherit via internal rule`, the block calculates the output word length and fraction length automatically. The equations that the block uses to calculate the ideal output word length and fraction length depend on the setting of the **Divide output by FFT length** check box.


- When you select the **Divide output by FFT length** check box, the ideal output word and fraction lengths are the same as the input word and fraction lengths.
- When you clear the **Divide output by FFT length** check box, the block computes the ideal output word and fraction lengths according to the following equations:

$$WL_{idealoutput} = WL_{input} + \text{floor}(\log_2(FFTlength - 1)) + 1$$

$$FL_{idealoutput} = FL_{input}$$

Using these ideal results, the internal rule then selects word lengths and fraction lengths that are appropriate for your hardware. For more information, see “Inherit via Internal Rule”.

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Output** parameter.

See “Control Data Types of Signals” (Simulink) for more information.

### Output Minimum — Minimum value block should output

`[]` (default) | scalar

Specify the minimum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Specify Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

### Output Maximum — Maximum value block should output

`[]` (default) | scalar

Specify the maximum value that the block should output. The default value is [ ] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Specify Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

**Lock data type settings against changes by the fixed-point tools** — Prevent fixed-point tools from overriding data types

off (default) | on

Select this parameter to prevent the fixed-point tools from overriding the data types you specify in the block dialog box.

## Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | double   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no                                      |
| <b>Multidimensional Signals</b> | yes                                     |
| <b>Variable-Size Signals</b>    | yes <sup>a</sup>                        |
| <b>Zero-Crossing Detection</b>  | no                                      |

<sup>a</sup> Variable-size signals are only supported when the Inherit FFT length from input dimensions checkbox is selected.

## Algorithms

### FFTW Implementation

The FFTW implementation provides an optimized FFT calculation including support for power-of-two and non-power-of-two transform lengths in both simulation and code generation. Generated code using the FFTW implementation will be restricted to MATLAB host computers. The data type must be floating-point. Refer to Simulink Coder for more details on generating code.

### Radix-2 Implementation

The Radix-2 implementation supports bit-reversed processing, fixed or floating-point data, and allows the block to provide portable C-code generation using the Simulink Coder. The dimension  $M$  of the  $M$ -by- $N$  input matrix must be a power of two. To work with other input sizes, use the Pad block to pad or truncate these dimensions to powers of two, or if possible choose the FFTW implementation.

With Radix-2 selected, the block implements one or more of the following algorithms:

- Butterfly operation
- Double-signal algorithm
- Half-length algorithm
- Radix-2 decimation-in-time (DIT) algorithm
- Radix-2 decimation-in-frequency (DIF) algorithm



### Radix-2 Algorithms for Real or Complex Input Complexity

| Parameter Settings   | Algorithms Used for IFFT Computation  |
|--|---|
| <input type="checkbox"/> Input is in bit-reversed order<br><input type="checkbox"/> Input is conjugate symmetric                       | Bit-reversal operation and radix-2 DIT  |
| <input checked="" type="checkbox"/> Input is in bit-reversed order<br><input type="checkbox"/> Input is conjugate symmetric            | Radix-2 DIT   |
| <input type="checkbox"/> Input is in bit-reversed order<br><input checked="" type="checkbox"/> Input is conjugate symmetric            | Bit-reversal operation and radix-2 DIT in conjunction with the half-length and double-signal algorithms |
| <input checked="" type="checkbox"/> Input is in bit-reversed order<br><input checked="" type="checkbox"/> Input is conjugate symmetric | Radix-2 DIT in conjunction with the half-length and double-signal algorithms                            |

### Radix-2 Optimization for the Table of Trigonometric Values

In certain situations, the block's Radix-2 algorithm computes all the possible trigonometric values of the twiddle factor

$$e^{j\frac{2\pi k}{K}}$$

where  $K$  is the greater value of either  $M$  or  $N$  and  $k = 0, \dots, K - 1$ . The block stores these values in a table and retrieves them during simulation. The number of table entries for fixed-point and floating-point is summarized in the following table:

| Number of Table Entries for N-Point FFT |        |
|---|--------|
| floating-point                          | $3N/4$ |
| fixed-point                             | $N$    |

## Version History

Introduced before R2006a

## References

- [1] Orfanidis, S. J. *Introduction to Signal Processing*. Upper Saddle River, NJ: Prentice Hall, 1996, p. 497.
- [2] Proakis, John G. and Dimitris G. Manolakis. *Digital Signal Processing*, 3rd ed. Upper Saddle River, NJ: Prentice Hall, 1996.
- [3] FFTW (<https://www.fftw.org>)
- [4] Frigo, M. and S. G. Johnson, "FFTW: An Adaptive Software Architecture for the FFT," *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Vol. 3, 1998, pp. 1381-1384.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Usage notes and limitations:

- When the following conditions apply, the executable generated from this block relies on prebuilt dynamic library files (.dll files) included with MATLAB:
  - **FFT implementation** is set to FFTW.
  - **Inherit FFT length from input dimensions** is cleared, and **FFT length** is set to a value that is not a power of two.

Use the packNGo function to package the code generated from this block and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see “How To Run a Generated Executable Outside MATLAB”.

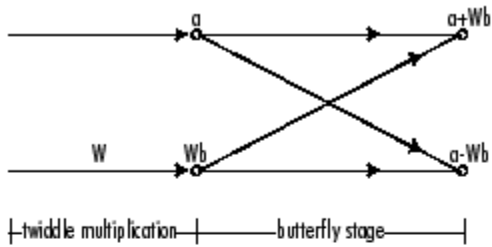
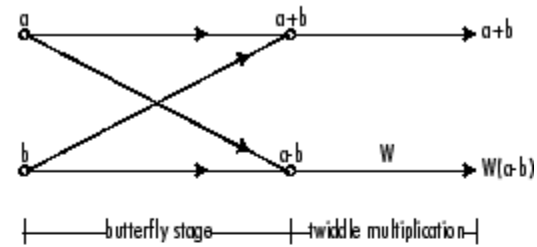
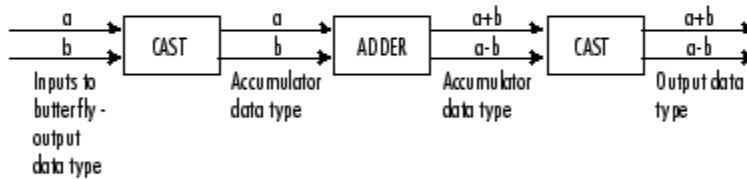
- When the FFT length is a power of two, you can generate standalone C and C++ code from this block.

### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

The following diagrams show the data types used within the IFFT block for fixed-point signals. You can set the sine table, accumulator, product output, and output data types displayed in the diagrams in the IFFT block dialog box, as discussed in “Parameters” on page 2-787.

The IFFT block first casts input to the output data type and then stores it in the output buffer. Each butterfly stage then processes signals in the accumulator data type, with the final output of the butterfly being cast back into the output data type. The block multiplies in a twiddle factor before each butterfly stage in a decimation-in-time IFFT, and after each butterfly stage in a decimation-in-frequency IFFT.

**Decimation-in-time IFFT****Decimation-in-frequency IFFT****Butterfly stage data types****Twiddle multiplication data types**

The output of the multiplier is in the accumulator data type because both of the inputs to the multiplier are complex. For details on the complex multiplication performed, see “Multiplication Data Types”.

---

**Note** When the block input is fixed point, all internal data types are signed fixed point.

---

**See Also****Objects**

`dsp.IFFT` | `dsp.FFT`

**Functions**

`bitrevorder` | `fft` | `ifft`

**Blocks**

FFT | IDCT | Pad

**Topics**

“Linear and Bit-Reversed Output Order”

## IIR Halfband Interpolator

Interpolate signal using polyphase IIR halfband filter



### Libraries:

DSP System Toolbox / Filtering / Multirate Filters

## Description

The IIR Halfband Interpolator block performs efficient polyphase interpolation of the input signal by a factor of 2. To design the halfband filter, you can use an elliptic design or a quasi-linear phase design. The block uses these design methods to compute the filter coefficients. To filter the inputs, the block uses a polyphase structure. The allpass filters in the polyphase structure are in a minimum multiplier form.

Elliptic design introduces nonlinear phase and creates the filter using fewer coefficients than the quasi-linear design. Quasi-linear phase design overcomes phase nonlinearity at the cost of additional coefficients.

Alternatively, instead of designing the halfband filter using a design method, you can specify the filter coefficients directly. When you choose this option, the allpass filters in the two branches of the polyphase implementation can be in a minimum multiplier form or a wave digital form.

You can also use the block to implement the synthesis portion of a two-band filter bank to synthesize a signal from lowpass and highpass subbands.

## Ports

### Input

**LP** — Data input

column vector | matrix

Specify the data input as a vector or a matrix. If the input signal is a matrix, the block treats each column of the matrix as an independent channel.

When you select the **Input highpass subband** parameter, this block acts as a halfband synthesis filter bank. The input at this port is then the lowpass subband output of a halfband analysis filter bank.

This port is unnamed until you select the **Input highpass subband** parameter.

Data Types: `single` | `double`

**HP** — Second input to the synthesis filter bank

column vector | matrix

Specify the second input to the synthesis filter bank as a column vector or a matrix. This signal is the highpass subband output of a halfband analysis filter bank. If the input signal is a matrix, the block treats each column of the matrix as an independent channel.

The size, data type, and complexity of both the inputs must be the same.

### Dependency

To enable this port, select the **Input highpass subband** parameter.

Data Types: `single` | `double`

**coeffs1** — Branch 1 allpass polynomial coefficients

*N*-by-1 vector | *N*-by-2 matrix

Specify the allpass polynomial filter coefficients of the first branch as an *N*-by-1 vector or *N*-by-2 matrix of *N* first-order or second-order allpass sections.

### Dependencies

To enable this parameter, set:

- **Filter specification** to Coefficients
- **Internal allpass structure** to Minimum multiplier
- Clear the **Make the first branch a pure delay** parameter
- Select the **Specify coefficients from input port** parameter

Data Types: `single` | `double`

**coeffs2** — Branch 2 allpass polynomial coefficients

*N*-by-1 vector | *N*-by-2 matrix

Specify the allpass polynomial filter coefficients of the second branch as an *N*-by-1 vector or *N*-by-2 matrix of *N* first-order or second-order allpass sections.

### Dependencies

To enable this parameter, set:

- **Filter specification** to Coefficients
- **Internal allpass structure** to Minimum multiplier
- Select the **Specify coefficients from input port** parameter

Data Types: `single` | `double`

### Output

**Output** — Output of interpolator

column vector | matrix

Output of the interpolator, returned as a column vector or a matrix. The number of rows in the interpolator output is twice the number of rows in the input signal.

Data Types: `single` | `double`

## Parameters

**Filter specification** — Filter design parameters

Transition width and stopband attenuation (default) | Filter order and stopband attenuation | Filter order and transition width | Coefficients

Select the parameters that the block uses to design the IIR halfband filter. Because the filter design has only two degrees of freedom, you can specify only two of the three parameters:

- Transition width and stopband attenuation (default) — Design the filter using **Transition width (Hz)** and **Stopband attenuation (dB)**. This design is the minimum order design.
- Filter order and transition width — Design the filter using **Filter order** and **Transition width (Hz)**.
- Filter order and stopband attenuation — Design the filter using **Filter order** and **Stopband attenuation (dB)**.
- Coefficients— Specify the filter coefficients directly using the enabled parameters.

**Transition width (Hz)** — Transition width

4.1e3 (default) | positive real scalar

Specify the transition width of the IIR halfband filter as a real positive scalar in Hz. The transition width must be less than 1/2 the sample rate of the input signal.

#### Dependencies

To enable this parameter, set **Filter specification** to Filter order and transition width or Transition width and stopband attenuation.

**Filter order** — Order of the IIR halfband filter

9 (default) | positive integer

Specify the filter order as a positive integer. If you set **Design method** to Elliptic, then **Filter order** must be an odd integer greater than one. If you set **Design method** to Quasi-linear phase, then **Filter order** must be a multiple of four.

#### Dependencies

To enable this parameter, set **Filter specification** to Filter order and transition width or Filter order and stopband attenuation.

**Stopband attenuation (dB)** — Minimum attenuation needed in stopband

80 (default) | positive real scalar

Specify the minimum attenuation needed in the stopband of the IIR halfband filter as a real positive scalar in dB.

#### Dependencies

To enable this parameter, set **Filter specification** to Filter order and stopband attenuation or Transition width and stopband attenuation.

**Design method** — Design method

Elliptic (default) | Quasi-linear phase

Specify the design method for the IIR halfband filter.

- **Elliptic** (default) — The filter has a nonlinear phase and uses few coefficients.
- **Quasi-linear phase** — The first branch of the polyphase filter structure is a pure delay, which results in an approximately linear phase response.

#### Dependencies

To enable this parameter, set **Filter specification** to any option except **Coefficients**.

**Internal allpass structure** — Filter structure in coefficient mode

Minimum multiplier (default) | Wave Digital Filter

Specify the internal allpass filter implementation structure as **Minimum multiplier** or **Wave Digital Filter**. Each structure uses a different coefficients set, independently stored in the corresponding coefficients property. The default is **Minimum multiplier**.

#### Dependencies

To enable this parameter, set **Filter specification** to **Coefficients**.

**Make the first branch a pure delay** — Make the first branch a pure delay

off (default) | on

When you select this check box, the first branch of the polyphase filter structure becomes a pure delay, and the **Branch 1 allpass polynomial coefficients** and **Branch 1 Wave Digital coefficients** parameters do not apply.

By default, this check box is not selected.

#### Dependencies

To enable this parameter, set **Filter specification** to **Coefficients**.

**Delay length in samples for branch 1** — Length of the delay

1 (default) | finite positive scalar

Specify the length of the first branch delay as a finite positive scalar. The default is 1.

#### Dependencies

To enable this parameter, set:

- **Filter specification** to **Coefficients**
- Select the **Make the first branch a pure delay** parameter

**Specify coefficients from input port** — Specify coefficients from input port

off (default) | on

When you select this check box, you can input the branch 1 allpass polynomial coefficients and branch 2 allpass polynomial coefficients through the input ports **coeffs1** and **coeffs2**. When you clear this check box, you specify the coefficients in the block dialog box through the **Branch 1 allpass polynomial coefficients** and **Branch 2 allpass polynomial coefficients** parameters.

**Dependencies**

To enable this parameter, set:

- **Filter specification** to Coefficients
- **Internal allpass structure** to Minimum multiplier

**Branch 1 allpass polynomial coefficients** — Allpass polynomial filter coefficients of first branch

[0.1284563; 0.7906755] (default) |  $N$ -by-1 |  $N$ -by-2

Specify the allpass polynomial filter coefficients of the first branch as an  $N$ -by-1 vector or  $N$ -by-2 matrix of  $N$  first-order or second-order allpass sections.

This parameter is tunable, that is, you can change its value during simulation.

**Dependencies**

To enable this parameter, set:

- **Filter specification** to Coefficients
- **Internal allpass structure** to Minimum multiplier
- Clear the **Make the first branch a pure delay** parameter
- Clear the **Specify coefficients from input port** parameter

**Branch 2 allpass polynomial coefficients** — Allpass polynomial filter coefficients of second branch

[0.4295667] (default) |  $N$ -by-1 |  $N$ -by-2

Specify the allpass polynomial filter coefficients of the second branch as an  $N$ -by-1 vector or  $N$ -by-2 matrix of  $N$  first-order or second-order allpass sections.

This parameter is tunable, that is, you can change its value during simulation.

**Dependencies**

To enable this parameter, set:

- **Filter specification** to Coefficients
- **Internal allpass structure** to Minimum multiplier
- Clear the **Specify coefficients from input port** parameter

**Branch 1 Wave Digital coefficients** — Allpass filter coefficients of first branch in wave digital filter form

[0.1284563; 0.7906755] (default) |  $N$ -by-1 |  $N$ -by-2

Specify the allpass filter coefficients of the first branch in wave digital filter (WDF) form as an  $N$ -by-1 vector or  $N$ -by-2 matrix of  $N$  first-order or second-order allpass sections.

The magnitude of each WDF coefficient must not be greater than 1.

**Dependencies**

To enable this parameter, set:



- **Filter specification** to Coefficients
- **Internal allpass structure** to Wave Digital Filter
- Clear the **Make the first branch a pure delay** parameter

**Branch 2 Wave Digital coefficients** — Allpass filter coefficients of second branch in wave digital filter form

[0.4295667] (default) |  $N$ -by-1 |  $N$ -by-2

Specify the allpass filter coefficients of the second branch in wave digital filter (WDF) form as an  $N$ -by-1 vector or  $N$ -by-2 matrix of  $N$  first-order or second-order allpass sections.

The magnitude of each WDF coefficient must not be greater than 1.

#### Dependencies

To enable this parameter, set:

- **Filter specification** to Coefficients
- **Internal allpass structure** to Wave Digital Filter

**Last section of branch 2 is first order** — Make the last section of the second branch as first order

off (default) | on

When you select this check box, the block treats the last section of the second branch as a first order section. When the coefficients of the second branch are in an  $N$ -by-2 matrix, the block ignores the second element of the last row of the matrix. The last section of the second branch then becomes a first-order section.

When you clear this check box, the block treats the last section of the second branch as a second-order section. When the coefficients of the second branch are in an  $N$ -by-1 matrix, the block ignores this parameter.

By default, this check box is cleared.

#### Dependencies

To enable this parameter, set **Filter specification** to Coefficients.

**Input highpass subband** — Input highpass subband

off (default) | on

When you select this check box, the block acts as a synthesis filter bank. The block accepts two inputs to synthesize: lowpass and highpass subbands. When you clear this check box, the block acts as an IIR halfband interpolator and accepts a single vector or matrix as input. By default, this check box is cleared.

**Inherit sample rate from input** — Inherit sample rate from input signal

off (default) | on

When you select this check box, the block inherits its sample rate from the input signal. The block calculates the sample rate based on the sample time of the input port. When you clear this check box, you specify the sample rate in **Input sample rate (Hz)**.

**Input sample rate (Hz)** — Sample rate of input signal

22050 (default) | positive real scalar

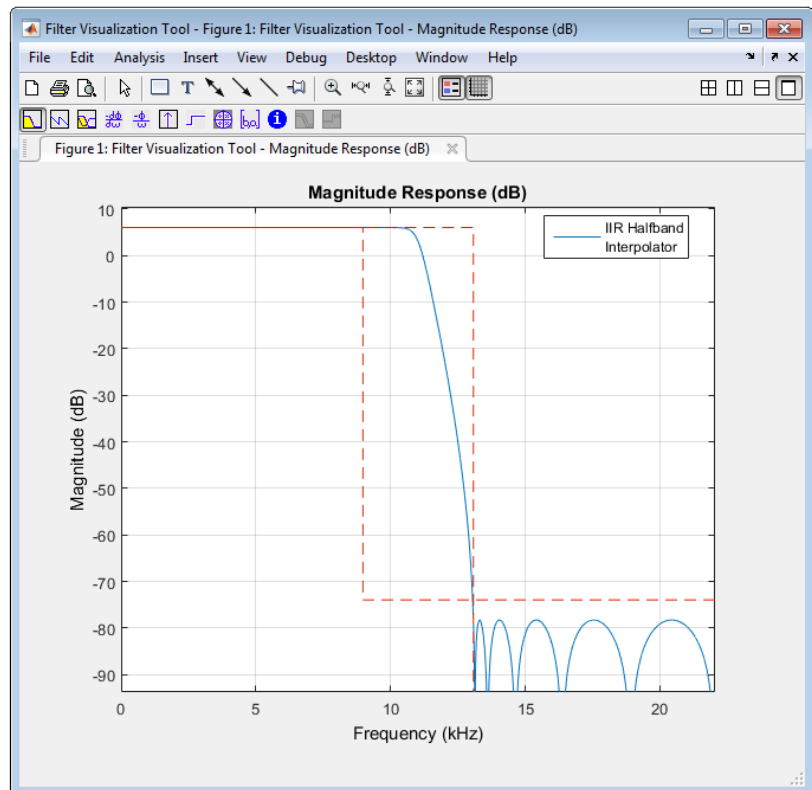
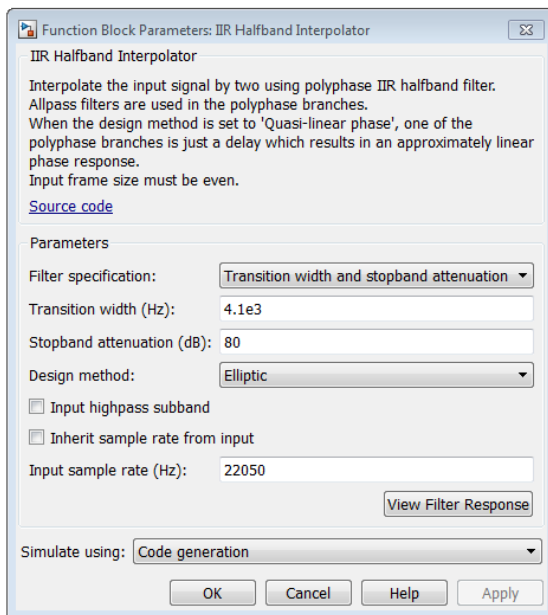
Specify the sample rate of the input signal as a scalar in Hz.

### Dependencies

To enable this parameter, clear the **Inherit sample rate from input** parameter.

**View Filter Response** — View Filter Response button

Click this button to open the Filter Visualization Tool (FVTool) and display the magnitude and phase response of the IIR Halfband Interpolator. The response is based on the values you specify in the block parameters dialog box. Changes made to these parameters update FVTool.



To update the magnitude response while FVTool is running, modify the dialog box parameters and click **Apply**.

**Simulate using** — Simulate using

Code generation (default) | Interpreted execution

Specify the type of simulation to run. You can set this parameter to:

- Code generation (default)

Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than Interpreted execution.

- Interpreted execution

Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than Code generation.

## Block Characteristics

|                                 |                 |
|---------------------------------|-----------------|
| <b>Data Types</b>               | double   single |
| <b>Direct Feedthrough</b>       | no              |
| <b>Multidimensional Signals</b> | no              |
| <b>Variable-Size Signals</b>    | yes             |
| <b>Zero-Crossing Detection</b>  | no              |

## Algorithms

### Polyphase Implementation with Halfband Filters

When you filter your signal, the IIR halfband interpolator uses an efficient polyphase implementation for halfband filters. You can use a polyphase implementation to move the upsampling operation after filtering. This change enables you to filter at a lower sampling rate.

IIR halfband filters are generally modeled using two parallel allpass filter branches.

$$H(z) = 0.5 * [A_1(z^2) + z^{-1}A_2(z^2)]$$

### Elliptic Design

The allpass filters for elliptic IIR halfband filter are given as

$$A_1(z) = \prod_{k=1}^{K_1} \frac{a_k^{(1)} + z^{-1}}{1 + a_k^{(1)}z^{-1}}$$

$$A_2(z) = \prod_{k=1}^{K_2} \frac{a_k^{(2)} + z^{-1}}{1 + a_k^{(2)}z^{-1}}$$

### Quasi-Linear Phase Design

A near-linear phase response for IIR halfband filters is achieved by making one of the branches a pure delay. In this design, the cost of the filter increases.

The allpass filters for quasi-linear phase IIR halfband filter are

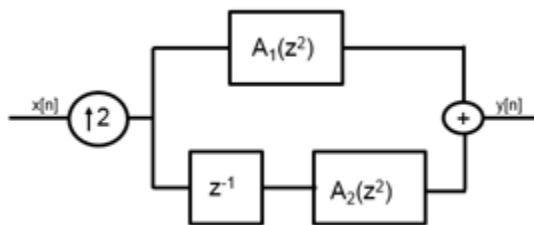
$$A_1(z) = z^{-k}$$

where,  $k$  is the length of the delay.

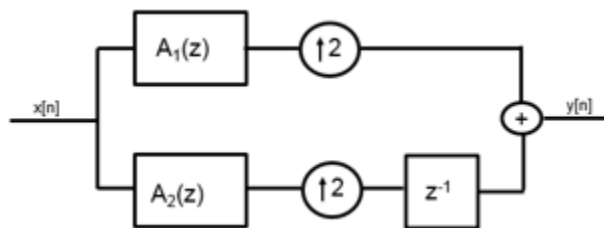
$$A_2(z) = \prod_{k=1}^{K_2^{(1)}} \frac{a_k + z^{-1}}{1 + a_k z^{-1}} \prod_{k=1}^{K_2^{(2)}} \frac{c_k + b_k z^{-1} + z^{-2}}{1 + b_k z^{-1} + c_k z^{-2}}$$

where  $N$  is the order of the IIR halfband filter.

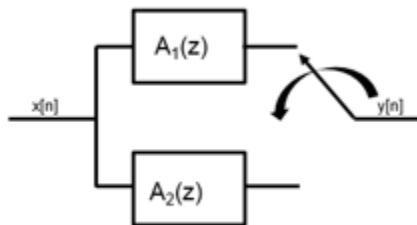
You can represent the upsampling-by-2 operation followed by the filtering operation using this figure.



Using the multirate noble identity for upsampling, you can move the upsampling operation after filtering. This enables you to filter at a lower rate.



To efficiently implement the halfband interpolator, this algorithm replaces the upsampling operator, delay block, and adder with a commutator switch. The commutator switch operates at twice the input sample rate. This is shown in the following figure.



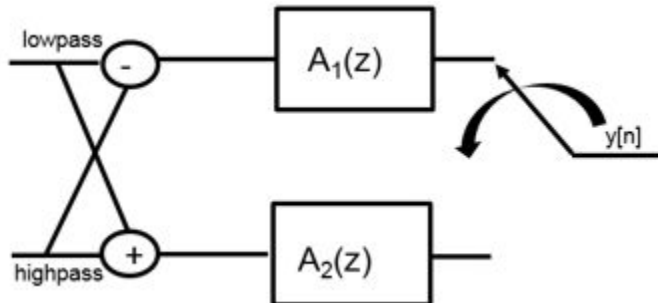
The commutator switch takes input samples from the two branches alternately, one sample at a time. This doubles the sampling rate of the input signal.

### Synthesis Filter Bank

Transfer function of the complementary high-pass filter branch of the synthesis filter bank is given by

$$G(z) = 0.5 * [A_1(z^2) - z^{-1}A_2(z^2)]$$

You can represent the synthesis filter bank as in this diagram.



The IIR halfband interpolator implements the synthesis portion of a two-band filter bank to synthesize a signal from lowpass and highpass subbands.

To summarize, the IIR halfband interpolator:

- Filters the input before upsampling
- Acts as a synthesis filter bank
- Has a nonlinear phase response and uses few coefficients with the elliptic design method
- Has near-linear phase response at the cost of additional coefficients with the quasi-linear phase design method, where one of the branches is a pure delay

## Version History

Introduced in R2015b

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

## See Also

### Objects

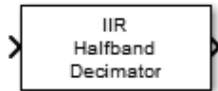
`dsp.IIRHalfbandInterpolator` | `dsp.IIRHalfbandDecimator`

### Blocks

IIR Halfband Decimator | FIR Halfband Interpolator | FIR Halfband Decimator

## IIR Halfband Decimator

Decimate signal using polyphase IIR halfband filter



### Libraries:

DSP System Toolbox / Filtering / Multirate Filters

### Description

The IIR Halfband Decimator block performs polyphase decimation of the input signal by a factor of 2. To design the halfband filter, you can specify the block to use an elliptic design or a quasi-linear phase design. The block uses these design methods to compute the filter coefficients. To filter the inputs, the block uses a polyphase structure. The allpass filters in the polyphase structure are in a minimum multiplier form.

Elliptic design introduces nonlinear phase and creates the filter using fewer coefficients than the quasi-linear design. Quasi-linear phase design overcomes phase nonlinearity at the cost of additional coefficients.

Alternatively, instead of designing the halfband filter using a design method, you can specify the filter coefficients directly. When you choose this option, the allpass filters in the two branches of the polyphase implementation can be in a minimum multiplier form or a wave digital form.

You can also use the block to implement the analysis portion of a two-band filter bank to filter a signal into lowpass and highpass subbands. For more details, see “Algorithms” on page 2-813.

### Ports

#### Input

**x** — Input signal  
column vector | matrix

Specify the input signal as a column vector or a matrix of size  $P$ -by- $Q$ . If the input signal is a matrix, the block treats each column of the matrix as an independent channel. The number of rows in the input signal must be a multiple of 2.

This block supports variable-size input signals.

This port is unnamed until you select the **Specify coefficients from input port** parameter.

Data Types: `single` | `double`

**coeffs1** — Branch 1 allpass polynomial coefficients  
 $N$ -by-1 vector |  $N$ -by-2 matrix

Specify the allpass polynomial filter coefficients of the first branch as an  $N$ -by-1 vector or  $N$ -by-2 matrix of  $N$  first-order or second-order allpass sections.

**Dependencies**

To enable this parameter, set:

- **Filter specification** to Coefficients
- **Internal allpass structure** to Minimum multiplier
- Clear the **Make the first branch a pure delay** parameter
- Select the **Specify coefficients from input port** parameter

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fixed point

**coeffs2** — Branch 2 allpass polynomial coefficients

$N$ -by-1 vector |  $N$ -by-2 matrix

Specify the allpass polynomial filter coefficients of the second branch as an  $N$ -by-1 vector or  $N$ -by-2 matrix of  $N$  first-order or second-order allpass sections.

**Dependencies**

To enable this parameter, set:

- **Filter specification** to Coefficients
- **Internal allpass structure** to Minimum multiplier
- Select the **Specify coefficients from input port** parameter

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fixed point

**Output**

**LP** — Lowpass subband of decimator output

column vector | matrix

Lowpass subband of the decimator output, returned as a column vector or a matrix of size  $P/2$ -by- $Q$ . As the filter is a halfband filter, the downsampling factor is always 2.

This port is unnamed until you select the **Output highpass subband** parameter.

Data Types: single | double

**HP** — Highpass subband of decimator output

column vector | matrix

Highpass subband of the decimator output, returned as a column vector or a matrix of size  $P/2$ -by- $Q$ . As the filter is a halfband filter, the downsampling factor is always 2.

This port is unnamed until you select the **Output highpass subband** parameter.

Data Types: single | double

**Parameters**

**Filter specification** — Filter design parameters

Transition width and stopband attenuation (default) | Filter order and transition width | Filter order and stopband attenuation | Coefficients

Select the parameters that the block uses to design the IIR halfband filter. Because the filter design has only two degrees of freedom, you can specify only two of the three parameters:

- Transition width and stopband attenuation (default) — Design the filter using **Transition width (Hz)** and **Stopband attenuation (dB)**. This design is the minimum order design.
- Filter order and transition width — Design the filter using **Filter order** and **Transition width (Hz)**.
- Filter order and stopband attenuation — Design the filter using **Filter order** and **Stopband attenuation (dB)**.
- Coefficients— Specify the filter coefficients directly using the enabled parameters.

**Transition width (Hz)** — Transition width in Hz

4.1e3 (default) | positive real scalar

Specify the transition width of the IIR halfband filter as a real positive scalar in Hz. The transition width must be less than 1/2 the sample rate of the input signal.

#### Dependencies

To enable this parameter, set **Filter specification** to Filter order and transition width or Transition width and stopband attenuation.

**Filter order** — Order of the IIR halfband filter

9 (default) | positive integer

Specify the filter order as a finite positive integer. If you set **Design method** to Elliptic, then **Filter order** must be an odd integer greater than one. If you set **Design method** to Quasi-linear phase, then **Filter order** must be a multiple of four.

#### Dependencies

To enable this parameter, set **Filter specification** to Filter order and transition width or Filter order and stopband attenuation.

**Stopband attenuation (dB)** — Minimum attenuation needed in stopband

80 (default) | positive real scalar

Specify the minimum attenuation needed in the stopband of the IIR halfband filter as a real positive scalar in dB.

#### Dependencies

To enable this parameter, set **Filter specification** to Filter order and stopband attenuation or Transition width and stopband attenuation.

**Design method** — Design method

Elliptic (default) | Quasi-linear phase



Specify the design method for the IIR halfband filter.

- **Elliptic** — The filter has a nonlinear phase and uses few coefficients.
- **Quasi-linear phase** — The first branch of the polyphase filter structure is a pure delay, which results in an approximately linear phase response.

#### Dependencies

To enable this parameter, set **Filter specification** to any option except **Coefficients**.

**Internal allpass structure** — Internal allpass filter implementation structure

Minimum multiplier (default) | Wave Digital Filter

Specify the internal allpass filter implementation structure as **Minimum multiplier** or **Wave Digital Filter**. Each structure uses a different coefficients set, independently stored in the corresponding coefficients property.

#### Dependencies

To enable this parameter, set **Filter specification** to **Coefficients**.

**Make the first branch a pure delay** — Make first branch a pure delay

off (default) | on

When you select this check box, the first branch of the polyphase filter structure becomes a pure delay, and the **Branch 1 allpass polynomial coefficients** and **Branch 1 Wave Digital coefficients** parameters do not apply.

By default, this check box is not selected.

#### Dependencies

To enable this parameter, set **Filter specification** to **Coefficients**.

**Delay length in samples for branch 1** — Length of delay

1 (default) | positive scalar

Specify the length of the first branch delay as a positive scalar.

#### Dependencies

To enable this parameter, set:

- **Filter specification** to **Coefficients**
- Select the **Make the first branch a pure delay** parameter

**Specify coefficients from input port** — Specify coefficients from input port

off (default) | on

When you select this check box, you can input the branch 1 allpass polynomial coefficients and branch 2 allpass polynomial coefficients through the input ports **coeffs1** and **coeffs2**. When you clear this check box, you specify the coefficients in the block dialog box through the **Branch 1 allpass polynomial coefficients** and **Branch 2 allpass polynomial coefficients** parameters.

**Dependencies**

To enable this parameter, set:

- **Filter specification** to Coefficients
- **Internal allpass structure** to Minimum multiplier

**Branch 1 allpass polynomial coefficients** — Allpass polynomial filter coefficients of first branch

[0.1284563; 0.7906755] (default) |  $N$ -by-1 |  $N$ -by-2

Specify the allpass polynomial filter coefficients of the first branch as an  $N$ -by-1 vector or  $N$ -by-2 matrix of  $N$  first-order or second-order allpass sections.

This parameter is tunable, that is, you can change its value during simulation.

**Tunable:** Yes

**Dependencies**

To enable this parameter, set:

- **Filter specification** to Coefficients
- **Internal allpass structure** to Minimum multiplier
- Clear the **Make the first branch a pure delay** parameter
- Clear the **Specify coefficients from input port** parameter

**Branch 2 allpass polynomial coefficients** — Allpass polynomial filter coefficients of second branch

[0.4295667] (default) |  $N$ -by-1 |  $N$ -by-2

Specify the allpass polynomial filter coefficients of the second branch as an  $N$ -by-1 vector or  $N$ -by-2 matrix of  $N$  first-order or second-order allpass sections.

This parameter is tunable, that is, you can change its value during simulation.

**Tunable:** Yes

**Dependencies**

To enable this parameter, set:

- **Filter specification** to Coefficients
- **Internal allpass structure** to Minimum multiplier
- Clear the **Specify coefficients from input port** parameter

**Branch 1 Wave Digital coefficients** — Allpass filter coefficients of first branch in wave digital filter form

[0.1284563; 0.7906755] (default) |  $N$ -by-1 |  $N$ -by-2

Specify the allpass filter coefficients of the first branch in wave digital filter (WDF) form as an  $N$ -by-1 vector or  $N$ -by-2 matrix of  $N$  first-order or second-order allpass sections.

The magnitude of each WDF coefficient must not be greater than 1.

**Dependencies**

To enable this parameter, set:

- **Filter specification** to Coefficients
- **Internal allpass structure** to Wave Digital Filter
- Clear the **Make the first branch a pure delay** parameter

**Branch 2 Wave Digital coefficients** — Allpass filter coefficients of second branch in wave digital filter form

[0.4295667] (default) |  $N$ -by-1 |  $N$ -by-2

Specify the allpass filter coefficients of the second branch in wave digital filter form as an  $N$ -by-1 vector or  $N$ -by-2 matrix of  $N$  first-order or second-order allpass sections.

The magnitude of each WDF coefficient must not be greater than 1.

**Dependencies**

To enable this parameter, set:

- **Filter specification** to Coefficients
- **Internal allpass structure** to Wave Digital Filter

**Last section of branch 2 is first order** — Last section of branch 2 is first order

off (default) | on

When you select this check box, the block treats the last section of the second branch as a first order section. When the coefficients of the second branch are in an  $N$ -by-2 matrix, the block ignores the second element of the last row of the matrix. The last section of the second branch then becomes a first-order section.

When you clear this check box, the block treats the last section of the second branch as a second-order section. When the coefficients of the second branch are in an  $N$ -by-1 matrix, the block ignores this parameter.

**Dependencies**

To enable this parameter, set **Filter specification** to Coefficients.

**Output highpass subband** — Output highpass subband

off (default) | on

When you select this check box, the block acts as an analysis filter bank, producing two power-complementary outputs. When you clear this check box, the block acts as an IIR halfband decimator and accepts a single vector or matrix as input.

**Inherit sample rate from input** — Inherit sample rate from input signal

off (default) | on

When you select this check box, the block inherits its sample rate from the input signal. When you clear this check box, you specify the sample rate in **Input sample rate (Hz)**.

**Input sample rate (Hz)** — Sample rate of input signal

44100 (default) | positive real scalar

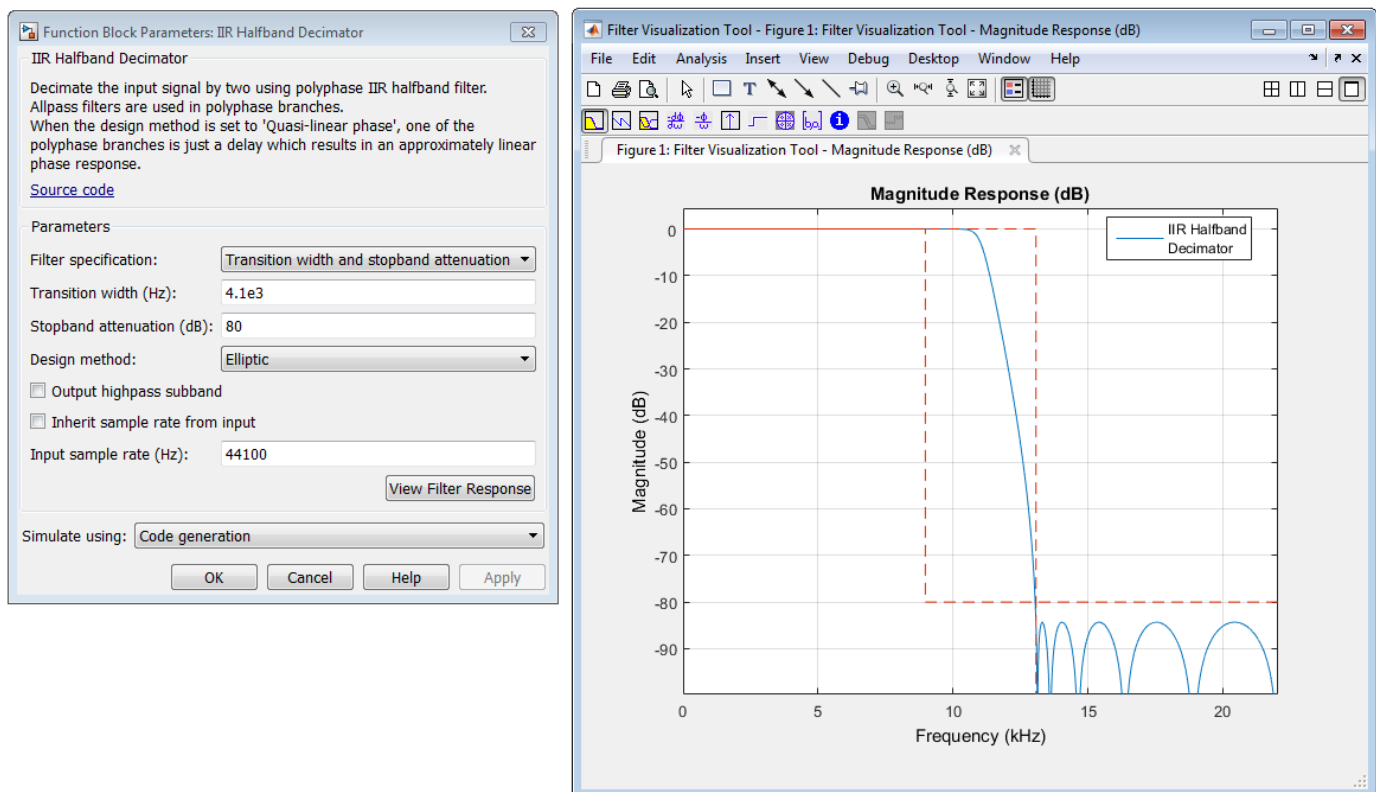
Specify the sample rate of the input signal as a scalar in Hz.

### Dependencies

To enable this parameter, clear the **Inherit sample rate from input** parameter.

**View Filter Response** — View Filter Response button

Click this button to open the Filter Visualization Tool (FVTool) and display the magnitude and phase response of the IIR Halfband Decimator. The response is based on the values you specify in the block parameters dialog box. Changes made to these parameters update FVTool.



To update the magnitude response while FVTool is running, modify the dialog box parameters and click **Apply**.

**Simulate using** — Simulate using

Code generation (default) | Interpreted execution

Specify the type of simulation to run. You can set this parameter to:

- Code generation (default)

Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than Interpreted execution.

- Interpreted execution

Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than Code generation.

## Block Characteristics

|                                 |                 |
|---------------------------------|-----------------|
| <b>Data Types</b>               | double   single |
| <b>Direct Feedthrough</b>       | no              |
| <b>Multidimensional Signals</b> | no              |
| <b>Variable-Size Signals</b>    | yes             |
| <b>Zero-Crossing Detection</b>  | no              |

## Algorithms

### Polyphase Implementation with Halfband Filters

When you filter your signal, the IIR halfband decimator uses an efficient polyphase implementation for halfband filters. You can use the polyphase implementation to move the downsample operation before filtering. This change enables you to filter at a lower sampling rate.

IIR halfband filters are generally modeled using two parallel allpass filter branches.

$$H(z) = 0.5 * [A_1(z^2) + z^{-1}A_2(z^2)]$$

### Elliptic Design

The allpass filters for elliptic IIR halfband filter are given as

$$A_1(z) = \prod_{k=1}^{K_1} \frac{a_k^{(1)} + z^{-1}}{1 + a_k^{(1)}z^{-1}}$$

$$A_2(z) = \prod_{k=1}^{K_2} \frac{a_k^{(2)} + z^{-1}}{1 + a_k^{(2)}z^{-1}}$$

### Quasi-Linear Phase Design

To achieve a near-linear phase response for IIR halfband filters, make one of the branches a pure delay. In this design, the cost of the filter increases.

The allpass filters for the quasi-linear phase IIR halfband filter are

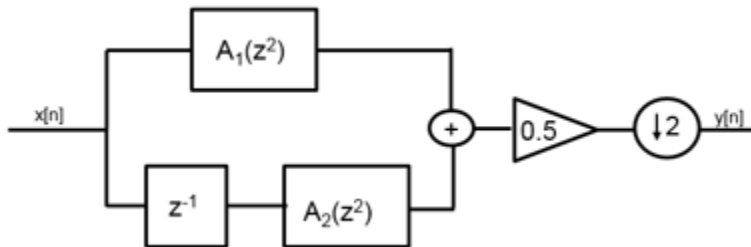
$$A_1(z) = z^{-k}$$

where  $k$  is the length of the delay.

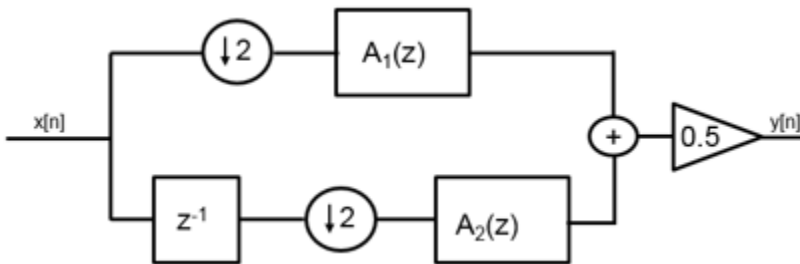
$$A_2(z) = \prod_{k=1}^{K_2^{(1)}} \frac{a_k + z^{-1}}{1 + a_k z^{-1}} \prod_{k=1}^{K_2^{(2)}} \frac{c_k + b_k z^{-1} + z^{-2}}{1 + b_k z^{-1} + c_k z^{-2}}$$

where  $N$  is the order of the IIR halfband filter.

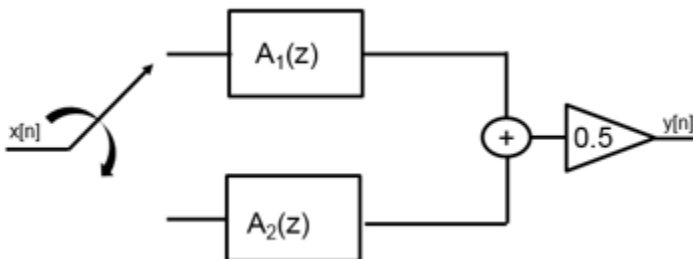
You can represent filtering the input signal and then downsampling it by 2 using this figure.



Using the multirate noble identity for downsampling, you can move the downsampling operation before the filtering operation. This change enables you to filter at a lower rate.



To implement the halfband decimator efficiently, this algorithm replaces the delay block and downsampling operator with a commutator switch.

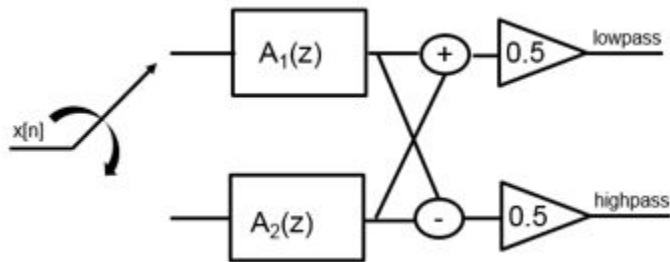


### Analysis Filter Bank

The transfer function of the complementary highpass filter branch of the analysis filter bank is given by

$$G(z) = 0.5 * [A_1(z^2) - z^{-1}A_2(z^2)]$$

You can represent the analysis filter bank as in this diagram.



The IIR halfband decimator generates two power-complementary output signals by adding and subtracting the two polyphase branch outputs respectively.

To summarize, the IIR halfband decimator:

- Decimates the input prior to filtering.
- Acts as an analysis filter bank.
- Has a nonlinear phase response and uses few coefficients with the elliptic design method.
- Has near-linear phase response at the cost of additional coefficients with the quasi-linear phase design method, where one of the branches is a pure delay

## Version History

Introduced in R2015b

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

## See Also

### Objects

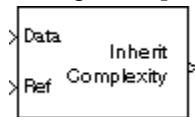
`dsp.IIRHalfbandInterpolator` | `dsp.IIRHalfbandDecimator`

### Blocks

IIR Halfband Interpolator | FIR Halfband Interpolator | FIR Halfband Decimator | CIC Compensation Decimator

## Inherit Complexity

Change complexity of input to match reference signal



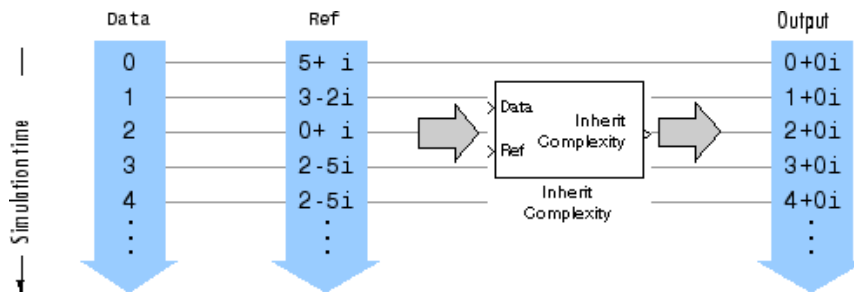
## Library

Signal Management / Signal Attributes

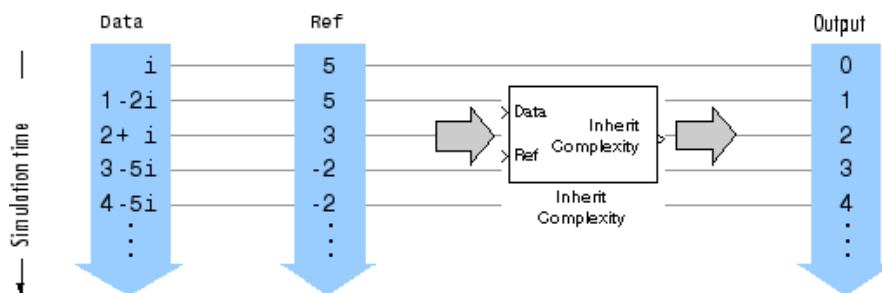
dpsigattribs

## Description

The Inherit Complexity block alters the input data at the Data port to match the complexity of the reference input at the Ref port. When the Data input is real, and the Ref input is complex, the block appends a zero-valued imaginary component,  $0i$ , to each element of the Data input.



When the Data input is complex, and the Ref input is real, the block outputs the real component of the Data input.



When both the Data input and Ref input are real, or when both the Data input and Ref input are complex, the block propagates the Data input with no change.



## Supported Data Types

| Port   | Supported Data Types   |
|--------|--|
| Data   | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed and unsigned)</li> <li>• Boolean</li> <li>• 8-, 16-, 32- and 64-bit signed integers</li> <li>• 8-, 16-, 32- and 64-bit unsigned integers</li> </ul> |
| Ref    | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed and unsigned)</li> <li>• Boolean</li> <li>• 8-, 16-, 32- and 64-bit signed integers</li> <li>• 8-, 16-, 32- and 64-bit unsigned integers</li> </ul> |
| Output | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed and unsigned)</li> <li>• Boolean</li> <li>• 8-, 16-, 32- and 64-bit signed integers</li> <li>• 8-, 16-, 32- and 64-bit unsigned integers</li> </ul> |

## See Also

|                            |                    |
|----------------------------|--------------------|
| Check Signal Attributes    | DSP System Toolbox |
| Complex to Magnitude-Angle | Simulink           |
| Complex to Real-Imag       | Simulink           |
| Magnitude-Angle to Complex | Simulink           |
| Real-Imag to Complex       | Simulink           |

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Generated code relies on the memcpy or memset function (`string.h`) under certain conditions.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

The Inherit Complexity block supports HDL code generation for these input types:

- Input complexities, such as real or complex, at the data and reference port
- Vector, matrix, and bus inputs

### HDL Architecture

This block has one default HDL architecture.

### HDL Block Properties

| General                          |  |
|----------------------------------|--|
| <b>ConstrainedOutputPipeline</b> | Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder). |
| <b>InputPipeline</b>             | Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).                 |
| <b>OutputPipeline</b>            | Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).               |

### Limitations

The block has these HDL code generation limitations:

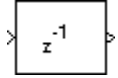
- Half-precision data type is not supported.
- Non-virtual bus input is not supported.

### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

## Integer Delay (Obsolete)

Delay input by integer number of sample periods



### Library

dspobslib

### Description

---

**Note** The Integer Delay block will be removed from the product in a future release. We strongly recommend replacing this block with the Delay block.

---

The Integer Delay block delays a discrete-time input by the number of sample intervals specified in the **Delay** parameter. Noninteger delay values are rounded to the nearest integer, and negative delays are clipped at 0.

#### Sample-Based Operation

When the input is a sample-based  $M$ -by- $N$  matrix, the block treats each of the  $M \times N$  matrix elements as an independent channel. The **Delay** parameter,  $v$ , can be an  $M$ -by- $N$  matrix of positive integers that specifies the number of sample intervals to delay each channel of the input, or a scalar integer by which to equally delay all channels.

For example, when the input is  $M$ -by-1 and  $v$  is the matrix  $[v(1) \ v(2) \ \dots \ v(M)]'$ , the first channel is delayed by  $v(1)$  sample intervals, the second channel is delayed by  $v(2)$  sample intervals, and so on. Note that when a channel is delayed for  $\Delta$  sample-time units, the output sample at time  $t$  is the input sample at time  $t - \Delta$ . When  $t - \Delta$  is negative, then the output is the corresponding value specified by the **Initial conditions** parameter.

A 1-D vector of length  $M$  is treated as an  $M$ -by-1 matrix, and the output is 1-D.

The **Initial conditions** parameter specifies the output of the block during the initial delay in each channel. The initial delay for a particular channel is the time elapsed from the start of the simulation until the first input in that channel is propagated to the output. Both fixed and time-varying initial conditions can be specified in a variety of ways to suit the dimensions of the input.

#### Fixed Initial Conditions

A fixed initial condition in sample-based mode can be specified as one of the following:

- Scalar value to be repeated at each sample time of the initial delay (for every channel). For a 2-by-2 input with the parameter settings below,

the block generates the following sequence of matrices at the start of the simulation,

$$\begin{bmatrix} -1 & -1 \\ -1 & -1 \end{bmatrix}, \begin{bmatrix} u_{11}^1 & -1 \\ -1 & -1 \end{bmatrix}, \begin{bmatrix} u_{11}^2 & u_{12}^1 \\ -1 & -1 \end{bmatrix}, \begin{bmatrix} u_{11}^3 & u_{12}^2 \\ u_{21}^1 & -1 \end{bmatrix}, \begin{bmatrix} u_{11}^4 & u_{12}^3 \\ u_{21}^2 & u_{22}^1 \end{bmatrix}, \dots$$

where  $u_{ij}^k$  is the  $ij$ th element of the  $k$ th matrix in the input sequence.

- Array of size  $M$ -by- $N$ -by- $d$ . In this case, you can set different fixed initial conditions for each element of a sample-based input. This setting is explained further in the Array bullet in “Time-Varying Initial Conditions” on page 2-820.

Initial conditions cannot be specified by full matrices.

### Time-Varying Initial Conditions

A time-varying initial condition in sample-based mode can be specified in one of the following ways:

- Vector of length  $d$ , where  $d$  is the maximum value specified for any channel in the **Delay** parameter. The vector can be a  $L$ -by- $d$ , 1-by- $d$ , or 1-by-1-by- $d$ . The  $d$  elements of the vector are output in sequence, one at each sample time of the initial delay.

For a scalar input and the parameters shown below,

the block outputs the sequence -1, -1, -1, 0, 1, ... at the start of the simulation.

- Array of dimension  $M$ -by- $N$ -by- $d$ , where  $d$  is the value specified for the **Delay** parameter (the maximum value when the **Delay** is a vector) and  $M$  and  $N$  are the number of rows and columns, respectively, in the input matrix. The  $d$  pages of the array are output in sequence, one at each sample time of the initial delay. For a 2-by-3 input, and the parameters below,

the block outputs the matrix sequence

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}, \begin{bmatrix} 2 & 4 & 6 \\ 1 & 3 & 5 \end{bmatrix}, \begin{bmatrix} 3 & 6 & 9 \\ 0 & 4 & 8 \end{bmatrix}$$

at the start of the simulation. Note that setting **Initial conditions** to an array with the same matrix for each entry implements constant initial conditions; a different constant initial condition for each input matrix element (channel).

Initial conditions cannot be specified by full matrices.

### Frame-Based Operation

When the input is a frame-based  $M$ -by- $N$  matrix, the block treats each of the  $N$  columns as an independent channel, and delays each channel as specified by the **Delay** parameter.

For frame-based inputs, the **Delay** parameter can be a scalar integer by which to equally delay all channels. It can also be a 1-by- $N$  row vector, each element of which serves as the delay for the corresponding channel of the  $N$ -channel input. Likewise, it can also be an  $M$ -by-1 column vector, each element of which serves as the delay for one of the corresponding  $M$  samples for each channel. The **Delay** parameter can be an  $M$ -by- $N$  matrix of positive integers as well; in this case, each element of each channel is delayed by the corresponding element in the delay matrix. For instance, if the fifth element of the third column of the delay matrix was 3, then the fifth element of the third channel of the input matrix is always delayed by three sample-time units.

When a channel is delayed for  $\Delta$  sample-time units, the output sample at time  $t$  is the input sample at time  $t - \Delta$ . When  $t - \Delta$  is negative, then the output is the corresponding value specified in the **Initial conditions** parameter.

The **Initial conditions** parameter specifies the output during the initial delay. Both fixed and time-varying initial conditions can be specified. The initial delay for a particular channel is the time elapsed from the start of the simulation until the first input in that channel is propagated to the output.

### Fixed Initial Conditions

The settings shown below specify fixed initial conditions. The value entered in the **Initial conditions** parameter is repeated at the output for each sample time of the initial delay. A fixed initial condition in frame-based mode can be one of the following:

- Scalar value to be repeated for all channels of the output at each sample time of the initial delay. For a general  $M$ -by- $N$  input with the parameter settings below,



The image shows a screenshot of a software interface with two input fields. The first field is labeled "Delay (samples):" and contains the number "5". The second field is labeled "Initial conditions:" and contains the number "0".

the first five samples in each of the  $N$  channels are zero. Notice that when the frame size is larger than the delay, all of these zeros are all included in the first output from the block.

- Array of size 1-by- $N$ -by- $D$ . In this case, you can also specify different fixed initial conditions for each channel. See “Time-Varying Initial Conditions” on page 2-821 for details.

Initial conditions cannot be specified by full matrices.

### Time-Varying Initial Conditions

The following settings specify time-varying initial conditions. For time-varying initial conditions, the values specified in the **Initial conditions** parameter are output in sequence during the initial delay. A time-varying initial condition in frame-based mode can be specified in the following ways:

- Vector of length  $D$ , where each of the  $N$  channels have the same initial conditions sequence specified in the vector.  $D$  is defined as follows:

- When an element of the delay entry is less than the frame size,

$$D = d + 1$$

where  $d$  is the maximum delay.

- When the all elements of the delay entry are greater than the input frame size,

$$D = d + \text{input frame size} - 1$$

Only the first  $d$  entries of the initial condition vector are used; the rest of the values are ignored, but you must include them nonetheless. For a two-channel ramp input  $[1:100; 1:100]'$  with a frame size of 4 and the parameter settings below,

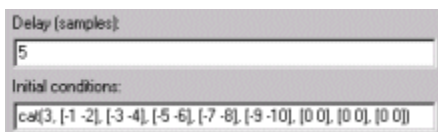


the block outputs the following sequence of frames at the start of the simulation.

$$\begin{bmatrix} -4 & -1 \\ -5 & -2 \\ 1 & -3 \\ 2 & -4 \end{bmatrix}, \begin{bmatrix} 3 & -5 \\ 4 & 1 \\ 5 & 2 \\ 6 & 3 \end{bmatrix}, \begin{bmatrix} 7 & 4 \\ 8 & 5 \\ 9 & 6 \\ 10 & 7 \end{bmatrix}, \dots$$

Note that since one of the delays, 2, is less than the frame size of the input, 4, the length of the **Initial conditions** vector is the sum of the maximum delay and 1 (5+1), which is 6. The first five entries of the initial conditions vector are used by the channel with the maximum delay, and the rest of the entries are ignored. Since the first channel is delayed for less than the maximum delay (2 sample time units), it only makes use of two of the initial condition entries.

- Array of size 1-by- $N$ -by- $D$ , where  $D$  is defined in “Time-Varying Initial Conditions” on page 2-821. In this case, the  $k$ th entry of each 1-by- $N$  entry in the array corresponds to an initial condition for the  $k$ th channel of the input matrix. Thus, a 1-by- $N$ -by- $D$  initial conditions input allows you to specify different initial conditions for each channel. For instance, for a two-channel ramp input  $[1:100; 1:100]'$  with a frame size of 4 and the parameter settings below,



the block outputs the following sequence of frames at the start of the simulation.

$$\begin{bmatrix} -1 & -2 \\ -3 & -4 \\ -5 & -6 \\ -7 & -8 \end{bmatrix}, \begin{bmatrix} -9 & -10 \\ 1 & 1 \\ 2 & 2 \\ 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 \\ 5 & 5 \\ 6 & 6 \\ 7 & 7 \end{bmatrix}, \dots$$

Note that the channels have distinct time varying initial conditions; the initial conditions for channel 1 correspond to the first entry of each length-2 row vector in the initial conditions array, and the initial conditions for channel 2 correspond to the second entry of each row vector in the

initial conditions array. Only the first five entries in the initial conditions array are used; the rest are ignored.

The 1-by- $N$ -by- $D$  array entry can also specify different fixed initial conditions for every channel; in this case, every 1-by- $N$  entry in the array would be identical, so that the initial conditions for each column are fixed over time.

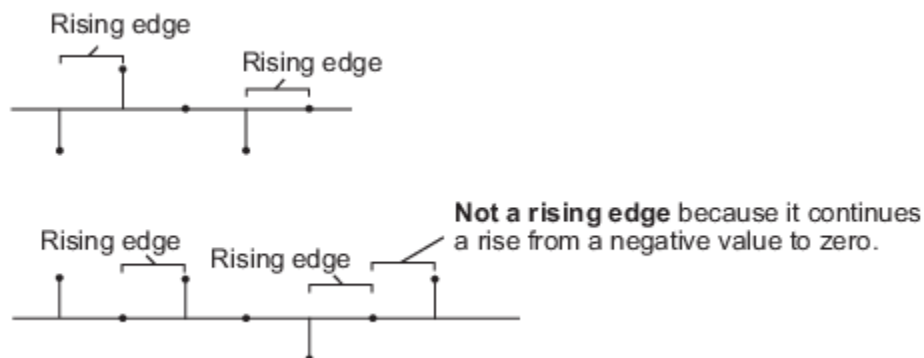
Initial conditions cannot be specified by full matrices.

### Resetting the Delay

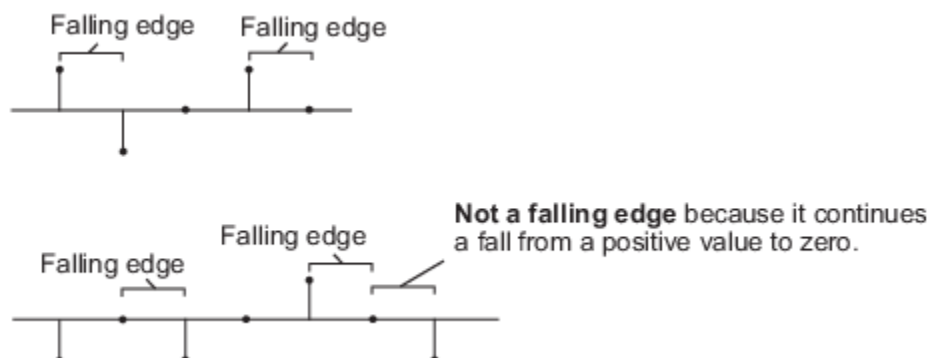
The block resets the delay whenever it detects a reset event at the optional Rst port. The reset sample time must be a positive integer multiple of the input sample time.

You specify the reset event in the **Reset port** parameter:

- None disables the Rst port.
- Rising edge — Triggers a reset operation when the Rst input does one of the following:
  - Rises from a negative value to a positive value or zero
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure)



- Falling edge — Triggers a reset operation when the Rst input does one of the following:
  - Falls from a positive value to a negative value or zero
  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero (see the following figure)



- **Either edge** — Triggers a reset operation when the Rst input is a **Rising edge** or **Falling edge** (as described above).
- **Non-zero sample** — Triggers a reset operation at each sample time that the Rst input is not zero.

---

**Note** When running simulations in the Simulink **MultiTasking** mode, sample-based reset signals have a one-sample latency, and frame-based reset signals have one frame of latency. Thus, there is a one-sample or one-frame delay between the time the block detects a reset event, and when it applies the reset. For more information on latency and the Simulink tasking modes, see “Excess Algorithmic Delay (Tasking Latency)” and “Time-Based Scheduling and Code Generation” (Simulink Coder).

---

## Parameters

### Delay

The number of sample periods to delay the input signal.

### Initial conditions

The value of the block's output during the initial delay.

### Reset port

Determines the reset event that causes the block to reset the delay. For more information, see “Resetting the Delay” on page 2-823.

## Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point (signed only)
- Boolean — The block accepts Boolean inputs to the Rst port, which is enabled by the **Reset port** parameter.
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

## See Also

Unit Delay

Variable Fractional Delay

Variable Integer Delay

Simulink

DSP System Toolbox

Simulink

## Version History

**Introduced in R2008b**



# Interpolation

Interpolate values of real input samples



## Libraries:

DSP System Toolbox / Signal Operations

## Description

The Interpolation block interpolates discrete, real inputs by using linear or FIR interpolation. The block accepts a vector, a matrix, or an  $N$ -D array. The block outputs a scalar, a vector, a matrix, or an  $N$ -D array of the interpolated values.

You must specify the interpolation points, the times at which to interpolate values in a one-based interpolation array  $I_{pts}$ . An entry of 1 in  $I_{pts}$  refers to the first sample of the input data, an entry of 2.5 refers to the sample halfway between the second and third input sample, and so on. Depending on the dimensions of the input data,  $I_{pts}$  can be a scalar, a length- $P$  row, or a column vector, a  $P$ -by- $N$  matrix, or an  $N$ -D array where  $P$  is the size of the first dimension of the  $N$ -D array. In most cases,  $P$  can be any positive integer. For more information about valid interpolation arrays, refer to the tables in “Pts” on page 2-0 and “Interpolation points” on page 2-0.

In most cases, the block applies  $I_{pts}$  across the first dimension of an  $N$ -D input array or to each input vector. You can set the block to apply the same interpolation array for all input data, static interpolation points entered on the block mask, or to use a different interpolation array for each  $N$ -D array, matrix, or vector input, time-varying interpolation points received through the Pts input port.

## Ports

### Input

**In** — Input signal

scalar | vector | matrix |  $N$ -D array

Input signal, specified as a scalar, vector, matrix, or  $N$ -D array.

This port is unnamed until **Source of interpolation points** is set to **Input port**.

Data Types: single | double

**Pts** — Interpolation points

scalar | vector | matrix |  $N$ -D array

Interpolation points, specified as a scalar, vector, matrix, or  $N$ -D array.

For more details on the block's behavior when the **Source of interpolation points** parameter is set to **Input port**, see “How the Block Applies Interpolation Arrays to Inputs” on page 2-829.

### Dependencies

This port is unavailable until **Source of interpolation points** is set to **Input port**.

Data Types: `single` | `double`

## Output

**Out** — Output signal  
 scalar | vector | matrix | *N*-D array

Output signal, specified as a vector, matrix, or *N*-D array.

This port is unnamed until `Source of interpolation points` is set to `Input port`.

Data Types: `single` | `double`

## Parameters

**Source of interpolation points** — Interpolation point input method

`Specify via dialog` (default) | `Input port`

Choose how you want to specify the interpolation points:

- `Specify via dialog` -- The `Interpolation points` parameter becomes available. Use this option for static interpolation points.
- `Input port` -- The `Pts` port appears on the block. The block uses the input to this port as the interpolation points. Use this option for time-varying interpolation points.

For more information, see “Specifying Static Interpolation Points” on page 2-828 and “Specifying Time-Varying Interpolation Points” on page 2-828.

**Interpolation points** — Interpolation points

`[1.1 4.8 2.67 1.6 3.2]'` (default) | scalar | vector | matrix | *N*-D array

The array of points in time at which to interpolate the input signal ( $I_{pts}$ ). An entry of 1 in  $I_{pts}$  refers to the first sample of the input, an entry of 2.5 refers to the sample halfway between the second and third input sample, and so on. See “How the Block Applies Interpolation Arrays to Inputs” on page 2-829.

**Tunable:** Yes

## Dependencies

This parameter becomes available only when the `Source of interpolation points` is set to `Specify via dialog`. For more information, see “Specifying Static Interpolation Points” on page 2-828.

**Interpolation mode** — Interpolation mode

`Linear` (default) | `FIR`

Sets the block to interpolate by either `Linear` or `FIR` interpolation. For more information, see “Linear Interpolation Mode” on page 2-831 and “FIR Interpolation Mode” on page 2-832.

**Interpolation filter half-length** — Half length of FIR interpolation filter

3 (default) | positive integer

Specify the half-length of the FIR interpolation filter ( $P$ ). To perform the interpolation in FIR mode, the block uses the nearest  $2 \times P$  low-rate samples. In most cases,  $P$  low-rate samples must appear below and above each interpolation point. If you interpolate at a low-rate sample point, the block includes that low-rate sample in the required  $2 \times P$  samples and requires only  $2 \times P - 1$  neighboring low-rate samples. If an interpolation point does not have the required number of neighboring low-rate samples, the block interpolates that point using linear interpolation.

#### Dependencies

This parameter becomes available only when the `Interpolation` mode is set to FIR. For more information, see “FIR Interpolation Mode” on page 2-832.

#### Interpolation points per input sample — Interpolation points per sample

3 (default)

Number of points per input sample, upsampling factor ( $L$ ), at which the block computes a unique FIR interpolation filter. To perform the FIR interpolation, the block uses a polyphase structure that has  $L$  filter arms of length  $2 \times P$ .

For example, if  $L=4$ , the block constructs a polyphase filter with four arms. The block then interpolates at points corresponding to  $1 + i/L$ ,  $2 + i/L$ ,  $3 + i/L$ ..., where the integers 1, 2, and 3 represent the low-rate samples, and  $i=0, 1, 2, 3$ . To interpolate at a point that does not directly correspond to an arm of the polyphase filter requires an extra computation. The block first rounds that point down to the nearest value that does correspond to an arm of the polyphase filter. To interpolate at the point 2.2, the block rounds 2.2 down to 2, and computes the FIR interpolation by using the first arm of the polyphase filter structure. Similarly, to interpolate the point 2.65, the block rounds the value down to 2.5 and uses the third arm of the polyphase filter structure.

#### Dependencies

This parameter becomes available only when the `Interpolation` mode is set to FIR. For more information, see “FIR Interpolation Mode” on page 2-832.

#### Normalized input bandwidth — Normalized input bandwidth

0.5 (default) | scalar in (0 1]

The bandwidth of the input divided by  $F_s/2$  (half the input sample frequency), specified as a scalar in the range (0 1].

#### Dependencies

This parameter becomes available only when the `Interpolation` mode is set to FIR. For more information, see “FIR Interpolation Mode” on page 2-832.

#### Out of range interpolation points — Error handling for out-of-range points

Clip (default) | Clip and warn | Error

Valid values in the interpolation array  $I_{pts}$  range from 1 to the number of samples in each channel of the input. For instance, given a length-5 input vector  $D$ , all entries of  $I_{pts}$  must range from 1 to 5.  $I_{pts}$  cannot contain entries such as 7 or -9 because there is no 7th or -9th entry in  $D$ .

The **Out of range interpolation points** parameter sets how the block handles interpolation points that fall outside the valid range. The parameter has these settings:

- **Clip** — The block replaces any out-of-range values in  $I_{pts}$  with the closest value in the valid range (from 1 to the number of input samples), and then proceeds with computations by using the clipped version of  $I_{pts}$ .
- **Clip and warn** — In addition to **Clip**, the block issues a warning at the MATLAB command line every time clipping occurs.
- **Error** — When the block encounters an out-of-range value in  $I_{pts}$ , the simulation stops, and the block issues an error at the MATLAB command line.

As an example of clipping, suppose the block is set to clip out-of-range interpolation points and gets these input vector and interpolation points:

- $D = [11 \ 22 \ 33 \ 44]'$
- $I_{pts} = [10 \ 2.6 \ -3]'$

Because  $D$  has four samples, valid interpolation points range from 1 to 4. The block clips the interpolation point 10 to 4 and the point -3 to 1, resulting in the clipped interpolation vector  $I_{ptsClipped} = [4 \ 2.6 \ 1]'$ .

## Block Characteristics

|                                 |                 |
|---------------------------------|-----------------|
| <b>Data Types</b>               | double   single |
| <b>Direct Feedthrough</b>       | no              |
| <b>Multidimensional Signals</b> | no              |
| <b>Variable-Size Signals</b>    | no              |
| <b>Zero-Crossing Detection</b>  | no              |

## More About

### Specifying Static Interpolation Points

To supply the block with a static interpolation array, that is an interpolation array applied to every vector or  $N$ -D array of input data), perform the following steps:

- 1 Set the **Source of interpolation points** parameter to **Specify via dialog**.
- 2 Enter the interpolation array in the **Interpolation points** parameter. To learn about interpolation arrays, see “How the Block Applies Interpolation Arrays to Inputs” on page 2-829.

### Specifying Time-Varying Interpolation Points

To supply the block with time-varying interpolation arrays, where the block uses a different interpolation array for each vector or  $N$ -D array input, perform the following steps:

- 1 Set the **Source of interpolation points** parameter to **Input port**. The **Pts** port appears on the block.

- 2 Generate a signal of interpolation arrays and supply it to the **Pts** port. The block uses the input to this port as the interpolation points. To learn about interpolation arrays, see “How the Block Applies Interpolation Arrays to Inputs” on page 2-829.

### How the Block Applies Interpolation Arrays to Inputs

The interpolation array  $I_{Pts}$  represents the points in time at which to interpolate values of the input signal. An entry of 1 in  $I_{Pts}$  refers to the first sample of the input, an entry of 2.5 refers to the sample halfway between the second and third input sample, and so on. In most cases, when  $I_{Pts}$  is a vector, it can be of any length.

Valid values in the interpolation array,  $I_{Pts}$ , range from 1 to the number of samples in each channel of the input. To learn how the block handles out-of-range interpolation values, see **Out of range interpolation points**.

Depending on the dimension of the input and the dimension of  $I_{Pts}$ , the block usually applies  $I_{Pts}$  to the input in one of the following ways:

- Applies the  $I_{Pts}$  array across the first dimension of an  $N$ -D array, resulting in an  $N$ -D array output.
- Applies the vector  $I_{Pts}$  to each input vector (as if the input vector were a single channel), resulting in a vector output with the same orientation as the input (row or column).

These tables summarize how the block applies the interpolation array  $I_{Pts}$  to all the possible types of inputs and shows the resulting output dimensions.

The first table describes the block's behavior when the **Source of interpolation points is Specify via dialog**.

### Specify Interpolation Points Through Dialog Box

| Input Dimensions             | Valid Dimensions of Interpolation Array / $I_{Pts}$ | How Block Applies $I_{Pts}$ to Input  | Output Dimensions (Frame-Based) |
|------------------------------|---|---|---------------------------------|
| $M$ -by- $N$ -by- $K$ matrix | $P$ -by-1 column                                    | Applies $I_{Pts}$ to the first dimension of the input   | $P$ -by- $N$ -by- $K$ array     |
|                              | $P$ -by- $N$ -by- $K$ matrix                        | Applies each column of $I_{Pts}$ (each element of $I_{Pts}$ ) to the corresponding column of the input matrix | $P$ -by- $N$ -by- $K$ array     |
| $M$ -by- $N$ matrix          | 1-by- $N$ row                                       | Applies each column of $I_{Pts}$ (each element of $I_{Pts}$ ) to the corresponding column of the input matrix | 1-by- $N$ row                   |
|                              | $P$ -by-1 column                                    | Applies $I_{Pts}$ to each input column  | $P$ -by- $N$ matrix             |
|                              | $P$ -by- $N$ matrix                                 | Applies the columns of $I_{Pts}$ to the corresponding columns of the input matrix                             |                                 |

| Input Dimensions                   | Valid Dimensions of Interpolation Array / Pts | How Block Applies / Pts to Input          | Output Dimensions (Frame-Based)                                  |
|------------------------------------|---|---|--|
| $M$ -by-1 column                   | $P$ -by-1 column                              | Applies $I_{Pts}$ to the input column     | $P$ -by-1 column   |
| 1-by- $N$ row<br>(not recommended) | 1-by- $N$ row                                 | Not Applicable. Block copies input vector | 1-by- $N$ row, a copy of the input vector                        |
|                                    | $P$ -by-1 column                              |   | $P$ -by- $N$ matrix where each row is a copy of the input vector |
|                                    | $P$ -by- $N$ matrix                           |   |  |

The next table describes the block's behavior when the **Source of interpolation points** is Input port.

**Specify Interpolation Points Through Input Port**

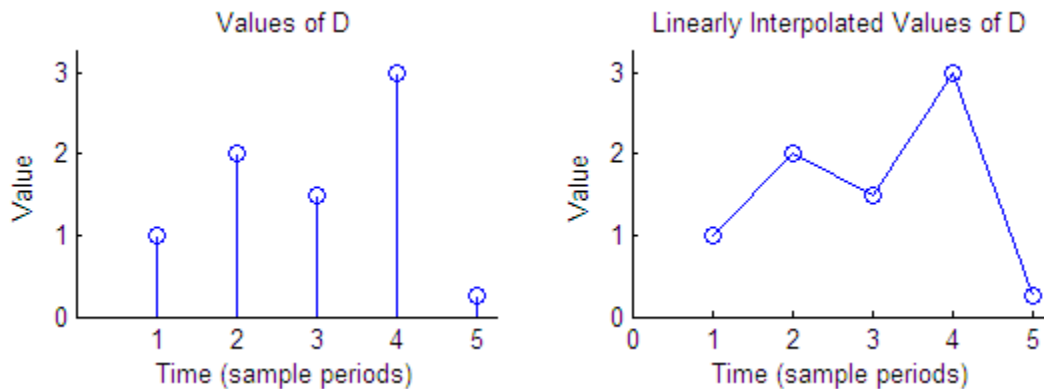
| Input Dimensions                   | Valid Dimensions of Interpolation Array / Pts    | How Block Applies / Pts to Input  | Output Dimensions (Frame-Based)                                  |
|------------------------------------|--|---|--|
| $M$ -by- $N$ -by- $K$ matrix       | Unoriented vector or column vector of length $P$ | Applies $I_{Pts}$ to the first dimension of the input   | $P$ -by- $N$ -by- $K$ array                                      |
|                                    | $P$ -by- $N$ -by- $K$ matrix                     | Applies each column of $I_{Pts}$ (each element of $I_{Pts}$ ) to the corresponding column of the input matrix | $P$ -by- $N$ -by- $K$ array                                      |
| $M$ -by- $N$ matrix                | 1-by- $N$ row                                    | Applies each column of $I_{Pts}$ (each element of $I_{Pts}$ ) to the corresponding column of the input matrix | 1-by- $N$ row  |
|                                    | $P$ -by-1 column                                 | Applies $I_{Pts}$ to each input column  | $P$ -by- $N$ matrix  |
|                                    | $P$ -by- $N$ matrix                              | Applies the columns of $I_{Pts}$ to the corresponding columns of the input matrix                             |  |
| $M$ -by-1 column                   | $P$ -by-1 column                                 | Applies $I_{Pts}$ to the input column   | $P$ -by-1 column   |
| 1-by- $N$ row<br>(not recommended) | 1-by- $N$ row                                    | Not Applicable. Block copies input vector   | 1-by- $N$ row, a copy of the input vector                        |
|                                    | $P$ -by-1 column                                 |   | $P$ -by- $N$ matrix where each row is a copy of the input vector |
|                                    | $P$ -by- $N$ matrix                              |   |  |

## Algorithms

### Linear Interpolation Mode

When Interpolation mode is set to Linear, the block interpolates data values by assuming that the data varies linearly between samples taken at adjacent sample times.

For instance, if the input signal  $D = [1 \ 2 \ 1.5 \ 3 \ 0.25]'$ , the plot on the left shows the samples in  $D$  and the plot on the right shows the linearly interpolated values between the samples in  $D$ .

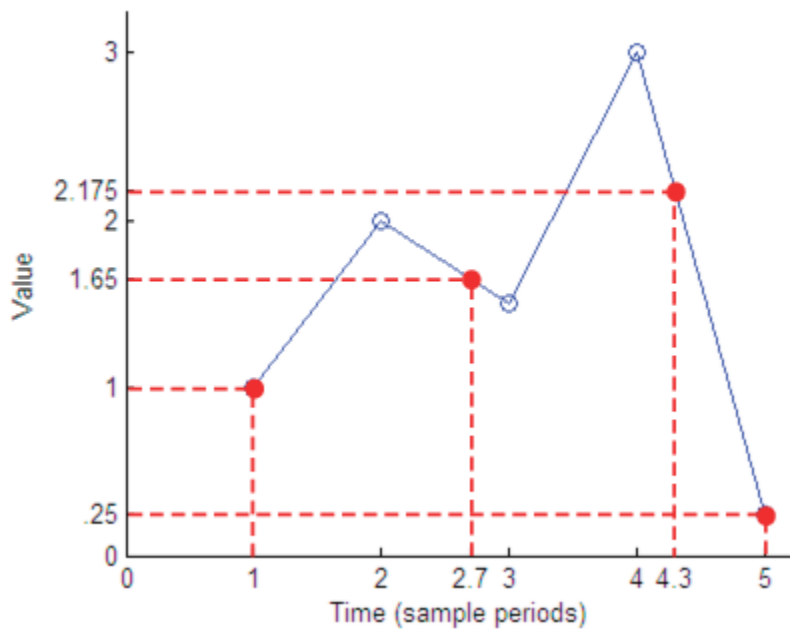


The following figure illustrates the case of a block in linear interpolation mode that is set to clip out-of-range interpolation points. The vector  $D$  supplies the input data and the vector  $I_{pts}$  supplies the interpolation points:

- $D = [1 \ 2 \ 1.5 \ 3 \ 0.25]'$
- $I_{pts} = [-4 \ 2.7 \ 4.3 \ 10]'$

The block clips the invalid interpolation points and outputs the linearly interpolated values in a vector,  $[1 \ 1.65 \ 2.175 \ 0.25]'$ .

Interpolated Values of D at Clipped Interpolation Points



$D = [1 \ 2 \ 1.5 \ 3 \ 0.25]'$   
 $lpts = [-4 \ 2.7 \ 4.3 \ 10]'$

Valid interpolation points range from 1 to 5, so -4 is clipped to 1 and 10 is clipped to 5.

Clipped  $lpts = [1 \ 2.7 \ 4.3 \ 5]'$

Output =  $[1 \ 1.65 \ 2.175 \ 0.25]'$

### FIR Interpolation Mode

When `Interpolation` mode is set to FIR, the block interpolates data values by using an FIR interpolation filter, specified by various block parameters. For more information, see “FIR Interpolation Mode” on page 2-1529 in the Variable Fractional Delay block reference.

## Version History

Introduced before R2006a

### Extended Capabilities

#### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### See Also

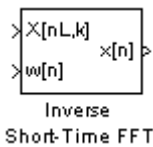
#### Blocks

FIR Halfband Interpolator | CIC Compensation Interpolator | IIR Halfband Interpolator | CIC Interpolation | FIR Interpolation



# Inverse Short-Time FFT

Recover time-domain signals by performing inverse short-time fast Fourier transform (FFT)



## Libraries:

DSP System Toolbox / Transforms / Matrix Operations

## Description

The Inverse Short-Time FFT block reconstructs the time-domain signal from the frequency-domain output of the Short-Time FFT block using a two-step process. First, the block implements the overlap-add algorithm.

$$x[n] = \frac{L}{W(0)} \sum_{p=-\infty}^{\infty} \left[ \frac{1}{N} \sum_{k=0}^{N-1} X[pL, k] e^{j2\pi kn/N} \right]$$

Then, the block rebuffers the signal in order to reconstruct the time-domain signal. Depending on the analysis window used by the Short-Time FFT block, the Inverse Short-Time FFT block might or might not achieve perfect reconstruction of the time-domain signal.

## Ports

### Input

**X[nL,k]** — Frequency-domain input signal  
unoriented vector | column vector | matrix

Specify the frequency-domain input signal as a vector or a matrix. The FFT length is equal to the number of rows in the input at the **x[nL,k]** port. Hence the number of input rows must be greater than or equal to the window length.

Data Types: single | double

**w[n]** — Analysis window  
vector

Specify the analysis window at the **w[n]** port.

Data Types: single | double

### Output

**x[n]** — Inverse short-time FFT output  
vector | matrix

The block outputs the inverse short-time FFT output at the port **x[n]**.

Data Types: single | double

## Parameters

**Analysis window length** — Length of analysis window

512 (default) | positive integer greater than 1

Enter the length of the analysis window as a positive integer greater than 1.

**Overlap between consecutive STFFT frames (in samples)** — Overlap length

384 (default) | nonnegative integer

Enter the number of samples of overlap for each frame of the input signal as a nonnegative integer. This value should be the same as the **Overlap between consecutive windows (in samples)** parameter in the dialog box of the Short-Time FFT block.

**Samples per output frame** — Number of samples per output frame

512 (default) | positive integer

Enter the desired frame size of the output signal as a positive integer.

**Input is conjugate symmetric** — Input to the block is conjugate symmetric

on (default) | off

Select this check box when the input to the block is both floating point and conjugate symmetric and you want real-valued outputs. If you select this check box when the input is not conjugate symmetric, the output of the block is invalid. This parameter cannot be used for fixed-point signals.

**Assert if analysis window does not support perfect signal reconstruction** — Error if analysis window does not support perfect signal reconstruction

off (default) | on

Select this check box to display an error when the analysis window used by the Short-Time FFT block does not support perfect signal reconstruction.

A time-domain signal input to the Short-time FFT block cannot be perfectly reconstructed.

To determine if the signal can be perfectly reconstructed, the Inverse Short-time FFT block uses the values you enter in the **Analysis window length** and **Reconstruction error tolerance** parameters.

**Reconstruction error tolerance** — Reconstruction error tolerance of original signal

1e-4 (default) | real scalar

Enter the amount of acceptable error in the reconstruction of the original signal as a real scalar.

### Dependency

To enable this parameter, select the **Assert if analysis window does not support perfect signal reconstruction** check box.

## Block Characteristics

|                                 |                 |
|---------------------------------|-----------------|
| <b>Data Types</b>               | double   single |
| <b>Direct Feedthrough</b>       | no              |
| <b>Multidimensional Signals</b> | no              |
| <b>Variable-Size Signals</b>    | no              |
| <b>Zero-Crossing Detection</b>  | no              |

## Version History

Introduced before R2006a

## References

[1] Quatieri, Thomas E. *Discrete-Time Speech Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 2001.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Usage notes and limitations:

- Generated code relies on the `memcpy` or `memset` function (`string.h`) under certain conditions.
- When the input length is not a power of 2, the executable generated from this block relies on prebuilt dynamic library files (`.dll` files) included with MATLAB. Use the `packNGo` function to package the code generated from this block and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see “How To Run a Generated Executable Outside MATLAB”.

When the input length is a power of 2, you can generate standalone C and C++ code from this block.

## See Also

### Objects

`dsp.ISTFT` | `dsp.SpectrumEstimator`

### Blocks

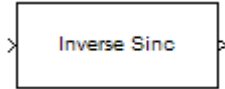
Short-Time FFT | Spectrum Estimator | Burg Method | Magnitude FFT | Periodogram | Spectrum Analyzer | Window Function | Yule-Walker Method

### Topics

“Spectral Analysis”

# Inverse Sinc Filter

Design inverse sinc filter



## Library

Filtering / Filter Designs

dspfdesign

## Description

This block brings the filter design capabilities of the `filterBuilder` function to the Simulink environment.

This block supports SIMD code generation. For details, see “Code Generation” on page 2-841.

## Dialog Box

See “Inverse Sinc Filter Design — Main Pane” on page 5-626 for more information about the parameters of this block. The **Data Types** and **Code Generation** panes are not available for blocks in the DSP System Toolbox Filter Designs library.

### View filter response

This button opens the Filter Visualization Tool (**FVTool**) from the Signal Processing Toolbox product. You can use the tool to display:

- Magnitude response, phase response, and group delay in the frequency domain.
- Impulse response and step response in the time domain.
- Pole-zero information.

The tool also helps you evaluate filter performance by providing information about filter order, stability, and phase linearity. For more information on FVTool, see the Signal Processing Toolbox documentation.

### Filter Specifications

In this group, you specify your filter format, such as the impulse response and the filter order.

#### Order mode

Select either **Minimum** (the default) or **Specify** from the drop-down list. Selecting **Specify** enables the **Order** option (see the following sections) so you can enter the filter order.

#### Response type

Select **Lowpass** or **Highpass** to design an inverse sinc lowpass or highpass filter.

**Filter type**

Select **Single-rate**, **Decimator**, **Interpolator**, or **Sample-rate converter**. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, the block specifies a single-rate filter.

- Selecting **Decimator** or **Interpolator** activates the **Decimation Factor** or the **Interpolation Factor** options respectively.
- Selecting **Sample-rate converter** activates both factors.

**Order**

Enter the filter order. This option is enabled only if you set the **Order mode** to **Specify**.

**Decimation Factor**

Enter the decimation factor. This option is enabled only if the **Filter type** is set to **Decimator** or **Sample-rate converter**. The default value is 2.

**Interpolation Factor**

Enter the interpolation factor. This option is enabled only if the **Filter type** is set to **Interpolator** or **Sample-rate converter**. The default value is 2.

**Frequency Specifications**

The parameters in this group allow you to specify your filter response curve.

Regions between specification values such as **Passband frequency** and **Stopband frequency** represent transition regions where the filter response is not constrained.

**Frequency constraints**

When **Order mode** is **Specify**, select the filter features that the block uses to define the frequency response characteristics. The list contains the following options, when available for the filter specifications.

- **Passband and stopband frequencies** — Define the filter by specifying the frequencies for the edges for the stop- and passbands.
- **Passband frequency** — For IIR filters, define the filter by specifying frequencies for the edges of the passband.
- **Stopband frequency** — For IIR filters, define the filter by specifying frequencies for the edges of the stopbands.
- **Cutoff (6dB) frequency** — For FIR filters, define the filter response by specifying the locations of the 6 dB point. The 6 dB point is the frequency for the point six decibels below the passband value.

**Frequency units**

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select **Normalized (0 to 1)** to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—**Hz**, **kHz**, **MHz**, or **GHz**. Selecting one of the unit options enables the **Input sample rate** parameter.

**Input sample rate**

$F_s$ , specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the

specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

### **Passband frequency**

Enter the frequency at the end of the passband. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

### **Stopband frequency**

Enter the frequency at the start of the stopband. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

### **Cutoff (6dB) frequency**

When **Frequency constraints** is Cutoff (6dB) frequency, specify the frequency of the 6 dB point. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

## **Magnitude Specifications**

Parameters in this group specify the filter response in the passbands and stopbands.

### **Magnitude units**

Specify the units for any parameter you provide in magnitude specifications. From the drop-down list, select one of the following options:

- Linear — Specify the magnitude in linear units.
- dB — Specify the magnitude in decibels (default)
- Squared — Specify the magnitude in squared units.

### **Passband ripple**

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels.

### **Stopband attenuation**

Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels.

## **Algorithm**

The parameters in this group allow you to specify the design method and structure of your filter.

### **Design Method**

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default FIR method is **Equiripple**.

### **Design Options**

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

#### **Density factor**

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 20 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

### Phase constraint

Specify the phase constraint of the filter as **Linear**, **Maximum**, or **Minimum**.

### Minimum order

When you select this parameter, the design method determines and design the minimum order filter to meet your specifications. Some filters do not provide this parameter. Select **Any**, **Even**, or **Odd** from the drop-down list to direct the design to be any minimum order, or minimum even order, or minimum odd order.

### Stopband Shape

Stopband shape lets you specify how the stopband changes with increasing frequency. Choose one of the following options;

- **Flat** — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.
- **Linear** — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.
- **1/f** — Specifies that the stopband attenuation changes exponentially as the frequency increases, where  $f$  is the frequency. Set the power (exponent) for the decay in **Stopband decay**.

### Stopband Decay

When you set **Stopband shape**, **Stopband decay** specifies the amount of decay applied to the stopband. the following conditions apply to **Stopband decay** based on the value of **Stopband Shape**:

- When you set **Stopband shape** to **Flat**, **Stopband decay** has no affect on the stopband.
- When you set **Stopband shape** to **Linear**, enter the slope of the stopband in units of dB/rad/s. The block applies that slope to the stopband.
- When you set **Stopband shape** to **1/f**, enter a value for the exponent  $n$  in the relation  $(1/f)^n$  to define the stopband decay. The block applies the  $(1/f)^n$  relation to the stopband to result in an exponentially decreasing stopband attenuation.

### Sinc frequency factor

A frequency dilation factor. The **Sinc frequency factor**,  $C$ , parameterizes the passband magnitude response for a lowpass design through  $H(\omega) = \text{sinc}(C\omega)^{-P}$  and through  $H(\omega) = \text{sinc}(C(1-\omega))^{-P}$  for a highpass design.

### Sinc power

Negative power of passband magnitude response. The **Sinc power**,  $P$ , parameterizes the passband magnitude response for a lowpass design through  $H(\omega) = \text{sinc}(C\omega)^{-P}$  and through  $H(\omega) = \text{sinc}(C(1-\omega))^{-P}$  for a highpass design.

## Filter Implementation

### Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure, and IIR filters use direct-form II filters with SOS.

### Use basic elements to enable filter customization

Select this check box to implement the filter as a subsystem of basic Simulink blocks. Clear the check box to implement the filter as a high-level subsystem. By default, this check box is cleared.

The high-level implementation provides better compatibility across various filter structures, especially filters that would contain algebraic loops when constructed using basic elements. On the other hand, using basic elements enables the following optimization parameters:

- **Optimize for zero gains** — Terminate chains that contain Gain blocks with a gain of zero.
- **Optimize for unit gains** — Remove Gain blocks that scale by a factor of one.
- **Optimize for delay chains** — Substitute delay chains made up of  $n$  unit delays with a single delay by  $n$ .
- **Optimize for negative gains** — Use subtraction in Sum blocks instead of negative gains in Gain blocks.

### Input processing

Specify how the block should process the input. The available options may vary depending on the settings of the **Filter Structure** and **Use basic elements for filter customization** parameters. You can set this parameter to one of the following options:

- **Columns as channels (frame based)** — When you select this option, the block treats each column of the input as a separate channel.
- **Elements as channels (sample based)** — When you select this option, the block treats each element of the input as a separate channel.

### Rate options

When the **Filter type** parameter specifies a multirate filter, select the rate processing rule for the block from following options:

- **Enforce single-rate processing** — When you select this option, the block maintains the sample rate of the input.
- **Allow multirate processing** — When you select this option, the block adjusts the rate at the output to accommodate an increased or reduced number of samples. To select this option, you must set the **Input processing** parameter to **Elements as channels (sample based)**.

### Use variable names for coefficients

Select this check box to enable the specification of coefficients using MATLAB variables. The available coefficient names differ depending on the filter structure. Using symbolic names allows tuning of filter coefficients in generated code. By default, this check box is cleared.



## Supported Data Types

| Port   | Supported Data Types   |
|--------|--|
| Input  | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> </ul> |
| Output | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> </ul> |

## Version History

Introduced in R2006b

### Extended Capabilities

#### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

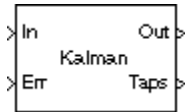
The Inverse Sinc Filter block supports SIMD code generation using Intel AVX2 technology under these conditions:

- **Filter type** is set to Single-rate, Decimator, or Interpolator.
- For **Filter type** that is set to Single-rate, **Structure** is set to Direct-form FIR or Direct-form FIR transposed.
- For **Filter type** that is set to Decimator, **Structure** is set to Direct-form FIR polyphase decimator and **Rate options** is set to Enforce single-rate processing.
- For **Filter type** that is set to Interpolator, **Rate options** is set to Enforce single-rate processing.
- **Input processing** is set to Columns as channels (frame based).
- Input signal has a data type of single or double.

The SIMD technology significantly improves the performance of the generated code.

## Kalman Adaptive Filter (Obsolete)

Compute filter estimates for inputs using Kalman adaptive filter algorithm



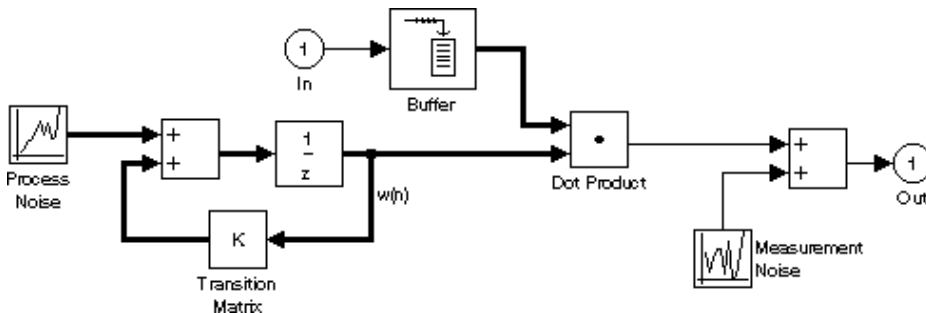
### Library

dspobslib

### Description

**Note** The Kalman Adaptive Filter block is still supported but is likely to be obsoleted in a future release. We strongly recommend replacing this block with the Kalman Filter block.

The Kalman Adaptive Filter block computes the optimal linear minimum mean-square estimate (MMSE) of the FIR filter coefficients using a one-step predictor algorithm. This Kalman filter algorithm is based on the following physical realization of a dynamic system.



The Kalman filter assumes that there are no deterministic changes to the filter taps over time (that is, the transition matrix is identity), and that the only observable output from the system is the filter output with additive noise. The corresponding Kalman filter is expressed in matrix form as

$$g(n) = \frac{K(n-1)u(n)}{u^H(n)K(n-1)u(n) + Q_M}$$

$$y(n) = u^H(n)\hat{w}(n)$$

$$e(n) = d(n) - y(n)$$

$$\hat{w}(n+1) = \hat{w}(n) + e(n)g(n)$$

$$K(n) = K(n-1) - g(n)u^H(n)K(n-1) + Q_p$$

The variables are as follows

| Variable         | Description  |
|------------------|--|
| $n$              | The current algorithm iteration                      |
| $u(n)$           | The buffered input samples at step $n$               |
| $K(n)$           | The correlation matrix of the state estimation error |
| $g(n)$           | The vector of Kalman gains at step $n$               |
| $\widehat{w}(n)$ | The vector of filter-tap estimates at step $n$       |
| $y(n)$           | The filtered output at step $n$                      |
| $e(n)$           | The estimation error at step $n$                     |
| $d(n)$           | The desired response at step $n$                     |
| $Q_M$            | The correlation matrix of the measurement noise      |
| $Q_P$            | The correlation matrix of the process noise          |

The correlation matrices,  $Q_M$  and  $Q_P$ , are specified in the parameter dialog by scalar variance terms to be placed along the matrix diagonals, thus ensuring that these matrices are symmetric. The filter algorithm based on this constraint is also known as the random-walk Kalman filter.

The implementation of the algorithm in the block is optimized by exploiting the symmetry of the input covariance matrix  $K(n)$ . This decreases the total number of computations by a factor of two.

The block icon has port labels corresponding to the inputs and outputs of the Kalman algorithm. Note that inputs to the In and Err ports must be sample-based scalars with the same complexity. The signal at the Out port is a scalar, while the signal at the Taps port is a sample-based vector.

| Block Ports | Corresponding Variables   |
|-------------|---|
| In          | $u$ , the scalar input, which is internally buffered into the vector $u(n)$ |
| Out         | $y(n)$ , the filtered scalar output   |
| Err         | $e(n)$ , the scalar estimation error  |
| Taps        | $\widehat{w}(n)$ , the vector of filter-tap estimates                       |

An optional Adapt input port is added when you select the **Adapt port** check box in the dialog. When this port is enabled, the block continuously adapts the filter coefficients while the Adapt input is nonzero. A zero-valued input to the Adapt port causes the block to stop adapting, and to hold the filter coefficients at their current values until the next nonzero Adapt input.

The **FIR filter length** parameter specifies the length of the filter that the Kalman algorithm estimates. The **Measurement noise variance** and the **Process noise variance** parameters specify the correlation matrices of the measurement and process noise, respectively. The **Measurement noise variance** must be a scalar, while the **Process noise variance** can be a vector of values to be placed along the diagonal, or a scalar to be repeated for the diagonal elements.

The **Initial value of filter taps** specifies the initial value  $\widehat{w}(0)$  as a vector, or as a scalar to be repeated for all vector elements. The **Initial error correlation matrix** specifies the initial value  $K(0)$ , and can be a diagonal matrix, a vector of values to be placed along the diagonal, or a scalar to be repeated for the diagonal elements.

## Parameters

### FIR filter length

The length of the FIR filter.

### Measurement noise variance

The value to appear along the diagonal of the measurement noise correlation matrix. Tunable (Simulink).

### Process noise variance

The value to appear along the diagonal of the process noise correlation matrix. Tunable (Simulink).

### Initial value of filter taps

The initial FIR filter coefficients.

### Initial error correlation matrix

The initial value of the error correlation matrix.

### Adapt port

Enables the Adapt port.

## References

Haykin, S. *Adaptive Filter Theory*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1996.

## Supported Data Types

- Double-precision floating point
- Single-precision floating point

## See Also

LMS Adaptive Filter (Obsolete)    DSP System Toolbox

RLS Adaptive Filter (Obsolete)    DSP System Toolbox

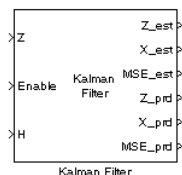
See “Noise Cancellation in Simulink Using Normalized LMS Adaptive Filter” for related information.

## Version History

**Introduced in R2008b**

# Kalman Filter

Predict or estimate states of dynamic systems



## Library

Filtering/Adaptive Filters

dspadpt3

## Description

Use the Kalman Filter block to predict or estimate the state of a dynamic system from a series of incomplete and/or noisy measurements. Suppose you have a noisy linear system that is defined by the following equations:

$$\begin{aligned}x_k &= Ax_{k-1} + w_{k-1} \\z_k &= Hx_k + v_k\end{aligned}$$

This block can use the previously estimated state,  $\hat{x}_{k-1}$ , to predict the current state at time  $k$ ,  $x_k^-$ , as shown by the following equation:

$$\begin{aligned}x_k^- &= A\hat{x}_{k-1} \\P_k^- &= A\hat{P}_{k-1}A^T + Q\end{aligned}$$

The block can also use the current measurement,  $z_k$ , and the predicted state,  $x_k^-$ , to estimate the current state value at time  $k$ ,  $\hat{x}_k$ , so that it is a more accurate approximation:

$$\begin{aligned}K_k &= P_k^- H^T (HP_k^- H^T + R)^{-1} \\ \hat{x}_k &= x_k^- + K_k(z_k - Hx_k^-) \\ \hat{P}_k &= (I - K_k H)P_k^-\end{aligned}$$

The variables in the previous equations are defined in the following table.

| Variable  | Definition      | Default Value or Initial Condition |
|-----------|-----------------|------------------------------------|
| $x$       | State           | N/A                                |
| $\hat{x}$ | Estimated state | <code>zeros([6, 1])</code>         |
| $x^-$     | Predicted state | N/A                                |

| Variable  | Definition                   | Default Value or Initial Condition   |
|-----------|------------------------------|--|
| $A$       | State transition matrix      | $\begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$ |
| $w$       | Process noise                | N/A  |
| $z$       | Measurement                  | N/A  |
| $H$       | Measurement matrix           | $\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$   |
| $v$       | Measurement noise            | N/A  |
| $\hat{P}$ | Estimated error covariance   | $10 \cdot \text{eye}(6)$   |
| $P$       | Predicted error covariance   | N/A  |
| $Q$       | Process noise covariance     | $0.05 \cdot \text{eye}(6)$   |
| $K$       | Kalman gain                  | N/A  |
| $R$       | Measurement noise covariance | $\text{eye}(4)$  |
| $I$       | Identity matrix              | N/A  |

In the previous equations,  $z$  is a vector of measurement values. Most of the time, the block processes  $Z$ , an M-by-N matrix, where M is the number of measurement values and N is the number of filters.

Use the **Number of filters** parameter to specify the number of filters to use to predict or estimate the current value.

Use the **Enable filters** parameter to specify which filters are enabled or disabled at each time step. If you select **Always**, the filters are always enabled. If you choose **Specify via input port <Enable>**, the Enable port appears on the block. The input to this port must be a row vector of 1s and 0s whose length is equal to the number of filters. For example, if there are 3 filters and the input to the Enable port is [1 0 1], only the first and third filter are enabled at this time step. If you select the **Reset the estimated state and estimated error covariance when filters are disabled** check box, the estimated and predicted states as well as the estimated error covariance that correspond to the disabled filters are reset to their initial values.

---

**Note** All filters have the same state transition matrix, measurement matrix, initial conditions, and noise covariance, but their state, measurement, enable, and MSE signals are unique. Within the state, measurement, enable, and MSE signals, each column corresponds to a filter.

---

Use the **Measurement matrix source** parameter to specify how to enter the measurement matrix values. If you select **Specify via dialog**, the **Measurement matrix** parameter appears in the dialog box. If you select **Input port <H>**, the H port appears on the block. Use this port to specify your measurement matrix.

## Parameters

### Number of filters

Specify the number of filters to use to predict or estimate the current value.

### Enable filters

Specify which filters are enabled or disabled at each time step. If you select **Always**, the filters are always enabled. If you choose **Specify via input port <Enable>**, the Enable port appears on the block.

### Reset the estimated state and estimated error covariance when filters are disabled

If you select this check box, the estimated and predicted states as well as the estimated error covariance that correspond to the disabled filters are reset to their initial values. This parameter is visible if, for the **Enable filters** parameter, you select **Specify via input port <Enable>**.

### Initial condition for estimated state

Enter the initial condition for the estimated state.

### Initial condition for estimated error covariance

Enter the initial condition for the estimated error covariance.

### State transition matrix

Enter the state transition matrix.

### Process noise covariance

Enter the process noise covariance.

### Measurement matrix source

Specify how to enter the measurement matrix values. If you select **Specify via dialog**, the **Measurement matrix** parameter appears in the dialog box. If you select **Input port <H>**, the H port appears on the block.

### Measurement matrix

Enter the measurement matrix values. This parameter is visible if you select **Specify via dialog** for the **Measurement matrix source** parameter.

### Measurement noise covariance

Enter the measurement noise covariance.

### Output estimated measurement <Z\_est>

Select this check box if you want the block to output the estimated measurement.

### Output estimated state <X\_est>

Select this check box if you want the block to output the estimated state.

### Output MSE of estimated state <MSE\_est>

Select this check box if you want the block to output the mean-squared error of the estimated state.

### Output predicted measurement <Z\_prd>

Select this check box if you want the block to output the predicted measurement.

### Output predicted state <X\_prd>

Select this check box if you want the block to output the predicted state.

**Output MSE of predicted state <MSE\_prb>**

Select this check box if you want the block to output the mean-squared error of the predicted state.

**References**

[1] Haykin, Simon. *Adaptive Filter Theory*. Upper Saddle River, NJ: Prentice Hall, 1996.

[2] Welch, Greg and Gary Bishop, "An Introduction to the Kalman Filter," TR 95-041, Department of Computer Science, University of North Carolina.

**Supported Data Types**

| Port    | Input/Output   | Supported Data Types  |
|---------|--|---|
| Z       | M-by-N measurement where M is the length of the measurement vector and N is the number of filters.                         | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> </ul>                    |
| Enable  | 1-by-N vector of 1s and 0s where N is the number of filters.   | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Boolean</li> </ul> |
| H       | M-by-P measurement matrix where M is the length of the measurement vector and P is the length of the filter state vectors. | Same as Z port  |
| Z_est   | M-by-N estimated measurement matrix where M is the length of the measurement vector and N is the number of filters.        | Same as Z port  |
| X_est   | P-by-N estimated state matrix where P is the length of the filter state vectors and N is the number of filters.            | Same as Z port  |
| MSE_est | 1-by-N vector that represents the mean-squared-error of the estimated state. N is the number of filters.                   | Same as Z port  |
| Z_prd   | M-by-N predicted measurement matrix where M is the length of the measurement vector and N is the number of filters.        | Same as Z port  |
| X_prd   | P-by-N predicted state matrix where P is the length of the filter state vectors and N is the number of filters.            | Same as Z port  |
| MSE_prd | 1-by-N vector that represents the mean-squared-error of the predicted state. N is the number of filters.                   | Same as Z port  |

**See Also**

|            |                    |
|------------|--------------------|
| LDL Solver | DSP System Toolbox |
|------------|--------------------|



## **Version History**

**Introduced in R2007a**

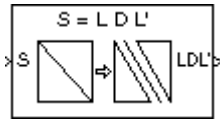
### **Extended Capabilities**

#### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

## LDL Factorization

Factor square Hermitian positive definite matrices into lower, upper, and diagonal components



### Libraries:

DSP System Toolbox / Math Functions / Matrices and Linear Algebra / Matrix Factorizations

### Description

The LDL Factorization block uniquely factors the square Hermitian positive definite input matrix  $S$  as

$$S = LDL^*$$

where  $L$  is a lower triangular square matrix with unity diagonal elements,  $D$  is a diagonal matrix, and  $L^*$  is the Hermitian (complex conjugate) transpose of  $L$ . Only the diagonal and lower triangle of the input matrix are used. Any imaginary component of the diagonal entries is disregarded.

LDL factorization requires half the computation of Gaussian elimination (LU decomposition), and is always stable. It is more efficient than Cholesky factorization because it avoids computing the square roots of the diagonal elements.

### Ports

#### Input

**Port\_1** — Input signal  
matrix

Input signal, specified as a square matrix.

The algorithm requires that the input be square and Hermitian positive definite. When the input is not positive definite, the block reacts as specified by the **Non-positive definite input** parameter.

If the input is fixed point, it must be signed fixed point with a power-of-two slope and zero bias.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `fixed point`

#### Output

**Port\_1** — Output signal  
matrix

The output is a composite matrix with lower triangle elements  $l_{ij}$  from  $L$ , diagonal elements  $d_{ij}$  from  $D$  and upper triangle elements  $u_{ij}$  from  $L^*$ , where  $L$  is a lower triangular square matrix that has unity diagonal elements,  $D$  is a diagonal matrix, and  $L^*$  is the Hermitian (complex conjugate) transpose of  $L$ .

The output format is shown below for a 5-by-5 matrix.

|          |          |          |          |          |
|----------|----------|----------|----------|----------|
| $d_{11}$ | $u_{12}$ | $u_{13}$ | $u_{14}$ | $u_{15}$ |
| $l_{21}$ | $d_{22}$ | $u_{23}$ | $u_{24}$ | $u_{25}$ |
| $l_{31}$ | $l_{32}$ | $d_{33}$ | $u_{34}$ | $u_{35}$ |
| $l_{41}$ | $l_{42}$ | $l_{43}$ | $d_{44}$ | $u_{45}$ |
| $l_{51}$ | $l_{52}$ | $l_{53}$ | $l_{54}$ | $d_{55}$ |

$$u_{ij} = l_{ji}^*$$

Data Types: single | double | int8 | int16 | int32 | fixed point

## Parameters

### Main Tab

**Non-positive definite input** — Non-positive definite input action

Warning (default) | Ignore | Error

Specify the action when nonpositive definite matrix inputs occur:

- Ignore — Proceed with the computation and do not issue an alert. The output is not a valid factorization. A partial factorization is present in the upper-left corner of the output.
- Warning — Display a warning message in the MATLAB Command Window and continue the simulation. The output is not a valid factorization. A partial factorization is present in the upper-left corner of the output.
- Error — Display an error dialog box and terminate the simulation.

### Data Types Tab

**Rounding mode** — Method of rounding operation

Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Specify the rounding mode for fixed-point operations as one of the following:

- Floor
- Ceiling
- Convergent
- Nearest
- Round
- Simplest
- Zero

For more details, see rounding mode.

**Saturate on integer overflow** — Method of overflow action

off (default) | on


When you select this parameter, the block saturates the result of its fixed-point operation. When you clear this parameter, the block wraps the result of its fixed-point operation. For details on `saturate` and `wrap`, see `overflow mode for fixed-point operations`.

### Intermediate product — Intermediate product data type

Inherit: `Same as input (default) | fixdt(1,16,0)`

Specify the intermediate product data type. As shown in *Fixed-Point Conversion* section in “Extended Capabilities” on page 2-0, the output of the multiplier is cast to the intermediate product data type before the next element of the input is multiplied into it. You can set the data type to:

- A rule that inherits a data type, for example, `Inherit: Same as input`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Alternatively, you can set the **Intermediate product** data type by using the **Data Type Assistant**. To use the assistant, click the **Show data type assistant** button .

For more information, see “Specify Data Types Using Data Type Assistant” (Simulink).


### Product output — Product output data type

Inherit: `Inherit via internal rule (default) | Inherit: Same as input | fixdt([],16,0)`

**Product output** specifies the data type of the output of a product operation in the LDL Factorization block.

- `Inherit: Inherit via internal rule` — The block inherits the product output data type based on an internal rule. For more information on this rule, see “Inherit via Internal Rule”.
- `Inherit: Same as input` — The block specifies the product output data type to be the same as the input data type.
- `fixdt([],16,0)` — The block specifies an autosigned, binary-point, scaled, fixed-point data type that has a word length of 16 bits and a fraction length of 0.

For more information, see “Multiplication Data Types” and *Fixed-Point Conversion* in “Extended Capabilities” on page 2-0.

Alternatively, you can set the **Product output** data type by using the **Data Type Assistant**. To use the assistant, click the **Show data type assistant** button .

For more information on the data type assistant, see “Specify Data Types Using Data Type Assistant” (Simulink).


### Accumulator — Accumulator data type

Inherit: `Inherit via internal rule (default) | Inherit: Same as input | Inherit: Same as product output | fixdt([],16,0)`

**Accumulator** specifies the data type of the output of an accumulation operation in the LDL Factorization block.

- **Inherit: Inherit via internal rule** — The block inherits the accumulator data type based on an internal rule. For more information on this rule, see “Inherit via Internal Rule”.
- **Inherit: Same as input** — The block specifies the accumulator data type to be the same as the input data type.
- **Inherit: Same as product output** — The block specifies the accumulator data type to be the same as the product output data type.
- `fixdt([], 16, 0)` — The block specifies an autosigned, binary-point, scaled, fixed-point data type that has a word length of 16 bits and a fraction length of 0.

For illustrations on how to use the accumulator data type in this block, see Fixed-Point Conversion in “Extended Capabilities” on page 2-0 .

Alternatively, you can set the **Accumulator** data type by using the **Data Type Assistant**. To use the assistant, click the **Show data type assistant** button .


For more information on the data type assistant, see “Specify Data Types Using Data Type Assistant” (Simulink).

#### **Output** — Output data type

`Inherit: Same as input (default) | fixdt([], 16, 0)`

**Output** specifies the data type of the output of the LDL Factorization block. For more information on the output data type, see the 'Fixed-Point Conversion' section in “Extended Capabilities” on page 2-0 .

- **Inherit: Same as input** — The block specifies the output data type to be the same as the input data type.
- `fixdt([], 16, 0)` — The block specifies an autosigned, binary-point, scaled, fixed-point data type that has a word length of 16 bits and a fraction length of 0.

Alternatively, you can set the **Output** data type by using the **Data Type Assistant**. To use the assistant, click the **Show data type assistant** button .

For more information on the data type assistant, see “Specify Data Types Using Data Type Assistant” (Simulink).

#### **Output Minimum** — Minimum value that block can output

`[] (default) | scalar`

Specify the minimum value that the block can output. Simulink software uses this minimum value to perform:

- Simulation range checking. See “Specify Signal Ranges” (Simulink).
- Automatic scaling of fixed-point data types.

#### **Output Maximum** — Maximum value that block can output

`[] (default) | scalar`

Specify the maximum value that the block can output. Simulink software uses this maximum value to perform:

- Simulation range checking. See “Specify Signal Ranges” (Simulink).
- Automatic scaling of fixed-point data types.

**Lock data type settings against changes by the fixed-point tools** — Prevent fixed-point tools from overriding data types  
off (default) | on

Select this parameter to prevent the fixed-point tools from overriding the data types you specify in the block dialog box.

## Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | double   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no                                      |
| <b>Multidimensional Signals</b> | no                                      |
| <b>Variable-Size Signals</b>    | no                                      |
| <b>Zero-Crossing Detection</b>  | no                                      |

## Version History

Introduced before R2006a

## References

- [1] Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Generated code relies on the memcpy or memset function (`string.h`) under certain conditions.

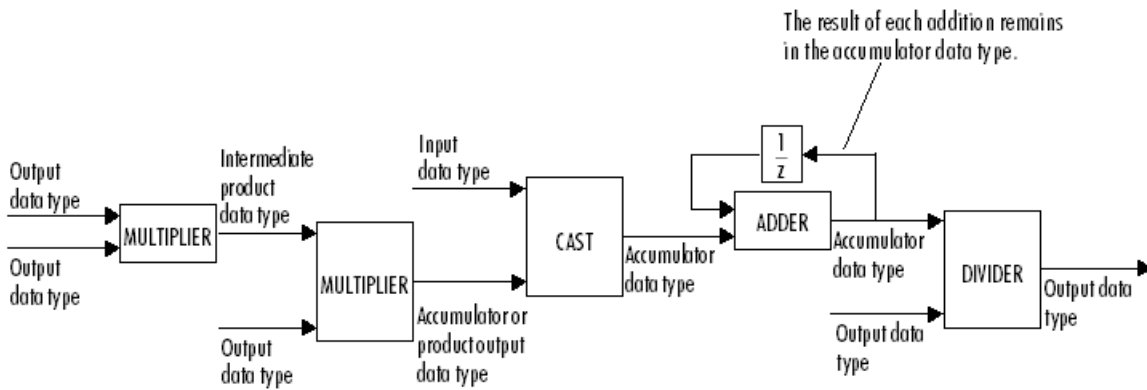
### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

If the input is fixed point, it must be a signed integer or a signed fixed point value with a power-of-two slope and zero bias.

### Fixed-Point Data Types

This diagram shows the data types used within the LDL Factorization block for fixed-point signals.



You can set the intermediate product, product output, accumulator, and output data types in the block dialog box.

When the input is real, the output of the second multiplier is in the product output data type. When the input is complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication performed, see "Multiplication Data Types".

## See Also

### Blocks

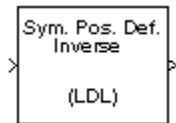
Cholesky Factorization | LDL Inverse | LDL Solver | LU Factorization | QR Factorization

### Topics

"Matrix Factorizations"

## LDL Inverse

Compute inverse of Hermitian positive definite matrix using LDL factorization



## Library

Math Functions / Matrices and Linear Algebra / Matrix Inverses

dspinverses

## Description

The LDL Inverse block computes the inverse of the Hermitian positive definite input matrix  $S$  by performing an LDL factorization.

$$S^{-1} = (LDL^*)^{-1}$$

$L$  is a lower triangular square matrix with unity diagonal elements,  $D$  is a diagonal matrix, and  $L^*$  is the Hermitian (complex conjugate) transpose of  $L$ . Only the diagonal and lower triangle of the input matrix are used, and any imaginary component of the diagonal entries is disregarded.

LDL factorization requires half the computation of Gaussian elimination (LU decomposition), and is always stable. It is more efficient than Cholesky factorization because it avoids computing the square roots of the diagonal elements.

The algorithm requires that the input be Hermitian positive definite. When the input is not positive definite, the block reacts with the behavior specified by the **Non-positive definite input** parameter. The following options are available:

- **Ignore** — Proceed with the computation and do not issue an alert. The output is not a valid inverse.
- **Warning** — Display a warning message in the MATLAB command window, and continue the simulation. The output is not a valid inverse.
- **Error** — Display an error dialog and terminate the simulation.

---

**Note** The **Non-positive definite input** parameter is a diagnostic parameter. Like all diagnostic parameters on the Configuration Parameters dialog, it is set to **Ignore** in the code generated for this block by Simulink Coder code generation software.

---

## Parameters

### Non-positive definite input

Response to nonpositive definite matrix inputs.



## References

Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

## Supported Data Types

- Double-precision floating point
- Single-precision floating point

## See Also

|                   |                    |
|-------------------|--------------------|
| Cholesky Inverse  | DSP System Toolbox |
| LDL Factorization | DSP System Toolbox |
| LDL Solver        | DSP System Toolbox |
| LU Inverse        | DSP System Toolbox |
| Pseudoinverse     | DSP System Toolbox |
| <code>inv</code>  | MATLAB             |

See “Matrix Inverses” for related information.

## Version History

Introduced before R2006a

## Extended Capabilities

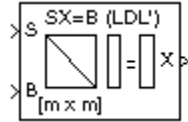
### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Generated code relies on the `memcpy` or `memset` function (`string.h`) under certain conditions.

## LDL Solver

Solve  $SX=B$  for  $X$  when  $S$  is square Hermitian positive definite matrix



## Library

Math Functions / Matrices and Linear Algebra / Linear System Solvers

dpsolvers

## Description

The LDL Solver block solves the linear system  $SX=B$  by applying LDL factorization to the matrix at the S port, which must be square ( $M$ -by- $M$ ) and Hermitian positive definite. Only the diagonal and lower triangle of the matrix are used, and any imaginary component of the diagonal entries is disregarded. The input to the B port is the right side  $M$ -by- $N$  matrix,  $B$ . The  $M$ -by- $N$  output matrix  $X$  is the unique solution of the equations.

A length- $M$  unoriented vector input for right side  $B$  is treated as an  $M$ -by-1 matrix.

When the input is not positive definite, the block reacts with the behavior specified by the **Non-positive definite input** parameter. The following options are available:

- **Ignore** — Proceed with the computation and do not issue an alert. The output is not a valid solution.
- **Warning** — Proceed with the computation and display a warning message in the MATLAB Command Window. The output is not a valid solution.
- **Error** — Display an error dialog and terminate the simulation.

---

**Note** The **Non-positive definite input** parameter is a diagnostic parameter. Like all diagnostic parameters on the Configuration Parameters dialog, it is set to **Ignore** in the code generated for this block by Simulink Coder code generation software.

---

## Algorithm

The LDL algorithm uniquely factors the Hermitian positive definite input matrix  $S$  as

$$S = LDL^*$$

where  $L$  is a lower triangular square matrix with unity diagonal elements,  $D$  is a diagonal matrix, and  $L^*$  is the Hermitian (complex conjugate) transpose of  $L$ .

The equation

$$LDL^*X = B$$

is solved for  $X$  by the following steps:

- 1 Substitute

$$Y = DL^*X$$

- 2 Substitute

$$Z = L^*X$$

- 3 Solve one diagonal and two triangular systems.

$$LY = B$$

$$DZ = Y$$

$$L^*X = Z$$

## Parameters

### Non-positive definite input

Response to nonpositive definite matrix inputs.

## Supported Data Types

- Double-precision floating point
- Single-precision floating point

## See Also

|                     |                    |
|---------------------|--------------------|
| Autocorrelation LPC | DSP System Toolbox |
| Cholesky Solver     | DSP System Toolbox |
| LDL Factorization   | DSP System Toolbox |
| LDL Inverse         | DSP System Toolbox |
| Levinson-Durbin     | DSP System Toolbox |
| LU Solver           | DSP System Toolbox |
| QR Solver           | DSP System Toolbox |

See “Linear System Solvers” for related information.

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Generated code relies on the `memcpy` or `memset` function (`string.h`) under certain conditions.

# Least Squares Polynomial Fit

Compute polynomial coefficients that best fit input data in least-squares sense



## Library

Math Functions / Polynomial Functions

dsppolyfun

## Description

The Least Squares Polynomial Fit block computes the coefficients of the  $n$ th order polynomial that best fits the input data in the least-squares sense, where you specify  $n$  in the **Polynomial order** parameter. A distinct set of  $n+1$  coefficients is computed for each column of the  $M$ -by- $N$  input,  $u$ .

For a given input column, the block computes the set of coefficients,  $c_1, c_2, \dots, c_{n+1}$ , that minimizes the quantity

$$\sum_{i=1}^M (u_i - \hat{u}_i)^2$$

where  $u_i$  is the  $i$ th element in the input column, and

$$\hat{u}_i = f(x_i) = c_1 x_i^n + c_2 x_i^{n-1} + \dots + c_{n+1}$$

The values of the independent variable,  $x_1, x_2, \dots, x_M$ , are specified as a length- $M$  vector by the **Control points** parameter. The same  $M$  control points are used for all  $N$  polynomial fits, and can be equally or unequally spaced. The equivalent MATLAB code is shown below.

```
c = polyfit(x,u,n) % Equivalent MATLAB code
```

For convenience, the block treats length- $M$  unoriented vector input as an  $M$ -by-1 matrix.

Each column of the  $(n+1)$ -by- $N$  output matrix,  $c$ , represents a set of  $n+1$  coefficients describing the best-fit polynomial for the corresponding column of the input. The coefficients in each column are arranged in order of descending exponents,  $c_1, c_2, \dots, c_{n+1}$ .

## Parameters

### Control points

The values of the independent variable to which the data in each input column correspond. For an  $M$ -by- $N$  input, this parameter must be a length- $M$  vector. Tunable (Simulink).

### Polynomial order

The order,  $n$ , of the polynomial to be used in constructing the best fit. The number of coefficients is  $n+1$ .

## Supported Data Types

- Double-precision floating point
- Single-precision floating point

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Generated code relies on the memcpy or memset function (string.h) under certain conditions.

## See Also

### Functions

polyfit

### Blocks

Detrend | Polynomial Evaluation | Polynomial Stability Test

### Topics

“Compute Polynomial Coefficients Using Least Squares Polynomial Fit Block”

# Levinson-Durbin

Solve linear system of equations using Levinson-Durbin recursion



## Library

Math Functions / Matrices and Linear Algebra / Linear System Solvers

dsp solvers

## Description

The Levinson-Durbin block solves the  $n$ th-order system of linear equations

$$Ra = b$$

in the cases where:

- $R$  is a Hermitian, positive-definite, Toeplitz matrix.
- $b$  is identical to the first column of  $R$  shifted by one element and with the opposite sign.

$$\begin{bmatrix} r(1) & r^*(2) & \dots & r^*(n) \\ r(2) & r(1) & \dots & r^*(n-1) \\ \vdots & \vdots & \ddots & \vdots \\ r(n) & r(n-1) & \dots & r(1) \end{bmatrix} \begin{bmatrix} a(2) \\ a(3) \\ \vdots \\ a(n+1) \end{bmatrix} = \begin{bmatrix} -r(2) \\ -r(3) \\ \vdots \\ -r(n+1) \end{bmatrix}$$

The input to the block,  $r = [r(1) \ r(2) \ \dots \ r(n+1)]$ , can be a vector or a matrix. If the input is a matrix, the block treats each column as an independent channel and solves it separately. Each channel of the input contains lags 0 through  $n$  of an autocorrelation sequence, which appear in the matrix  $R$ .

The block can output the polynomial coefficients,  $A$ , the reflection coefficients,  $K$ , and the prediction error power,  $P$ , in various combinations. The **Output(s)** parameter allows you to enable the  $A$  and  $K$  outputs by selecting one of the following settings:

- **A** — For each channel, port A outputs  $A = [1 \ a(2) \ a(3) \ \dots \ a(n+1)]$ , the solution to the Levinson-Durbin equation.  $A$  has the same dimension as the input. You can also view the elements of each output channel as the coefficients of an  $n$ th-order autoregressive (AR) process.
- **K** — For each channel, port K outputs  $K = [k(1) \ k(2) \ \dots \ k(n)]$ , which contains  $n$  reflection coefficients and has the same dimension as the input, less one element. A scalar input channel

causes an error when you select K. You can use reflection coefficients to realize a lattice representation of the AR process described later in this page.

- **A and K** — The block outputs both representations at their respective ports. A scalar input channel causes an error when you select **A and K**.

Select the **Output prediction error power (P)** check box to output the prediction error power for each channel,  $P$ . For each channel,  $P$  represents the power of the output of an FIR filter with taps  $A$  and input autocorrelation described by  $r$ , where  $A$  represents a prediction error filter and  $r$  is the input to the block. In this case,  $A$  is a whitening filter.  $P$  has one element per input channel.

When you select the **If the value of lag 0 is zero, A=[1 zeros], K=[zeros], P=0** check box (default), an input channel whose  $r(1)$  element is zero generates a zero-valued output. When you clear this check box, an input with  $r(1) = 0$  generates NaNs in the output. In general, an input with  $r(1) = 0$  is invalid because it does not construct a positive-definite matrix  $R$ . Often, however, blocks receive zero-valued inputs at the start of a simulation. The check box allows you to avoid propagating NaNs during this period.

### Applications

One application of the Levinson-Durbin formulation implemented by this block is in the Yule-Walker AR problem, which concerns modeling an unknown system as an autoregressive process. You would model such a process as the output of an all-pole IIR filter with white Gaussian noise input. In the Yule-Walker problem, the use of the signal's autocorrelation sequence to obtain an optimal estimate leads to an  $Ra = b$  equation of the type shown above, which is most efficiently solved by Levinson-Durbin recursion. In this case, the input to the block represents the autocorrelation sequence, with  $r(1)$  being the zero-lag value. The output at the block's A port then contains the coefficients of the autoregressive process that optimally models the system. The coefficients are ordered in descending powers of  $z$ , and the AR process is minimum phase. The prediction error,  $G$ , defines the gain for the unknown system, where  $G = \sqrt{P}$ :

$$H(z) = \frac{G}{A(z)} = \frac{G}{1 + a(2)z^{-1} + \dots + a(n+1)z^{-n}}$$

The output at the block's K port contains the corresponding reflection coefficients,  $[k(1) \ k(2) \ \dots \ k(n)]$ , for the lattice realization of this IIR filter. The Yule-Walker AR Estimator block implements this autocorrelation-based method for AR model estimation, while the Yule-Walker Method block extends the method to spectral estimation.

Another common application of the Levinson-Durbin algorithm is in linear predictive coding, which is concerned with finding the coefficients of a moving average (MA) process (or FIR filter) that predicts the next value of a signal from the current signal sample and a finite number of past samples. In this case, the input to the block represents the signal's autocorrelation sequence, with  $r(1)$  being the zero-lag value, and the output at the block's A port contains the coefficients of the predictive MA process (in descending powers of  $z$ ).

$$H(z) = A(z) = 1 + a(2)z^{-1} + \dots + a(n+1)z^{-n}$$

These coefficients solve the following optimization problem:

$$\min_{\{a_i\}} E \left[ \left| x_n - \sum_{i=1}^N a_i x_{n-i} \right|^2 \right]$$



Again, the output at the block's K port contains the corresponding reflection coefficients,  $[k(1) k(2) \dots k(n)]$ , for the lattice realization of this FIR filter. The Autocorrelation LPC block in the Linear Prediction library implements this autocorrelation-based prediction method.

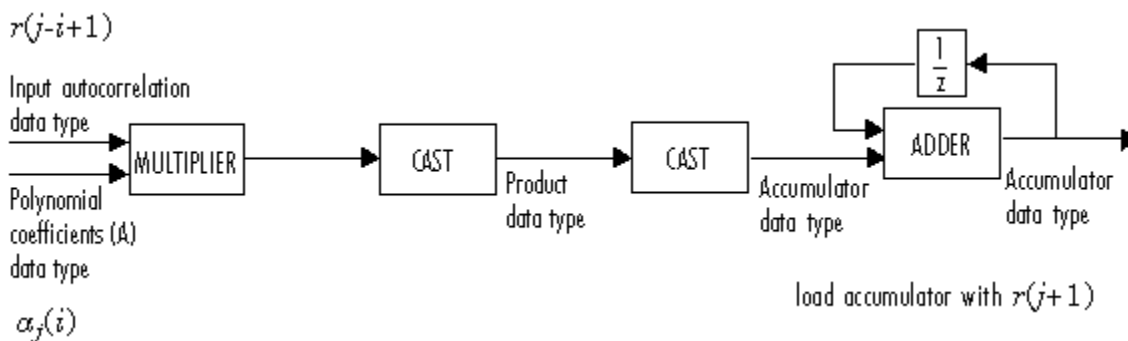
**Fixed-Point Data Types**

The diagrams in this section show the data types used within the Levinson-Durbin block for fixed-point signals.

After initialization the block performs  $n$  updates. At the  $(j+1)$  update,

$$\text{value in accumulator} = r(j+1) + \sum a_j(i) \times r(j-i+1)$$

The following diagram displays the fixed-point data types used in this calculation:



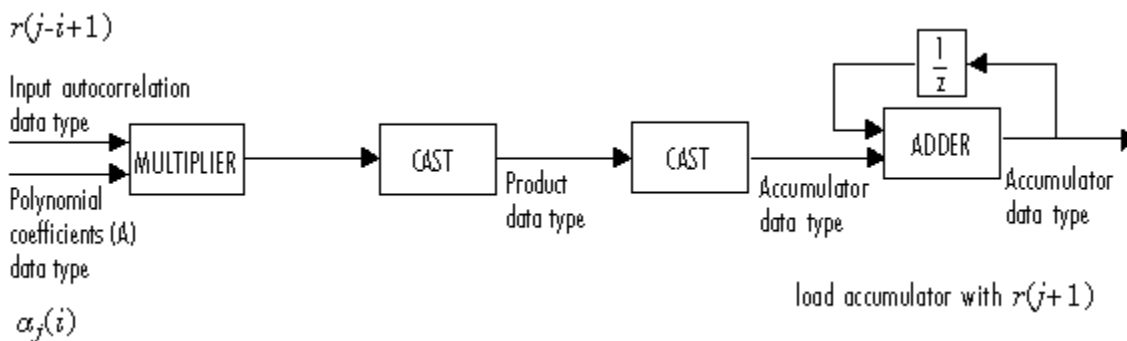
The block then updates the reflection coefficients  $K$  according to

$$K_{j+1} = \text{value in accumulator} / P_j$$

The block then updates the prediction error power  $P$  according to

$$P_{j+1} = P_j - P_j \times K_{j+1} \times \text{conj}(K_{j+1})$$

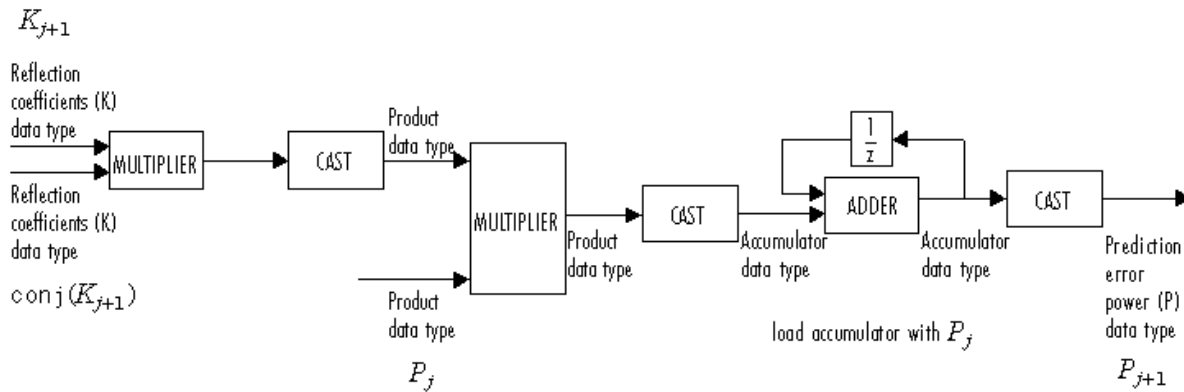
The next diagram displays the fixed-point data types used in this calculation:



The polynomial coefficients  $A$  are then updated according to

$$a_{j+1}(i) = a_j(i) + K_{j+1} \times \text{conj}(a_j(j-1+i))$$

This diagram displays the fixed-point data types used in this calculation:



## Algorithm

The algorithm requires  $O(n^2)$  operations for each input channel. This implementation is therefore much more efficient for large  $n$  than standard Gaussian elimination, which requires  $O(n^3)$  operations per channel.

## Parameters

### Main Tab

### Output(s)

Specify the solution representation of  $Ra = b$  to output: model coefficients (A), reflection coefficients (K), or both (A and K). When the input is a scalar or row vector, you must set this parameter to A.

### Output prediction error power (P)

Select to output the prediction error at port P.

### If the value of lag 0 is zero, A=[1 zeros], K=[zeros], P=0

When you select this check box and the first element of the input,  $r(1)$ , is zero, the block outputs the following vectors, as appropriate:

- $A = [1 \text{ zeros}(1, n)]$
- $K = [\text{zeros}(1, n)]$
- $P = 0$

When you clear this check box, the block outputs a vector of NaNs for each channel whose  $r(1)$  element is zero.

### Data Types Tab

---

**Note** Floating-point inheritance takes precedence over the data type settings defined on this pane. When inputs are floating point, the block ignores these settings, and all internal data types are floating point.

---

### Rounding mode

Specify the rounding mode for fixed-point operations as one of the following:

- Floor
- Ceiling
- Convergent
- Nearest
- Round
- Simplest
- Zero

For more details, see rounding mode.

### Saturate on integer overflow

When you select this parameter, the block saturates the result of its fixed-point operation. When you clear this parameter, the block wraps the result of its fixed-point operation. For details on `saturate` and `wrap`, see overflow mode for fixed-point operations.

### Product output

Specify the product output data type. See “Fixed-Point Data Types” on page 2-865 and “Multiplication Data Types” for illustrations depicting the use of the product output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as input`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`


Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Product output** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in *Simulink User's Guide* (Simulink) for more information.

### Accumulator

Specify the accumulator data type. See “Fixed-Point Data Types” on page 2-865 for illustrations depicting the use of the accumulator data type in this block. You can set it to:


- A rule that inherits a data type, for example, `Inherit: Same as input`
- A rule that inherits a data type, for example, `Inherit: Same as product output`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Accumulator** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in *Simulink User's Guide* (Simulink) for more information.

### Polynomial coefficients (A)


Specify the polynomial coefficients (A) data type. See “Fixed-Point Data Types” on page 2-865 for illustrations depicting the use of the A data type in this block. You can set it to an expression that evaluates to a valid data type, for example, `fixdt(1,16,15)`.

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **A** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in *Simulink User's Guide* (Simulink) for more information.

### Reflection coefficients (K)

Specify the polynomial coefficients (A) data type. See “Fixed-Point Data Types” on page 2-865 for illustrations depicting the use of the K data type in this block. You can set it to an expression that evaluates to a valid data type, for example, `fixdt(1,16,15)`.


Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **K** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in *Simulink User's Guide* (Simulink) for more information.

### Prediction error power (P)

Specify the prediction error power (P) data type. See “Fixed-Point Data Types” on page 2-865 for illustrations depicting the use of the P data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as input`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **P** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in *Simulink User's Guide* (Simulink) for more information.

### Minimum

Specify the minimum values that the polynomial coefficients, reflection coefficients, or prediction error power should have. The default value is `[]` (unspecified). Simulink uses this value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters” (Simulink))
- Automatic scaling of fixed-point data types

### Maximum

Specify the maximum values that the polynomial coefficients, reflection coefficients, or prediction error power should have. The default value is `[]` (unspecified). Simulink uses this value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters” (Simulink))
- Automatic scaling of fixed-point data types

### Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

### Product Output Data Type Assistant Parameters

#### Mode

Select how you would like to specify the data type properties of the **Product output** data type. You can choose:

- **Inherit** — Lets you specify a rule for inheriting a data type, for example, `Inherit: Inherit via internal rule`
- **Fixed point** — Lets you specify the fixed-point attributes of the data type.
- **Expression** — Lets you specify an expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

### **Signedness**

Specify the **Signedness** for the **Product output** data type.

### **Scaling**

Specify the **Scaling** for the **Product output** data type.

For more information see “Scaling” in the DSP System Toolbox User's Guide.

### **Word length**

Specify the **Word length** for the **Product output** data type.

### **Fraction length**

Specify the **Fraction length** for the **Product output** data type.

### **Data type override**

Specify the data type override mode. You can select one of the following options:

- **Inherit** — Inherits the data type override setting specified for the model.
- **Off** — Ignores the data type override setting specified for the model and uses the fixed-point data type you specify

This parameter appears only when you set the **Mode** parameter to **Built in** or **Fixed Point**. For more information, see “Specify Data Types Using Data Type Assistant” (Simulink).

## **Accumulator Data Type Assistant Parameters**

### **Mode**

Select how you would like to specify the data type properties of the **Accumulator** data type. You can choose:

- **Inherit** — Lets you specify a rule for inheriting a data type, for example, `Inherit: Inherit via internal rule`
- **Fixed point** — Lets you specify the fixed-point attributes of the data type.
- **Expression** — Lets you specify an expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

**Signedness**

Specify the **Signedness** for the **Accumulator** data type.

**Scaling**

Specify the **Scaling** for the **Accumulator** data type.

For more information see “Scaling” in the DSP System Toolbox User's Guide.

**Word length**

Specify the **Word length** for the **Accumulator** data type.

**Fraction length**

Specify the **Fraction length** for the **Accumulator** data type.

**Data type override**

Specify the data type override mode. You can select one of the following options:

- **Inherit** — Inherits the data type override setting specified for the model.
- **Off** — Ignores the data type override setting specified for the model and uses the fixed-point data type you specify

This parameter appears only when you set the **Mode** parameter to **Built in** or **Fixed Point**. For more information, see “Specify Data Types Using Data Type Assistant” (Simulink).

**Polynomial Coefficients (A) Data Type****Mode**

Select how you would like to specify the data type properties of the **Polynomial coefficients (A)** data type. You can choose:

- **Inherit** — Lets you specify a rule for inheriting a data type, for example, **Inherit: Inherit via internal rule**
- **Fixed point** — Lets you specify the fixed-point attributes of the data type.
- **Expression** — Lets you specify an expression that evaluates to a valid data type, for example, **fixdt(1,16,0)**

**Signedness**

Specify the **Signedness** for the **Polynomial coefficients (A)** data type.

**Scaling**

Specify the **Scaling** for the **Polynomial coefficients (A)** data type.

For more information see “Scaling” in the DSP System Toolbox User's Guide.

#### **Word length**

Specify the **Word length** for the **Polynomial coefficients (A)** data type.

#### **Fraction length**

Specify the **Fraction length** for the **Polynomial coefficients (A)** data type.

#### **Data type override**

Specify the data type override mode. You can select one of the following options:

- `Inherit` — Inherits the data type override setting specified for the model.
- `Off` — Ignores the data type override setting specified for the model and uses the fixed-point data type you specify

This parameter appears only when you set the **Mode** parameter to `Built in` or `Fixed Point`. For more information, see “Specify Data Types Using Data Type Assistant” (Simulink).

### **Reflection Coefficients (K) Data Type**

#### **Mode**

Select how you would like to specify the data type properties of the **Reflection coefficients (K)** data type. You can choose:

- `Inherit` — Lets you specify a rule for inheriting a data type, for example, `Inherit: Inherit via internal rule`
- `Fixed point` — Lets you specify the fixed-point attributes of the data type.
- `Expression` — Lets you specify an expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

#### **Signedness**

Specify the **Signedness** for the **Reflection coefficients (K)** data type.

#### **Scaling**

Specify the **Scaling** for the **Reflection coefficients (K)** data type.

For more information see “Scaling” in the DSP System Toolbox User's Guide.

#### **Word length**

Specify the **Word length** for the **Reflection coefficients (K)** data type.

#### **Fraction length**

Specify the **Fraction length** for the **Reflection coefficients (K)** data type.

#### **Data type override**

Specify the data type override mode. You can select one of the following options:

- **Inherit** — Inherits the data type override setting specified for the model.
- **Off** — Ignores the data type override setting specified for the model and uses the fixed-point data type you specify

This parameter appears only when you set the **Mode** parameter to **Built in** or **Fixed Point**. For more information, see “Specify Data Types Using Data Type Assistant” (Simulink).

#### **Prediction Error Power Data Type Assistant Parameters**

##### **Mode**

Select how you would like to specify the data type properties of the **Prediction error power (P)** data type. You can choose:

- **Inherit** — Lets you specify a rule for inheriting a data type, for example, `Inherit: Inherit via internal rule`
- **Fixed point** — Lets you specify the fixed-point attributes of the data type.
- **Expression** — Lets you specify an expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

##### **Signedness**

Specify the **Signedness** for the **Prediction error power (P)** data type.

##### **Scaling**

Specify the **Scaling** for the **Prediction error power (P)** data type.

For more information see “Scaling” in the DSP System Toolbox User's Guide.

##### **Word length**

Specify the **Word length** for the **Prediction error power (P)** data type.

##### **Fraction length**

Specify the **Fraction length** for the **Prediction error power (P)** data type.

##### **Data type override**

Specify the data type override mode. You can select one of the following options:

- **Inherit** — Inherits the data type override setting specified for the model.



- `Off` — Ignores the data type override setting specified for the model and uses the fixed-point data type you specify

This parameter appears only when you set the **Mode** parameter to `Built in` or `Fixed Point`. For more information, see “Specify Data Types Using Data Type Assistant” (Simulink).

## References

Golub, G. H. and C. F. Van Loan. Sect. 4.7 in *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

Ljung, L. *System Identification: Theory for the User*. Englewood Cliffs, NJ: Prentice Hall, 1987. Pgs. 278-280.

Kay, Steven M. *Modern Spectral Estimation: Theory and Application*. Englewood Cliffs, NJ: Prentice Hall, 1988.

## Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point (signed only)
- 8-, 16-, and 32-bit signed integers

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Generated code relies on the `memcpy` or `memset` function (`string.h`) under certain conditions.

### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

If the input is fixed point, it must be a signed integer or a signed fixed point value with a power-of-two slope and zero bias.

## See Also

### Functions

`levinson`

### Blocks

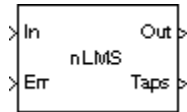
Cholesky Solver | LDL Solver | Autocorrelation LPC | LU Solver | QR Solver | Yule-Walker AR Estimator | Yule-Walker Method

**Topics**

“Linear System Solvers”

## LMS Adaptive Filter (Obsolete)

Compute filter estimates for input using LMS adaptive filter algorithm



### Library

dspobslib

### Description

**Note** The LMS Adaptive Filter block is still supported but is likely to be obsoleted in a future release. We strongly recommend replacing this block with the LMS Filter block.

The LMS Adaptive Filter block implements an adaptive FIR filter using the stochastic gradient algorithm known as the normalized least mean-square (LMS) algorithm.

$$y(n) = \widehat{w}^H(n-1)u(n)$$

$$e(n) = d(n) - y(n)$$

$$\widehat{w}(n) = \widehat{w}(n-1) + \frac{u(n)}{a + u^H(n)u(n)}\mu e^*(n)$$

The variables are as follows.

| Variable         | Description                                    |
|------------------|--|
| $n$              | The current algorithm iteration                |
| $u(n)$           | The buffered input samples at step $n$         |
| $\widehat{w}(n)$ | The vector of filter-tap estimates at step $n$ |
| $y(n)$           | The filtered output at step $n$                |
| $e(n)$           | The estimation error at step $n$               |
| $d(n)$           | The desired response at step $n$               |
| $\mu$            | The adaptation step size                       |

To overcome potential numerical instability in the tap-weight update, a small positive constant ( $a = 1e-10$ ) has been added in the denominator.

To turn off normalization, clear the **Use normalization** check box in the parameter dialog. The block then computes the filter-tap estimate as

$$\widehat{w}(n) = \widehat{w}(n-1) + u(n)\mu e^*(n)$$

The block icon has port labels corresponding to the inputs and outputs of the LMS algorithm. Note that inputs to the In and Err ports must be sample-based scalars. The signal at the Out port is a scalar, while the signal at the Taps port is a sample-based vector.

| Block Ports | Corresponding Variables   |
|-------------|---|
| In          | $u$ , the scalar input, which is internally buffered into the vector $u(n)$ |
| Out         | $y(n)$ , the filtered scalar output   |
| Err         | $e(n)$ , the scalar estimation error  |
| Taps        | $\hat{w}(n)$ , the vector of filter-tap estimates                           |

An optional **Adapt** input port is added when you select the **Adapt input** check box in the dialog. When this port is enabled, the block continuously adapts the filter coefficients while the **Adapt** input is nonzero. A zero-valued input to the **Adapt** port causes the block to stop adapting, and to hold the filter coefficients at their current values until the next nonzero **Adapt** input.

The **FIR filter length** parameter specifies the length of the filter that the LMS algorithm estimates. The **Step size** parameter corresponds to  $\mu$  in the equations. Typically, for convergence in the mean square,  $\mu$  must be greater than 0 and less than 2. The **Initial value of filter taps** specifies the initial value  $\hat{w}(0)$  as a vector, or as a scalar to be repeated for all vector elements. The **Leakage factor** specifies the value of the leakage factor,  $1 - \mu\alpha$ , in the leaky LMS algorithm below. This parameter must be between 0 and 1.

$$\hat{w}(n+1) = (1 - \mu\alpha)\hat{w}(n) + \frac{u(n)}{u^H(n)u(n)}\mu e^*(n)$$

## Examples

See the `lmsadtde` demo.

## Parameters

### FIR filter length

The length of the FIR filter.

### Step-size

The step-size, usually in the range (0, 2). Tunable (Simulink).

### Initial value of filter taps

The initial FIR filter coefficients.

### Leakage factor

The leakage factor, in the range [0, 1]. Tunable (Simulink).

### Use normalization

Select this check box to compute the filter-tap estimate using the normalized equations.

### Adapt input

Enables the **Adapt** port when selected.

## References

Haykin, S. *Adaptive Filter Theory*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1996.

## Supported Data Types

- Double-precision floating point
- Single-precision floating point

## See Also

Kalman Adaptive Filter (Obsolete)

DSP System Toolbox

RLS Adaptive Filter (Obsolete)

DSP System Toolbox

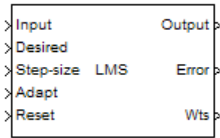
See “Noise Cancellation in Simulink Using Normalized LMS Adaptive Filter” for related information.

## Version History

**Introduced in R2008b**

## LMS Filter

Compute output, error, and weights using LMS adaptive algorithm



### Libraries:

DSP System Toolbox / Filtering / Adaptive Filters  
 DSP System Toolbox HDL Support / Filtering

## Description

The LMS Filter block can implement an adaptive FIR filter by using five different algorithms. The block estimates the filter weights or coefficients needed to minimize the error,  $e(n)$ , between the output signal  $y(n)$  and the desired signal,  $d(n)$ . The output is the filtered input signal, which is the estimate of the desired signal. The Error port outputs the result of subtracting the output signal from the desired signal.

Under specific conditions, this block also supports SIMD code generation. For details, see “Code Generation” on page 2-886.

## Ports

### Input

**Input** — Input signal  
 scalar | vector

Connect the signal that you want to filter, specified as a scalar or column vector.

When the input is fixed-point, it must be signed.

When you set the **Algorithm** parameter to Sign-Error LMS, Sign-Data LMS, or Sign-Sign LMS, the data input through the **Input** port must be real.

Data Types: single | double | fixed point

**Desired** — Desired signal  
 vector | matrix

Connect the desired signal, specified as a vector or matrix. The desired signal must have the same data type, complexity, and dimensions as the Input signal.

When **Input** is fixed-point, the desired signal must be a signed fixed-point.

When you set the **Algorithm** parameter to Sign-Error LMS, Sign-Data LMS, or Sign-Sign LMS, the data input through the **Desired** port must be real.

Data Types: single | double | fixed point

**Step-size** — Step-size  
 scalar

Enter the step size  $\mu$ . For convergence of the normalized LMS equations,  $0 < \mu < 2$ . Input type must match the type of the Input port.

When **Input** is fixed-point, the step-size must be a signed fixed-point.

#### Dependencies

This port appears only when you set the **Specify step size** via parameter to **Input port**.

Data Types: `single` | `double` | `fixed point`

#### **Adapt** — Update filter weights

scalar | vector

When the input to this port is greater than zero, the block continuously updates the filter weights. When the input to this port is less than or equal to zero, the filter weights remain at their current values.

#### Dependencies

This port appears only when you set the **Adapt port** parameter to **on**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `Boolean`

#### **Reset** — Reset filter weights

scalar | vector

Signal to reset the value of the filter weights to their initial values, specified as a vector. The block resets the filter weights whenever a reset event is detected at the Reset port. The reset signal rate must be the same rate as the data signal input.

For reset event types, see the **Reset** parameter.

#### Dependencies

This port appears only when you set the **Reset port** parameter to **Rising edge**, **Falling edge**, **Either edge**, or **Non-zero sample**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `Boolean`

#### Output

##### **Output** — Output signal

scalar | vector

Outputs the filtered input signal, which is the estimate of the desired signal. It is the same size and complexity as the input signal.

The output signal has the same data type as the desired signal.

Data Types: `single` | `double` | `fixed point`

##### **Error** — Error between Output and Desired signals

scalar | vector

Outputs the result of subtracting the output signal from the desired signal.

The error signal has the same data type as the desired signal.

Data Types: `single` | `double` | `fixed point`

**Wts** — Updated filter weights  
`scalar` | `vector`

For each iteration, the block outputs the current updated filter weights from this port.

The weights data type must match the type of the `Input` port for floating-point signals. Obeys the `Weights` parameter for fixed-point signals.

### Dependencies

This port appears only when you set the `Output filter weights` parameter to `On`.

Data Types: `single` | `double` | `fixed point`

## Parameters

### Main Tab

**Algorithm** — Select algorithm

`LMS` (default) | `Normalized LMS` | `Sign-Error LMS` | `Sign-Data LMS` | `Sign-Sign LMS`

Choose the algorithm used to calculate the filter weights.

**Filter length** — Filter length

`32` (default) | `scalar`

Enter the length of the FIR filter weights vector.

**Specify step size via** — Specify step size via

`Dialog` (default) | `Input port`

- `Dialog` -- Specify step size by using the **Step size ( $\mu$ )** parameter.
- `Input port` -- Specify step size by using the `Step-size` port.

**Step size ( $\mu$ )** — Step size

`0.1` (default) | `positive scalar`

Enter the step size  $\mu$ . For convergence of the normalized LMS equations,  $0 < \mu < 2$ .

**Tunable:** Yes

### Dependencies

This parameter appears only when you set the `Specify step size via` parameter to `Dialog`.

**Leakage factor (0 to 1)** — Leakage factor

`1.0` (default) | `scalar`

Enter the leakage factor,  $0 < 1 - \mu\alpha \leq 1$ .



**Tunable:** Yes

**Initial value of filter weights** — Initial value of filter weights

0 (default) | vector | scalar

Enter the initial filter weights  $\mathbf{w}(0)$  as a vector or a scalar. When you enter a scalar, the block uses the scalar value to create a vector of filter weights. This vector length is equal to the filter length and all of its values are equal to the scalar value.

**Adapt port** — Enable Adapt port

on (default) | off

Select this check box to enable the **Adapt** input port.

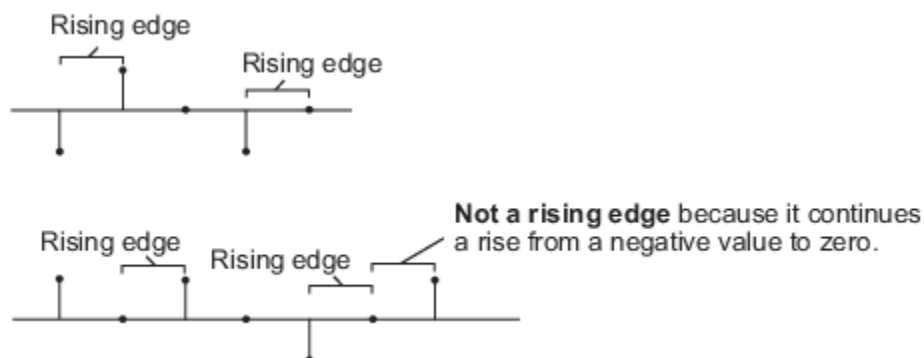
**Reset port** — Reset port

None (default) | Rising edge | Falling edge | Either edge | Non-zero sample

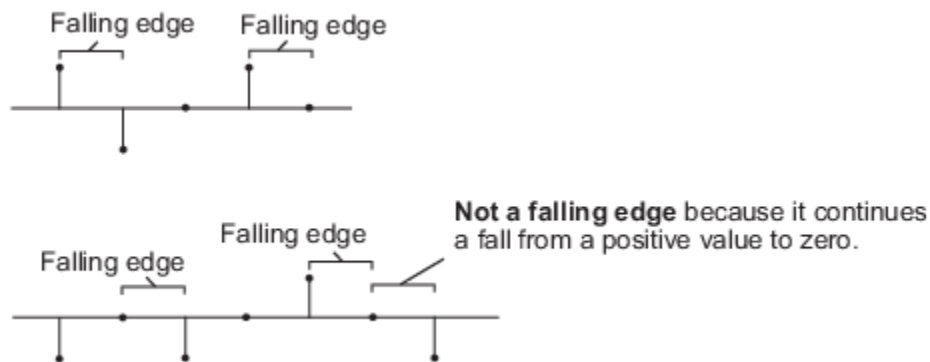
When you want to reset the value of filter weights to their initial values, use the **Reset port** parameter. The reset signal must be the same rate as the data signal input.

Select **None** to disable the **Reset** port. To enable the **Reset** port, select one of the following from the list:

- **Rising edge** — Triggers a reset operation when the **Reset** input does one of the following:
  - Rises from a negative value to a positive value or zero
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure)



- **Falling edge** — Triggers a reset operation when the **Reset** input does one of the following:
  - Falls from a positive value to a negative value or zero
  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero (see the following figure)



- **Either edge** — Triggers a reset operation when the **Reset** input is a **Rising edge** or **Falling edge**
- **Non-zero sample** — Triggers a reset operation at each sample time that the Reset input is not zero

**Output filter weights** — Output filter weights

on (default) | off

Select the **Output filter weights** parameter to export the filter weights from the **Wts** port. For each iteration, the block outputs the current updated filter weights from this port.

### Data Type Tab

**Rounding mode** — Method of rounding operation

Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Specify the rounding mode for fixed-point operations as one of the following:

- Floor
- Ceiling
- Convergent
- Nearest
- Round
- Simplest
- Zero

For more details, see rounding mode.

**Saturate on integer overflow** — Method of overflow action

off (default) | on

When you select this parameter, the block saturates the result of its fixed-point operation. When you clear this parameter, the block wraps the result of its fixed-point operation. For details on saturate and wrap, see overflow mode for fixed-point operations.

**Parameters** — Parameters

Same word length as first input (default) | Specify word length | Binary point scaling

Choose how you specify the word length and the fraction length of the leakage factor and step size:

- **Same word length as first input** -- The word length of the leakage factor and step size match that of the first input to the block. In this mode, the fraction length of the leakage factor and step size is automatically set to the binary-point only scaling that provides you with the best precision possible given the value and word length of the coefficients.
- **Specify word length** -- You can enter the word length of the leakage factor and step size, in bits. In this mode, the fraction length of the leakage factor and step size is automatically set to the binary-point only scaling that provides you with the best precision possible given the value and word length of the coefficients.
- **Binary point scaling** -- You can enter the word length and the fraction length of the leakage factor and step size, in bits. The leakage factor and the step size must have the same word length, but the fraction lengths can differ.

For the **Specify step size via** parameter, if you choose **Input port**, the word length of the leakage factor is the same as the word length of the step size input at the **Step-size** port. The fraction length of the leakage factor is automatically set to the best precision possible based on the word length of the leakage factor.

#### Dependencies

This parameter is visible only if you set the **Specify step size via** parameter to **Dialog**

**Weights** — Word and fraction length of filter weights

Same as first input (default) | Binary point scaling

Choose how you specify the word length and fraction length of the filter weights of the block:

- **Same as first input** -- The word length and fraction length of the filter weights match those of the first input to the block.
- **Binary point scaling** -- You can enter the word length and the fraction length of the filter weights, in bits.

**Product and quotient** — Word and fraction length of product and quotient

Same as first input (default) | Binary point scaling

Choose how you specify the word length and fraction length of  $\mathbf{u}'\mathbf{u}$ ,  $\mathbf{W}'\mathbf{u}$ ,  $\mu \cdot e$ ,  $Q \cdot \mathbf{u}$ , and the quotient,  $Q$ . Here,  $\mathbf{u}$  is the input vector,  $\mathbf{W}$  is the vector of filter weights,  $\mu$  is the step size,  $e$  is the error, and  $Q$  is the quotient, which is defined as  $Q = \frac{\mu \cdot e}{\mathbf{u}'\mathbf{u}}$

- **Same as first input** -- The word length and fraction length of these quantities match those of the first input to the block.
- **Binary point scaling** -- You can enter the word length and the fraction length of these quantities, in bits. The word length of the quantities must be the same, but the fraction lengths can differ.

**Accumulator** — Word and fraction lengths of accumulators

Same as `first input` (default) | `Binary point scaling`

Use this parameter to specify how you want to designate the word and fraction lengths of the accumulators for the  $\mathbf{u}'\mathbf{u}$  and  $\mathbf{W}'\mathbf{u}$  operations.

---

**Note** Do not use this parameter to designate the word and fraction lengths of the accumulator for the  $Q \cdot \mathbf{u}$  operation. The accumulator data type for this quantity is automatically set to be the same as the product data type. The minimum, maximum, and overflow information for this accumulator is logged as part of the product information. Autoscaling treats this product and accumulator as one data type.

---

- `Same as first input` -- These characteristics match those of the input to the block.
- `Binary point scaling` -- You can enter the word length and the fraction length of the accumulators, in bits. The word length of both the accumulators must be the same, but the fraction lengths can differ.

For illustrations depicting the use of the accumulator data type in this block, see “Fixed Point” on page 2-887 and “Multiplication Data Types”.

**Lock data type settings against changes by the fixed-point tools** — Prevent fixed-point tools from overriding data types

`off` (default) | `on`

Select this parameter to prevent the fixed-point tools from overriding the data types you specify in the block dialog box.

## Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | <code>double</code>   <code>fixed point</code>   <code>integer</code>   <code>single</code> |
| <b>Direct Feedthrough</b>       | <code>no</code>   |
| <b>Multidimensional Signals</b> | <code>no</code>   |
| <b>Variable-Size Signals</b>    | <code>yes</code>  |
| <b>Zero-Crossing Detection</b>  | <code>no</code>   |

## More About

### LMS Filter Algorithms

When you select LMS for the `Algorithm` parameter, the block calculates the filter weights by using the least mean-square (LMS) algorithm. This algorithm is defined by these equations.

$$\begin{aligned}
 y(n) &= \mathbf{w}^T(n-1)\mathbf{u}(n) \\
 e(n) &= d(n) - y(n) \\
 \mathbf{w}(n) &= \alpha\mathbf{w}(n-1) + f(\mathbf{u}(n), e(n), \mu)
 \end{aligned}$$

The various LMS adaptive filter algorithms available in this block are defined as:

- LMS --

$$f(\mathbf{u}(n), e(n), \mu) = \mu e(n) \mathbf{u}^*(n)$$

- Normalized LMS --

$$f(\mathbf{u}(n), e(n), \mu) = \mu e(n) \frac{\mathbf{u}^*(n)}{\varepsilon + \mathbf{u}^H(n) \mathbf{u}(n)}$$

In Normalized LMS, to overcome potential numerical instability in the update of the weights, a small positive constant,  $\varepsilon$ , has been added in the denominator. For double-precision floating-point input,  $\varepsilon$  is the output of the `eps` function. For single-precision floating-point input,  $\varepsilon$  is the output of `eps("single")`. For fixed-point input,  $\varepsilon$  is 0.

- Sign-Error LMS --

$$f(\mathbf{u}(n), e(n), \mu) = \mu \text{sign}(e(n)) \mathbf{u}^*(n)$$

- Sign-Data LMS --

$$f(\mathbf{u}(n), e(n), \mu) = \mu e(n) \text{sign}(\mathbf{u}(n))$$

where  $\mathbf{u}(n)$  is real.

- Sign-Sign LMS --

$$f(\mathbf{u}(n), e(n), \mu) = \mu \text{sign}(e(n)) \text{sign}(\mathbf{u}(n))$$

where  $\mathbf{u}(n)$  is real.

| Variable          | Description  |
|-------------------|--|
| $n$               | The current time index   |
| $\mathbf{u}(n)$   | The vector of buffered input samples at step $n$   |
| $\mathbf{u}^*(n)$ | The complex conjugate of the vector of buffered input samples at step $n$                              |
| $\mathbf{w}(n)$   | The vector of filter weight estimates at step $n$  |
| $y(n)$            | The filtered output at step $n$  |
| $e(n)$            | The estimation error at step $n$   |
| $d(n)$            | The desired response at step $n$   |
| $\mu$             | The adaptation step size   |
| $\alpha$          | The leakage factor ( $0 < \alpha \leq 1$ )   |
| $\varepsilon$     | A constant that corrects any potential numerical instability that occurs during the update of weights. |

## Version History

Introduced before R2006a

## References

- [1] Hayes, M.H. *Statistical Digital Signal Processing and Modeling*. New York: John Wiley & Sons, 1996.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Generated code relies on the `memcpy` or `memset` function (`string.h`) under certain conditions.

The LMS Filter block supports SIMD code generation using Intel AVX2 technology under these conditions:

- **Algorithm** is set to LMS or Normalized LMS.
- Input signal is real-valued.
- Input signal has a data type of `single` or `double`.

The SIMD technology significantly improves the performance of the generated code.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

### HDL Architecture

By default, the LMS Filter implementation uses a linear sum for the FIR section of the filter.

The LMS Filter implements a tree summation (which has a shorter critical path) under the following conditions:

- The LMS Filter is used with real data.
- The word length of the Accumulator **W'u** data type is at least `ceil(log2(filter length))` bits wider than the word length of the Product **W'u** data type.
- The Accumulator **W'u** data type has the same fraction length as the Product **W'u** data type.

### HDL Block Properties

|                                  |  |
|----------------------------------|--|
| <b>ConstrainedOutputPipeline</b> | Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see "ConstrainedOutputPipeline" (HDL Coder). |
| <b>InputPipeline</b>             | Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "InputPipeline" (HDL Coder).                 |
| <b>OutputPipeline</b>            | Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "OutputPipeline" (HDL Coder).               |

### Complex Data Support

This block supports code generation for complex signals.

### Restrictions

- HDL Coder does not support the Normalized LMS algorithm of the LMS Filter.
- The Reset port supports only Boolean and unsigned inputs.
- The Adapt port supports only Boolean inputs.
- **Filter length** must be greater than or equal to 2.

### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

If the input is fixed point, it must be a signed integer or a signed fixed point value with a power-of-two slope and zero bias.

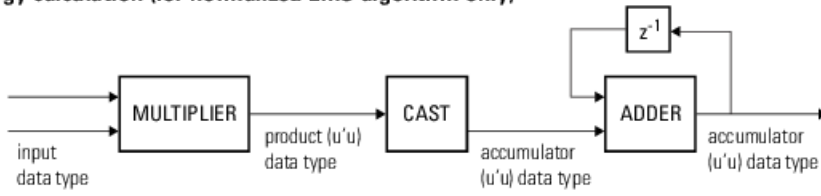
### Fixed-Point Data Types

The following diagrams show the data types used within the LMS Filter block for fixed-point signals. The table summarizes the definitions of variables used in the diagrams.

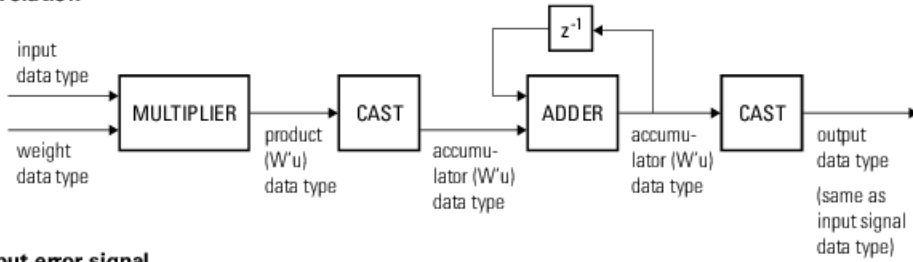
| Variable                            | Definition   |
|-------------------------------------|--|
| <b>u</b>                            | Input vector   |
| <b>W</b>                            | Vector of filter weights   |
| $\mu$                               | Step size  |
| $e$                                 | Error  |
| $Q$                                 | Quotient, $Q = \frac{\mu \cdot e}{\mathbf{u}'\mathbf{u}}$                |
| Product $\mathbf{u}'\mathbf{u}$     | Product data type in Energy calculation diagram                          |
| Accumulator $\mathbf{u}'\mathbf{u}$ | Accumulator data type in Energy calculation diagram                      |
| Product $\mathbf{W}'\mathbf{u}$     | Product data type in Convolution diagram                                 |
| Accumulator $\mathbf{W}'\mathbf{u}$ | Accumulator data type in Convolution diagram                             |
| Product $\mu \cdot e$               | Product data type in Product of step size and error diagram              |
| Product $Q \cdot \mathbf{u}$        | Product and accumulator data type in Weight update diagram. <sup>1</sup> |

<sup>1</sup>The accumulator data type for this quantity is automatically set to be the same as the product data type. The minimum, maximum, and overflow information for this accumulator is logged as part of the product information. Autoscaling treats this product and accumulator as one data type.

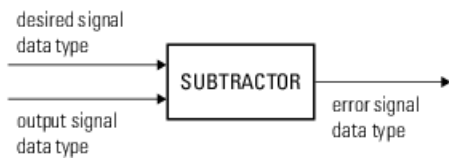
**Energy calculation (for normalized LMS algorithm only)**



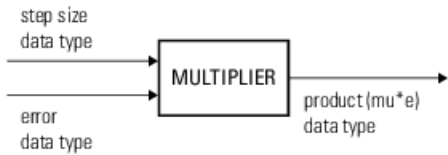
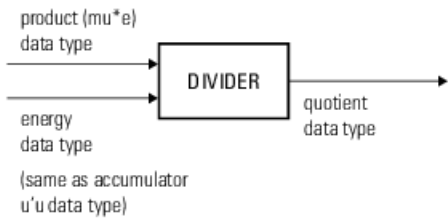
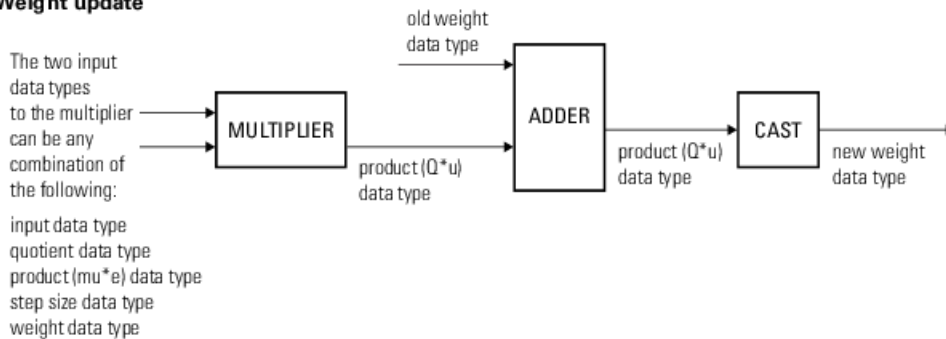
**Convolution**



**Output error signal**





**Product of step size and error (for LMS and Sign-Data LMS algorithms only)****Quotient (for normalized LMS only)****Weight update**

You can set the data type of the parameters, weights, products, quotient, and accumulators in the block mask. Fixed-point inputs, outputs, and mask parameters of this block must have these characteristics:

- The input signal and the desired signal must have the same word length, but their fraction lengths can differ.
- The step size and leakage factor must have the same word length, but their fraction lengths can differ.
- The output signal and the error signal have the same word length and the same fraction length as the desired signal.
- The quotient and the product output of the  $\mathbf{u}'\mathbf{u}$ ,  $\mathbf{W}'\mathbf{u}$ ,  $\mu \cdot e$ , and  $Q \cdot \mathbf{u}$  operations must have the same word length, but their fraction lengths can differ.
- The accumulator data type of the  $\mathbf{u}'\mathbf{u}$  and  $\mathbf{W}'\mathbf{u}$  operations must have the same word length, but their fraction lengths can differ.

The output of the multiplier is in the product output data type if at least one of the inputs to the multiplier is real. If both of the inputs to the multiplier are complex, the result of the multiplication is in the accumulator data type. For details, see “Multiplication Data Types”.

## **See Also**

### **Blocks**

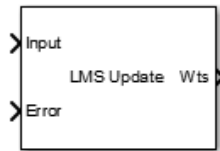
[LMS Update](#) | [RLS Filter](#) | [Block LMS Filter](#) | [Fast Block LMS Filter](#)

### **Topics**

["Noise Cancellation in Simulink Using Normalized LMS Adaptive Filter"](#)

# LMS Update

Estimate weights of LMS adaptive filter



## Libraries:

DSP System Toolbox / Filtering / Adaptive Filters

## Description

The LMS Update block estimates the weights of an LMS adaptive filter. Specify data and error as inputs and the block computes the filter weights based on the algorithm specified in the **Algorithm** parameter. For more details on the algorithms, see “Algorithms” on page 2-896.

You can configure the block to operate as a tapped-delay line FIR filter or as an adaptive linear combiner using the **Adaptive filter mode** parameter. The block can also adapt multiple filters independently when you specify the **Number of adaptive filters** parameter to a value greater than 1.

You can use this block to compute the adaptive filter weights in applications such as system identification, inverse modeling, and filtered-x LMS algorithms, which are used in acoustic noise cancellation. For more details, see “References” on page 2-897.

## Input/Output Ports

### Input

**Input** — Data input

scalar | column vector

Data input to the adaptive filter, specified as one of these:

- Scalar -- When you set the **Adaptive filter mode** parameter to Tapped delay-line FIR filter, the input to the block should be a scalar.

The block algorithm buffers the input samples to generate the vector  $\mathbf{u}(n) = [u(n), u(n-1), \dots, u(n-L+1)]$ , where  $L$  is the length of the adaptive filter and  $n$  is the time index. The block algorithm then passes the vector through the filter.

- Column vector of length  $L$  -- When you set the **Adaptive filter mode** parameter to Adaptive linear combiner, the block does not buffer the input samples, and the input to the block should be in the form of a column vector. The block multiplies the vector with the filter weights to generate the filter output. In this case, there is no relationship between the input samples.

When you set the **Number of adaptive filters** parameter to a value greater than 1, the block assumes that all filters run on the same input.

When you set the **Algorithm** parameter to Sign-Error LMS, Sign-Data LMS, or Sign-Sign LMS, the data input through the **Input** port must be real.

Data Types: `single` | `double`

**Error** — Error

scalar | column vector

Specify the error between the output signal and the desired signal as a:

- Scalar when the number of adaptive filters  $N = 1$ .
- Column vector of length  $N$  when  $N > 1$ .

When you set the **Algorithm** parameter to `Sign–Error LMS`, `Sign–Data LMS`, or `Sign–Sign LMS`, the error input through the **Error** port must be real.

Data Types: `single` | `double`

**Mu** — Filter adaptation step size

0.1 (default) | scalar in the range [0,1]

Specify the filter adaptation step size as a scalar in the range [0,1].

**Dependency**

To enable this port, set **Step size source** to `Input` port.

Data Types: `single` | `double`

**Adapt** — Update filter weights

real scalar | vector

The input to this port can be one of these:

- Scalar -- If the number of adaptive filters ( $N$ ) is greater than 1, then the block updates the filter weights for all filters if the value at the port is nonzero and does not update if the value at the port is 0.
- Vector of length  $N$  -- If the number of adaptive filters ( $N$ ) is greater than 1, then the block determines whether to update each filter during run time depending on the value of the corresponding element in the **Adapt** vector input. When the **Adapt** port input element is not 0, the block updates the filter weights. When the input is 0, the filter weights do not change.

For an example, see “Adapt Multiple Filters Using LMS Update block”.

**Dependency**

To enable this port, select the **Enable adapt input** parameter on the block dialog.

Data Types: `single` | `double` | `Boolean` | `int16` | `int32` | `int64` | `int8` | `uint16` | `uint32` | `uint64` | `uint8`

**Reset** — Reset filter weights

real scalar

When the input to this port is not 0, the block resets the filter weights to their initial values. When the input to this port is 0, the filter weights do not change.

**Dependency**

To enable this port, select the **Enable reset input** parameter on the block dialog.

Data Types: `single` | `double` | `Boolean` | `int16` | `int32` | `int64` | `int8` | `uint16` | `uint32` | `uint64` | `uint8`

## Output

**Wts** — Filter weights

row vector | matrix

The block outputs the filter weights as a 1-by- $L$  row vector or a  $N$ -by- $L$  matrix, where  $N$  is the number of adaptive filters, and  $L$  is the length of each filter.

Data Types: `single` | `double`

## Parameters

**Algorithm** — LMS adaptive algorithm

`LMS` (default) | `Normalized LMS` | `Sign-Error LMS` | `Sign-Data LMS` | `Sign-Sign LMS`

The block uses one of the listed algorithms to compute the filter weights. For more details on the algorithms, see “Algorithms” on page 2-896.

**Filter length** — Length of the filter

32 (default) | positive integer

Specify the length of each adaptive filter  $L$  as a positive integer.

When there are  $N$  adaptive filters, the weights vector the block generates through the **Wts** output port is an  $N$ -by- $L$  matrix.

**Number of adaptive filters** — Number of adaptive filters

1 (default) | positive integer

Specify the number of adaptive filters  $N$  the block supports as a positive integer.

When you specify this value to be an integer greater than 1, each filter can be chosen to adapt independently during run time by providing the **Adapt** input as a logical vector of length  $N$ . All filters are assumed to run on the same input and have a common step size, leakage factor, and reset input.

The weights the block generates through the **Wts** output port is an  $N$ -by- $L$  matrix, where  $L$  is the length of each filter.

For an example that shows how to adapt multiple filters, see “Adapt Multiple Filters Using LMS Update block”.

**Step size source** — Method to specify the step size

Property (default) | Input port

- Property — Specify the filter adaptation size using the **Step size (mu)** parameter.
- Input port — Pass filter adaptation size using the **Mu** input port.

**Step size ( $\mu$ )** — Size of the adaptation step

0.1 (default) | nonnegative scalar

**Step size ( $\mu$ )** indicates the amount by which the filter weights are updated in each iteration. Choose an optimal step size so that the filter is stable and the convergence speed is optimal.

This parameter is tunable. You can change its value even during the simulation.

**Tunable:** Yes

**Dependency**

To enable this parameter, set **Step size source** to Property.

**Leakage factor (0 to 1)** — Leakage factor

1.0 (default) | real scalar in the range [0,1]

**Leakage factor (0 to 1)** prevents unbounded growth of the filter coefficients by reducing the drift of the coefficients from their optimum values. A leakage factor of 1.0 indicates no leakage. If you encounter coefficient drift, that is, large fluctuation about the optimum solution, decrease the leakage factor until the coefficient fluctuation becomes small.

This parameter is tunable. You can change its value even during the simulation.

**Tunable:** Yes

**Initial value of filter weights** — Initial value of filter weights

0 (default) | real scalar | vector | matrix

Specify the initial value of filter weights as one of the following:

- Scalar
- Vector of size 1-by- $L$  --  $N$  is set to 1
- Matrix of size  $N$ -by- $L$  --  $N > 1$  and  $L > 1$

where  $N$  is the number of adaptive filters specified in **Number of adaptive filters** and  $L$  is the length of each filter specified in **Filter length**.

This parameter specifies the initial value of the filter weights  $\mathbf{w}(n-1)$ . The block uses this value to compute the weights  $\mathbf{w}(n)$ , when  $n = 1$ . For more information, see "Algorithms" on page 2-896.

**Adaptive filter mode** — Adaptive filter mode

Tapped delay-line FIR filter (default) | Adaptive linear combiner

Specify the adaptive filter mode as one of the following:

- **Tapped delay-line FIR filter** -- The block assumes the variable portion of the adaptive filter to be a tapped delay-line FIR filter. In this mode, the input samples are related using this equation:

$$\mathbf{u}(n) = [u(n), u(n-1), \dots, u(n-L+1)]$$

where  $n$  is the current time index and  $L$  is the length of the adaptive filter specified in the **Filter length** parameter.

The block accepts scalar input samples in this mode. The block algorithm buffers the samples to form the  $\mathbf{u}(n)$  vector.

- **Adaptive linear combiner** -- The block does not buffer the input samples, and the input you provide to the block must be a column vector equal in length to the **Filter length** parameter. In this case, there is no relationship between the input samples.

For more details, see “Algorithms” on page 2-896.

**Enable adapt input** — Update the filter weights

off (default) | on

When you select this check box, the **Adapt** input port appears on the block. When the input to this port is greater than 0, the block updates the filter weights. When the input to this port is less than or equal to 0, the filter weights do not change.

**Enable reset input** — Reset the filter weights

off (default) | on

When you select this check box, the **Reset** input port appears on the block. When the input to this port is greater than 0, the block resets the filter weights to their initial values. When the input to this port is less than or equal to 0, the filter weights do not change.

**Simulate using** — Type of simulation to run

Code generation (default) | Interpreted execution

Specify the type of simulation to run as one of the following:

- **Code generation** -- Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than **Interpreted execution**.
- **Interpreted execution** -- Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than **Code generation**.

## Block Characteristics

|                                 |                 |
|---------------------------------|-----------------|
| <b>Data Types</b>               | double   single |
| <b>Direct Feedthrough</b>       | no              |
| <b>Multidimensional Signals</b> | no              |
| <b>Variable-Size Signals</b>    | no              |
| <b>Zero-Crossing Detection</b>  | no              |

## Algorithms

In a closed loop adaptive filter, the weights or coefficients of the filter are adjusted until the error is minimized. The error is the difference between the filter output and the desired signal. In this block, the error signal is specified through the **Error** input port.

The block computes filter weight estimates using  $\mathbf{w}(n) = \alpha \mathbf{w}(n-1) + f(\mathbf{u}(n), e(n), \mu)$ .

The function  $f(\mathbf{u}(n), e(n), \mu)$  is defined according to the LMS algorithm you specify through the **Algorithm** parameter:

- **LMS** —  $f(\mathbf{u}(n), e(n), \mu) = \mu e(n) \mathbf{u}^*(n)$
- **Normalized LMS** —  $f(\mathbf{u}(n), e(n), \mu) = \mu e(n) \frac{\mathbf{u}^*(n)}{\varepsilon + \mathbf{u}^H(n) \mathbf{u}(n)}$

In the Normalized LMS algorithm,  $\varepsilon$  is a small positive constant that overcomes the potential numerical instability in the update of weights.

For double-precision floating-point input,  $\varepsilon$  is the output of the `eps` function. For single-precision floating-point input,  $\varepsilon$  is the output of `eps("single")`. For fixed-point input,  $\varepsilon$  is 0.

- **Sign-Error LMS** —  $f(\mathbf{u}(n), e(n), \mu) = \mu \text{sign}(e(n)) \mathbf{u}^*(n)$
- **Sign-Data LMS** —  $f(\mathbf{u}(n), e(n), \mu) = \mu e(n) \text{sign}(\mathbf{u}(n))$ , where  $\mathbf{u}(n)$  is real
- **Sign-Sign LMS** —  $f(\mathbf{u}(n), e(n), \mu) = \mu \text{sign}(e(n)) \text{sign}(\mathbf{u}(n))$ , where  $\mathbf{u}(n)$  is real

In the equations:

- $n$  — The current time index.
- $\mathbf{u}(n)$  — The vector of input samples at step  $n$ .

When you set the **Adaptive filter mode** parameter to `Tapped delay-line FIR filter`, the input to the block is a scalar value and the block buffers the input samples to generate the  $\mathbf{u}(n)$  vector.

When you set the **Adaptive filter mode** parameter to `Adaptive linear combiner`, the block does not buffer the samples. The input to the block should be in the form of a column vector. In this case, there is no relationship between the input samples.

- $\mathbf{u}^*(n)$  — The complex conjugate of the vector of input samples at step  $n$ .
- $\mathbf{w}(n)$  — The vector of filter weight estimates at step  $n$ .
- $e(n)$  — The estimation error at step  $n$ .
- $\mu$  — The adaptation step size.
- $\alpha$  — The leakage factor ( $0 \leq \alpha \leq 1$ ).

### Multifilter Support

You can update the weights of multiple adaptive filters using the same LMS Update block. To enable this, specify the **Number of adaptive filters** parameter to be a value greater than 1. You can then adapt each filter independently during run time by providing the input at the **Adapt** port as a logical vector of length  $N$ . The block algorithm assumes that all filters run on the same input, adapt using the same algorithm, and have the same filter length, step size, leakage factor, and reset input.



## Version History

Introduced in R2016b

## References

- [1] Madisetti, Vijay, and Douglas Williams. "Introduction to Adaptive Filters." *The Digital Signal Processing Handbook*. Boca Raton, FL: CRC Press, 1999.
- [2] Akhtar, M. T., M. Abe, M. Kawamata. "Modified-filtered-x LMS algorithm based active noise control systems with improved online secondary-path modeling." IEEE Symposium on Circuits and Systems, 2004.
- [3] Orfanidis, Sophocles J. *Optimum Signal Processing. An Introduction*. 2nd Ed. Englewood Cliffs, NJ: Prentice-Hall, 1996.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

## See Also

### Blocks

LMS Filter | RLS Filter | Block LMS Filter | Fast Block LMS Filter

### Objects

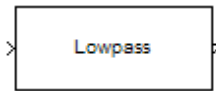
`dsp.AdaptiveLatticeFilter` | `dsp.BlockLMSFilter` | `dsp.LMSFilter` | `dsp.RLSFilter`

### Topics

"Noise Cancellation in Simulink Using Normalized LMS Adaptive Filter"

# Lowpass Filter

Design FIR or IIR lowpass filter



## Libraries:

DSP System Toolbox / Filtering / Filter Designs

## Description

The Lowpass Filter block independently filters each channel of the input signal over time using the filter design specified by the block parameters. You can control whether the block implements an IIR or FIR lowpass filter using the **Filter type** parameter. You can specify the passband and stopband edge frequencies in Hz or in normalized frequency units.

This block supports ARM Cortex code generation. Under certain conditions, this block also supports SIMD code generation. For details, see “Code Generation” on page 2-903.

## Ports

### Input

**Port\_1** — Input signal to filter

column vector | matrix

Input signal, specified as a real- or complex-valued column vector or matrix. If the input signal is a matrix, each column of the matrix is treated as an independent channel. The number of rows in the input signal denotes the channel length.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fixed point

### Output

**Port\_1** — Filtered signal

vector | matrix

Filtered signal, returned as a vector or matrix. The output has the same size, data type, and complexity characteristics as the input.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fixed point

## Parameters

### Main

**Filter type** — FIR or IIR filter

FIR (default) | IIR

Specify whether the block implements an FIR lowpass filter or an IIR lowpass filter.

**Design minimum order filter** — Design filter with minimum order

on (default) | off

When you select this check box, the block designs a filter with the minimum order and the specified passband, stopband frequency, passband ripple, and stopband attenuation.

When you clear this check box, you can specify the **Filter order** as a positive integer.

**Filter order** — Order of lowpass filter

50 (default) | positive integer

Specify the filter order of the lowpass filter as a positive integer.

#### Dependencies

To enable this parameter, clear the **Design minimum order filter** check box.

**Passband edge frequency (Hz)** — Passband edge frequency

8e3 (default) | real positive scalar

Specify the passband edge frequency of the lowpass filter as a real positive scalar in Hz or in normalized frequency units.

If you set the **Sample rate mode** parameter to:

- **Specify on dialog or Inherit from input port** -- The value of the passband edge frequency is in Hz and must be less than half the value of the input sample rate.
- **Use normalized frequency (0 to 1)** -- The value of the passband edge frequency is in normalized frequency units. The value must be a positive scalar less than 1.0.

**Stopband edge frequency (Hz)** — Stopband edge frequency

12e3 (default) | real positive scalar

Specify the stopband edge frequency of the lowpass filter as a real positive scalar in Hz or in normalized frequency units.

If you set the **Sample rate mode** parameter to:

- **Specify on dialog or Inherit from input port** -- The value of the stopband edge frequency is in Hz and must be less than half the value of the input sample rate.
- **Use normalized frequency (0 to 1)** -- The value of the stopband edge frequency is in normalized frequency units. The value must be a positive scalar less than 1.0.

#### Dependencies

To enable this parameter, select the **Design minimum order filter** check box.

**Maximum passband ripple (dB)** — Maximum passband ripple

0.1 (default) | real positive scalar

Specify the maximum ripple of the filter response in the passband as a real positive scalar in dB.

**Minimum stopband attenuation (dB)** — Minimum stopband attenuation

80 (default) | real positive scalar

Specify the minimum attenuation in the stopband as a real positive scalar in dB.

**Sample rate mode** — Mode to specify the input sample rate

Specify on dialog (default) | Inherit from input port | Use normalized frequency (0 to 1)

Specify the input sample rate using one of these options:

- **Specify on dialog** -- Specify the input sample rate in the block dialog box using the **Input sample rate (Hz)** parameter.
- **Inherit from input port** -- The block inherits the sample rate from the input signal.
- **Use normalized frequency (0 to 1)** -- Specify the passband edge and the stopband edge frequencies in normalized frequency units (0 to 1).

**Input sample rate (Hz)** — Input sample rate

44100 (default) | scalar

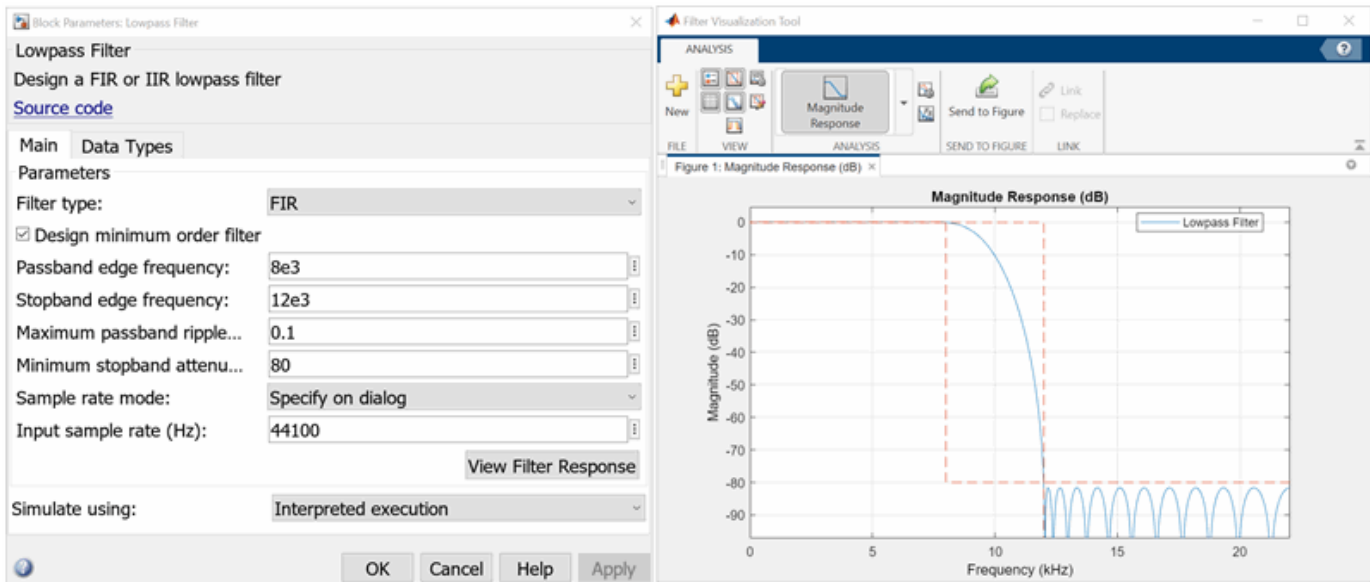
Specify the input sample rate as a scalar in Hz.

### **Dependencies**

To enable this parameter, set the **Sample rate mode** parameter to **Specify on dialog**.

**View Filter Response** — Open Filter Visualization Tool  
button

Opens the Filter Visualization Tool (`fvtool`) and displays the magnitude and phase response of the Lowpass Filter. The response is based on the block dialog box parameters. Changes made to these parameters update FVTool.



To update the magnitude response while FVTool is running, modify the dialog box parameters and click **Apply**.

**Simulate using** — Type of simulation to run

Interpreted execution (default) | Code generation

Type of simulation to run:

- Interpreted execution (default)

Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than Code generation.

- Code generation

Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than Interpreted execution.

## Data Types

**Rounding mode** — Rounding method

Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero


Rounding method for the output fixed-point operations.

**Coefficients** — Coefficient data type

fixdt(1,16) (default) | fixdt(1,16,0) | <data type expression>

Fixed-point data type of the coefficients, specified as one of the following:

- `fixdt(1,16)` — Signed fixed-point data type of word length 16, with binary point scaling. The block determines the fraction length automatically from the coefficient values in such a way that the coefficients occupy maximum representable range without overflowing.
- `fixdt(1,16,0)` — Signed fixed-point data type of word length 16 and fraction length 0. You can change the fraction length to any other integer value.
- `<data type expression>` — Specify the data type using an expression that evaluates to a data type object, for example, numeric type (`fixdt([ ],16,15)`). Specify the sign mode of this data type as `[ ]` or `true`.
- Refresh Data Type — Refresh to the default data type.

Click the **Show data type assistant** button  to display the data type assistant, which helps you set the data type. For more information, see “Specify Data Types Using Data Type Assistant” (Simulink).

## Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | double   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no                                      |
| <b>Multidimensional Signals</b> | no                                      |
| <b>Variable-Size Signals</b>    | yes                                     |
| <b>Zero-Crossing Detection</b>  | no                                      |

## Algorithms

### FIR Lowpass Filter

For the minimum order design, the algorithm uses generalized Remez FIR filter design algorithm. For the specified order design, the algorithm uses the constrained equiripple FIR filter design algorithm. The designed filter is then implemented as a linear phase Type-1 filter with a `Direct form` structure.

### IIR Lowpass Filter

In the IIR configuration, the algorithm uses the elliptic design method to compute the SOS and scale values required to meet the filter design specifications. The algorithm uses the SOS and scale values to setup a `Direct form I` biquadratic IIR filter, which forms the basis of the IIR version of the lowpass filter.

## Version History

### Introduced in R2015b

#### R2023a: Support for normalized frequencies

When you set the **Sample rate mode** parameter to `Use normalized frequency (0 to 1)`, you can specify the passband and stopband edge frequencies in normalized frequency units (0 to 1).

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

The Lowpass Filter block supports ARM Cortex code generation. To learn more about ARM Cortex code generation, see “Code Generation for ARM Cortex-M and ARM Cortex-A Processors”.

The Lowpass Filter block also supports SIMD code generation using Intel AVX2 technology under these conditions:

- **Filter type** is set to FIR.
- Input signal has a data type of `single` or `double`.

The SIMD technology significantly improves the performance of the generated code.

### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

## See Also

### Blocks

Highpass Filter

### Objects

`dsp.LowpassFilter` | `dsp.HighpassFilter`

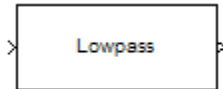
### Topics

“Filter Frames of a Noisy Sine Wave Signal in Simulink”

“Lowpass IIR Filter Design in Simulink”

## Lowpass Filter (Obsolete)

Design lowpass filter



## Compatibility

---

**Note** The Lowpass Filter (Obsolete) block has been replaced by the Lowpass Filter block. Existing instances of the Lowpass Filter (Obsolete) block will continue to operate. For new models, use the Lowpass Filter block.

---

## Library

Filtering / Filter Designs

dspfdesign

## Description

This block brings the filter design capabilities of the `filterBuilder` function to the Simulink environment.

## Parameters

### View filter response

This button opens the Filter Visualization Tool (**FVTool**) from the Signal Processing Toolbox product. You can use the tool to display:

- Magnitude response, phase response, and group delay in the frequency domain.
- Impulse response and step response in the time domain.
- Pole-zero information.

The tool also helps you evaluate filter performance by providing information about filter order, stability, and phase linearity. For more information on FVTool, see the Signal Processing Toolbox documentation.

### Filter Specifications

In this group, you specify your filter format, such as the impulse response and the filter order.

### Impulse response

Select either FIR or IIR from the drop-down list. FIR is the default. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly.



---

**Note** The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

---

### Order mode

Select **Minimum** (the default) or **Specify**. Selecting **Specify** enables the **Order** option so you can enter the filter order. When you set the **Impulse response** to **IIR**, you can specify different numerator and denominator orders. To specify a different denominator order, you must select the **Denominator order** check box.

### Order

Enter the filter order. This option is enabled only if you set the **Order mode** to **Specify**.

### Denominator order

Select this check box to specify a different denominator order. This option is enabled only if you set the **Impulse response** to **IIR** and the **Order mode** to **Specify**.

### Filter type

Select **Single-rate**, **Decimator**, **Interpolator**, or **Sample-rate converter**. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, the block specifies a single-rate filter.

- Selecting **Decimator** or **Interpolator** activates the **Decimation Factor** or the **Interpolation Factor** options respectively.
- Selecting **Sample-rate converter** activates both factors.

### Decimation Factor

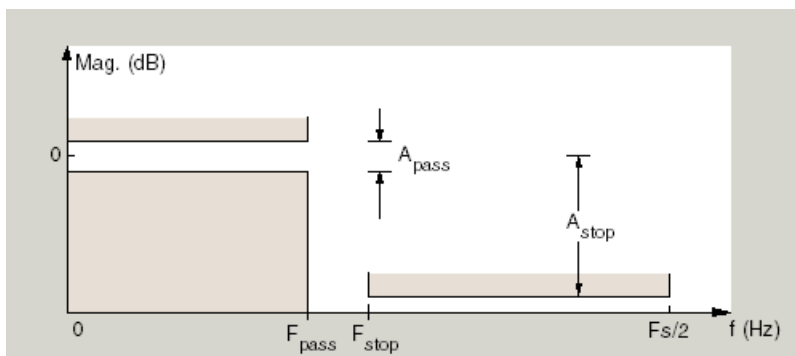
Enter the decimation factor. This option is enabled only if the **Filter type** is set to **Decimator** or **Sample-rate converter**. The default value is 2.

### Interpolation Factor

Enter the interpolation factor. This option is enabled only if the **Filter type** is set to **Interpolator** or **Sample-rate converter**. The default value is 2.

### Frequency Specifications

The parameters in this group allow you to specify your filter response curve. Graphically, the filter specifications look similar to the one shown in the following figure.



In the figure, regions between specification values such as  $F_{pass}$  and  $F_{stop}$  represent transition regions where the filter response is not constrained.

### Frequency constraints

When **Order mode** is *Specify*, select the filter features that the block uses to define the frequency response characteristics. The list contains the following options, when available for the filter specifications.

- **Passband and Stopband frequencies** — Define the filter by specifying the frequencies for the edges for the stop- and passbands.
- **Passband frequency** — Define the filter by specifying the edge of the passband.
- **Stopband frequency** — Define the filter by specifying the edge of the stopband.
- **Halfband power (3dB) frequency** — Define the filter response by specifying the location of the 3 dB point. The 3 dB point is the frequency for the point three decibels below the passband value.
- **Cutoff (6dB) frequency** — For FIR filters, define the filter response by specifying the location of the 6 dB point. The 6 dB point is the frequency for the point six decibels below the passband value.

### Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select **Normalized (0–1)** to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—Hz, kHz, MHz, or GHz. Selecting one of the unit options enables the **Input sample rate** parameter.

### Input sample rate

**Input sample rate**, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

### Passband frequency

Enter the frequency at the end of the passband. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

### Stopband frequency

Enter the frequency at the start of the stopband. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

### Half power (3dB) frequency

When **Frequency constraints** is **Half power (3dB) frequency**, specify the frequency of the 3 dB point. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

### Cutoff (6dB) frequency

When **Frequency constraints** is **Cutoff (6dB) frequency**, specify the frequency of the 6 dB point. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

### Magnitude Specifications

Parameters in this group specify the filter response in the passbands and stopbands.

### Magnitude constraints

This option is only available when you specify the order of your filter design. Depending on the setting of the **Frequency constraints** parameter, some combination of the following options will

be available for the **Magnitude constraints** parameter: Unconstrained, and Passband ripple and stopband attenuation.

### **Magnitude units**

Specify the units for any parameter you provide in magnitude specifications. From the drop-down list, select one of the following options:

- Linear — Specify the magnitude in linear units.
- dB — Specify the magnitude in decibels (default)

### **Passband ripple**

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels.

### **Stopband attenuation**

Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels.

### **Algorithm**

The parameters in this group allow you to specify the design method and structure of your filter.

### **Design Method**

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is usually Elliptic, and the default FIR method is Equiripple.

### **Scale SOS filter coefficients to reduce chance of overflow**

Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

### **Design Options**

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

#### **Density factor**

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 16 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

#### **Phase constraint**

Specify the phase constraint of the filter as Linear, Maximum, or Minimum.

#### **Minimum order**

When you select this parameter, the design method determines and design the minimum order filter to meet your specifications. Some filters do not provide this parameter. Select

Any, Even, or Odd from the drop-down list to direct the design to be any minimum order, or minimum even order, or minimum odd order.

### Match Exactly

Specifies that the resulting filter design matches either the passband or stopband or both bands when you select passband or stopband or both from the drop-down list.

### Stopband Shape

Stopband shape lets you specify how the stopband changes with increasing frequency. Choose one of the following options:

- **Flat** — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.
- **Linear** — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.
- **1/f** — Specifies that the stopband attenuation changes exponentially as the frequency increases, where  $f$  is the frequency. Set the power (exponent) for the decay in **Stopband decay**.

### Stopband Decay

When you set Stopband shape, Stopband decay specifies the amount of decay applied to the stopband. The following conditions apply to Stopband decay based on the value of Stopband Shape:

- When you set **Stopband shape** to **Flat**, **Stopband decay** has no effect on the stopband.
- When you set **Stopband shape** to **Linear**, enter the slope of the stopband in units of dB/rad/s. The block applies that slope to the stopband.
- When you set **Stopband shape** to **1/f**, enter a value for the exponent  $n$  in the relation  $(1/f)^n$  to define the stopband decay. The block applies the  $(1/f)^n$  relation to the stopband to result in an exponentially decreasing stopband attenuation.

## Filter Implementation

### Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure, and IIR filters use direct-form II filters with SOS.

### Use basic elements to enable filter customization

Select this check box to implement the filter as a subsystem of basic Simulink blocks. Clear the check box to implement the filter as a high-level subsystem. By default, this check box is cleared.

The high-level implementation provides better compatibility across various filter structures, especially filters that would contain algebraic loops when constructed using basic elements. On the other hand, using basic elements enables the following optimization parameters:

- **Optimize for zero gains** — Terminate chains that contain Gain blocks with a gain of zero.
- **Optimize for unit gains** — Remove Gain blocks that scale by a factor of one.
- **Optimize for delay chains** — Substitute delay chains made up of  $n$  unit delays with a single delay by  $n$ .
- **Optimize for negative gains** — Use subtraction in Sum blocks instead of negative gains in Gain blocks.

### Optimize for unit scale values

Select this check box to scale unit gains between sections in SOS filters. This parameter is available only for SOS filters (Impulse response: IIR).

### Input processing

Specify how the block should process the input. The available options may vary depending on the settings of the **Filter Structure** and **Use basic elements for filter customization** parameters. You can set this parameter to one of the following options:

- **Columns as channels (frame based)** — When you select this option, the block treats each column of the input as a separate channel.
- **Elements as channels (sample based)** — When you select this option, the block treats each element of the input as a separate channel.

---

**Note** The **Inherited** (this choice will be removed – see release notes) option will be removed in a future release. See “Frame-Based Processing” in the *DSP System Toolbox Release Notes* for more information.

---

### Rate options

When the **Filter type** parameter specifies a multirate filter, select the rate processing rule for the block from following options:

- **Enforce single-rate processing** — When you select this option, the block maintains the sample rate of the input.
- **Allow multirate processing** — When you select this option, the block adjusts the rate at the output to accommodate an increased or reduced number of samples. To select this option, you must set the **Input processing** parameter to **Elements as channels (sample based)**.

### Use symbolic names for coefficients

Select this check box to enable the specification of coefficients using MATLAB variables. The available coefficient names differ depending on the filter structure. Using symbolic names allows tuning of filter coefficients in generated code. By default, this check box is cleared.

## Supported Data Types

| Port   | Supported Data Types   |
|--------|--|
| Input  | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> </ul> |
| Output | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> </ul> |

## Version History

Introduced in R2006b

## LPC to LSF/LSP Conversion

Convert linear prediction coefficients to line spectral pairs or line spectral frequencies



### Library

Estimation / Linear Prediction

`dsplp`

### Description

The LPC to LSF/LSP Conversion block takes a vector or matrix of linear prediction coefficients (LPCs) and converts it to a vector or matrix of line spectral pairs (LSPs) or line spectral frequencies (LSFs). When converting LPCs to LSFs, the block outputs match those of the `poly2lsf` function.

The block input must be either a matrix, a column vector, or an unoriented vector. Each channel of the input must have at least two samples.

The input LPCs for each channel,  $1, a_1, a_2, \dots, a_m$ , must be the denominator of the transfer function of a stable all-pole filter with the form given in the first equation of “Requirements for Valid Outputs” on page 2-910. A length- $M+1$  input channel yields a length- $M$  output channel.

See other sections of this reference page to learn about how to ensure that you get valid outputs, how to detect invalid outputs, how the block computes the LSF/LSP values, and more.

### Requirements for Valid Outputs

To get valid outputs, your inputs and the **Root finding coarse grid points** parameter value must meet these requirements:

- The input LPCs for each channel,  $1, a_1, a_2, \dots, a_m$ , must come from the denominator of the following transfer function,  $H(z)$ , of a stable all-pole filter (all roots of  $H(z)$  must be inside the unit circle). Note that the first term in  $H(z)$ 's denominator must be 1. When the input LPCs do not come from a transfer function of the following form, the block outputs are invalid.

$$H(z) = \frac{1}{1 + a_1z^{-1} + a_2z^{-2} + \dots + a_mz^{-m}}$$

- The **Root finding coarse grid points** parameter value must be large enough so that the block can find all the LSP or LSF values. (The output LSFs and LSPs are roots of polynomials related to the input LPC polynomial; the block looks for these roots to produce the output. For details, see “LSF and LSP Computation Method: Chebyshev Polynomial Method for Root Finding” on page 2-914.) When you do not set **Root finding coarse grid points** to a high enough value relative to the number of LPCs, the block might not find all the LSPs or LSFs and yield invalid outputs as described in “Root Finding Method Limitations: Failure to Find Roots” on page 2-916.

To learn about recognizing invalid inputs and outputs and parameters for dealing with them, see “Handling and Recognizing Invalid Inputs and Outputs” on page 2-912.

### Setting Outputs to LSFs or LSPs

Set the **Output** parameter to one of the following settings to determine whether the block outputs LSFs or LSPs:

- **LSF in radians (0 pi)** — Block outputs the LSF values between 0 and  $\pi$  radians in increasing order. The block does not output the guaranteed LSF values, 0 and  $\pi$ .
- **LSF normalized in range (0 0.5)** — Block outputs normalized LSF values in increasing order, computed by dividing the LSF values between 0 and  $\pi$  radians by  $2\pi$ . The block does not output the guaranteed normalized LSF values, 0 and 0.5.
- **LSP in range (-1 1)** — Block outputs LSP values in decreasing order, equal to the cosine of the LSF values between 0 and  $\pi$  radians. The block does not output the guaranteed LSP values, -1 and 1.

### Adjusting Output Computation Time and Accuracy with Root Finding Parameters

The values  $n$  and  $k$  determine the block's output computation time and accuracy, where

- $n$  is the value of the **Root finding coarse grid points** parameter (choose this value with care; see the note below).
- $k$  is the value of the **Root finding bisection refinement** parameter.
- Decreasing the values of  $n$  and  $k$  decreases the output computation time, but also decreases output accuracy:
  - The upper bound of block's computation time is proportional to  $k \cdot (n - 1)$ .
  - Each LSP output is within  $1/(n \cdot 2^k)$  of the actual LSP value.
  - Each LSF output is within  $\Delta LSF$  of the actual LSF value,  $LSF_{act}$ , where

$$\Delta LSF = \left| \arccos(LSF_{act}) - \arccos\left(LSF_{act} + 1/\left(n \cdot 2^k\right)\right) \right|$$

---

**Note** When the value of the **Root finding coarse grid points** parameter is too small relative to the number of LPCs, the block might output invalid data as described in “Requirements for Valid Outputs” on page 2-910. Also see “Handling and Recognizing Invalid Inputs and Outputs” on page 2-912.

---

### Notable Input and Output Properties

- To get valid outputs, your input LPCs and the value of the **Root finding coarse grid points** parameter must meet the requirements described in “Requirements for Valid Outputs” on page 2-910.
- Length- $L+1$  input channel yields length- $L$  output channel
- **Output** parameter determines the output type (see “Setting Outputs to LSFs or LSPs” on page 2-911):
  - LSFs — frequencies,  $w_k$ , where  $0 < w_k < \pi$  and  $w_k < w_{k+1}$
  - Normalized LSFs —  $w_k / 2\pi$

- LSPs —  $\cos(w_k)$

### Handling and Recognizing Invalid Inputs and Outputs

The block outputs invalid data when your input LPCs and the value of the **Root finding coarse grid points** parameter do not meet the requirements described in “Requirements for Valid Outputs” on page 2-910. The following topics describe what invalid outputs look like, and how to set the block parameters provided for handling invalid inputs and outputs:

- “What Invalid Outputs Look Like” on page 2-912
- “Parameters for Handling Invalid Inputs and Outputs” on page 2-912

### What Invalid Outputs Look Like

The channels of an invalid output have the same dimensions, sizes, and frame statuses as the channels of a valid output. However, invalid output channels do not contain all the LSP or LSF values. Instead, they contain none or some of the LSP and LSF values and the rest of the output is filled with place holder values (-1, 0.5, or  $\pi$ ) depending on the **Output** parameter setting).

In short, all invalid outputs in a channel end in one of the place holder values (-1, 0.5, or  $\pi$ ) as illustrated in the following table. To learn how to use the block's parameters for handling invalid inputs and outputs, see the next section.

| Output Parameter Setting                | Place Holder | Sample Invalid Outputs   |
|---|--------------|--|
| LSF in radians ( $\theta$ $\pi$ )       | $\pi$        | $[w_1 \ w_2 \ w_3 \ \pi \ \pi \ \pi \ \pi \ \pi]$                              |
| LSF normalized in range ( $\theta$ 0.5) | 0.5          | $\begin{bmatrix} w_1 \\ w_2 \\ 0.5 \end{bmatrix}$                              |
| LSP in range (-1 1)                     | -1           | $\begin{bmatrix} \cos(w_{13}) \\ \cos(w_{23}) \\ -1 \\ -1 \\ -1 \end{bmatrix}$ |

### Parameters for Handling Invalid Inputs and Outputs

You must set how the block handles invalid inputs and outputs by setting these parameters:

- **Show output validity status (1=valid, 0=invalid)** — Set this parameter to activate a second output port that outputs a vector with one Boolean element per channel; 1 when the output of the corresponding channel is valid, and 0 when the output is invalid. The LSF and LSP outputs are invalid when the block fails to find all the LSF or LSP values or when the input LPCs are unstable (for details, see “Requirements for Valid Outputs” on page 2-910). See the previous section to learn how to recognize invalid outputs.
- **If current output is invalid, overwrite with previous output** — Select this check box to cause the block to overwrite invalid outputs with the previous output. When you set this parameter you also need to consider these parameters:
  - **When first output is invalid, overwrite with user-defined values** — When the first input is unstable, you can overwrite the invalid first output with either



- The default values, by clearing this check box
- Values you specify, by selecting this check box

The default initial overwrite values are the LSF or LSP representations of an all-pass filter. The vector that is used to overwrite invalid output is stored as an internal state.

- **User-defined LSP/LSF values for overwriting invalid first output** — Specify a vector of values for overwriting an invalid first output if you selected the **When first output is invalid, overwrite with user-defined values** parameter. For multichannel inputs, provide a matrix with the same number of channels as the input, or one vector that will be applied to every channel. The vector or matrix of LSP/LSF values you specify should have the same dimension, size, and frame status as the other outputs.
- **If first input value is not 1** — The block output in any channel is invalid when the first coefficient in an LPC vector is not 1; this parameter determines what the block does when given such inputs:
  - **Ignore** — Proceed with computations as if the first coefficient is 1.
  - **Normalize** — Divide the input LPCs by the value of the first coefficient before computing the output.
  - **Normalize and warn** — In addition to **Normalize**, display a warning message at the MATLAB command line.
  - **Error** — Stop the simulation and display an error message at the MATLAB command line.

## Parameters

### Output

Specifies whether to convert the input linear prediction polynomial coefficients (LPCs) to LSP in range  $(-1, 1)$ , LSF in radians  $(0, \pi)$ , or LSF normalized in range  $(0, 0.5)$ . See “Setting Outputs to LSFs or LSPs” on page 2-911 for descriptions of the three settings.

### Root finding coarse grid points

The value  $n$ , where the block divides the interval  $(-1, 1)$  into  $n$  subintervals of equal length, and looks for roots (LSP values) in each subinterval. You must pick  $n$  large enough or the block output might be invalid as described in “Requirements for Valid Outputs” on page 2-910. To learn how the block uses this parameter to compute the output, see “LSF and LSP Computation Method: Chebyshev Polynomial Method for Root Finding” on page 2-914. Also see “Adjusting Output Computation Time and Accuracy with Root Finding Parameters” on page 2-911. Tunable (Simulink).

### Root finding bisection refinement

The value  $k$ , where each LSP output is within  $1/(n \cdot 2^k)$  of the actual LSP value, where  $n$  is the value of the **Root finding coarse grid points** parameter. To learn how the block uses this parameter to compute the output, see “LSF and LSP Computation Method: Chebyshev Polynomial Method for Root Finding” on page 2-914. Also see “Adjusting Output Computation Time and Accuracy with Root Finding Parameters” on page 2-911. Tunable (Simulink).

### Show output validity status

Set this parameter to activate a second output port that outputs a vector with one Boolean element per channel; 1 when the output of the corresponding channel is valid, and 0 when the output is invalid. The LSF and LSP outputs are invalid when the block fails to find all the LSF or LSP values or when the input LPCs are unstable (for details, see “Requirements for Valid Outputs” on page 2-910).

**If current output is invalid, overwrite with previous output**

Selecting this check box causes the block to overwrite invalid outputs with the previous output. Setting this parameter activates other parameters for taking care of initial overwrite values (when the very first output of the block is invalid). For more information, see “Parameters for Handling Invalid Inputs and Outputs” on page 2-912.

**When first output is invalid, overwrite with user-defined values**

When the first input is unstable, you can overwrite the invalid first output with either

- The default values, by clearing this check box
- Values you specify, by selecting this check box

The default initial overwrite values are the LSF or LSP representations of an all-pass filter. The vector that is used to overwrite invalid output is stored as an internal state. For more information, see “Parameters for Handling Invalid Inputs and Outputs” on page 2-912.

**User-defined LSP/LSF values for overwriting invalid first output**

Specify a vector of values for overwriting an invalid first output if you selected the **When first output is invalid, overwrite with user-defined values** parameter. For multichannel inputs, provide a matrix with the same number of channels as the input, or one vector that will be applied to every channel. The vector or matrix of LSP/LSF values you specify should have the same dimension, size, and frame status as the other outputs.

**If first input value is not 1**

Determines what the block does when the first coefficient of an input is not 1. The block can either proceed with computations as when the first coefficient is 1 (**Ignore**); divide the input LPCs by the value of the first coefficient before computing the output (**Normalize**); in addition to **Normalize**, display a warning message at the MATLAB command line (**Normalize and warn**); stop the simulation and display an error message at the MATLAB command line (**Error**). For more information, see “Parameters for Handling Invalid Inputs and Outputs” on page 2-912.

## Theory

**LSF and LSP Computation Method: Chebyshev Polynomial Method for Root Finding**

---

**Note** To learn the principles on which the block's LSP and LSF computation method is based, see the reference listed in “References” on page 2-917.

---

To compute LSP outputs for each channel, the block relies on the fact that LSP values are the roots of two particular polynomials related to the input LPC polynomial; the block finds these roots using the Chebyshev polynomial root finding method, described next. To compute LSF outputs, the block computes the arc cosine of the LSPs, outputting values ranging from 0 to  $\pi$  radians.

**Root Finding Method**

LSPs, which are the roots of two particular polynomials, always lie in the range (-1, 1). (The guaranteed roots at 1 and -1 are factored out.) The block finds the LSPs by looking for a sign change of the two polynomials' values between points in the range (-1, 1). The block searches a maximum of  $k(n - 1)$  points, where

- $n$  is the value of the **Root finding coarse grid points** parameter.

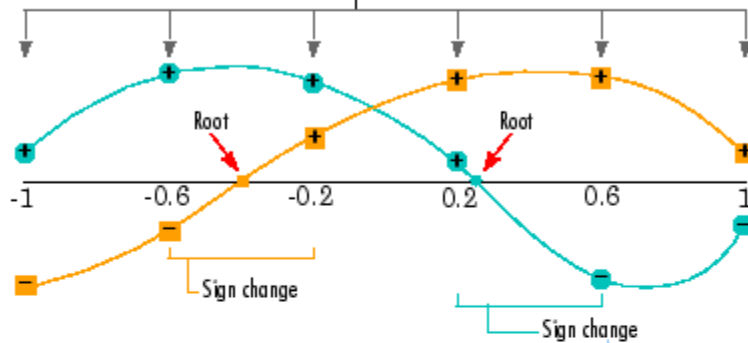
- $k$  is the value of the **Root finding bisection refinement** parameter.

The block's method for choosing which points to check consists of the following two steps:

- 1 Coarse Root Finding** — The block divides the interval  $[-1, 1]$  into  $n$  intervals, each of length  $2/n$ , and checks the signs of both polynomials' values at the endpoints of the intervals. The block starts checking signs at 1, and continues checking signs at  $1 - 4/n$ ,  $1 - 6/n$ , and so on at steps of length  $2/n$ , outputting any point if it is a root. The block stops searching in these situations:
  - a** The block finds a sign change of a polynomial's values between two adjacent points. An interval containing a sign change is guaranteed to contain a root, so the block further searches the interval as described in Step 2, Root Finding Refinement.
  - b** The block finds and outputs all  $M$  roots (given a length- $M+1$  LPC input).
  - c** The block fails to find all  $M$  roots and yields invalid outputs as described in “Handling and Recognizing Invalid Inputs and Outputs” on page 2-912.
- 2 Root Finding Refinement** — When the block finds a sign change in an interval,  $[a, b]$ , it searches for the root guaranteed to lie in the interval by following these steps:
  - a Check if Midpoint Is a Root** — The block checks the sign of the midpoint of the interval  $[a, b]$ . The block outputs the midpoint if it is a root, and continues Step 1, Coarse Root Finding, at the next point,  $a - 2/n$ . Otherwise, the block selects the half-interval with endpoints of opposite sign (either  $[a, (a + b)/2]$  or  $[(a + b)/2, b]$ ) and executes Step 2b, Stop or Continue Root Finding Refinement.
  - b Stop or Continue Root Finding Refinement** — When the block has repeated Step 2a  $k$  times ( $k$  is the value of the **Root finding bisection refinement** parameter), the block linearly interpolates the root by using the half-interval's endpoints, outputs the result as an LSP value, and returns to Step 1, Coarse Root Finding. Otherwise, the block repeats Step 2a using the half-interval.

**Coarse Root Finding:** LSPs are roots of two particular polynomials related to the input LPCs. Check signs of the two polynomials at evenly-spaced points to find all intervals containing a sign change. Output any roots (LSPs) found.

**Root finding coarse grid points = 5**  
Divide  $[-1, 1]$  into five intervals of equal length and check signs of the polynomials' values at the endpoints of the intervals: 1, 0.6, 0.2, -0.2, -0.6, -1.



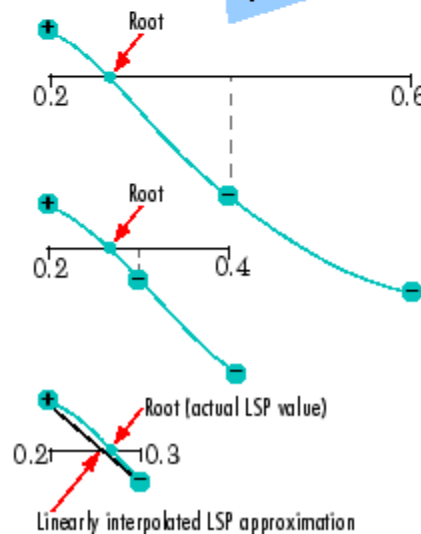
**Root Finding Refinement:** Whenever Coarse Root Finding identifies an interval containing a sign change, repeatedly bisect the interval to better approximate the root (LSP value).

**Bisection 1:** Check the sign of the polynomial at the midpoint of the interval and select the half-interval with endpoints of opposite sign:  $[0.2, 0.4]$

**Bisection 2:** Similar to Bisection 1

**Bisection 3:** The last bisection. Since the midpoint of this interval is not the root, linearly interpolate the root and output the result as an LSP value.

**Root finding bisection refinement = 3**  
Bisect all sign change intervals found in the Coarse Root Finding up to three times to find the root. When the root is not found in the last bisection, linearly interpolate the root.



## Coarse Root Finding and Root Finding Refinement

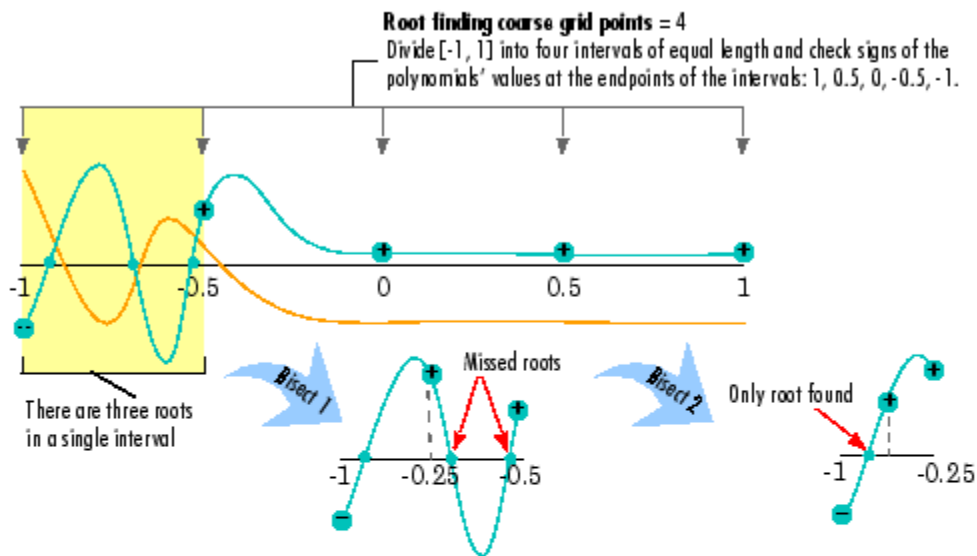
### Root Finding Method Limitations: Failure to Find Roots

The block root finding method described above can fail, causing the block to produce invalid outputs (for details on invalid outputs, see “Handling and Recognizing Invalid Inputs and Outputs” on page 2-912).

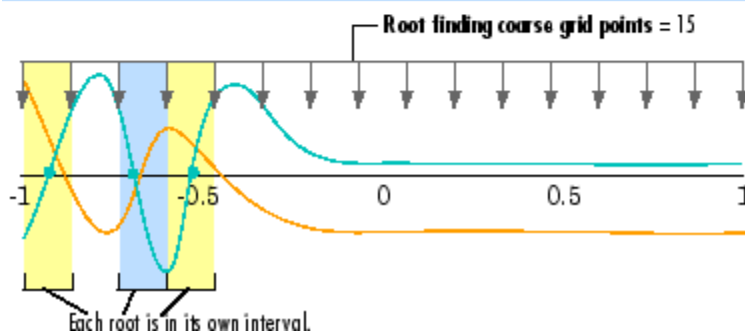
In particular, the block can fail to find some roots if the value of the **Root finding coarse grid points** parameter,  $n$ , is too small. If the polynomials oscillate quickly and have roots that are very close together, the root finding might be too coarse to identify roots that are very close to each other, as illustrated in “Fixing a Failed Root Finding” on page 2-917.

For higher-order input LPC polynomials, you should increase the **Root finding coarse grid points** value to ensure the block finds all the roots and produces valid outputs.

**Root Finding Fails:** The root search divides the interval  $[-1, 1]$  into four intervals, but all three roots are in a single interval. The block can only find one root per interval, so two of the roots are never found.



**Fix Root Finding so it Succeeds:** Increasing the value of the **Root finding course grid points** parameter to 15 ensures that each root is in its own interval, so all roots are found.



## Fixing a Failed Root Finding

### Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Boolean — Supported only by the optional output port that appears when you set the parameter, **Show output validity status (1=valid, 0=invalid)**

### References

Kabal, P. and Ramachandran, R. "The Computation of Line Spectral Frequencies Using Chebyshev Polynomials." *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. ASSP-34 No. 6, December 1986. pp. 1419-1426.

## See Also

LSF/LSP to LPC Conversion  
LPC to/from RC  
LPC/RC to Autocorrelation  
poly2lsf

DSP System Toolbox  
DSP System Toolbox  
DSP System Toolbox  
Signal Processing Toolbox

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Generated code relies on the `memcpy` or `memset` function (`string.h`) under certain conditions.

# LSF/LSP to LPC Conversion

Convert line spectral frequencies or line spectral pairs to linear prediction coefficients



## Library

Estimation / Linear Prediction

dsplp

## Description

The LSF/LSP to LPC Conversion block takes a vector or matrix of line spectral pairs (LSPs) or line spectral frequencies (LSFs) and converts it to a vector or matrix of linear prediction polynomial coefficients (LPCs). When converting LSFs to LPCs, the block outputs match those of the `lsf2poly` function.

The block input can be an  $N$ -by- $M$  matrix or an unoriented vector. Each column of the matrix is treated as a channel. When the input is an unoriented vector, the input is treated as one channel. Each input channel must be in the same format, which you specify in the **Input** parameter:

- LSF in range  $(0 \pi)$  — Vector of LSF values between 0 and  $\pi$  radians in increasing order. Do not include the guaranteed LSF values, 0 and  $\pi$ .
- LSF normalized in range  $(0 0.5)$  — Vector of normalized LSF values in increasing order, (compute by dividing the LSF values between 0 and  $\pi$  radians by  $2\pi$ ). Do not include the guaranteed normalized LSF values, 0 and 0.5.
- LSP in range  $(-1 1)$  — Vector of LSP values in decreasing order, equal to the cosine of the LSF values between 0 and  $\pi$  radians. Do not include the guaranteed LSP values, -1 and 1.

## Parameters

### Input

Specifies whether to convert LSP in range  $(-1 1)$ , LSF in range  $(0 \pi)$ , or LSF normalized in range  $(0 0.5)$  to linear prediction coefficients (LPCs).

## Supported Data Types

- Double-precision floating point
- Single-precision floating point

## References

Kabal, P. and Ramachandran, R. "The Computation of Line Spectral Frequencies Using Chebyshev Polynomials." *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. ASSP-34 No. 6, December 1986. pp. 1419-1426.

## See Also

LPC to LSF/LSP Conversion

LPC to/from RC

LPC/RC to Autocorrelation

lsf2poly

DSP System Toolbox

DSP System Toolbox

DSP System Toolbox

Signal Processing Toolbox

## Version History

**Introduced before R2006a**

## Extended Capabilities

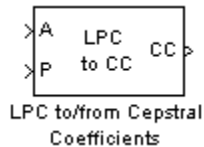
### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.



## LPC to/from Cepstral Coefficients

Convert linear prediction coefficients to cepstral coefficients or cepstral coefficients to linear prediction coefficients



### Library

Estimation / Linear Prediction

dsplp

### Description

The LPC to/from Cepstral Coefficients block either converts linear prediction coefficients (LPCs) to cepstral coefficients (CCs) or cepstral coefficients to linear prediction coefficients. Set the **Type of conversion** parameter to `LPCs to cepstral coefficients` or `Cepstral coefficients to LPCs` to select the domain into which you want to convert your coefficients. The LPC port corresponds to LPCs, and the CC port corresponds to the CCs. For more information, see “Algorithm” on page 2-922.

The block input can be an  $N$ -by- $M$  matrix or an unoriented vector. Each column of the matrix is treated as a channel. When the input is an unoriented vector, the input is treated as one channel.

Consider a signal  $x(n)$  as the input to an FIR analysis filter represented by LPCs. The output of this analysis filter,  $e(n)$ , is known as the prediction error signal. The power of this error signal is denoted by  $P$ , the prediction error power.

When you select `LPCs to cepstral coefficients` from the **Type of conversion** list, you can specify the prediction error power in two ways. From the **Specify P** list, choose `via input port` to input the prediction error power using input port  $P$ . The input to the port must be a vector with length equal to the number of input channels. Select `assume P equals 1` to set the prediction error power equal to 1 for all channels.

When you select `LPCs to cepstral coefficients` from the **Type of conversion** list, the **Output size same as input size** check box appears. When you select this check box, the length of the input vector of LPCs is equal to the output vector of CCs. When you do not select this check box, enter a positive scalar for the **Length of output cepstral coefficients** parameter.

When you select `LPCs to cepstral coefficients` from the **Type of conversion** list, you can use the **If first input value is not 1** parameter to specify the behavior of the block when the first coefficient of the LPC vector is not 1. The following options are available:

- `Replace it with 1` — Changes the first value of the coefficient vector to 1. The other coefficient values are unchanged.

- **Normalize** — Divides the entire vector of coefficients by the first coefficient so that the first coefficient of the LPC vector is 1.
- **Normalize and Warn** — Divides the entire vector of coefficients by the first coefficient so that the first coefficient of the LPC vector is 1. The block displays a warning message telling you that your vector of coefficients has been normalized.
- **Error** — Displays an error telling you that the first coefficient of the LPC vector is not 1.

When you select **Cepstral coefficients to LPCs** from the **Type of conversion** list, the **Output P** check box appears on the block. Select this check box when you want to output the prediction error power from output port P.

## Algorithm

The cepstral coefficients are the coefficients of the Fourier transform representation of the logarithm magnitude spectrum. Consider a sequence,  $x(n)$ , having a Fourier transform  $X(\omega)$ . The cepstrum,  $c_x(n)$ , is defined by the inverse Fourier transform of  $C_x(\omega)$ , where  $C_x(\omega) = \log_e X(\omega)$ . See the Real Cepstrum block reference page for information on computing cepstrum coefficients from time-domain signals.

### LPC to CC

When in this mode, this block uses a recursion technique to convert LPCs to CCs. The LPC vector is defined by  $[a_0 \ a_1 \ a_2 \ \dots \ a_p]$  and the CC vector is defined by  $[c_0 \ c_1 \ c_2 \ \dots \ c_p \ \dots \ c_{n-1}]$ . The recursion is defined by the following equations:

$$c_0 = \log_e P$$

$$c_m = -a_m + \frac{1}{m} \sum_{k=1}^{m-1} [-(m-k) \cdot a_k \cdot c_{(m-k)}], 1 \leq m \leq p$$

$$c_m = \sum_{k=1}^p \left[ \frac{-(m-k)}{m} \cdot a_k \cdot c_{(m-k)} \right], p < m < n$$

### CC to LPC

When in this mode, this block uses a recursion technique to convert CCs to LPCs. The CC vector is defined by  $[c_0 \ c_1 \ c_2 \ \dots \ c_p \ \dots \ c_n]$  and the LPC vector is defined by  $[a_0 \ a_1 \ a_2 \ \dots \ a_p]$ . The recursion is defined by the following equations

$$a_m = -c_m - \frac{1}{m} \sum_{k=1}^{m-1} [(m-k) \cdot c_{(m-k)} \cdot a_k]$$

$$P = \exp(C_0)$$

where  $m = 1, 2, \dots, p$ .

## Parameters

### Type of conversion

Choose **LPCs to cepstral coefficients** or **Cepstral coefficients to LPCs** to specify the domain into which you want to convert your coefficients.

**Specify P**

Choose `via input port` to input the values of prediction error power using input port P. Select `assume P equals 1` to set the prediction error power equal to 1.

**Output size same as input size**

When you select this check box, the length of the input vector of LPCs is equal to the output vector of CCs.

**Length of output cepstral coefficients**

Enter a positive scalar that is the length of each output channel of CCs.

**If first input value is not 1**

Select what you would like the block to do when the first coefficient of the LPC vector is not 1. You can choose `Replace it with 1`, `Normalize`, `Warn`, and `Error`.

**Output P**

Select this check box to output the prediction error power for each channel from output port P.

**References**

Papamichalis, Panos E. *Practical Approaches to Speech Coding*. Englewood Cliffs, NJ: Prentice Hall, 1987.

**Supported Data Types**

- Double-precision floating point
- Single-precision floating point

**See Also**

|                           |                    |
|---------------------------|--------------------|
| Levinson-Durbin           | DSP System Toolbox |
| LPC to LSF/LSP Conversion | DSP System Toolbox |
| LSF/LSP to LPC Conversion | DSP System Toolbox |
| LPC to/from RC            | DSP System Toolbox |
| LPC/RC to Autocorrelation | DSP System Toolbox |
| Real Cepstrum             | DSP System Toolbox |
| Complex Cepstrum          | DSP System Toolbox |

**Version History**

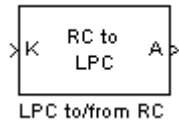
Introduced before R2006a

**Extended Capabilities****C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

## LPC to/from RC

Convert linear prediction coefficients to reflection coefficients or reflection coefficients to linear prediction coefficients



## Library

Estimation / Linear Prediction

dsp\lp

## Description

The LPC to/from RC block either converts linear prediction coefficients (LPCs) to reflection coefficients (RCs) or reflection coefficients to linear prediction coefficients. Set the **Type of conversion** parameter to **LPC to RC** or **RC to LPC** to select the domain into which you want to convert your coefficients. The A port corresponds to LPC coefficients, and the K port corresponds to the RC coefficients. For more information, see “Algorithm” on page 2-925.

The block input can be an  $N$ -by- $M$  matrix or an unoriented vector. Each column of the matrix is treated as a channel. When the input is an unoriented vector, the input is treated as one channel.

Consider a signal  $x(n)$  as the input to an FIR analysis filter represented by LPC coefficients. The output of the analysis filter,  $e(n)$ , is known as the prediction error signal. The power of this error signal is denoted by  $P$ . When the zero lag autocorrelation coefficient of  $x(n)$  is one, the autocorrelation sequence and prediction error power are said to be normalized.

Select the **Output normalized prediction error power** check box to enable port P. The normalized prediction error power output at P is a vector with one element per input channel. Each element varies between zero and one.

Select the **Output LPC filter stability** check box to output the stability of the filter represented by the LPCs or RCs. The synthesis filter represented by the LPCs is stable when the absolute value of each of the roots of the LPC polynomial is less than one. The lattice filter represented by the RCs is stable when the absolute value of each reflection coefficient is less than 1. When the filter is stable, the block outputs a Boolean value of 1 for each input channel at the S port. When the filter is unstable, the block outputs a Boolean value of 0 for each input channel at the S port.

If **first input value is not 1** parameter specifies the behavior of the block when the first coefficient of the LPC coefficient vector in any channel is not 1. The following options are available:

- **Replace it with 1** — Changes the first value of the coefficient channel to 1. The other coefficient values are unchanged.
- **Normalize** — Divides the entire channel of coefficients by the first coefficient so that the first coefficient of the LPC coefficient vector is 1.

- **Normalize and Warn** — Divides the entire channel of coefficients by the first coefficient so that the first coefficient of the LPC coefficient vector is 1. The block displays a warning message telling you that your vector of coefficients has been normalized.
- **Error** — Displays an error telling you that the first coefficient of the LPC coefficient channel is not 1.

## Algorithm

### LPC to RC

When in this mode, this block uses backward Levinson recursion to convert linear prediction coefficients (LPCs) to reflection coefficients (RCs). For a given Nth order LPC vector  $LPC_N = [1 \ a_{N1} \ a_{N2} \ \dots \ a_{NN}]$ , the block calculates the Nth reflection coefficient value using the formula  $\gamma_N = -a_{NN}$ . The block then finds the lower order LPC vectors,  $LPC_{N-1}, LPC_{N-2}, \dots, LPC_1$ , using the following recursion.

for  $p = N, N - 1, \dots, 2$ ,

$$\gamma_p = a_{pp}$$

$$F = 1 - \gamma_p^2$$

$$a_{p-1,m} = \frac{a_{p,m}}{F} - \frac{\gamma_p a_{p,p-m}}{F}, \quad 1 \leq m < p$$

end

Finally,  $\gamma_1 = -a_{11}$ . The reflection coefficient vector is  $[\gamma_1, \gamma_2, \dots, \gamma_N]$ .

### RC to LPC

When in this mode, this block uses Levinson recursion to convert reflection coefficients (RCs) to linear prediction coefficients (LPCs). In this case, the input to the block is  $RC = [\gamma_1 \ \gamma_2 \ \dots \ \gamma_N]$ . The zeroth order LPC vector term is 1. Starting with this term, the block uses recursion to calculate the higher order LPC vectors,  $LPC_2, LPC_3, \dots, LPC_N$ , until it has calculated the entire LPC matrix.

$$LPC_{matrix} = \begin{bmatrix} LPC_0 \\ LPC_1 \\ LPC_2 \\ \dots \\ LPC_N \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & \dots & 0 \\ 1 & a_{11} & 0 & 0 & \dots & 0 \\ 1 & a_{21} & a_{22} & 0 & \dots & 0 \\ 1 & a_{31} & a_{32} & a_{33} & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 1 & a_{N1} & a_{N2} & a_{N3} & \dots & a_{NN} \end{bmatrix}$$

This LPC matrix consists of LPC vectors of order 0 through  $N$  found by using the Levinson recursion. The following are the formulas for the recursion steps, for  $p = 0, 1, \dots, N - 1$ .

$$a_{p+1,m} = a_{p,m} + \gamma_{p+1} a_{p,p+1-m}, \quad 1 \leq m \leq p$$

$$a_{p+1,p+1} = \gamma_{p+1}$$

## Parameters

### Type of conversion

Select `LPC to RC` or `RC to LPC` to select the domain into which you want to convert your coefficients.

### Output normalized prediction error power

Select this check box to output the normalized prediction error power at port P.

### Output LPC filter stability

Select this check box to output the stability of the filter. When the filter represented by the LPCs or RCs is stable, the block outputs a Boolean value of 1 for each input channel at the S port. When the filter represented by the LPCs or RCs is unstable, the block outputs a Boolean value of 0 for each input channel at the S port.

### If first input value is not 1

Select what you would like the block to do when the first coefficient of the LPC coefficient vector is not 1. You can choose `Replace it with 1`, `Normalize`, `Normalize and Warn`, and `Error`.

## References

Makhoul, J *Linear Prediction: A tutorial review*. Proc. IEEE. 63, 63, 56 (1975).

Markel, J.D. and A. H. Gray, Jr., *Linear Prediction of Speech*. New York, Springer-Verlag, 1976.

## Supported Data Types

- Double-precision floating-point
- Single-precision floating-point

## See Also

|                           |                    |
|---------------------------|--------------------|
| Levinson-Durbin           | DSP System Toolbox |
| LPC to LSF/LSP Conversion | DSP System Toolbox |
| LSF/LSP to LPC Conversion | DSP System Toolbox |
| LPC/RC to Autocorrelation | DSP System Toolbox |

## Version History

Introduced before R2006a

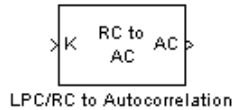
## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

# LPC/RC to Autocorrelation

Convert linear prediction coefficients or reflection coefficients to autocorrelation coefficients



## Library

Estimation / Linear Prediction

dsp\lp

## Description

The LPC/RC to Autocorrelation block either converts linear prediction coefficients (LPCs) to autocorrelation coefficients (ACs) or reflection coefficients (RCs) to autocorrelation coefficients (ACs). Set the **Type of conversion** parameter to `LPC to autocorrelation` or `RC to autocorrelation` to select the domain from which you want to convert your coefficients. The A port corresponds to LPC coefficients, and the K port corresponds to the RC coefficients.

The block input can be an  $N$ -by- $M$  matrix or an unoriented vector. Each column of the matrix is treated as a channel. When the input is an unoriented vector, the input is treated as one channel.

Use the **Specify P** parameter to set the value of the prediction error power. You can set this parameter to 1 by selecting `Assume P=1`. When you select `Via input port`, a P port appears on the block. You can use this port to input the value of the actual, non-unity prediction error power for each channel. The length of this vector must equal the number of channels in the input.

The **If first input value is not 1** parameter specifies the behavior of the block when the first coefficient of the LPC coefficient vector is not 1. The following options are available:

- **Replace it with 1** — The block changes the first value of the coefficient vector to 1. The rest of the coefficient values are unchanged.
- **Normalize** — The block divides the entire vector of coefficients by the first coefficient so that the first coefficient of the LPC coefficient vector is 1.
- **Normalize and Warn** — The block divides the entire vector of coefficients by the first coefficient so that the first coefficient of the LPC coefficient vector is 1. The block displays a warning message telling you that your vector of coefficients has been normalized.
- **Error** — The block displays an error telling you that the first coefficient of the LPC coefficient vector is not 1.

## Parameters

### Type of conversion

From the list select `LPC to autocorrelation` or `RC to autocorrelation` to specify the domain from which you want to convert your coefficients.

**Specify P**

From the list select Assume P=1 or Via input port to specify the value of prediction error power.

**If first input value is not 1**

Select what you would like the block to do when the first coefficient of the LPC coefficient vector is not 1. You can choose Replace it with 1, Normalize, Normalize and Warn, and Error.

**References**

Orfanidis, S.J. *Optimum Signal Processing*. New York, McGraw-Hill, 1988.

Makhoul, J. *Linear Prediction: A tutorial review*. Proc. IEEE. 63, 63, 56 (1975).

Markel, J.D. and A. H. Gray, Jr., *Linear Prediction of Speech*. New York, Springer-Verlag, 1976.

**Supported Data Types**

- Double-precision floating point
- Single-precision floating point

**See Also**

Levinson-Durbin

DSP System Toolbox

LPC to LSF/LSP Conversion

DSP System Toolbox

LSF/LSP to LPC Conversion

DSP System Toolbox

LPC to/from RC

DSP System Toolbox

**Version History**

Introduced before R2006a

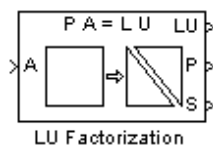
**Extended Capabilities****C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.



# LU Factorization

Factor square matrix into lower and upper triangular components



## Libraries:

DSP System Toolbox / Math Functions / Matrices and Linear Algebra / Matrix Factorizations

## Description

The LU Factorization block factors a row-permuted version of the square input matrix  $A$  as  $A_p = L*U$ , where  $L$  is a unit-lower triangular matrix,  $U$  is an upper triangular matrix, and  $A_p$  contains the rows of  $A$  permuted as indicated by the permutation index vector  $P$ . The block uses the pivot matrix  $A_p$  instead of the exact input matrix  $A$  because it improves the numerical accuracy of the factorization.

## Ports

### Input

**A** — Input matrix

$M$ -by- $M$  square matrix

Specify the input matrix as an  $M$ -by- $M$  square matrix through this port. If the input matrix is fixed point, it must be signed integer or signed fixed point with power-of-two slope and zero bias.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `fixed point`

### Output

**LU** — Lower and upper triangular factors

$M$ -by- $M$  square matrix

Composite matrix containing both the lower triangle elements of  $L$  and the upper triangle elements of  $U$ . The composite matrix improves the efficiency of the block. The matrix output is in a different format than the output of the `lu` function, which returns  $L$  and  $U$  as separate matrices. To convert the output from the **LU** port of the block to separate  $L$  and  $U$  matrices, use the following code:

```
L = tril(LU,-1)+eye(size(LU)); U = triu(LU);
```

If you compare the results produced by these equations to the actual output of the `lu` function, you might see slightly different values. These differences are due to a rounding error, and are expected. For more information on the LU factorizations, see the `lu` function page.

This composite matrix has the same size, data type, and complexity as the input matrix  $A$ .

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `fixed point`

**P** — Permutation vector

vector

Permutation vector, returned as a vector of length  $M$ . The block permutes the input square matrix  $A$  using the permutation vector  $P$  and factors the permuted matrix into unit-lower triangular  $L$  and upper triangular matrix  $U$ .

Consider the following input matrix.

$$A = \begin{bmatrix} 1 & -2 & 3 \\ 4 & 0 & 6 \\ 2 & -1 & 3 \end{bmatrix}$$

With a permutation index vector  $P$  that is equal to  $P = \begin{bmatrix} 2 \\ 1 \\ 3 \end{bmatrix}$ , the permuted matrix  $A_p$  becomes:

$$A_p = \begin{bmatrix} 4 & 0 & 6 \\ 1 & -2 & 3 \\ 2 & -1 & 3 \end{bmatrix}$$

The block factors this matrix into upper and lower triangular matrices. For a more detailed version of this example, see “Factorize Matrix Using LU Factorization Block”.

Data Types: `single` | `double` | `uint32`

**S** — Singularity of input matrix  
logical scalar

You can determine the singularity of the input matrix  $A$  by enabling the optional output port **S**. When  $A$  is singular, the block outputs a 1 at port **S**. When  $A$  is nonsingular, the block outputs a 0.

Data Types: `Boolean`

## Parameters

### Main Tab

**Show singularity status** — Show singularity status

`off` (default) | `on`

Select to output the singularity of the input at port **S**, which outputs Boolean data type values of 1 or 0. An output of 1 indicates that the current input is singular, and an output of 0 indicates the current input is nonsingular.

### Data Types Tab

**Rounding mode** — Method of rounding operation

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

Specify the rounding mode for fixed-point operations as one of the following:

- `Floor`
- `Ceiling`

- Convergent
- Nearest
- Round
- Simplest
- Zero

For more details, see rounding mode.

#### **Saturate on integer overflow** — Method of overflow action

off (default) | on

When you select this parameter, the block saturates the result of its fixed-point operation. When you clear this parameter, the block wraps the result of its fixed-point operation. For details on saturate and wrap, see overflow mode for fixed-point operations.

#### **Product output** — Product output data type

Inherit: `Inherit via internal rule` (default) | Inherit: `Same as input` | `fixdt(1,16,0)`

**Product output** specifies the data type of the output of a product operation in the LU Factorization block. For more information on the product output data type, see “Fixed Point” on page 2-933 and “Multiplication Data Types”.

- `Inherit: Inherit via internal rule` — The block inherits the product output data type based on an internal rule. For more information on this rule, see “Inherit via Internal Rule”.
- `Inherit: Same as input` — The block specifies the product output data type to be the same as the input data type.
- `fixdt(1,16,0)` — The block specifies a signed, binary-point, scaled, fixed-point data type with a word length of 16 bits and a fraction length of 0.

Alternatively, you can set the **Product output** data type by using the **Data Type Assistant**. To use

the assistant, click the **Show data type assistant** button .

For more information on the data type assistant, see “Specify Data Types Using Data Type Assistant” (Simulink) in *Simulink User's Guide* (Simulink).


#### **Accumulator** — Accumulator data type

Inherit: `Inherit via internal rule` (default) | Inherit: `Same as input` | Inherit: `Same as product output` | `fixdt(1,16,0)`

**Accumulator** specifies the data type of the output of an accumulation operation in the LU Factorization block. For more information on how to use the accumulator data type in this block, see the “Fixed Point” on page 2-933 section.

- `Inherit: Inherit via internal rule` — The block inherits the accumulator data type based on an internal rule. For more information on this rule, see “Inherit via Internal Rule”.
- `Inherit: Same as input` — The block specifies the accumulator data type to be the same as the input data type.

- **Inherit**: Same as product output — The block specifies the accumulator data type to be the same as the product output data type.
- `fixdt(1,16,0)` — The block specifies a signed, binary-point scaled, fixed-point data type with a word length of 16 bits and a fraction length of 0.

Alternatively, you can set the **Accumulator** data type by using the **Data Type Assistant**. To use the assistant, click the **Show data type assistant** button .


For more information on the data type assistant, see “Specify Data Types Using Data Type Assistant” (Simulink) in *Simulink User's Guide* (Simulink).

#### **Output** — Output data type

**Inherit**: Same as input (default) | `fixdt(1,16,0)`

**Output** specifies the data type of the output of the LU Factorization block. For more information on the output data type, see the “Fixed Point” on page 2-933 section.

- **Inherit**: Same as input — The block specifies the output data type to be the same as the input data type.
- `fixdt(1,16,0)` — The block specifies a signed, binary-point scaled, fixed-point data type with a word length of 16 bits and a fraction length of 0.

Alternatively, you can set the **Output** data type by using the **Data Type Assistant**. To use the assistant, click the **Show data type assistant** button .

For more information on the data type assistant, see “Specify Data Types Using Data Type Assistant” (Simulink) in *Simulink User's Guide* (Simulink).

#### **Output Minimum** — Minimum value the block can output

`[]` (default) | scalar

Specify the minimum value the block can output. Simulink software uses this minimum value to perform:

- Simulation range checking. See “Specify Signal Ranges” (Simulink).
- Automatic scaling of fixed-point data types.

#### **Output Maximum** — Maximum value block can output

`[]` (default) | scalar

Specify the maximum value the block can output. Simulink software uses this maximum value to perform:

- Simulation range checking. See “Specify Signal Ranges” (Simulink).
- Automatic scaling of fixed-point data types.

**Lock data type settings against changes by the fixed-point tools** — Prevent fixed-point tools from overriding data types

`off` (default) | `on`

Select this parameter to prevent the fixed-point tools from overriding the data types you specify in the block dialog box.

## Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | double   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no                                      |
| <b>Multidimensional Signals</b> | no                                      |
| <b>Variable-Size Signals</b>    | no                                      |
| <b>Zero-Crossing Detection</b>  | no                                      |

## Version History

Introduced before R2006a

## References

[1] Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

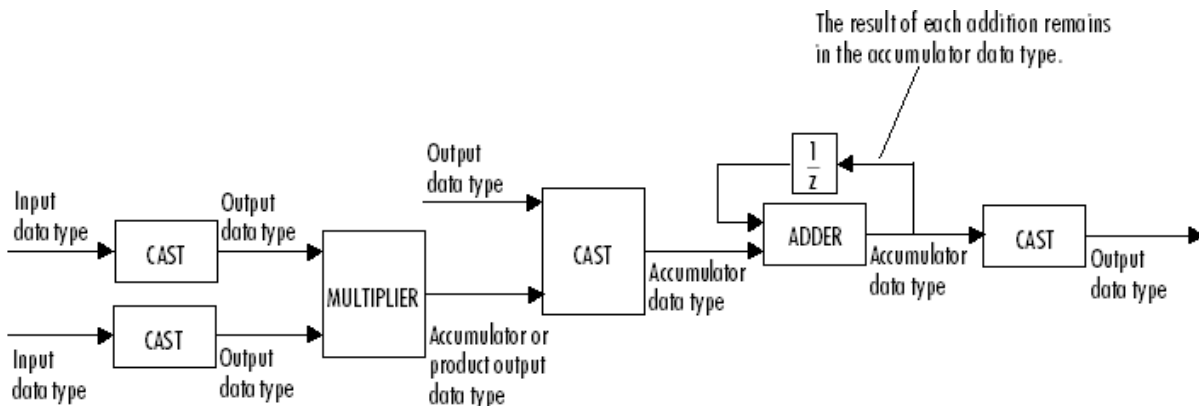
Generated code relies on the memcpy or memset function (string.h) under certain conditions.

### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

If the input is fixed point, it must be a signed integer or a signed fixed point value with a power-of-two slope and zero bias.

The following diagram shows the data types used within the LU Factorization block for fixed-point signals.



You can set the product output, accumulator, and output data types in the block dialog as discussed below.

The output of the multiplier is in the product output data type when the input is real. When the input is complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication performed, see “Multiplication Data Types”.

## **See Also**

### **Functions**

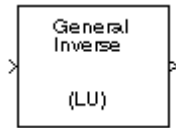
lu

### **Blocks**

Cholesky Factorization | Autocorrelation LPC | LU Solver | LDL Factorization | LU Inverse | Permute Matrix | QR Factorization

# LU Inverse

Compute inverse of square matrix using LU factorization



## Libraries:

DSP System Toolbox / Math Functions / Matrices and Linear Algebra / Matrix Inverses

## Description

The LU Inverse block computes the inverse of the square input matrix  $A$  by factoring and inverting its row-pivoted variant  $A_p$ .

$$A_p^{-1} = (LU)^{-1}$$

$L$  is a lower triangular square matrix with unity diagonal elements, and  $U$  is an upper triangular square matrix. The block outputs the inverse matrix  $A^{-1}$ .

## Ports

### Input

**Input** — Input square matrix  
square matrix

Specify the input as an  $M$ -by- $M$  square matrix.

Data Types: `single` | `double`

### Output

**Output** — Matrix inverse  
square matrix

Matrix inverse, returned as an  $M$ -by- $M$  square matrix.

Data Types: `single` | `double`

## Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | <code>double</code>   <code>single</code> |
| <b>Direct Feedthrough</b>       | no  |
| <b>Multidimensional Signals</b> | no  |
| <b>Variable-Size Signals</b>    | no  |
| <b>Zero-Crossing Detection</b>  | no  |

## Version History

Introduced before R2006a

## References

[1] Golub, Gene H., and Charles F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Generated code relies on the memcpy or memset function (`string.h`) under certain conditions.

## See Also

### Functions

`inv`

### Blocks

Cholesky Inverse | LDL Inverse | LU Factorization | LU Solver

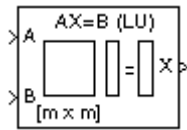
### Topics

“Matrix Inverses”



# LU Solver

Solve  $AX = B$  when  $A$  is a square matrix



## Libraries:

DSP System Toolbox / Math Functions / Matrices and Linear Algebra / Linear System Solvers

## Description

The LU Solver block solves the linear system  $AX = B$  by applying LU factorization, where:

- $A$  is an  $M$ -by- $M$  square matrix input through the **A** port.
- $B$  is an  $M$ -by- $N$  matrix input through the **B** port.
- $X$  is the  $M$ -by- $N$  output matrix and is the unique solution to the equations.

## Ports

### Input

**A** — Input  $A$  matrix  
 $M$ -by- $M$  matrix

Specify the input  $M$ -by- $M$  square matrix  $A$  through this port. Inputs  $A$  and  $B$  must have the same number of rows.

Data Types: `single` | `double`

**B** —  $B$  matrix  
 $M$ -by- $N$  matrix |  $M$ -by-1 vector

Specify the  $B$  matrix in the  $AX = B$  equation as an  $M$ -by- $N$  matrix or an  $M$ -by-1 vector.

When you specify an  $M$ -by-1 vector, the block treats the length- $M$  unoriented vector input at port **B** as an  $M$ -by-1 matrix. Inputs  $A$  and  $B$  must have the same number of rows.

Data Types: `single` | `double`

### Output

**X** — Solution to system of equations  
 $M$ -by- $N$  matrix |  $M$ -by-1 matrix

The block returns the output as an  $M$ -by- $N$  matrix or a  $M$ -by-1 matrix. The size of  $X$  matrix is same as the size of the  $B$  matrix.

Data Types: `single` | `double`

## Block Characteristics

|                                 |                 |
|---------------------------------|-----------------|
| <b>Data Types</b>               | double   single |
| <b>Direct Feedthrough</b>       | no              |
| <b>Multidimensional Signals</b> | no              |
| <b>Variable-Size Signals</b>    | no              |
| <b>Zero-Crossing Detection</b>  | no              |

## Algorithms

The LU algorithm factors a row-permuted variant ( $A_p$ ) of the square input matrix  $A$  as

$$A_p = LU$$

where  $L$  is a lower triangular square matrix with unity diagonal elements, and  $U$  is an upper triangular square matrix.

The matrix factors are substituted for  $A_p$  in

$$A_p X = B_p$$

where  $B_p$  is the row-permuted variant of  $B$ , and the resulting equation

$$LUX = B_p$$

is solved for  $X$  by substituting  $Y = UX$ , and solving two triangular systems.

$$LY = B_p$$

$$UX = Y$$

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

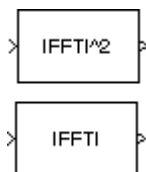
Generated code relies on the memcopy or memset function (`string.h`) under certain conditions.

## See Also

Autocorrelation LPC | Cholesky Solver | LDL Solver | Levinson-Durbin | LU Factorization | LU Inverse | QR Solver

# Magnitude FFT

Compute nonparametric estimate of spectrum using periodogram method



## Libraries:

DSP System Toolbox / Estimation / Power Spectrum Estimation  
 DSP System Toolbox / Transforms

## Description

The Magnitude FFT block computes a nonparametric estimate of the spectrum using the periodogram method.

## Ports

### Input

**Input** — Input  
 vector | matrix

Specify the input  $u$  to the block as a vector or a matrix of size  $M$ -by- $N$ .

The block treats an  $M$ -by- $N$  matrix input as  $M$  sequential time samples from  $N$  independent channels.

The Magnitude FFT block supports real and complex floating-point inputs. The block also supports real fixed-point inputs in the `Magnitude` and `Magnitude squared` modes, and complex fixed-point inputs in the `Magnitude squared` mode.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `fixed point`

### Output

**Output** — Magnitude FFT output  
 vector | matrix

Magnitude FFT output, returned as a vector or a matrix.

The block computes a separate estimate for each of the  $N$  independent channels and generates an  $N_{fft}$ -by- $N$  matrix output. Each column of the output matrix contains the estimate of the power spectral density of the corresponding input column at  $N_{fft}$  equally spaced frequency points in the range  $[0, F_s)$ , where  $F_s$  represents the sample rate of the signal.

When you set the **Output** parameter to `Magnitude squared`, the block output for an  $M$ -by- $N$  input  $u$  is equivalent to:

```
y = abs(fft(u,nfft)).^2    % M ≤ nfft
```

When you set the **Output** parameter to `Magnitude`, the block output for an input  $u$  is equivalent to:

```
y = abs(fft(u,nfft))      % M ≤ nfft
```

When  $M > N_{fft}$ , the block wraps the input to  $N_{fft}$  before computing the FFT using one of the above equations:

```
y(:,k) = datawrap(u(:,k),nfft) % 1 ≤ k ≤ N
```

When  $M > N_{fft}$ , the block can also truncate the input:

```
y(:,k) = abs(fft(u,nfft)) % 1 ≤ k ≤ N
```

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `fixed point`

## Parameters

**Output** — Type of output

`Magnitude squared (default)` | `Magnitude`

Specify whether the block computes the magnitude FFT or the magnitude-squared FFT of the input signal. For more details on how this block computes the output based on this setting, see the **Output** port description.

**FFT implementation** — FFT implementation

`Auto (default)` | `Radix-2` | `FFTW`

Set this parameter to one of these values:

- **Auto** -- Use this option to let the block choose the FFT implementation. For transform lengths that are not a power of 2, the block restricts the generated code to MATLAB host computers.
- **Radix-2** -- Use this option for bit-reversed processing, fixed or floating-point data, or for portable C-code generation using Simulink Coder. The first dimension  $M$  of the input matrix must be a power of two. To work with other input sizes, use the Pad block to pad or truncate the dimensions to powers of two, or if possible choose the FFTW algorithm.
- **FFTW** -- Use this option to support an input signal of arbitrary length. The block restricts the generated code with the FFTW implementation to MATLAB host computers.

**Inherit FFT length from input dimensions** — Inherit FFT length from input frame size

`off (default)` | `on`

Select this parameter to use the input frame size as the number of data points on which to perform the FFT. When you select this check box, the input frame size must be a power of two. When you do not select this check box, the **FFT length** parameter specifies the number of data points.

**FFT length** — FFT length

`256 (default)` | `positive integer`

Enter the number of data points  $N_{fft}$  on which to perform the FFT. When  $N_{fft}$  is larger than the input frame size, the block pads each frame with zeros as needed. When  $N_{fft}$  is smaller than the input frame size, the block wraps each frame as needed.

When you set the **FFT implementation** parameter to `Radix-2`, this value must be a power of 2.

## Dependency

To enable this parameter, clear the **Inherit FFT length from input dimensions** parameter.

**Wrap input data when FFT length is shorter than input length** — Choose to wrap or truncate input

on (default) | off

Choose to wrap or truncate the input depending on the **FFT length**.

When you select this parameter and if the **FFT length** is shorter than the input length, a modulo-length data wrapping occurs before the FFT operation. When you clear this parameter, the block truncates the input data to the FFT length before the FFT operation.

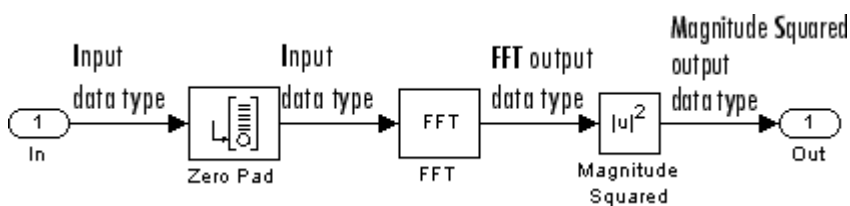
## Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | double   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no                                      |
| <b>Multidimensional Signals</b> | no                                      |
| <b>Variable-Size Signals</b>    | no                                      |
| <b>Zero-Crossing Detection</b>  | no                                      |

## More About

### Fixed-Point Data Types

This diagram shows the data types used within the Magnitude FFT subsystem block for fixed-point signals.



The settings for the fixed-point parameters of the FFT block in the diagram above are as follows:

- Sine table — Same word length as input
- Integer rounding mode — Floor
- Saturate on integer overflow — Not selected
- Product output — Inherit via internal rule
- Accumulator — Inherit via internal rule
- Output — Inherit via internal rule

The settings for the fixed-point parameters of the Magnitude Squared block in the diagram are as follows:

- Integer rounding mode — Floor
- Saturate on integer overflow — Selected
- Output — Inherit via internal rule

## Version History

Introduced before R2006a

## References

- [1] FFTW (<https://www.fftw.org>)
- [2] Frigo, M. and S. G. Johnson, "FFTW: An Adaptive Software Architecture for the FFT," *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Vol. 3, 1998, pp. 1381-1384.
- [3] Oppenheim, A. V. and R. W. Schaffer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1989.
- [4] Orfanidis, S. J. *Introduction to Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1995.
- [5] Proakis, J. and D. Manolakis. *Digital Signal Processing*. 3rd ed. Englewood Cliffs, NJ: Prentice-Hall, 1996.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Usage notes and limitations:

- Generated code relies on the `memcpy` or `memset` function (`string.h`) under certain conditions.
- When the following conditions apply, the executable generated from this block relies on prebuilt dynamic library files (`.dll` files) included with MATLAB:
  - You set **FFT implementation** to FFTW.
  - You clear the **Inherit FFT length from input dimensions** parameter and set **FFT length** to a value that is not a power of two.

Use the `packNGo` function to package the code generated from this block and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see "How To Run a Generated Executable Outside MATLAB".

- When the FFT length is a power of two, you can generate standalone C and C++ code from this block.

### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

## See Also

### Functions

pwelch

### Blocks

Burg Method | Short-Time FFT | Spectrum Analyzer | Yule-Walker Method

### Topics

“Spectral Analysis”

## Matrix 1-Norm

Compute 1-norm of matrix



### Libraries:

DSP System Toolbox / Math Functions / Matrices and Linear Algebra / Matrix Operations

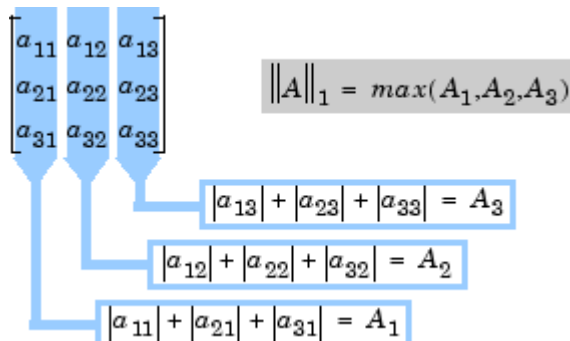
## Description

The Matrix 1-Norm block computes the 1-norm or maximum column-sum of an  $M$ -by- $N$  input matrix  $A$ .

$$y = \|A\|_1 = \max_{1 \leq j \leq N} \sum_{i=1}^M |a_{ij}|$$

Equivalent MATLAB code is given by:

```
y = max(sum(abs(A)))
```



## Ports

### Input

**Port\_1** — Input signal

vector | matrix

Input signal, specified as a vector or matrix.

The block treats length- $M$  unoriented vector input as an  $M$ -by-1 matrix.

The Matrix 1-Norm block supports real and complex floating-point inputs and real fixed-point inputs.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

Complex Number Support: Yes



## Output

**Port\_1** — Output signal  
scalar

Matrix 1-norm or maximum column-sum of the input, returned as a scalar. The output  $y$  is always a scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

## Parameters

### Main

There are no parameters on the **Main Tab**.

### Data Types

**Rounding mode** — Method of rounding operation

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

Specify the rounding mode for fixed-point operations as one of the following:

- Floor
- Ceiling
- Convergent
- Nearest
- Round
- Simplest
- Zero

For more details, see Rounding Modes.

---

**Note** The **Rounding mode** and **Saturate on integer overflow** parameters have no effect on numerical results when all these conditions are met:

- **Accumulator** data type is `Inherit`: `Inherit via internal rule`.
- **Output** data type is `Inherit`: `Same as accumulator`.

With these data-type settings, the block operates in a full-precision mode.

---

**Saturate on integer overflow** — Overflow handling method

`off` (default) | `on`

Select this parameter to saturate the result of the fixed-point operation. Clear this parameter to wrap the result of the fixed-point operation. For details on *saturate* and *wrap*, see Overflow Handling for fixed-point operations.

---

**Note** The **Rounding mode** and **Saturate on integer overflow** parameters have no effect on numeric results when all these conditions are met:

- **Product output** data type is `Inherit: Inherit via internal rule`.
- **Accumulator** data type is `Inherit: Inherit via internal rule`.

With these data-type settings, the block operates in a full-precision mode.


---

#### **Accumulator** — Accumulator data type

`Inherit: Inherit via internal rule (default) | Inherit: Same as input | fixdt([],16,0)`

**Accumulator** specifies the data type of the output of an accumulation operation in the Matrix 1-Norm block.

- `Inherit: Inherit via internal rule` — The block inherits the accumulator data type based on an internal rule. For more information on this rule, see “Inherit via Internal Rule”.
- `Inherit: Same as input` — The block specifies the accumulator data type to be the same as the input data type.
- `fixdt([],16,0)` — The block specifies an autosigned, binary-point, scaled, fixed-point data type with a word length of 16 bits and a fraction length of 0.

Alternatively, you can set the **Accumulator** data type by using the **Data Type Assistant**. To use the assistant, click the **Show data type assistant** button .

For more information on the data type assistant, see “Specify Data Types Using Data Type Assistant” (Simulink).


For illustrations on how to use the accumulator data type in this block, see Fixed-Point Conversion in “Extended Capabilities” on page 2-0 .

#### **Output** — Output data type

`Inherit: Same as accumulator (default) | Inherit: Same as input | fixdt([],16,0)`

**Output** specifies the data type of the output of the Matrix 1-Norm block.

- `Inherit: Same as input` — The block specifies the output data type to be the same as the input data type.
- `Inherit: Same as accumulator` — The block specifies the output data type to be the same as the accumulator data type.
- `fixdt([],16,0)` — The block specifies an autosigned, binary-point, scaled, fixed-point data type with a word length of 16 bits and a fraction length of 0.

Alternatively, you can set the **Output** data type by using the **Data Type Assistant**. To use the assistant, click the **Show data type assistant** button .

For more information on the data type assistant, see “Specify Data Types Using Data Type Assistant” (Simulink).

For more information on the output data type, see Fixed-Point Conversion in “Extended Capabilities” on page 2-0 .

**Output Minimum** — Minimum value that block can output

[ ] (default) | scalar

Specify the minimum value the block can output. Simulink uses this minimum value to perform:

- Simulation range checking. See “Specify Signal Ranges” (Simulink).
- Automatic scaling of fixed-point data types.

**Output Maximum** — Maximum value that block can output

[ ] (default) | scalar

Specify the maximum value the block can output. Simulink uses this maximum value to perform:

- Simulation range checking. See “Specify Signal Ranges” (Simulink).
- Automatic scaling of fixed-point data types.

**Lock data type settings against changes by the fixed-point tools** — Prevent fixed-point tools from overriding data types

off (default) | on

Select this parameter to prevent the fixed-point tools from overriding the data types you specify in the block dialog box.

---

**Note** Floating-point inheritance takes precedence over the data type settings defined on this tab. When inputs are floating point, the block ignores these settings and all internal data types are floating point.

---

## Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | double   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no                                      |
| <b>Multidimensional Signals</b> | no                                      |
| <b>Variable-Size Signals</b>    | no                                      |
| <b>Zero-Crossing Detection</b>  | no                                      |

## Version History

Introduced before R2006a

## References

- [1] Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

This diagram shows the data types used within the Matrix 1-Norm block for fixed-point signals.



The block calculations are all done in the accumulator data type until the max operation is performed. The result is then cast to the output data type. You can set the accumulator and output data types in the block dialog box.

## See Also

### Functions

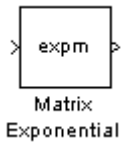
norm

### Blocks

Normalization | Reciprocal Condition

# Matrix Exponential

Compute matrix exponential



## Libraries:

DSP System Toolbox / Math Functions / Matrices and Linear Algebra / Matrix Operations

## Description

The Matrix Exponential block computes the matrix exponential of a square matrix using a scaling and squaring algorithm with a Pade approximation.

## Ports

### Input

**Input 1** — Data input  
matrix

Specify the input data as a square matrix.

Data Types: `single` | `double`

### Output

**Output 1** — Matrix exponential output  
matrix

Matrix exponential output of the input square matrix.

Data Types: `single` | `double`

## Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | <code>double</code>   <code>single</code> |
| <b>Direct Feedthrough</b>       | no  |
| <b>Multidimensional Signals</b> | no  |
| <b>Variable-Size Signals</b>    | no  |
| <b>Zero-Crossing Detection</b>  | no  |

## Version History

Introduced before R2006a

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

Generated code relies on the memcpy or memset function (`string.h`) under certain conditions.

## **See Also**

### **Functions**

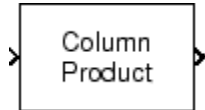
`expm`

### **Blocks**

Array-Vector Multiply | Product | Matrix Product | Dot Product

# Matrix Product

Multiply matrix elements along rows, columns, or entire input



## Library

Math Functions / Matrices and Linear Algebra / Matrix Operations

dspmtx3

## Description

The Matrix Product block multiplies the elements of an  $M$ -by- $N$  input matrix  $u$  along its rows, its columns, or over all its elements.

When the **Multiply over** parameter is set to **Rows**, the block multiplies across the elements of each row and outputs the resulting  $M$ -by-1 matrix. The block treats length- $N$  unoriented vector input as a 1-by- $N$  matrix.

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ u_{21} & u_{22} & u_{23} \\ u_{31} & u_{32} & u_{33} \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} \left( \prod_{j=1}^3 u_{1j} \right) \\ \left( \prod_{j=1}^3 u_{2j} \right) \\ \left( \prod_{j=1}^3 u_{3j} \right) \end{bmatrix}$$

When the **Multiply over** parameter is set to **Columns**, the block multiplies down the elements of each column and outputs the resulting 1-by- $N$  matrix. The block treats length- $M$  unoriented vector input as an  $M$ -by-1 matrix.

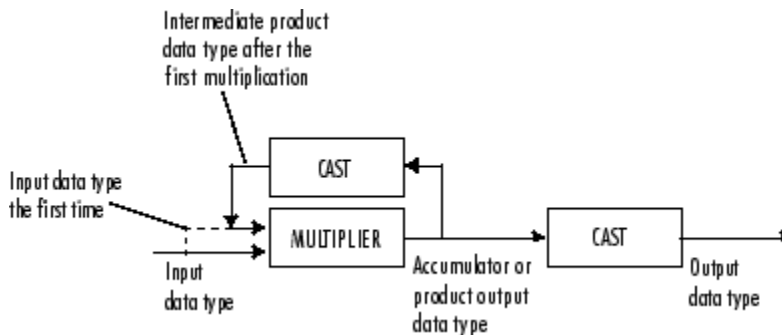
$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ u_{21} & u_{22} & u_{23} \\ u_{31} & u_{32} & u_{33} \end{bmatrix} \\ \Downarrow \\ [y_1 \ y_2 \ y_3] = \left[ \left( \prod_{i=1}^3 u_{i1} \right) \left( \prod_{i=1}^3 u_{i2} \right) \left( \prod_{i=1}^3 u_{i3} \right) \right]$$

When the **Multiply over** parameter is set to **Entire input**, the block multiplies all the elements of the input together and outputs the resulting scalar.

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ u_{21} & u_{22} & u_{23} \\ u_{31} & u_{32} & u_{33} \end{bmatrix} = y = \left( \prod_{i=1}^3 \prod_{j=1}^3 u_{ij} \right)$$

## Fixed-Point Data Types

The following diagram shows the data types used within the Matrix Product block for fixed-point signals.



The output of the multiplier is in the product output data type when at least one of the inputs to the multiplier is real. When both of the inputs to the multiplier are complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication performed, see “Multiplication Data Types”. You can set the accumulator, product output, intermediate product, and output data types in the block dialog as discussed in “Parameters” on page 2-952 below.

## Parameters

### Main Tab

#### Multiply over

Indicate whether to multiply together the elements of each row, each column, or the entire input.

#### Data Types Tab

---

**Note** Floating-point inheritance takes precedence over the data type settings defined on this pane. When inputs are floating point, the block ignores these settings, and all internal data types are floating point.

---

#### Rounding mode

Specify the rounding mode for fixed-point operations as one of the following:

- Floor
- Ceiling
- Convergent
- Nearest
- Round
- Simplest
- Zero

For more details, see rounding mode.




### Saturate on integer overflow

When you select this parameter, the block saturates the result of its fixed-point operation. When you clear this parameter, the block wraps the result of its fixed-point operation. For details on `saturate` and `wrap`, see `overflow mode for fixed-point operations`.

### Intermediate product

Specify the intermediate product data type. As shown in “Fixed-Point Data Types” on page 2-952, the output of the multiplier is cast to the intermediate product data type before the next element of the input is multiplied into it. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as input`
- An expression that evaluates to a valid data type, for example, `fixdt([], 16, 0)`


Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Product output** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in *Simulink User's Guide* (Simulink) for more information.

### Product output

Specify the product output data type. See “Fixed-Point Data Types” on page 2-952 and “Multiplication Data Types” for illustrations depicting the use of the product output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.
- A rule that inherits a data type, for example, `Inherit: Same as input`.
- An expression that evaluates to a valid data type, for example, `fixdt([], 16, 0)`


Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Product output** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in *Simulink User's Guide* (Simulink) for more information.

### Accumulator

Specify the accumulator data type. See “Fixed-Point Data Types” on page 2-952 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.
- A rule that inherits a data type, for example, `Inherit: Same as input`.
- A rule that inherits a data type, for example, `Inherit: Same as product output`.
- An expression that evaluates to a valid data type, for example, `fixdt([], 16, 0)`


Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Accumulator** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in *Simulink User's Guide* (Simulink) for more information.

**Output**

Specify the output data type. See “Fixed-Point Data Types” on page 2-952 for illustrations depicting the use of the output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as product output`
- A rule that inherits a data type, for example, `Inherit: Same as input`.
- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Output** parameter.

See “Control Data Types of Signals” (Simulink) in *Simulink User's Guide* (Simulink) for more information.

**Minimum**

Specify the minimum value that the block should output. The default value is `[]` (unspecified). Simulink uses this value to perform:

- Simulation range checking (see “Specify Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

**Maximum**

Specify the maximum value that the block should output. The default value is `[]` (unspecified). Simulink uses this value to perform:

- Simulation range checking (see “Specify Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

**Lock data type settings against changes by the fixed-point tools**

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

**Intermediate Product Data Type Assistant Parameters****Mode**

Select how you would like to specify the data type properties of the **Intermediate product** data type. You can choose:

- **Inherit** — Lets you specify a rule for inheriting a data type, for example, `Inherit: Inherit via internal rule`
- **Fixed point** — Lets you specify the fixed-point attributes of the data type.
- **Expression** — Lets you specify an expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

**Signedness**

Specify the **Signedness** for the **Intermediate product** data type.

**Scaling**

Specify the **Scaling** for the **Intermediate product** data type.

For more information see “Scaling” in the DSP System Toolbox User's Guide.

#### **Word length**

Specify the **Word length** for the **Intermediate product** data type.

#### **Fraction length**

Specify the **Fraction length** for the **Intermediate product** data type.

#### **Data type override**

Specify the data type override mode. You can select one of the following options:

- **Inherit** — Inherits the data type override setting specified for the model.
- **Off** — Ignores the data type override setting specified for the model and uses the fixed-point data type you specify

This parameter appears only when you set the **Mode** parameter to **Built in** or **Fixed Point**. For more information, see “Specify Data Types Using Data Type Assistant” (Simulink).

### **Product Output Data Type Assistant Parameters**

#### **Mode**

Select how you would like to specify the data type properties of the **Product output** data type. You can choose:

- **Inherit** — Lets you specify a rule for inheriting a data type, for example, `Inherit: Inherit via internal rule`
- **Fixed point** — Lets you specify the fixed-point attributes of the data type.
- **Expression** — Lets you specify an expression that evaluates to a valid data type, for example, `fixdt([], 16, 0)`

#### **Signedness**

Specify the **Signedness** for the **Product output** data type.

#### **Scaling**

Specify the **Scaling** for the **Product output** data type.

For more information see “Scaling” in the DSP System Toolbox User's Guide.

#### **Word length**

Specify the **Word length** for the **Product output** data type.

**Fraction length**

Specify the **Fraction length** for the **Product output** data type.

**Data type override**

Specify the data type override mode. You can select one of the following options:

- **Inherit** — Inherits the data type override setting specified for the model.
- **Off** — Ignores the data type override setting specified for the model and uses the fixed-point data type you specify

This parameter appears only when you set the **Mode** parameter to **Built in** or **Fixed Point**. For more information, see “Specify Data Types Using Data Type Assistant” (Simulink).

**Accumulator Data Type Assistant Parameters****Mode**

Select how you would like to specify the data type properties of the **Accumulator** data type. You can choose:

- **Inherit** — Lets you specify a rule for inheriting a data type, for example, `Inherit: Inherit via internal rule`
- **Fixed point** — Lets you specify the fixed-point attributes of the data type.
- **Expression** — Lets you specify an expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

**Signedness**

Specify the **Signedness** for the **Accumulator** data type.

**Scaling**

Specify the **Scaling** for the **Accumulator** data type.

For more information see “Scaling” in the DSP System Toolbox User's Guide.

**Word length**

Specify the **Word length** for the **Accumulator** data type.

**Fraction length**

Specify the **Fraction length** for the **Accumulator** data type.

**Data type override**

Specify the data type override mode. You can select one of the following options:

- **Inherit** — Inherits the data type override setting specified for the model.
- **Off** — Ignores the data type override setting specified for the model and uses the fixed-point data type you specify

This parameter appears only when you set the **Mode** parameter to **Built in** or **Fixed Point**. For more information, see “Specify Data Types Using Data Type Assistant” (Simulink).

### Output Data Type Assistant Parameters

#### Mode

Select how you would like to specify the data type properties of the **Output** data type. You can choose:

- **Inherit** — Lets you specify a rule for inheriting a data type, for example, `Inherit via internal rule`
- **Fixed point** — Lets you specify the fixed-point attributes of the data type.
- **Expression** — Lets you specify an expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

#### Signedness

Specify the **Signedness** for the **Output** data type.

#### Scaling

Specify the **Scaling** for the **Output** data type.

For more information see “Scaling” in the DSP System Toolbox User's Guide.

#### Word length

Specify the **Word length** for the **Output** data type.

#### Fraction length

Specify the **Fraction length** for the **Output** data type.

#### Data type override

Specify the data type override mode. You can select one of the following options:

- **Inherit** — Inherits the data type override setting specified for the model.
- **Off** — Ignores the data type override setting specified for the model and uses the fixed-point data type you specify

This parameter appears only when you set the **Mode** parameter to **Built in** or **Fixed Point**. For more information, see “Specify Data Types Using Data Type Assistant” (Simulink).

## Supported Data Types

| Port   | Supported Data Types  |
|--------|---|
| Input  | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul> |
| Output | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul> |

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Generated code relies on the memcpy or memset function (string.h) under certain conditions.

### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

## See Also

### Functions

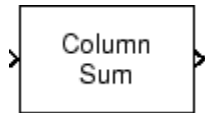
prod

### Blocks

Array-Vector Multiply | Matrix Sum

## Matrix Sum (Obsolete)

Sum matrix elements along rows, columns, or entire input



## Library

Math Functions / Matrices and Linear Algebra / Matrix Operations

`dspobslib`

## Description

The Matrix Sum block sums the elements of an  $M$ -by- $N$  input matrix  $u$  along its rows, its columns, or over all its elements.

When the **Sum over** parameter is set to **Rows**, the block sums across the elements of each row and outputs the resulting  $M$ -by-1 matrix. A length- $N$  1-D vector input is treated as a 1-by- $N$  matrix.

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ u_{21} & u_{22} & u_{23} \\ u_{31} & u_{32} & u_{33} \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} \left( \sum_{j=1}^3 u_{1j} \right) \\ \left( \sum_{j=1}^3 u_{2j} \right) \\ \left( \sum_{j=1}^3 u_{3j} \right) \end{bmatrix}$$

When the **Sum over** parameter is set to **Columns**, the block sums down the elements of each column and outputs the resulting 1-by- $N$  matrix. A length- $M$  1-D vector input is treated as a  $M$ -by-1 matrix.

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ u_{21} & u_{22} & u_{23} \\ u_{31} & u_{32} & u_{33} \end{bmatrix} \\ \Downarrow \\ [y_1 \ y_2 \ y_3] = \left[ \left( \sum_{i=1}^3 u_{i1} \right) \left( \sum_{i=1}^3 u_{i2} \right) \left( \sum_{i=1}^3 u_{i3} \right) \right]$$

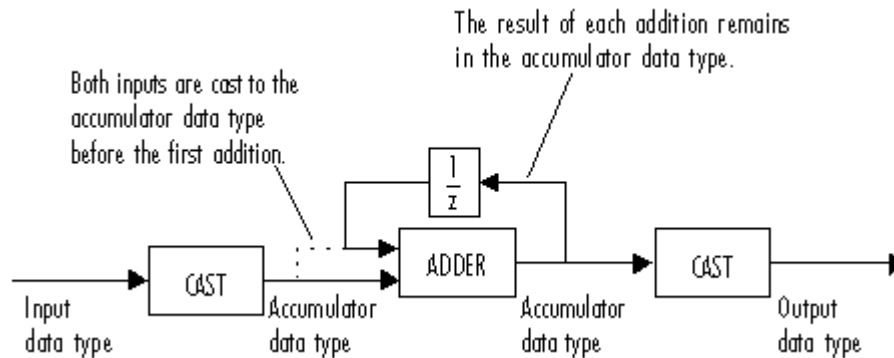
When the **Sum over** parameter is set to **Entire** input, the block sums all the elements of the input together and outputs the resulting scalar.

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ u_{21} & u_{22} & u_{23} \\ u_{31} & u_{32} & u_{33} \end{bmatrix} = y = \left( \sum_{i=1}^3 \sum_{j=1}^3 u_{ij} \right)$$

The output of the Matrix Sum block has the same frame status as the input. This block accepts real and complex fixed-point and floating-point inputs except for complex unsigned fixed-point inputs.

### Fixed-Point Data Types

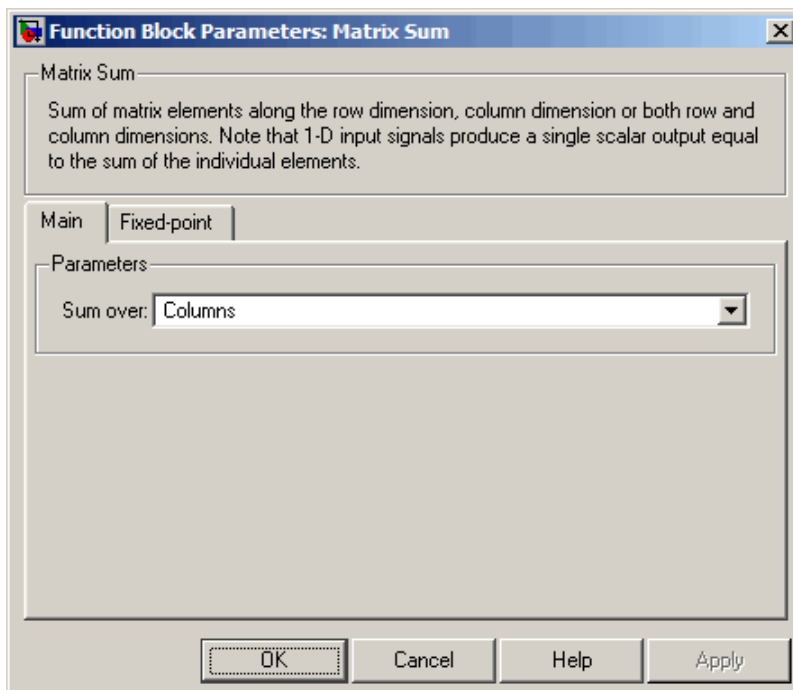
The following diagram shows the data types used within the Matrix Sum block for fixed-point signals.



You can set the accumulator and output data types in the block dialog as discussed in “Dialog Box” on page 2-960 below.

### Dialog Box

The **Main** pane of the Matrix Sum block dialog appears as follows.

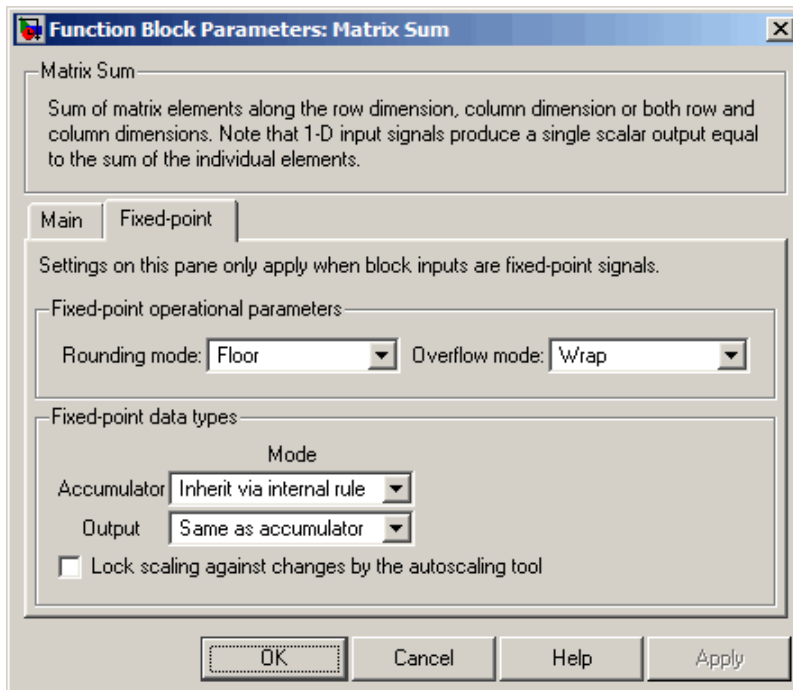


#### Sum over

Indicate whether to sum the elements of each row, each column, or of the entire input.

The **Fixed-point** pane of the Matrix Sum block dialog appears as follows.





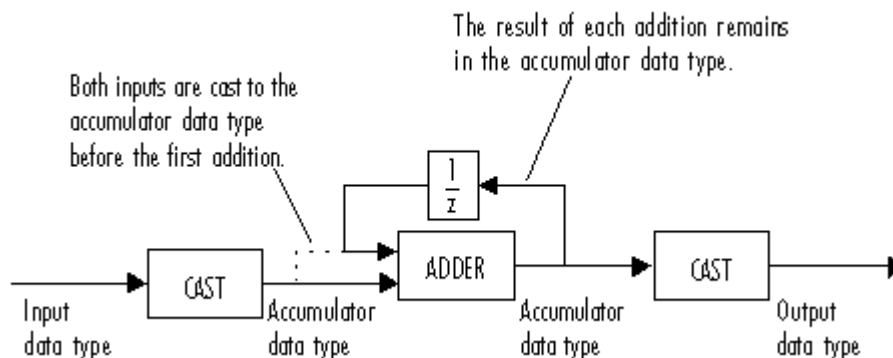
### Rounding mode

Select the rounding mode for fixed-point operations.

### Overflow mode

Select the overflow mode for fixed-point operations.

### Accumulator



As depicted above, the elements of the block input are cast to the accumulator data type before they are added together. The output of the adder remains in the accumulator data type as each element of the input is added to it. Use this parameter to specify how you would like to designate this accumulator word and fraction lengths:

- When you select *Inherit via internal rule*, the accumulator word length and fraction length are calculated automatically. For information about how the accumulator word and fraction lengths are calculated when an internal rule is used, see “*Inherit via Internal Rule*”.
- When you select *Same as input*, these characteristics match those of the input to the block.

- When you select **Binary point scaling**, you can enter the word length and the fraction length of the accumulator, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the accumulator. This block requires power-of-two slope and a bias of zero.

### Output

Choose how you specify the output word length and fraction length:

- When you select **Same as accumulator**, these characteristics match those of the accumulator.
- When you select **Same as input**, these characteristics match those of the input to the block.
- When you select **Binary point scaling**, you can enter the word length and the fraction length of the output, in bits.
- When you select **Slope and bias scaling**, you can enter the word length, in bits, and the slope of the output. This block requires power-of-two slope and a bias of zero.

### Lock scaling against changes by the autoscaling tool

Select this parameter to prevent any fixed-point scaling you specify in this block mask from being overridden by the autoscaling feature of the Fixed-Point Tool. See the `fxptdlg` reference page for more information.

## Supported Data Types

| Port   | Supported Data Types  |
|--------|---|
| Input  | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul> |
| Output | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul> |

### See Also

Matrix Product  
Matrix Multiply  
sum

DSP System Toolbox  
DSP System Toolbox  
MATLAB

## Version History

Introduced in R2008b

# Matrix Viewer

(To be removed) Display matrices as color images

---

**Note** The Matrix Viewer block will be removed in a future release. Use a different scope instead, see “Scopes and Data Logging”.

---



## Library

Sinks

dspsnks4

## Description

The Matrix Viewer block displays an  $M$ -by- $N$  matrix input by mapping the matrix element values to a specified range of colors. The display is updated as each new input is received. This block treats an unoriented length  $M$  vector input as an  $M$ -by-1 matrix.

You can use the Matrix Viewer block in models running in Normal or Accelerator simulation modes. The software does not support this block in models running in Rapid Accelerator or External mode. For more information about these modes, see “How Acceleration Modes Work” (Simulink) in the *Simulink User's Guide*.

## Image Properties

Select the **Image Properties** tab to show the image property parameters, which control the colormap and display.

You specify the mapping of matrix element values to colors in the **Colormap matrix**, **Minimum input value**, and **Maximum input value** parameters. For a colormap with  $L$  colors, the colormap matrix has dimension  $L$ -by-3, with one row for each color and one column for each element of the RGB triple that defines the color. Examples of RGB triples are

```
[ 1  0  0 ] (red)
[ 0  0  1 ] (blue)
[0.8 0.8 0.8] (light gray)
```

MATLAB provides a number of functions for generating predefined colormaps, such as `hot`, `cool`, `bone`, and `autumn`. Each of these functions accepts the colormap size as an argument, and can be used in the **Colormap matrix** parameter. For example, when you specify `gray(128)` for the **Colormap matrix** parameter, the matrix is displayed in 128 shades of gray. The color in the first row of the colormap matrix represents the value specified by the **Minimum input value** parameter, and the color in the last row represents the value specified by the **Maximum input value** parameter.

Values between the minimum and maximum are quantized and mapped to the intermediate rows of the colormap matrix.

The documentation for the MATLAB `colormap` function provides complete information about specifying colormap matrices, and includes a complete list of the available colormap functions.

### Axis Properties

Select the **Axis Properties** tab to show the axis property parameters, which control labeling and positioning.

The **Axis origin** parameter determines where the first element of the input matrix,  $U(1,1)$ , is displayed. When you specify `Upper left corner`, the matrix is displayed in matrix orientation, with  $U(1,1)$  in the upper-left corner.

$$\begin{bmatrix} U_{11} & U_{12} & U_{13} & U_{14} \\ U_{21} & U_{22} & U_{23} & U_{24} \\ U_{31} & U_{32} & U_{33} & U_{34} \\ U_{41} & U_{42} & U_{43} & U_{44} \end{bmatrix}$$

When you specify `Lower left corner`, the matrix is flipped vertically to image orientation, with  $U(1,1)$  in the lower-left corner.

$$\begin{bmatrix} U_{41} & U_{42} & U_{43} & U_{44} \\ U_{31} & U_{32} & U_{33} & U_{34} \\ U_{21} & U_{22} & U_{23} & U_{24} \\ U_{11} & U_{12} & U_{13} & U_{14} \end{bmatrix}$$

**Axis zoom**, when selected, causes the image display to completely fill the figure window. Axis titles are not displayed. This option can also be selected from the pop-up menu that is displayed when you right-click in the figure window. When **Axis zoom** is cleared, the axis labels and titles are displayed in a gray border surrounding the image axes.

To customize the axis tick range, set **Axis tick mode** to `User-defined` and specify a two-element row vector, `[Minvalue Maxvalue]`, in **X-tick range** and **Y-tick range**. The matrix data does not change even when the axes ticks change. For example, consider displaying a matrix on axes with ticks ranging from 0 to 10. When you set **X-tick range** and **Y-tick range** to `[0 1]`, the range of ticks changes from `[0 10]` to `[0 1]`.

### Figure Window

The image title in the figure title bar is the same as the block title. The axis tick marks reflect the size of the input matrix; the x-axis is numbered from 1 to  $N$  (number of columns), and the y-axis is numbered from 1 to  $M$  (number of rows).

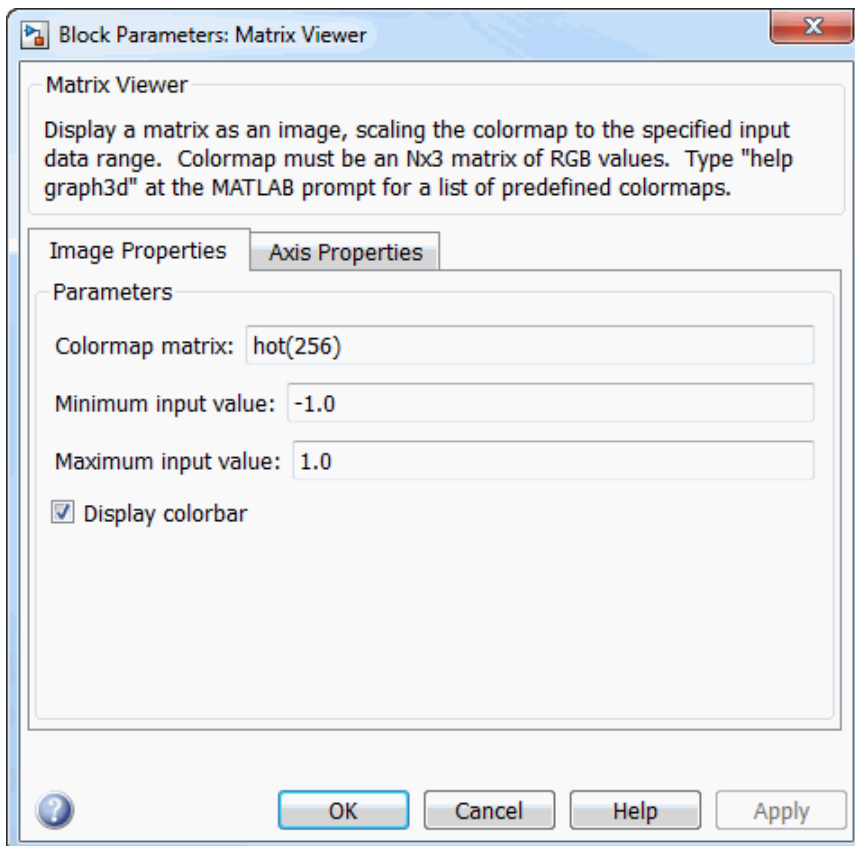
Right-click the image in the figure window to access the following menu items:

- **Refresh** erases all data on the scope display except for the most recent image.
- **Autoscale** recomputes the minimum and maximum input values to fit the range of values observed in a series of 10 consecutive inputs. The numerical limits selected by the autoscale feature are shown in the **Minimum input value** and **Maximum input value** parameters, where you can make further adjustments to them manually.

- **Axis zoom**, when selected, causes the image to completely fill the figure window. Axis titles are not displayed. When **Axis zoom** is cleared, the axis labels and titles are displayed in a gray border surrounding the scope axes. This option can also be set in the Axis Properties pane of the parameter dialog box.
- **Colorbar**, when selected, displays a bar with the specified colormap to the right of the image axes.
- **Save Position** automatically updates the **Figure position** parameter in the **Axis Properties** pane to reflect the figure window's current position and size on the screen. To make the scope window open at a particular location on the screen when the simulation runs, drag the window to the desired location, resize it, and select **Save Position**. The parameter dialog box must be closed when you select **Save Position** for the **Figure position** parameter to be updated.

## Dialog Box

The **Image Properties** pane of the Matrix Viewer block appears as follows.



### Colormap matrix

A 3-column matrix defining the colormap as a set of RGB triples, or a call to a colormap-generating function such as `hot` or `spring`.

- See the `colormap` function for a list of colormap-generating functions.
- A color name or a short name: 'red' (or 'r'), 'green' (or 'g'), 'blue' (or 'b'), 'cyan' (or 'c'), 'magenta' (or 'm'), 'yellow' (or 'y'), 'black' (or 'k'), 'white' (or 'w').

- An RGB triplet specified as a 1-by-3 vector whose elements are the intensities of the red, green, and blue components of a color. The intensities must be in the range [0, 1]. For example, [0.7 0.8 1] specifies light blue.

Tunable (Simulink).

### Minimum input value

The input value to be mapped to the color defined in the first row of the colormap matrix. Right-click in the figure window and select `Autoscale` from pop-up menu to set this parameter to the minimum value observed in a series of 10 consecutive matrix inputs. Tunable (Simulink).

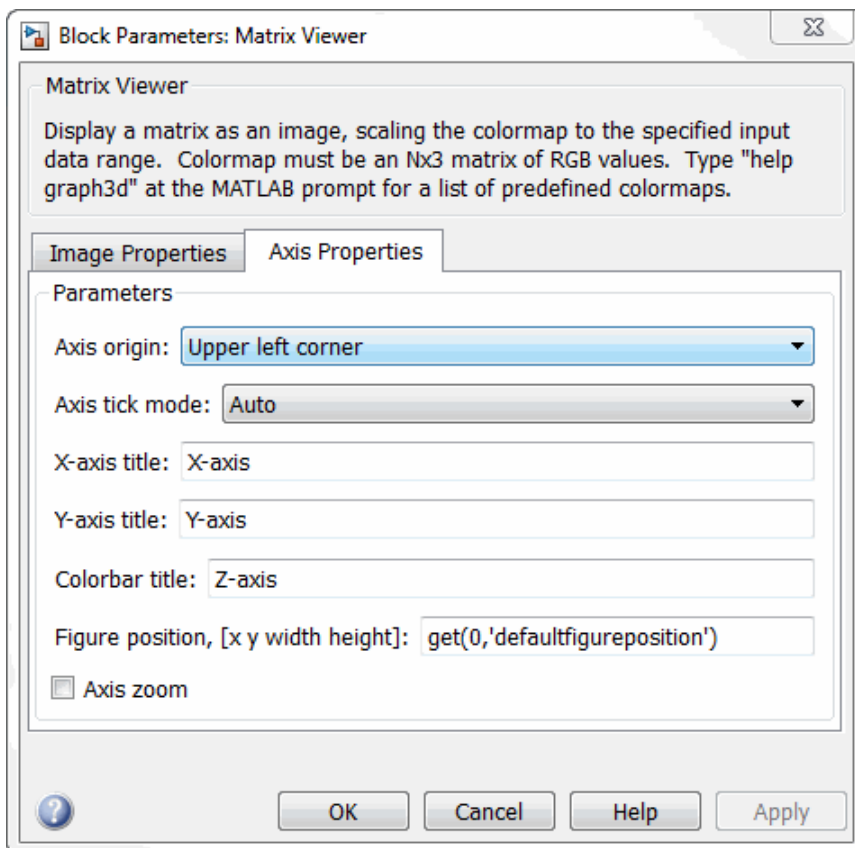
### Maximum input value

The input value to be mapped to the color defined in the last row of the colormap matrix. Right-click in the figure window and select `Autoscale` from the pop-up menu to set this parameter to the maximum value observed in a series of 10 consecutive matrix inputs. Tunable (Simulink).

### Display colorbar

Select to display a bar with the selected colormap to the right of the image axes. Tunable (Simulink).

The **Axis Properties** pane of the Matrix Viewer block appears as follows.



### Axis origin

The position within the axes where the first element of the input matrix,  $U(1,1)$ , is plotted; bottom left or top left. Tunable (Simulink).

**Axis tick mode**

The mode used to specify the axis tick range. When you select **Auto**, the x-axis tick range and y-axis tick range are chosen based on the input signal. When you select **User-defined**, the axis tick range is set based on the values you specify in **X-tick range** and **Y-tick range**. The default is **Auto**.

**X-tick range**

x-axis tick range, specified as a two-element row vector of real scalars,  $[X_{\min} \ X_{\max}]$ . This parameter applies only when you set **Axis tick mode** to **User-defined**. When you change **X-tick range** during simulation, the axis range on the Matrix Viewer block updates in real time. The default is  $[0 \ 1]$ .

**Y-tick range**

y-axis tick range, specified as a two-element row vector of real scalars,  $[Y_{\min} \ Y_{\max}]$ . This parameter applies only when you set **Axis tick mode** to **User-defined**. When you change **Y-tick range** during simulation, the axis range on the Matrix Viewer block updates in real time. The default is  $[0 \ 1]$ .

**X-axis title**

The text to be displayed below the x-axis. Tunable (Simulink).

**Y-axis title**

The text to be displayed to the left of the y-axis. Tunable (Simulink).

**Colorbar title**

The text to be displayed to the right of the color bar, when **Display colorbar** is currently selected. Tunable (Simulink).

**Figure position, [x y width height]**

A 4-element vector of the form  $[x \ y \ width \ height]$  specifying the position of the figure window, where  $(0, 0)$  is the lower-left corner of the display. Tunable (Simulink).

**Axis zoom**

Resizes the image to fill the figure window. Tunable (Simulink).

**Supported Data Types**

| Port  | Supported Data Types   |
|-------|--|
| Input | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed and unsigned)</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul> |

**Version History**

**Introduced before R2006a**

**R2020b: Matrix Viewer block will be removed**

*Not recommended starting in R2020b*

The Matrix Viewer block will be removed in a future release. Use a different scope instead, see “Scopes and Data Logging”.

## **Extended Capabilities**

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

This block can be used for simulation visibility in subsystems that generate HDL code, but is not included in the hardware implementation.

### **Fixed-Point Conversion**

Design and simulate fixed-point systems using Fixed-Point Designer™.

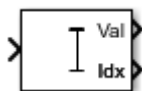
## **See Also**

Spectrum Analyzer | [colormap](#) | [image](#)



# Maximum

Maximum values of input or sequence of inputs



## Libraries:

DSP System Toolbox / Statistics

DSP System Toolbox HDL Support / Statistics

## Description

The Maximum block identifies the value and position of the largest element in each row or column of the input, or along vectors of a specified dimension of the input. It can also compute the maximum value of the entire input. The Maximum block can also track the maximum values in a sequence of inputs over a period of time. The **Mode** parameter specifies the block's mode of operation and can be set to one of the following:

- **Value** — The block outputs the maximum values in the specified dimension.
- **Index** — The block outputs the index array of the maximum values in the specified dimension.
- **Value and Index** — The block outputs the maximum values and the corresponding index array in the specified dimension.
- **Running** — The block tracks the maximum values in a sequence of inputs over a period of time.

You can specify the dimension using the **Find the maximum value over** parameter.

---

**Note** The **Running** mode in the Maximum block will be removed in a future release. To compute the running maximum in Simulink, use the Moving Maximum block instead.

---

## Ports

### Input

**In** — Data input

vector | matrix |  $N$ -D array

The block accepts real-valued or complex-valued multichannel and multidimensional inputs. The input can be floating-point, fixed-point, or Boolean. Real fixed-point inputs can be either signed or unsigned. Complex fixed-point inputs must be signed.

This port is unnamed until you set the **Mode** parameter to **Running** and the **Reset port** parameter to any option other than **None**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point`

**Rst** — Reset port

scalar

Specify the reset event that causes the block to reset the running maximum. The sample time of the **Rst** input must be a positive integer multiple of the input sample time.

### Dependencies

To enable this port, set the **Mode** parameter to **Running** and the **Reset port** parameter to any option other than **None**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `Boolean`

### Output

**Val** — Maximum values along the specified dimension

`scalar` | `vector` | `matrix` | *N-D array*

The data type of the maximum value matches the data type of the input.

When the **Mode** parameter is set to either **Value and Index** or **Value**, the following applies:

- The size of the dimension for which the block computes the maximum value is 1. The sizes of all other dimensions match those of the input array. For example, when the input is an *M*-by-*N*-by-*P* array, with the dimension set to 1, the block outputs a 1-by-*N*-by-*P* array. When the dimension is set to 3, the block outputs a two-dimensional *M*-by-*N* matrix.
- When the input is an *M*-by-*N* matrix, with the dimension set to 1, the block outputs a 1-by-*N* matrix.

If you specify the block to compute the maximum value over the entire input, the block outputs a scalar.

When the **Mode** parameter is set **Running**, the block tracks the maximum value of each channel in a time sequence of *M*-by-*N* inputs. In this mode, you must also specify the **Input processing** parameter as one of the following:

- **Elements as channels (sample based)** — The block treats each element of the input as a separate channel. For a three-dimensional input signal of size *M*-by-*N*-by-*P*, the block outputs an *M*-by-*N*-by-*P* array. Each  $y_{ijk}$  element of the output contains the maximum value observed in element  $u_{ijk}$  for all inputs since the last reset.

When a reset event occurs, the running maximum  $y_{ijk}$  in the current frame is reset to the element  $u_{ijk}$ .

- **Columns as channels (frame based)** — The block treats each column of the input as a separate channel. This option does not support an *N*-dimensional input signal, where  $N > 2$ . For a two-dimensional input signal of size *M*-by-*N*, the block outputs an *M*-by-*N* matrix. Each element  $y_{ij}$  of the output contains the maximum value observed in the *j*th column of all inputs since the last reset, up to and including element  $u_{ij}$  of the current input.

When a reset event occurs, the running maximum for each channel becomes the maximum value of all the samples in the current input frame, up to and including the current input sample.

The block resets the running maximum whenever a reset event is detected at the optional **Rst** port. The reset sample time must be a positive integer multiple of the input sample time.

### Dependencies

To enable this port, set the **Mode** parameter to either **Value and Index** or **Value**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point`

**Idx** — Index of the maximum values along the specified dimension  
 scalar | vector | matrix | *N*-D array

When the input is `double`, the index values are also `double`. Otherwise, the index values are `uint32`.

### Dependencies

To enable this port, set the **Mode** parameter to either `Value` and `Index` or `Index`.

Data Types: `double` | `uint32`

## Parameters

### Main Tab

**Mode** — Mode in which the block operates  
`Value` and `Index` (default) | `Value` | `Index` | `Running`

When the **Mode** parameter is set to:

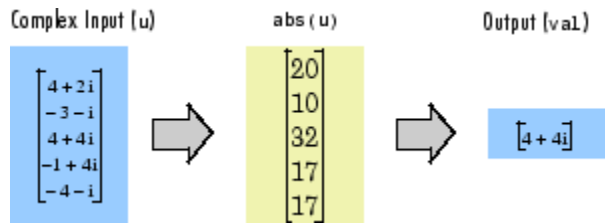
- **Value** — The block computes the maximum value in each row or column of the input, or along vectors of a specified dimension of the input. It can also compute the maximum value of the entire input at each sample time, and outputs the array, *y*. Each element in the output is the maximum value in the corresponding column, row, vector, or entire input. The output *y* depends on the setting of the **Find the maximum value over** parameter. Consider a three dimensional input signal of size *M*-by-*N*-by-*P*. Set **Find the maximum value over** to:
  - **Each row** — The output *y* at each sample time consists of an *M*-by-1-by-*P* array, where each element contains the maximum value of each vector over the second dimension of the input. For an *M*-by-*N* matrix input, the output at each sample time is an *M*-by-1 column vector.
  - **Each column** — The output *y* at each sample time consists of a 1-by-*N*-by-*P* array, where each element contains the maximum value of each vector over the first dimension of the input. For an *M*-by-*N* matrix input, the output at each sample time is a 1-by-*N* row vector.

In this mode, the block treats length-*M* unoriented vector inputs as *M*-by-1 column vectors.

- **Entire input** — The output *y* at each sample time is a scalar that contains the maximum value in the *M*-by-*N*-by-*P* input matrix.
- **Specified dimension** — The output *y* at each sample time depends on **Dimension**. If **Dimension** is set to 1, the output is the same as when you select **Each column**. If **Dimension** is set to 2, the output is the same as when you select **Each row**. If **Dimension** is set to 3, the output at each sample time is an *M*-by-*N* matrix containing the maximum value of each vector over the third dimension of the input.

### Complex Inputs

For complex inputs, the block selects the value in each row or column of the input, along vectors of a specified dimension of the input, or of the entire input that has the maximum magnitude squared as shown in the following figure. For complex value  $u = a + bi$ , the magnitude squared is  $a^2 + b^2$ .



- **Index** — The block computes the maximum value in each row or column of the input, along vectors of a specified dimension of the input, or of the entire input, and outputs the index array  $I$ . Each element in  $I$  is an integer indexing the maximum value in the corresponding column, row, vector, or entire input. The output  $I$  depends on the setting of the **Find the maximum value over** parameter. Consider a three-dimensional input signal of size  $M$ -by- $N$ -by- $P$ :
  - **Each row** — The output  $I$  at each sample time consists of an  $M$ -by-1-by- $P$  array, where each element contains the index of the maximum value of each vector over the second dimension of the input. For an input that is an  $M$ -by- $N$  matrix, the output at each sample time is an  $M$ -by-1 column vector.
  - **Each column** — The output  $I$  at each sample time consists of a 1-by- $N$ -by- $P$  array, where each element contains the index of the maximum value of each vector over the first dimension of the input. For an input that is an  $M$ -by- $N$  matrix, the output at each sample time is a 1-by- $N$  row vector.

In this mode, the block treats length- $M$  unoriented vector inputs as  $M$ -by-1 column vectors.

- **Entire input** — The output  $I$  at each sample time is a 1-by-3 vector that contains the location of the maximum value in the  $M$ -by- $N$ -by- $P$  input matrix. For an input that is an  $M$ -by- $N$  matrix, the output is a 1-by-2 vector.
- **Specified dimension** — The output  $I$  at each sample time depends on **Dimension**. If **Dimension** is set to 1, the output is the same as when you select **Each column**. If **Dimension** is set to 2, the output is the same as when you select **Each row**. If **Dimension** is set to 3, the output at each sample time is an  $M$ -by- $N$  matrix containing the indices of the maximum values of each vector over the third dimension of the input.

When a maximum value occurs more than once, the computed index corresponds to the first occurrence. For example, when the input is the column vector  $[3 \ 2 \ 1 \ 2 \ 3]'$ , the computed one-based index of the maximum value is 1, rather than 5 when **Each column** is selected.

- **Value and Index** — The block outputs the maximum value in each row or column of the input, along vectors of a specified dimension of the input, or of the entire input, and the corresponding index array  $I$ .
- **Running** — The block tracks the maximum value of each channel in a time sequence of  $M$ -by- $N$  inputs. In this mode, you must also specify the **Input processing** parameter as one of the following:
  - **Elements as channels (sample based)** — The block treats each element of the input as a separate channel. For a three-dimensional input signal of size  $M$ -by- $N$ -by- $P$ , the block outputs an  $M$ -by- $N$ -by- $P$  array. Each  $y_{ijk}$  element of the output contains the maximum value observed in element  $u_{ijk}$  for all inputs since the last reset.
 

When a reset event occurs, the running maximum  $y_{ijk}$  in the current frame is reset to the element  $u_{ijk}$ .
  - **Columns as channels (frame based)** — The block treats each column of the input as a separate channel. This option does not support an  $N$ -dimensional input signal, where  $N > 2$ .

For a two-dimensional input signal of size  $M$ -by- $N$ , the block outputs an  $M$ -by- $N$  matrix. Each element  $y_{ij}$  of the output contains the maximum value observed in the  $j$ th column of all inputs since the last reset, up to and including element  $u_{ij}$  of the current input.

When a reset event occurs, the running maximum for each channel becomes the maximum value of all the samples in the current input frame, up to and including the current input sample.

The block resets the running maximum whenever a reset event is detected at the optional **Rst** port. The reset sample time must be a positive integer multiple of the input sample time.

### Running Mode for Variable-Size Inputs

When the input is a variable-size signal, and you set the **Mode** to **Running**, then:

- If you set the **Input processing** parameter to **Elements as channels (sample based)**, the state is reset.
- If you set the **Input processing** parameter to **Columns as channels (frame based)**, then:
  - When the input size difference is in the number of channels (columns), the state is reset.
  - When the input size difference is in the length of channels (rows), there is no reset and the running operation is carried out as usual.

**Index base** — Base of the maximum value index

One (default) | Zero

Specify whether the index of the maximum value is reported using one-based or zero-based numbering.

### Dependencies

To enable this parameter, set **Mode** to either **Index** or **Value** and **Index**.

**Find the maximum value over** — Dimension over which the block computes the maximum value

Each column (default) | Each row | Entire input | Specified dimension

- **Each column** — The block outputs the maximum value over each column.
- **Each row** — The block outputs the maximum value over each row.
- **Entire input** — The block outputs the maximum value over the entire input.
- **Specified dimension** — The block outputs the maximum value over the dimension, specified in the **Dimension** parameter.

### Dependencies

To enable this parameter, set **Mode** to **Value** and **Index**, **Value**, or **Index**.

**Dimension** — Custom dimension

1 (default) | scalar

Specify the dimension (one-based value) of the input signal over which the block computes the maximum. The value of this parameter must be greater than 0 and less than the number of dimensions in the input signal.

### Dependencies

To enable this parameter, set **Find the maximum value over** to Specified dimension.

**Input processing** — Method to process the input in running mode

Columns as channels (frame based) (default) | Elements as channels (sample based)

- **Columns as channels (frame based)** — The block treats each column of the input as a separate channel. This option does not support an  $N$ -dimensional input signal, where  $N > 2$ . For a two-dimensional input signal of size  $M$ -by- $N$ , the block outputs an  $M$ -by- $N$  matrix. Each element  $y_{ij}$  of the output contains the maximum value observed in the  $j$ th column of all inputs since the last reset, up to and including element  $u_{ij}$  of the current input.

When a reset event occurs, the running maximum for each channel becomes the maximum value of all the samples in the current input frame, up to and including the current input sample.

- **Elements as channels (sample based)** — The block treats each element of the input as a separate channel. For a three-dimensional input signal of size  $M$ -by- $N$ -by- $P$ , the block outputs an  $M$ -by- $N$ -by- $P$  array. Each  $y_{ijk}$  element of the output contains the maximum value observed in element  $u_{ijk}$  for all inputs since the last reset.

When a reset event occurs, the running maximum  $y_{ijk}$  in the current frame is reset to the element  $u_{ijk}$ .

### Dependencies

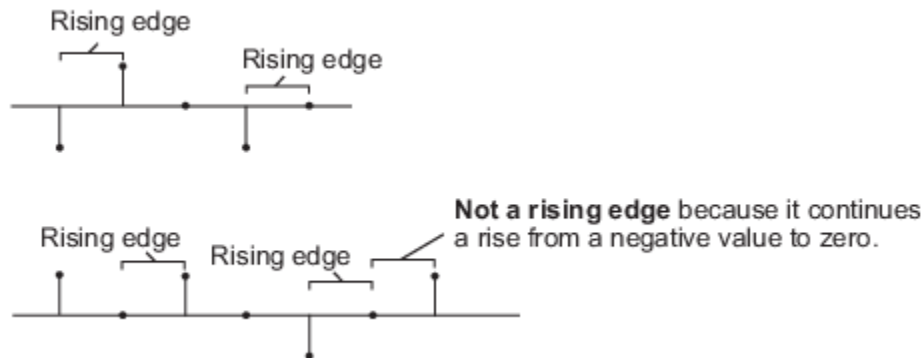
To enable this parameter, set **Mode** to Running.

**Reset port** — Reset event

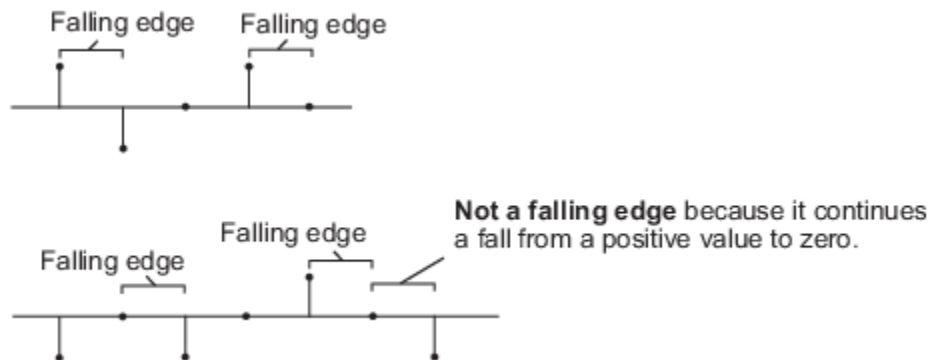
None (default) | Rising edge | Falling edge | Either edge | Non-zero sample

The block resets the running maximum whenever a reset event is detected at the optional **Rst** port. The reset sample time must be a positive integer, which is a multiple of the input sample time.

- **None** — Disables the **Rst** port.
- **Rising edge** — Triggers a reset operation when the **Rst** input does one of the following:
  - Rises from a negative value to a positive value or zero.
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero.



- **Falling edge** — Triggers a reset operation when the **Rst** input does one of the following:
  - Falls from a positive value to a negative value or zero.
  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero.



- **Either edge** — Triggers a reset operation when the **Rst** input is a Rising edge or Falling edge.
- **Non-zero sample** — Triggers a reset operation at each sample time that the **Rst** input is not zero.

---

**Note** When running simulations in the Simulink `MultiTasking` mode, reset signals have a one-sample latency. Therefore, when the block detects a reset event, there is a one-sample delay at the reset port rate before the block applies the reset. For more information on latency and the Simulink tasking modes, see “Excess Algorithmic Delay (Tasking Latency)” and “Time-Based Scheduling and Code Generation” (Simulink Coder).

---

### Dependencies

To enable this parameter, set **Mode** to `Running`.

### Data Types Tab

---

**Note** To use these parameters, the data input must be complex and fixed-point.

---

**Rounding mode** — Method of rounding operation

Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Specify the rounding mode for fixed-point operations as one of the following:

- Floor
- Ceiling
- Convergent
- Nearest
- Round
- Simplest
- Zero

For more details, see rounding mode.


**Saturate on integer overflow** — Method of overflow action  
off (default) | on

When you select this parameter, the block saturates the result of its fixed-point operation. When you clear this parameter, the block wraps the result of its fixed-point operation. For details on saturate and wrap, see overflow mode for fixed-point operations.

**Product output** — Product output data type  
Inherit: Same as input (default) | fixdt([],16,0)

**Product output** specifies the data type of the output of a product operation in the Maximum block. For more information on the product output data type, see “Multiplication Data Types”.

- **Inherit: Same as input** — The block specifies the product output data type to be the same as the input data type.
- **fixdt([],16,0)** — The block specifies an autosigned, binary-point, scaled, fixed-point data type with a word length of 16 bits and a fraction length of 0.

Alternatively, you can set the **Product output** data type by using the **Data Type Assistant**. To use the assistant, click the **Show data type assistant** button .

For more information on the data type assistant, see “Specify Data Types Using Data Type Assistant” (Simulink) in *Simulink User's Guide* (Simulink).


**Accumulator** — Accumulator data type  
Inherit: Same as product output (default) | Inherit: Same as input | fixdt([],16,0)

**Accumulator** specifies the data type of the output of an accumulation operation in the Maximum block.

- **Inherit: Same as product output** — The block specifies the accumulator data type to be the same as the product output data type.
- **Inherit: Same as input** — The block specifies the accumulator data type to be the same as the input data type.



- `fixdt([], 16, 0)` — The block specifies an autosigned, binary-point, scaled, fixed-point data type with a word length of 16 bits and a fraction length of 0.

Alternatively, you can set the **Accumulator** data type by using the **Data Type Assistant**. To use the assistant, click the **Show data type assistant** button .

For more information on the data type assistant, see “Specify Data Types Using Data Type Assistant” (Simulink) in *Simulink User's Guide* (Simulink).

**Lock data type settings against changes by the fixed-point tools** — Prevent fixed-point tools from overriding data types  
off (default) | on

Select this parameter to prevent the fixed-point tools from overriding the data types you specify in the block dialog box.

## Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | double   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no                                      |
| <b>Multidimensional Signals</b> | no                                      |
| <b>Variable-Size Signals</b>    | yes                                     |
| <b>Zero-Crossing Detection</b>  | no                                      |

## Algorithms

### Maximum

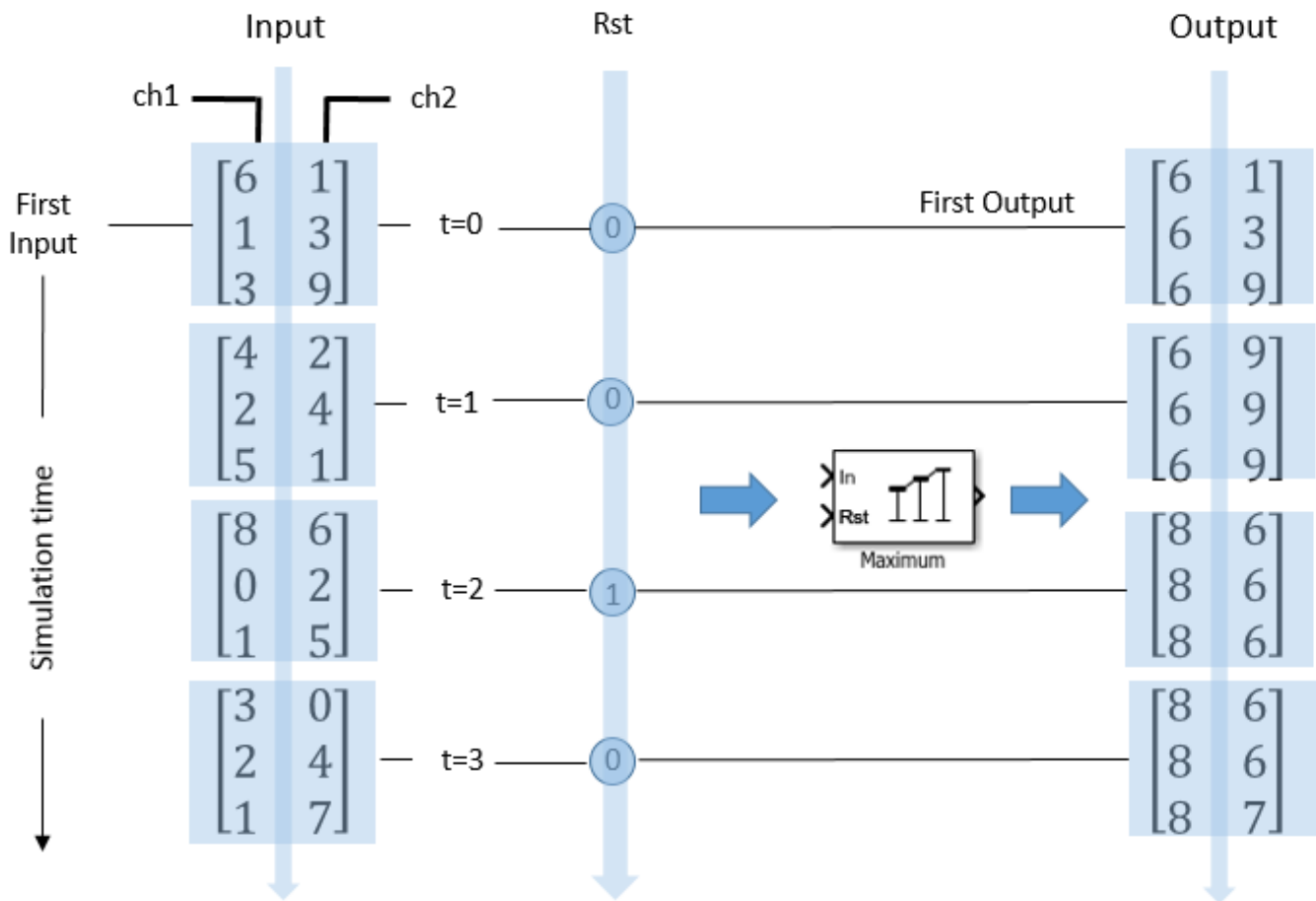
When you set **Mode** to one of **Value**, **Index**, or **Value and Index**, and specify a dimension, the block produces results identical to the MATLAB `max` function, when it is called as `[y, I] = max(u, [], D)`.

- `u` is the data input.
- `D` is the dimension.
- `y` is the maximum value.
- `I` is the index of the maximum value.

The maximum value along the entire input is identical to calling the `max` function as `[y, I] = max(u(:))`.

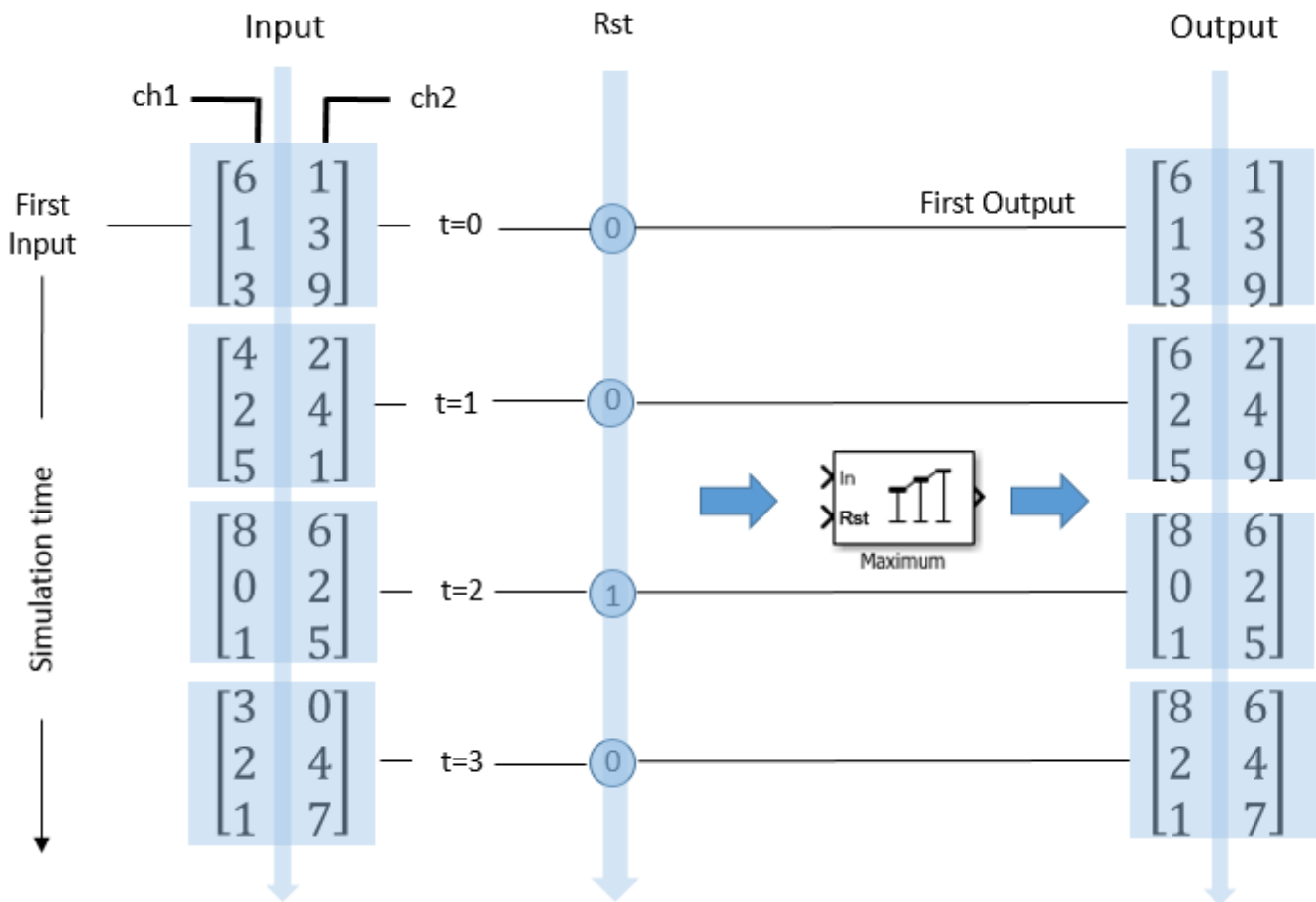
### Running Maximum

When you set **Mode** to **Running**, and **Input processing** to **Columns as channels (frame based)**, the block treats each column of the input as a separate channel. In this example, the block processes a two-channel signal with a frame size of three under these settings.



The block outputs the maximum value over each channel since the last reset. At  $t = 2$ , the reset event occurs. The maximum value in the second column changes to 6, even though 6 is less than 9, which was the maximum value since the previous reset event.

When you set **Mode** to Running, and **Input processing** to Elements as channels (sample based), the block treats each element of the input as a separate channel. In this example, the block processes a two-channel signal with a frame size of three under these settings.



Each  $y_{ij}$  element of the output contains the maximum value observed in element  $u_{ij}$  for all inputs since the last reset. The reset event occurs at  $t = 2$ . When a reset event occurs, the running maximum,  $y_{ij}$ , in the current frame is reset to element  $u_{ij}$ .

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

### HDL Architecture

This block has multi-cycle implementations that introduce additional latency in the generated code. To see the added latency, view the generated model or validation model. See “Generated Model and Validation Model” (HDL Coder).

| Architecture    | Additional cycles of latency                  | Description  |
|-----------------|---|--|
| default<br>Tree | 0   | Generates a tree structure of comparators.   |
| Cascade         | 1, when block has a single vector input port. | This implementation is optimized for latency * area, with medium speed. See “Cascade Architecture Best Practices” (HDL Coder). |

### HDL Block Properties

|                                  |  |
|----------------------------------|--|
| <b>ConstrainedOutputPipeline</b> | Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder). |
| <b>InputPipeline</b>             | Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).                 |
| <b>OutputPipeline</b>            | Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).               |
| <b>InstantiateStages</b>         | Generate a VHDL® entity or Verilog® module for each cascade stage. The default is off. See also “InstantiateStages” (HDL Coder).   |
| <b>SerialPartition</b>           | Specify partitions for Cascade-serial implementations as a vector of the lengths of each partition. The default setting uses the minimum number of stages. See also “SerialPartition” (HDL Coder).                                       |

### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

The parameters on the **Data Types Tab** of the block are used only for complex fixed-point inputs. The sum of the squares of the real and imaginary parts of such an input are formed before a comparison is made, as described under the 'Mode' parameter in “Main Tab” on page 2-971. The results of the squares of the real and imaginary parts are placed into the product output data type. The result of the sum of the squares is placed into the accumulator data type. These parameters are ignored for other types of inputs.

### See Also

#### Functions

max | cummax

#### Objects

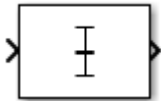
dsp.MovingMinimum | dsp.MovingMaximum

**Blocks**

Minimum | Moving Maximum | Moving Minimum | Mean

# Mean

Find mean value of input or sequence of inputs



**Libraries:**  
DSP System Toolbox / Statistics

## Description

The Mean block computes the mean of each row or column of the input, or along vectors of a specified dimension of the input. It can also compute the mean of the entire input. You can specify the dimension using the **Find the mean value over** parameter. The Mean block can also track the mean value in a sequence of inputs over a period of time. To track the mean value in a sequence of inputs, select the **Running mean** parameter.

---

**Note** The **Running** mode in the Mean block will be removed in a future release. To compute the running mean in Simulink, use the Moving Average block instead.

---

## Ports

### Input

**I** — Data input  
vector | matrix | *N*-D array

The block accepts real-valued or complex-valued multichannel and multidimensional inputs. The input data type must be double precision, single precision, integer, or fixed point with power-of-two slope and zero bias.

This port is unnamed until you select the **Running mean** parameter and set the **Reset port** parameter to any option other than **None**.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fixed point

**Rst** — Reset port  
scalar

Specify the reset event that causes the block to reset the running mean. The sample time of the **Rst** input must be a positive integer multiple of the input sample time.

### Dependencies

To enable this port, select the **Running mean** parameter and set the **Reset port** parameter to any option other than **None**.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean

## Output

**Port\_1** — Mean value along the specified dimension

scalar | vector | matrix |  $N$ -D array

The data type of the output matches the data type of the input.

When you do not select the **Running mean** parameter, the block computes the mean value in each row or column of the input, or along vectors of a specified dimension of the input. It can also compute the mean of the entire input at each individual sample time. Each element in the output array  $y$  is the mean value of the corresponding column, row, or entire input. The output array  $y$  depends on the setting of the **Find the mean value over** parameter. Consider a three-dimensional input signal of size  $M$ -by- $N$ -by- $P$ . When you set **Find the mean value over** to:

- **Entire input** — The output at each sample time is a scalar that contains the mean value of the  $M$ -by- $N$ -by- $P$  input matrix.
- **Each row** — The output at each sample time consists of an  $M$ -by-1-by- $P$  array, where each element contains the mean value of each vector over the second dimension of the input. For an  $M$ -by- $N$  matrix input, the output at each sample time is an  $M$ -by-1 column vector.
- **Each column** — The output at each sample time consists of a 1-by- $N$ -by- $P$  array, where each element contains the mean value of each vector over the first dimension of the input. For an  $M$ -by- $N$  matrix input, the output at each sample time is a 1-by- $N$  row vector.

In this mode, the block treats length- $M$  unoriented vector inputs as  $M$ -by-1 column vectors.

- **Specified dimension** — The output at each sample time depends on the value of the **Dimension** parameter. If you set the **Dimension** to 1, the output is the same as when you select **Each column**. If you set the **Dimension** to 2, the output is the same as when you select **Each row**. If you set the **Dimension** to 3, the output at each sample time is an  $M$ -by- $N$  matrix containing the mean value of each vector over the third dimension of the input.

When you select **Running mean**, the block tracks the mean value of each channel in a time sequence of inputs. In this mode, you must also specify a value for the **Input processing** parameter.

- **Elements as channels (sample based)** — The block treats each element of the input as a separate channel. For a three-dimensional input signal of size  $M$ -by- $N$ -by- $P$ , the block outputs an  $M$ -by- $N$ -by- $P$  array. Each element  $y_{ijk}$  of the output contains the mean value of the element  $u_{ijk}$  for all inputs since the last reset.

When a reset event occurs, the running mean  $y_{ijk}$  in the current frame is reset to the element  $u_{ijk}$ .

- **Columns as channels (frame based)** — The block treats each column of the input as a separate channel. This option does not support an  $N$ -dimensional input signal, where  $N > 2$ . For a two-dimensional input signal of size  $M$ -by- $N$ , the block outputs an  $M$ -by- $N$  matrix. Each element  $y_{ij}$  of the output contains the mean of the values in the  $j$ th column of all inputs since the last reset, up to and including the element  $u_{ij}$  of the current input.

When a reset event occurs, the running mean for each channel becomes the mean value of all the samples in the current input frame, up to and including the current input sample.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fixed point

## Parameters

### Main Tab

**Running mean** — Option to select running mean  
off (default) | on

When you select the **Running mean** parameter, the block tracks the mean value of each channel in a time sequence of inputs.

**Find the mean value over** — Dimension over which the block computes the mean value  
Each column (default) | Entire input | Each row | Specified dimension

- **Each column** — The block outputs the mean value over each column.
- **Each row** — The block outputs the mean value over each row.
- **Entire input** — The block outputs the mean value over the entire input.
- **Specified dimension** — The block outputs the mean value over the dimension, specified in the **Dimension** parameter.

### Dependencies

To enable this parameter, clear the **Running mean** parameter.

**Dimension** — Custom dimension  
1 (default) | scalar

Specify the dimension (one-based value) of the input signal over which the mean is computed. The value of this parameter must be greater than 0 and less than the number of dimensions in the input signal.

### Dependencies

To enable this parameter, set **Find the mean value over** to **Specified dimension**.

**Input processing** — Method to process the input in running mode  
Columns as channels (frame based) (default) | Elements as channels (sample based)

- **Columns as channels (frame based)** — The block treats each column of the input as a separate channel. This option does not support an  $N$ -dimensional input signal, where  $N > 2$ . For a two-dimensional input signal of size  $M$ -by- $N$ , the block outputs an  $M$ -by- $N$  matrix. Each element  $y_{ij}$  of the output contains the mean of the values in the  $j$ th column of all inputs since the last reset, up to and including the element  $u_{ij}$  of the current input.

When a reset event occurs, the running mean for each channel becomes the mean value of all the samples in the current input frame, up to and including the current input sample.

- **Elements as channels (sample based)** — The block treats each element of the input as a separate channel. For a three-dimensional input signal of size  $M$ -by- $N$ -by- $P$ , the block outputs an  $M$ -by- $N$ -by- $P$  array. Each element  $y_{ijk}$  of the output contains the mean value of the element  $u_{ijk}$  for all inputs since the last reset.



When a reset event occurs, the running mean  $y_{ijk}$  in the current frame is reset to the element  $u_{ijk}$ .

**Variable-Size Inputs**

When your inputs are of variable size, and you select the **Running mean** parameter, then:

- If you set the **Input processing** parameter to **Elements as channels (sample based)**, the state is reset.
- If you set the **Input processing** parameter to **Columns as channels (frame based)**, then:
  - When the input size difference is in the number of channels (number of columns), the state is reset.
  - When the input size difference is in the length of channels (number of rows), there is no reset and the running operation is carried out as usual.

**Dependencies**

To enable this parameter, select the **Running mean** parameter.

**Reset port** — Reset event

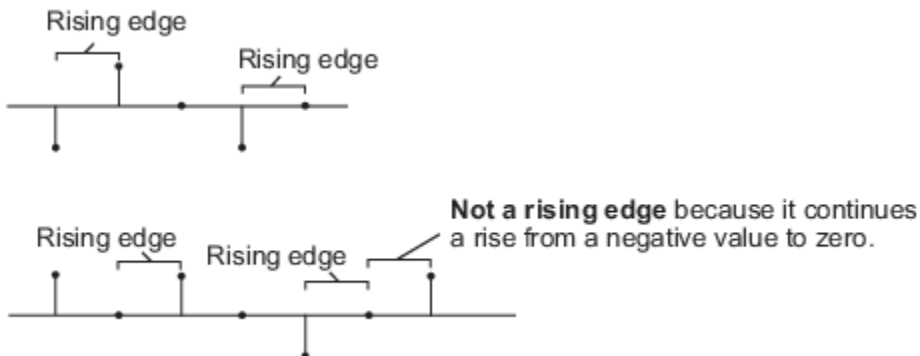
None (default) | Rising edge | Falling edge | Either edge | Non-zero sample

The block resets the running mean whenever a reset event is detected at the optional **Rst** port. The reset sample time must be a positive integer multiple of the input sample time.

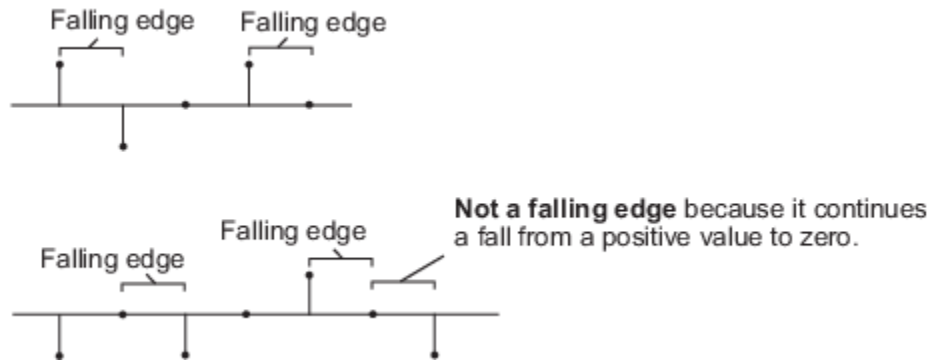
When a reset event occurs while the **Input processing** parameter is set to **Elements as channels (sample based)**, the running mean for each channel is initialized to the value in the corresponding channel of the current input. Similarly, when the **Input processing** parameter is set to **Columns as channels (frame based)**, the running mean for each channel becomes the mean value of all the samples in the current input frame, up to and including the current input sample.

Use this parameter to specify the reset event.

- **None** — Disables the **Rst** port.
- **Rising edge** — Triggers a reset operation when the **Rst** input does one of the following:
  - Rises from a negative value to either a positive value or zero.
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero.



- **Falling edge** — Triggers a reset operation when the **Rst** input does one of the following:
  - Falls from a positive value to a negative value or zero.
  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero.



- **Either edge** — Triggers a reset operation when the **Rst** input is a Rising edge or Falling edge.
- **Non-zero sample** — Triggers a reset operation at each sample time, when the **Rst** input is not zero.

---

**Note** When running simulations in the Simulink multitasking mode, reset signals have a one-sample latency. Therefore, when the block detects a reset event, there is a one-sample delay at the reset port rate before the block applies the reset. For more information on latency and the Simulink tasking modes, see “Excess Algorithmic Delay (Tasking Latency)” and “Time-Based Scheduling and Code Generation” (Simulink Coder).

---

### Dependencies

To enable this parameter, select the **Running mean** parameter.

### Data Types Tab

---

**Note** To use these parameters, the data input must be fixed-point. For all other inputs, the parameters on the **Data Types** tab are ignored.

---

**Rounding mode** — Method of rounding operation

Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Specify the rounding mode for fixed-point operations as one of the following:

- Floor
- Ceiling
- Convergent
- Nearest

- Round
- Simplest
- Zero

For more details, see rounding mode.


**Saturate on integer overflow** — Method of overflow action  
off (default) | on

When you select this parameter, the block saturates the result of its fixed-point operation. When you clear this parameter, the block wraps the result of its fixed-point operation. For details on saturate and wrap, see overflow mode for fixed-point operations.

**Accumulator** — Accumulator data type  
Inherit: Same as input (default) | fixdt([],16,0)

**Accumulator** specifies the data type of the output of an accumulation operation in the Mean block. See “Fixed Point” on page 2-990 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- **Inherit: Same as input** — The block specifies the accumulator data type to be the same as the input data type.
- **fixdt([],16,0)** — The block specifies an autosigned, binary-point, scaled, fixed-point data type with a word length of 16 bits and a fraction length of 0.


Alternatively, you can set the **Accumulator** data type by using the **Data Type Assistant**. To use the assistant, click the **Show data type assistant** button .

For more information on the data type assistant, see “Specify Data Types Using Data Type Assistant” (Simulink) in *Simulink User's Guide* (Simulink).

**Output** — Output data type  
Inherit: Same as accumulator (default) | fixdt([],16,0)

**Output** specifies the data type of the output of the Mean block. See “Fixed Point” on page 2-990 for illustrations depicting the use of the output data type in this block. You can set it to:

- **Inherit: Same as accumulator** — The block specifies the output data type to be the same as the accumulator data type.
- **fixdt([],16,0)** — The block specifies an autosigned, binary-point, scaled, fixed-point data type with a word length of 16 bits and a fraction length of 0.

Alternatively, you can set the **Output** data type by using the **Data Type Assistant**. To use the assistant, click the **Show data type assistant** button .

For more information on the data type assistant, see “Specify Data Types Using Data Type Assistant” (Simulink) in *Simulink User's Guide* (Simulink).

**Output Minimum** — Minimum value the block can output  
[] (default) | scalar

Specify the minimum value that the block can output. The default value is [] (unspecified). Simulink uses this value to perform:

- Simulation range checking. See “Specify Signal Ranges” (Simulink).
- Automatic scaling of fixed-point data types.

**Output Maximum** — Maximum value the block can output  
[] (default) | scalar

Specify the maximum value that the block can output. The default value is [] (unspecified). Simulink uses this value to perform:

- Simulation range checking. See “Specify Signal Ranges” (Simulink).
- Automatic scaling of fixed-point data types.

**Lock data type settings against changes by the fixed-point tools** — Prevent fixed-point tools from overriding data types  
off (default) | on

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block.

## Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | double   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no                                      |
| <b>Multidimensional Signals</b> | no                                      |
| <b>Variable-Size Signals</b>    | yes                                     |
| <b>Zero-Crossing Detection</b>  | no                                      |

## Algorithms

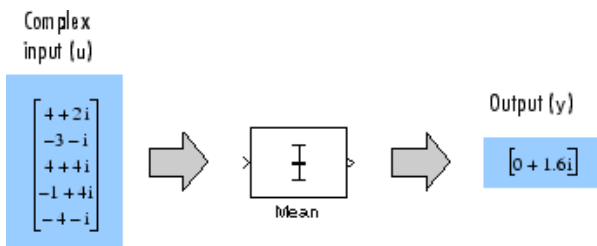
### Mean

When you clear the **Running mean** parameter and specify a dimension, the block produces results identical to the MATLAB mean function, when it is called as  $y = \text{mean}(u, D)$ .

- $u$  is the data input.
- $D$  is the dimension.
- $y$  is the mean value.

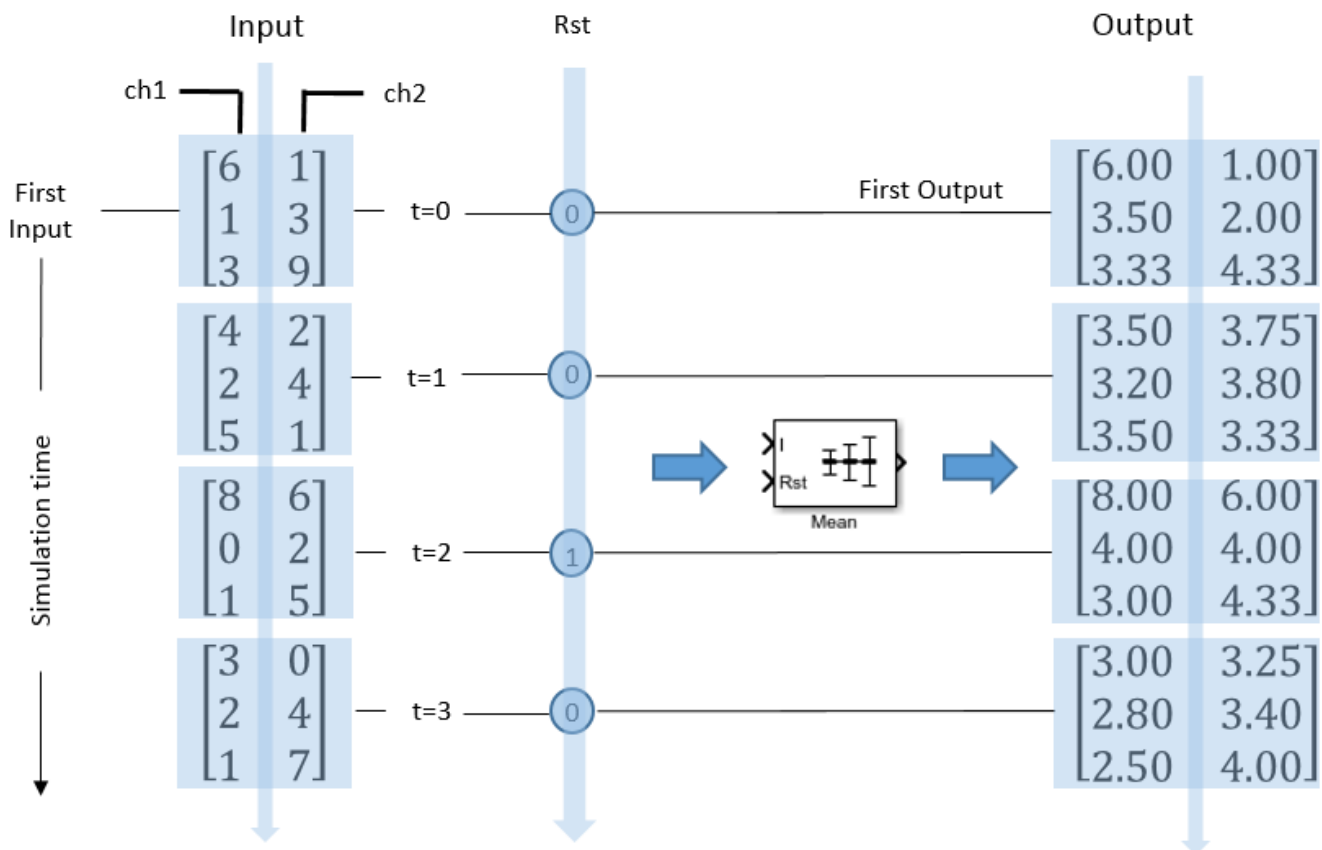
The mean value along the entire input is identical to calling the mean function as  $y = \text{mean}(u(:))$ .

The mean of a complex input is computed independently for the real and imaginary components.



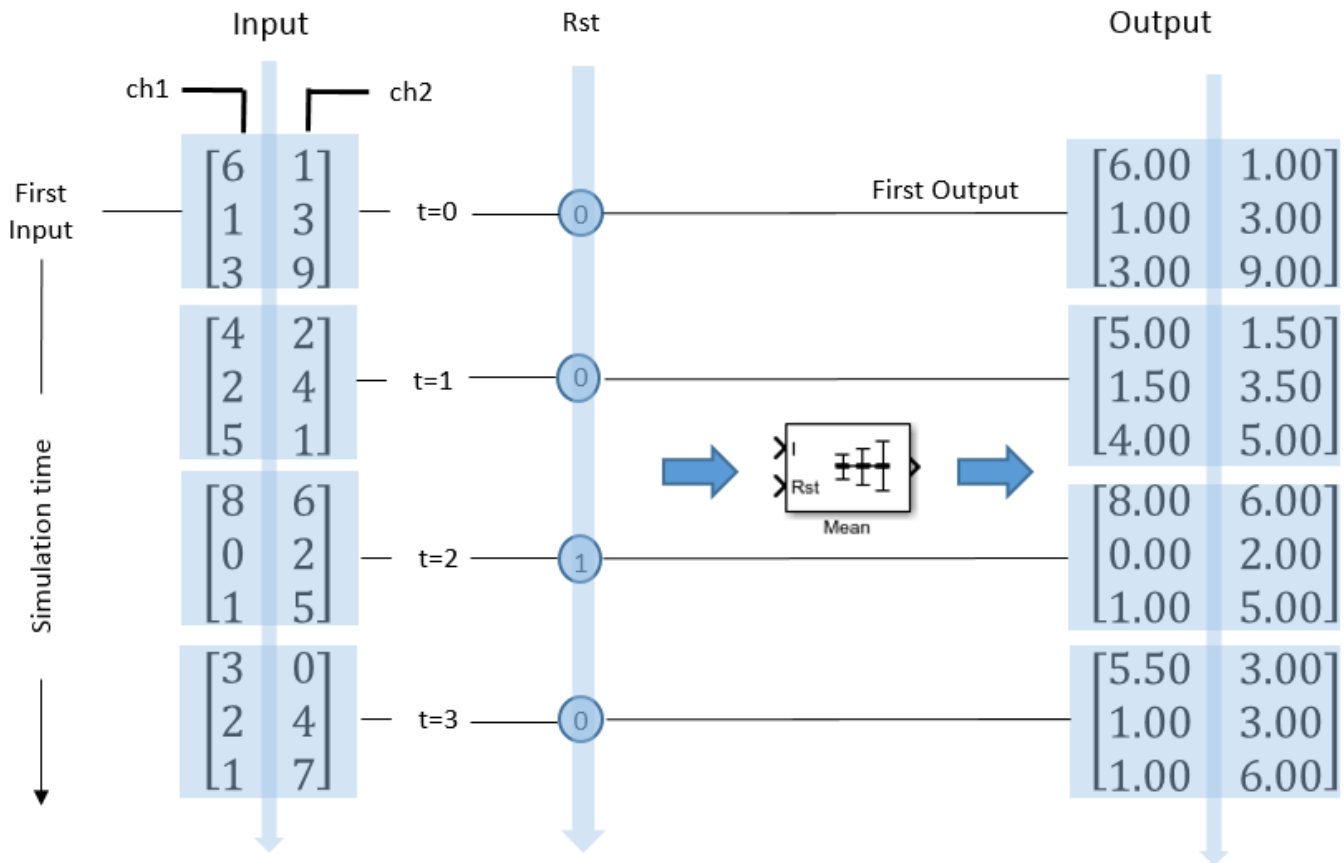
### Running Mean

When you select the **Running mean** parameter, and set the **Input processing** parameter to Columns as channels (frame based), the block treats each column of the input as a separate channel. In this example, the block processes a two-channel signal with a frame size of three under these settings.



The block outputs the mean value over each channel since the last reset. At  $t = 2$ , the reset event occurs. The window of data in the second column now contains only 6.

When you select the **Running mean** parameter, and set the **Input processing** parameter to Elements as channels (sample based), the block treats each element of the input as a separate channel. In this example, the block processes a two-channel signal with a frame size of three under these settings.



Each  $y_{ij}$  element of the output contains the mean value observed in element  $u_{ij}$  for all inputs since the last reset. The reset event occurs at  $t = 2$ . When a reset event occurs, the running mean,  $y_{ij}$ , in the current frame is reset to element  $u_{ij}$ .

## Version History

Introduced before R2006a

## Extended Capabilities

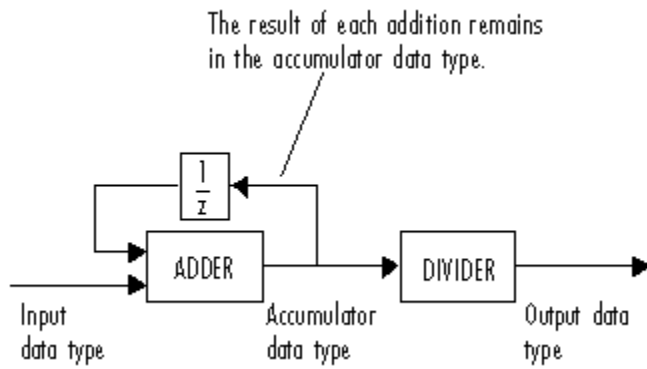
### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

The following diagram shows the data types used within the Mean block for fixed-point signals.



## See Also

### Functions

mean

### Objects

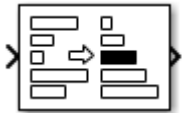
`dsp.MovingAverage` | `dsp.MedianFilter`

### Blocks

Moving Average | Maximum | Median | Median Filter | Minimum

# Median

Median value of input



## Libraries:

DSP System Toolbox / Statistics

## Description

The Median block computes the median of each row or column of the input, or along vectors of a specified dimension of the input. It can also compute the median of the entire input. You can specify the dimension using the **Find the median value over** parameter. While computing the median, the block first sorts the input values. If the number of values is odd, the median is the middle value. If the number of values is even, the median is the average of the two middle values. To sort the data, you can specify the **Sort algorithm** parameter as either `Quick sort` or `Insertion sort`. The block sorts complex inputs according to their magnitude.

## Ports

### Input

**Port\_1** — Data input

vector | matrix |  $N$ -D array

The block accepts real-valued or complex-valued multichannel and multidimensional inputs. The input data type must be double precision, single precision, integer, or fixed point, with power-of-two slope and zero bias.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fixed point`

### Output

**Port\_1** — Median value along specified dimension

vector | matrix |  $N$ -D array

The block computes the median value in each row or column of the input, or along vectors of a specified dimension of the input. It can also compute the median of the entire input. Each element in the output array  $y$  is the median value of the corresponding column, row, or entire input. The output array  $y$  depends on the setting of the **Find the median value over** parameter.

Consider a three-dimensional input signal of size  $M$ -by- $N$ -by- $P$ . When you set **Find the median value over** to:

- **Entire input** — The output at each sample time is a scalar that contains the median value of the  $M$ -by- $N$ -by- $P$  input matrix.
- **Each row** — The output at each sample time consists of an  $M$ -by-1-by- $P$  array, where each element contains the median value of each vector over the second dimension of the input. For an  $M$ -by- $N$  matrix input, the output is an  $M$ -by-1 column vector.



- **Each column** — The output at each sample time consists of a 1-by- $N$ -by- $P$  array, where each element contains the median value of each vector over the first dimension of the input. For an  $M$ -by- $N$  matrix input, the output at each sample time is a 1-by- $N$  row vector.

In this mode, the block treats length- $M$  unoriented vector inputs as  $M$ -by-1 column vectors.

- **Specified dimension** — The output at each sample time depends on the value of the **Dimension** parameter. If you set the **Dimension** to 1, the output is the same as when you select **Each column**. If you set the **Dimension** to 2, the output is the same as when you select **Each row**. If you set the **Dimension** to 3, the output at each sample time is an  $M$ -by- $N$  matrix containing the median value of each vector over the third dimension of the input.

The data type of the output matches the data type of the input.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fixed point`

## Parameters

### Main Tab

**Sort algorithm** — Sort method

`Quick sort` (default) | `Insertion sort`

Specify the sorting algorithm as either `Quick sort` or `Insertion sort`.

**Find the median value over** — Dimension over which the median is computed

`Each column` (default) | `Entire input` | `Each row` | `Specified dimension`

- **Each column** — The block outputs the median value over each column.
- **Each row** — The block outputs the median value over each row.
- **Entire input** — The block outputs the median value over the entire input.
- **Specified dimension** — The block outputs the median value over the dimension specified in the **Dimension** parameter.

**Dimension** — Custom dimension

1 (default) | `scalar`

Specify the dimension (one-based value) of the input signal over which the block computes the median. The value of this parameter must be greater than 0 and less than or equal to the number of dimensions in the input signal.

### Dependencies

To enable this parameter, set **Find the median value over** to `Specified dimension`.

### Data Types Tab

---

**Note** To use these parameters, the data input must be `fixed point`. For all other inputs, the parameters on the **Data Types** tab are ignored.

---

**Rounding mode** — Method of rounding operation

Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Specify the rounding mode for fixed-point operations. For more details, see rounding mode.

**Saturate on integer overflow** — Method of overflow action

off (default) | on

When you select this parameter, the block saturates the result of its fixed-point operation. When you clear this parameter, the block wraps the result of its fixed-point operation. For details on saturate and wrap, see overflow mode for fixed-point operations.


**Product output** — Product output data type

Inherit: Same as input (default) | fixdt([],16,0)

Specify the data type of the output of a product operation in the Median block. For more information, see “Fixed Point” on page 2-997 and “Multiplication Data Types”.

You can set this parameter to:

- **Inherit: Same as input** — The product output data type is the same as the input data type.
- **fixdt([],16,0)** — The product output data type is an autosigned, binary-point, scaled, fixed-point data type with a word length of 16 bits and a fraction length of 0.

Alternatively, you can set the **Product output** data type by using the **Data Type Assistant**. To use the assistant, click the **Show data type assistant** button .

For more information on the data type assistant, see “Specify Data Types Using Data Type Assistant” (Simulink) in *Simulink User's Guide* (Simulink).


**Accumulator** — Accumulator data type

Inherit: Same as product output (default) | Inherit: Same as input | fixdt([],16,0)

Specify the data type of the output of an accumulation operation in the Median block. For more details, see “Fixed Point” on page 2-997.

You can set this parameter to:

- **Inherit: Same as product output** — The accumulator data type is the same as the product output data type.
- **Inherit: Same as input** — The accumulator data type is the same as the input data type.
- **fixdt([],16,0)** — The accumulator data type is an autosigned, binary-point, scaled, fixed-point data type with a word length of 16 bits and a fraction length of 0.

Alternatively, you can set the **Accumulator** data type by using the **Data Type Assistant**. To use the assistant, click the **Show data type assistant** button .

For more information on the data type assistant, see “Specify Data Types Using Data Type Assistant” (Simulink) in *Simulink User's Guide* (Simulink).

**Output** — Output data type

Inherit: Same as accumulator (default) | Inherit: Same as input | Inherit: Same as product output | fixdt([],16,0)

**Output** specifies the data type of the output of the Median block. For more details, see “Fixed Point” on page 2-997.

You can set this parameter to:

- **Inherit: Same as accumulator** — The output data type is the same as the accumulator data type.
- **Inherit: Same as input** — The output data type is the same as the input data type.
- **Inherit: Same as product output** — The output data type is the same as the product output data type.
- **fixdt([],16,0)** — The output data type is an autosigned, binary-point, scaled, fixed-point data type with a word length of 16 bits and a fraction length of 0.

Alternatively, you can set the **Output** data type by using the **Data Type Assistant**. To use the

assistant, click the **Show data type assistant** button .

For more information on the data type assistant, see “Specify Data Types Using Data Type Assistant” (Simulink) in *Simulink User's Guide* (Simulink).

**Output Minimum** — Minimum output value

[] (default) | scalar

Specify the minimum value that the block can output. The default value is [] (unspecified). Simulink uses this value to perform:

- Simulation range checking. See “Specify Signal Ranges” (Simulink).
- Automatic scaling of fixed-point data types.

**Output Maximum** — Maximum output value

[] (default) | scalar

Specify the maximum value that the block can output. The default value is [] (unspecified). Simulink uses this value to perform:

- Simulation range checking. See “Specify Signal Ranges” (Simulink).
- Automatic scaling of fixed-point data types.

**Lock data type settings against changes by the fixed-point tools** — Prevent fixed-point tools from overriding data types

off (default) | on

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block.

## Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | double   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no                                      |
| <b>Multidimensional Signals</b> | no                                      |
| <b>Variable-Size Signals</b>    | yes                                     |
| <b>Zero-Crossing Detection</b>  | no                                      |

## Algorithms

### Median

The median of a set of data is calculated using the following steps:

- 1 The values are sorted using the specified sorting algorithm.
- 2 If the number of values is odd, the median is the middle value.
- 3 If the number of values is even, the median is the average of the two middle values.

The block produces results identical to the MATLAB `median` function when called as  $y = \text{median}(u, D)$ , where

- $u$  is the data input.
- $D$  is the dimension.
- $y$  is the median value.

When the block calculates the median value along the entire input, the result is identical to calling the `median` function as  $y = \text{median}(u(:))$ .

When the input is complex, the block sorts the data according to the magnitude of each value. The magnitude in this case is defined as the sum of the squares of the real and imaginary components of the complex input.

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Generated code relies on the `memcpy` or `memset` function (`string.h`) under certain conditions.

### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

For fixed-point inputs, you can specify the **Accumulator**, **Product output**, and **Output** data types in the block dialog box. Not all of these fixed-point parameters are applicable to all types of fixed-point inputs. The table shows when each kind of data type and scaling is used.

$M$  is the length of the sorted data along the specified dimension. X indicates that the particular data type is applicable.

|  | <b>Output data type</b> | <b>Accumulator data type</b> | <b>Product output data type</b> |
|--|-------------------------|------------------------------|---------------------------------|
| <b>Even <math>M</math></b>             | X                       | X                            | Not Applicable                  |
| <b>Odd <math>M</math></b>              | X                       | Not Applicable               | Not Applicable                  |
| <b>Odd <math>M</math> and complex</b>  | X                       | X                            | X                               |
| <b>Even <math>M</math> and complex</b> | X                       | X                            | X                               |

When  $M$  is even, the **Accumulator** and **Output** data types and scalings are used for fixed-point signals. While calculating the average of the two central rows of the input matrix, the result of the sum is stored in the **Accumulator** data type and scaling. The total result of the average, which is the median of the data, is stored in the **Output** data type and scaling.

When the fixed-point inputs are complex, both the **Accumulator** and the **Product output** data types are used in addition to the **Output** data type. Before sorting the data, the block computes the sum of the squares of the real and imaginary components of the complex input. The results of the squares are stored in the **Product output** data type and scaling. The result of the sum of the squares is stored in the **Accumulator** data type and scaling.

For fixed-point inputs that are both complex and have even  $M$ , the **Accumulator** data type also stores the sum of the two central rows of the input matrix. The average of the two central rows, which is the median of the data, is stored in the **Output** data type.

## See Also

### Functions

median

### Objects

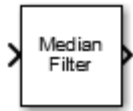
dsp.MedianFilter | dsp.HampelFilter | dsp.MovingAverage

### Blocks

Median Filter | Hampel Filter | Variance | Standard Deviation | Sort | Minimum

# Median Filter

Median filter



## Libraries:

DSP System Toolbox / Filtering / Filter Designs  
 DSP System Toolbox / Statistics

## Description

The Median Filter block computes the moving median of the input signal along each channel independently over time. The block uses the sliding window method to compute the moving median. In this method, a window of specified length moves over each channel sample by sample, and the block computes the median of the data in the window. This block performs median filtering on the input data over time. For more details, see “Algorithms” on page 2-999.

## Input/Output Ports

### Input

**Port\_1** — Data input

column vector | row vector | matrix

Data over which the block computes moving median. The block accepts real-valued or complex-valued multichannel inputs, that is,  $m$ -by- $n$  size inputs, where  $m \geq 1$ , and  $n \geq 1$ . The block also accepts variable-size inputs. During simulation, you can change the size of each input channel. However, the number of channels cannot change.

Data Types: single | double

### Output

**Port\_1** — Moving median output

column vector | row vector | matrix

The size of the moving median output matches the size of the input. The block uses the sliding window method to compute the moving median. For more details, see “Algorithms” on page 2-999.

Data Types: single | double

## Parameters

**Window length** — Length of the sliding window

5 (default) | positive scalar integer

**Window length** specifies the length of the sliding window in samples.

**Simulate using** — Type of simulation to run

Code generation (default) | Interpreted execution

Specify the type of simulation to run as one of the following:

- **Code generation** -- Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than **Interpreted execution**.
- **Interpreted execution** -- Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than **Code generation**.

## Block Characteristics

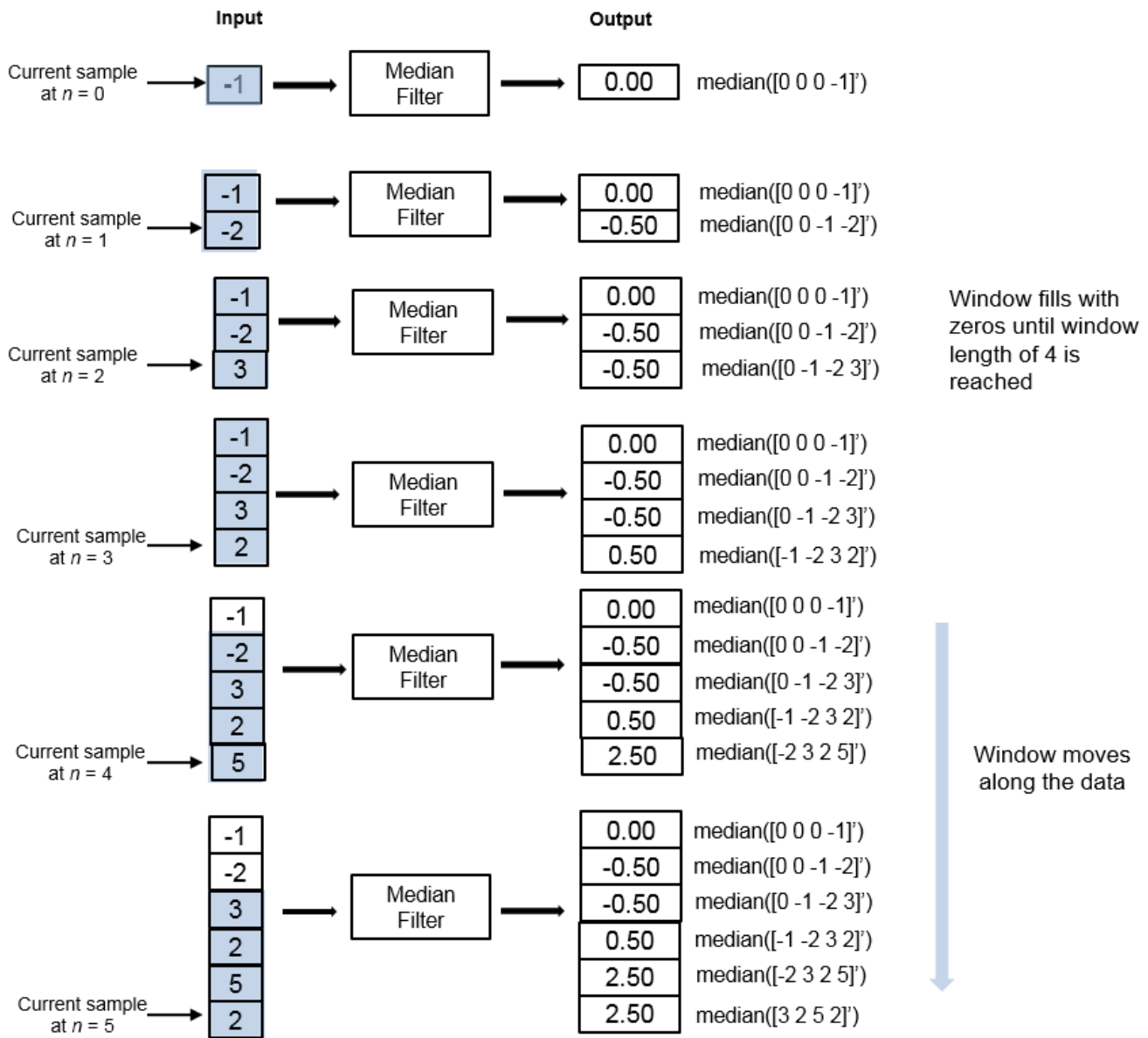
|                                 |                 |
|---------------------------------|-----------------|
| <b>Data Types</b>               | double   single |
| <b>Multidimensional Signals</b> | No              |
| <b>Variable-Size Signals</b>    | Yes             |

## Algorithms

### Sliding Window Method

In the sliding window method, the output for each input sample is the median of the current sample and the  $Len - 1$  previous samples.  $Len$  is the length of the window in samples. To compute the first  $Len - 1$  outputs, when the window does not have enough data yet, the algorithm fills the window with zeros. As an example, to compute the median value when the second input sample comes in, the algorithm fills the window with  $Len - 2$  zeros. The data vector,  $x$ , is then the two data samples followed by  $Len - 2$  zeros. This object performs median filtering on the input data over time.

Consider an example of computing the moving median of a streaming input data using the sliding window method. The algorithm uses a window length of 4. With each input sample that comes in, the window of length 4 moves along the data.



## Version History

Introduced in R2016b

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.



## See Also

### Blocks

[Median](#) | [Moving RMS](#) | [Moving Maximum](#) | [Moving Minimum](#) | [Moving Standard Deviation](#) | [Moving Variance](#) | [Moving Average](#)

### Objects

[dsp.MedianFilter](#) | [dsp.MovingRMS](#) | [dsp.MovingMaximum](#) | [dsp.MovingMinimum](#) | [dsp.MovingStandardDeviation](#) | [dsp.MovingVariance](#) | [dsp.MovingAverage](#)

### Topics

[“What Are Moving Statistics?”](#)

[“Streaming Signal Statistics”](#)

[“Remove High-Frequency Noise from Gyroscope Data”](#)

## MIDI Controls

Output values from controls on MIDI control surface



---

**Note** In a future release, the MIDI Controls block will require Audio Toolbox.

---

### Library

Sources

dspsrcs4

### Description

The MIDI Controls block outputs values from controls on a MIDI control surface in real time.

Use the **MIDI device** parameter to specify the name of the MIDI control surface device from which to receive control values. You can choose:

- Default
- Specify other

If you choose **Default**, the block looks for a MATLAB preference with a group named `midi` and preference named `DefaultDevice`. You can set this preference using the MATLAB `setpref` function. For example, if the desired device is named `BCF2000`, you can type the following command at the MATLAB command line:

```
>> setpref('midi', 'DefaultDevice', 'BCF2000');
```

If the block does not find this preference, it then attempts to choose a device using an algorithm that is unspecified and platform dependent.

If you choose **Specify other**, then a **MIDI device name** edit box appears for you to enter a MATLAB expression for the device name. Enter any MATLAB expression that can evaluate to a string. Literal names must be enclosed in quotes, (for example, `'BCF2000'`).

You can determine the name of your MIDI device using `midiid`, discussed in “Identifying MIDI Device Names and Control Numbers” on page 2-1003.

Use the **MIDI controls** parameter to specify the controls on the MIDI device to which the block should respond. This parameter also determines the size of the block output port. You can choose:

- Respond to any control

- Respond to specified controls

If you choose **Respond to any control**, then the block output will be a scalar. This scalar outputs the value from any and all controls that are manipulated on the MIDI device. Use this option in simple cases when you need only a single control value and the control to which it responds is unimportant.

If you choose **Respond to specified controls**, then a **MIDI control numbers** edit box opens. In this box, enter a MATLAB expression for the device control numbers. Enter any MATLAB expression that can evaluate to a row vector of real double-precision values. The block outputs a 1-D vector with one element corresponding to the output of each specified control.

Use the **Initial values** parameter to specify the value of the block output when simulation starts. The MIDI protocol transmits control values only when a control changes. This protocol provides no means for the block to query the current value of a control. Thus, the block must have some initial value to output until it receives a control change from the device.

Use the **Send initial values to device at start** check box to synchronize the device controls with the block outputs when simulation starts. Some MIDI control surfaces are bidirectional, meaning that they not only send control values but can also receive them. For example, some devices have motorized controls that move to the appropriate position when they receive a control value. If you have such a bidirectional device, select this check box. The block attempts to send the initial values to the device when the simulation starts. No diagnostic message appears if the attempt fails.

The generated code for this block relies on prebuilt .dll files. You can run this code outside the MATLAB environment, or redeploy. However you must account for these extra .dll files when doing so. The packNGo function creates a single .zip file containing all of the pieces required to run or rebuild this code. See packNGo for more information.

## Output Port

The MIDI Controls block output is a vector whose width is determined by the **MIDI controls** and **MIDI control numbers** parameters previously described. The output data type can be either real double-precision floating point, or uint8 integer if the output mode is 'Raw MIDI'. The output values range from 0.0 to 1.0, inclusively, and in the raw mode, they range from 0 to 127, inclusively. The output port back inherits its sample time.

## Identifying MIDI Device Names and Control Numbers

To specify a particular control on a particular MIDI device, you must know the name assigned to the device by the operating system. In addition, a number is always associated with the control. You can interactively discover this information using the MATLAB function, `midiid`. To use `midiid`, note that you will need to have the Audio Toolbox installed. Follow these steps to identify device names and control numbers:

- 1 Verify that MIDI control surface device is correctly connected to the host computer running MATLAB.

---

**Note** For the most consistent behavior, MathWorks recommends that you connect your MIDI control surface device to your computer before starting MATLAB. In some circumstances MATLAB may not be able to find your device if you connect it after starting your MATLAB session. Also, it may not find your device if you disconnect it and reconnect it during your MATLAB session.

---

- 2 Type the following command at the MATLAB command line.

```
>> [ctlnum devname] = midiid
```

You are prompted to move the control in which you are interested.

```
>> [ctlnum devname] = midiid
Move the control you wish to identify; type ^C to abort.
Waiting for control message ...
```

- 3 Move the control. `midiid` detects the movement and returns the device name and control number.

```
>> [ctlnum devname] = midiid
Move the control you wish to identify; type ^C to abort.
Waiting for control message ... done
ctlnum =
    1081
devname =
    BCF2000
>>
```

- 4 Use the device name in the block dialog, or set it as the default device using `setpref`. Then, enter the control number in the block dialog. Concatenate the number with other control numbers as needed.

## Parameters

### MIDI device

Specify whether to use a default MIDI device, or specify a particular device by name.

### MIDI device name

Specify the name of a particular MIDI control surface device from which to receive control values.

### MIDI controls

Specify whether to respond to any control on the MIDI device or respond to particular specified controls.

### MIDI control numbers

Specify particular controls to which the block should respond.

### Initial values

Specify initial values to output when simulation starts.

### Send initial values to device at start

Select this check box to attempt to synchronize a bidirectional MIDI device with block initial values when simulation starts.

### Output mode

Specify the mode in which the control values are generated. When you set **Output mode** to **Normalized (0-1)**, the block generates control values in the range [0 1]. In this mode, control values are represented as a fraction of a full-scale. Hence, you can easily scale this range to your particular application. When you set **Output mode** to **RAW MIDI (0-127)**, the block generates byte-oriented MIDI control values in the range [0 127]. By default, this parameter is set to **Normalized (0-1)**.

## Supported Data Types

| Port   | Supported Data Types   |
|--------|--|
| Output | <ul style="list-style-type: none"><li>• Double-precision floating point, uint8 integer</li></ul> |

## Version History

Introduced in R2012a

## Extended Capabilities

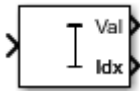
### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Host computer only. Excludes Simulink Desktop Real-Time code generation.

# Minimum

Minimum values of input or sequence of inputs



## Libraries:

DSP System Toolbox / Statistics

DSP System Toolbox HDL Support / Statistics

## Description

The Minimum block identifies the value and position of the smallest element in each row or column of the input, or along vectors of a specified dimension of the input. It can also compute the minimum value of the entire input. The Minimum block can also track the minimum values in a sequence of inputs over a period of time. The **Mode** parameter specifies the block's mode of operation and can be set to one of the following:

- **Value** — The block outputs the minimum values in the specified dimension.
- **Index** — The block outputs the index array of the minimum values in the specified dimension.
- **Value and Index** — The block outputs the minimum values and the corresponding index array in the specified dimension.
- **Running** — The block tracks the minimum values in a sequence of inputs over a period of time.

You can specify the dimension using the **Find the minimum value over** parameter.

---

**Note** The **Running** mode in the Minimum block will be removed in a future release. To compute the running minimum in Simulink, use the Moving Minimum block instead.

---

## Ports

### Input

**In** — Data input

vector | matrix | *N*-D array

The block accepts real-valued or complex-valued multichannel and multidimensional inputs. The input can be floating-point, fixed-point, or Boolean. Real fixed-point inputs can be either signed or unsigned. Complex fixed-point inputs must be signed.

This port is unnamed until you set the **Mode** parameter to **Running** and the **Reset port** parameter to any option other than **None**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point`

**Rst** — Reset port

scalar

Specify the reset event that causes the block to reset the running minimum. The sample time of the **Rst** input must be a positive integer multiple of the input sample time.

## Dependencies

To enable this port, set the **Mode** parameter to **Running** and the **Reset port** parameter to any option other than **None**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean`

## Output

**Val** — Minimum values along the specified dimension

`scalar` | `vector` | `matrix` | *N-D array*

The data type of the minimum value matches the data type of the input.

When the **Mode** parameter is set to either **Value and Index** or **Value**, the following applies:

- The size of the dimension for which the block computes the minimum value is 1. The sizes of all other dimensions match those of the input array. For example, when the input is an *M*-by-*N*-by-*P* array, with the dimension set to 1, the block outputs a 1-by-*N*-by-*P* array. When the dimension is set to 3, the block outputs a two-dimensional *M*-by-*N* matrix.
- When the input is an *M*-by-*N* matrix, with the dimension set to 1, the block outputs a 1-by-*N* matrix.

If you specify the block to compute the minimum value over the entire input, the block outputs a scalar.

When the **Mode** parameter is set **Running**, the block tracks the minimum value of each channel in a time sequence of *M*-by-*N* inputs. In this mode, you must also specify the **Input processing** parameter as one of the following:

- **Elements as channels (sample based)** — The block treats each element of the input as a separate channel. For a three-dimensional input signal of size *M*-by-*N*-by-*P*, the block outputs an *M*-by-*N*-by-*P* array. Each  $y_{ijk}$  element of the output contains the minimum value observed in element  $u_{ijk}$  for all inputs since the last reset.

When a reset event occurs, the running minimum  $y_{ijk}$  in the current frame is reset to the element  $u_{ijk}$ .

- **Columns as channels (frame based)** — The block treats each column of the input as a separate channel. This option does not support an *N*-dimensional input signal, where  $N > 2$ . For a two-dimensional input signal of size *M*-by-*N*, the block outputs an *M*-by-*N* matrix. Each element  $y_{ij}$  of the output contains the minimum value observed in the *j*th column of all inputs since the last reset, up to and including element  $u_{ij}$  of the current input.

When a reset event occurs, the running minimum for each channel becomes the minimum value of all the samples in the current input frame, up to and including the current input sample.

The block resets the running minimum whenever a reset event is detected at the optional **Rst** port. The reset sample time must be a positive integer multiple of the input sample time.

## Dependencies

To enable this port, set the **Mode** parameter to either **Value and Index** or **Value**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point`

**Idx** — Index of the minimum values along the specified dimension  
 scalar | vector | matrix | *N*-D array

When the input is double, the index values are also double. Otherwise, the index values are uint32.

### Dependencies

To enable this port, set the **Mode** parameter to either Value and Index or Index.

Data Types: double | uint32

## Parameters

### Main Tab

**Mode** — Mode in which the block operates  
 Value and Index (default) | Value | Index | Running

When the **Mode** parameter is set to:

- **Value**— The block computes the minimum value in each row or column of the input, along vectors of a specified dimension of the input, or of the entire input at each sample time, and outputs the array *y*. Each element in *y* is the minimum value in the corresponding column, row, vector, or entire input. The output *y* depends on the setting of the **Find the minimum value over** parameter. Consider a three dimensional input signal of size *M*-by-*N*-by-*P*. Set **Find the minimum value over** to:
  - **Each row** — The output *y* at each sample time consists of an *M*-by-1-by-*P* array, where each element contains the minimum value of each vector over the second dimension of the input. For an input that is an *M*-by-*N* matrix, the output at each sample time is an *M*-by-1 column vector.
  - **Each column** — The output *y* at each sample time consists of a 1-by-*N*-by-*P* array, where each element contains the minimum value of each vector over the first dimension of the input. For an input that is an *M*-by-*N* matrix, the output at each sample time is a 1-by-*N* row vector.

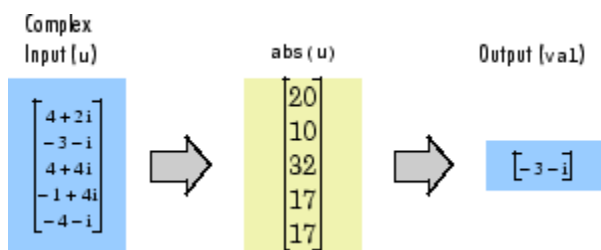
In this mode, the block treats length-*M* unoriented vector inputs as *M*-by-1 column vectors.

- **Entire input** — The output *y* at each sample time is a scalar that contains the minimum value in the *M*-by-*N*-by-*P* input matrix.
- **Specified dimension** — The output *y* at each sample time depends on **Dimension**. If **Dimension** is set to 1, the output is the same as when you select **Each column**. If **Dimension** is set to 2, the output is the same as when you select **Each row**. If **Dimension** is set to 3, the output at each sample time is an *M*-by-*N* matrix containing the minimum value of each vector over the third dimension of the input.

### Complex Inputs

For complex inputs, the block selects the value in each row or column of the input, along vectors of a specified dimension of the input, or of the entire input that has the minimum magnitude squared in the following figure. For complex value  $u = a + bi$ , the magnitude squared is  $a^2 + b^2$ .





- **Index** — The block computes the minimum value in each row or column of the input, along vectors of a specified dimension of the input, or of the entire input, and outputs the index array  $I$ . Each element in  $I$  is an integer indexing the minimum value in the corresponding column, row, vector, or entire input. The output  $I$  depends on the setting of the **Find the minimum value over** parameter. Consider a three dimensional input signal of size  $M$ -by- $N$ -by- $P$ :
  - **Each row** — The output  $I$  at each sample time consists of an  $M$ -by-1-by- $P$  array, where each element contains the index of the minimum value of each vector over the second dimension of the input. For an input that is an  $M$ -by- $N$  matrix, the output at each sample time is an  $M$ -by-1 column vector.
  - **Each column** — The output  $I$  at each sample time consists of a 1-by- $N$ -by- $P$  array, where each element contains the index of the minimum value of each vector over the first dimension of the input. For an input that is an  $M$ -by- $N$  matrix, the output at each sample time is a 1-by- $N$  row vector.

In this mode, the block treats length- $M$  unoriented vector inputs as  $M$ -by-1 column vectors.

- **Entire input** — The output  $I$  at each sample time is a 1-by-3 vector that contains the location of the minimum value in the  $M$ -by- $N$ -by- $P$  input matrix. For an input that is an  $M$ -by- $N$  matrix, the output is a 1-by-2 vector.
- **Specified dimension** — The output  $I$  at each sample time depends on **Dimension**. If **Dimension** is set to 1, the output is the same as when you select **Each column**. If **Dimension** is set to 2, the output is the same as when you select **Each row**. If **Dimension** is set to 3, the output at each sample time is an  $M$ -by- $N$  matrix containing the indices of the minimum values of each vector over the third dimension of the input.

When a minimum value occurs more than once, the computed index corresponds to the first occurrence. For example, when the input is the column vector  $[3 \ 2 \ 1 \ 2 \ 3]'$ , the computed one-based index of the minimum value is 1, rather than 5 when **Each column** is selected.

- **Value and Index** — The block outputs the minimum value in each row or column of the input, along vectors of a specified dimension of the input, or of the entire input, and the corresponding index array  $I$ .
- **Running** — The block tracks the minimum value of each channel in a time sequence of  $M$ -by- $N$  inputs. The block resets the running minimum whenever a reset event is detected at the optional **Rst** port. The reset sample time must be a positive integer multiple of the input sample time. In this mode, you must also specify the **Input processing** parameter as one of the following:
  - **Elements as channels (sample based)** — The block treats each element of the input as a separate channel. For a three-dimensional input signal of size  $M$ -by- $N$ -by- $P$ , the block outputs an  $M$ -by- $N$ -by- $P$  array. Each  $y_{ijk}$  element of the output contains the minimum value observed in element  $u_{ijk}$  for all inputs since the last reset.

When a reset event occurs, the running minimum  $y_{ijk}$  in the current frame is reset to the element  $u_{ijk}$ .

- **Columns as channels (frame based)** — The block treats each column of the input as a separate channel. This option does not support an  $N$ -dimensional input signal, where  $N > 2$ . For a two-dimensional input signal of size  $M$ -by- $N$ , the block outputs an  $M$ -by- $N$  matrix. Each element  $y_{ij}$  of the output contains the minimum value observed in the  $j$ th column of all inputs since the last reset, up to and including element  $u_{ij}$  of the current input.

When a reset event occurs, the running minimum for each channel becomes the minimum value of all the samples in the current input frame, up to and including the current input sample.

The block resets the running minimum whenever a reset event is detected at the optional **Rst** port. The reset sample time must be a positive integer multiple of the input sample time.

### Running Mode for Variable-Size Inputs

When the input is a variable-size signal, and you set the **Mode** to **Running**, then:

- If you set the **Input processing** parameter to **Elements as channels (sample based)**, the state is reset.
- If you set the **Input processing** parameter to **Columns as channels (frame based)**, then:
  - When the input size difference is in the number of channels (columns), the state is reset.
  - When the input size difference is in the length of channels (rows), there is no reset and the running operation is carried out as usual.

**Index base** — Base of the minimum value index  
One (default) | Zero

Specify whether the index of the minimum value is reported using one-based or zero-based numbering.

### Dependencies

To enable this parameter, set **Mode** to either **Index** or **Value** and **Index**.

**Find the minimum value over** — Dimension over which the block computes the minimum value  
Each column (default) | Each row | Entire input | Specified dimension

- **Each column** — The block outputs the minimum value over each column.
- **Each row** — The block outputs the minimum value over each row.
- **Entire input** — The block outputs the minimum value over the entire input.
- **Specified dimension** — The block outputs the minimum value over the dimension specified in the **Dimension** parameter.

### Dependencies

To enable this parameter, set **Mode** to **Value** and **Index**, **Value**, or **Index**.

**Dimension** — Custom dimension  
1 (default) | scalar

Specify the dimension (one-based value) of the input signal over which the block computes the minimum. The value of this parameter must be greater than 0 and less than the number of dimensions in the input signal.

#### Dependencies

To enable this parameter, set **Find the minimum value over** to Specified dimension.

**Input processing** — Method to process the input in running mode

Columns as channels (frame based) (default) | Elements as channels (sample based)

- **Columns as channels (frame based)** — The block treats each column of the input as a separate channel. This option does not support an  $N$ -dimensional input signal, where  $N > 2$ . For a two-dimensional input signal of size  $M$ -by- $N$ , the block outputs an  $M$ -by- $N$  matrix. Each element  $y_{ij}$  of the output contains the minimum value observed in the  $j$ th column of all inputs since the last reset, up to and including element  $u_{ij}$  of the current input.

When a reset event occurs, the running minimum for each channel becomes the minimum value of all the samples in the current input frame, up to and including the current input sample.

- **Elements as channels (sample based)** — The block treats each element of the input as a separate channel. For a three-dimensional input signal of size  $M$ -by- $N$ -by- $P$ , the block outputs an  $M$ -by- $N$ -by- $P$  array. Each  $y_{ijk}$  element of the output contains the minimum value observed in element  $u_{ijk}$  for all inputs since the last reset.

When a reset event occurs, the running minimum  $y_{ijk}$  in the current frame is reset to the element  $u_{ijk}$ .

#### Dependencies

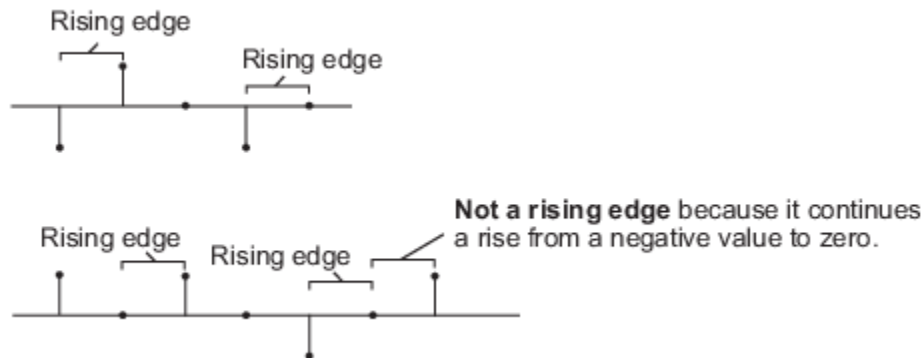
To enable this parameter, set **Mode** to Running.

**Reset port** — Reset event

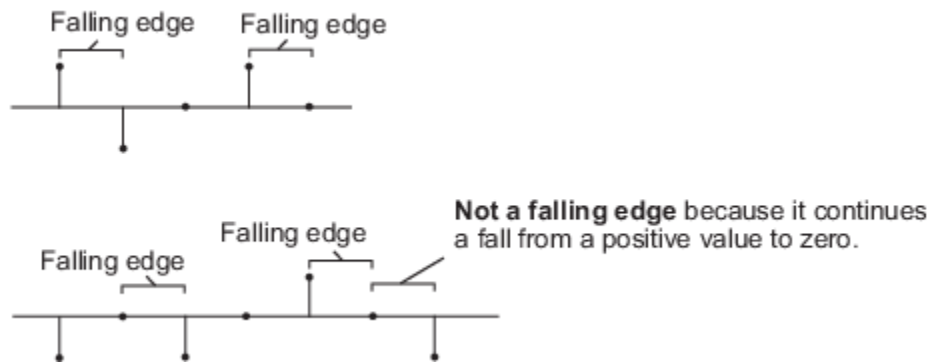
None (default) | Rising edge | Falling edge | Either edge | Non-zero sample

The block resets the running minimum whenever a reset event is detected at the optional **Rst** port. The reset sample time must be a positive integer which is a multiple of the input sample time.

- **None** — Disables the **Rst** port.
- **Rising edge** — Triggers a reset operation when the **Rst** input does one of the following:
  - Rises from a negative value to a positive value or zero.
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero.



- **Falling edge** — Triggers a reset operation when the **Rst** input does one of the following:
  - Falls from a positive value to a negative value or zero.
  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero.



- **Either edge** — Triggers a reset operation when the **Rst** input is a Rising edge or Falling edge.
- **Non-zero sample** — Triggers a reset operation at each sample time that the **Rst** input is not zero.

---

**Note** When running simulations in the Simulink `MultiTasking` mode, reset signals have a one-sample latency. Therefore, when the block detects a reset event, there is a one-sample delay at the reset port rate before the block applies the reset. For more information on latency and the Simulink tasking modes, see “Excess Algorithmic Delay (Tasking Latency)” and “Time-Based Scheduling and Code Generation” (Simulink Coder).

---

### Dependencies

To enable this parameter, set **Mode** to `Running`.

### Data Types Tab

---

**Note** To use these parameters, the data input must be complex fixed-point.

---

**Rounding mode** — Method of rounding operation

Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Specify the rounding mode for fixed-point operations as one of the following:

- Floor
- Ceiling
- Convergent
- Nearest
- Round
- Simplest
- Zero

For more details, see rounding mode.


**Saturate on integer overflow** — Method of overflow action  
off (default) | on

When you select this parameter, the block saturates the result of its fixed-point operation. When you clear this parameter, the block wraps the result of its fixed-point operation. For details on saturate and wrap, see overflow mode for fixed-point operations.

**Product output** — Product output data type  
Inherit: Same as input (default) | fixdt([],16,0)

**Product output** specifies the data type of the output of a product operation in the Minimum block. For more information on the product output data type, see “Multiplication Data Types”.

- **Inherit: Same as input** — The block specifies the product output data type to be the same as the input data type.
- **fixdt([],16,0)** — The block specifies an autosigned, binary-point, scaled, fixed-point data type with a word length of 16 bits and a fraction length of 0.

Alternatively, you can set the **Product output** data type by using the **Data Type Assistant**. To use the assistant, click the **Show data type assistant** button .


For more information on the data type assistant, see “Specify Data Types Using Data Type Assistant” (Simulink) in *Simulink User's Guide* (Simulink).

**Accumulator** — Accumulator data type  
Inherit: Same as product output (default) | Inherit: Same as input | fixdt([],16,0)

**Accumulator** specifies the data type of the output of an accumulation operation in the Minimum block.

- **Inherit: Same as product output** — The block specifies the accumulator data type to be the same as the product output data type.
- **Inherit: Same as input** — The block specifies the accumulator data type to be the same as the input data type.

- `fixdt([], 16, 0)` — The block specifies an autosigned, binary-point, scaled, fixed-point data type with a word length of 16 bits and a fraction length of 0.

Alternatively, you can set the **Accumulator** data type by using the **Data Type Assistant**. To use the assistant, click the **Show data type assistant** button .

For more information on the data type assistant, see “Specify Data Types Using Data Type Assistant” (Simulink) in *Simulink User's Guide* (Simulink).

**Lock data type settings against changes by the fixed-point tools** — Prevent fixed-point tools from overriding data types  
off (default) | on

Select this parameter to prevent the fixed-point tools from overriding the data types you specify in the block dialog box.

## Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | double   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no                                      |
| <b>Multidimensional Signals</b> | no                                      |
| <b>Variable-Size Signals</b>    | yes                                     |
| <b>Zero-Crossing Detection</b>  | no                                      |

## Algorithms

### Minimum

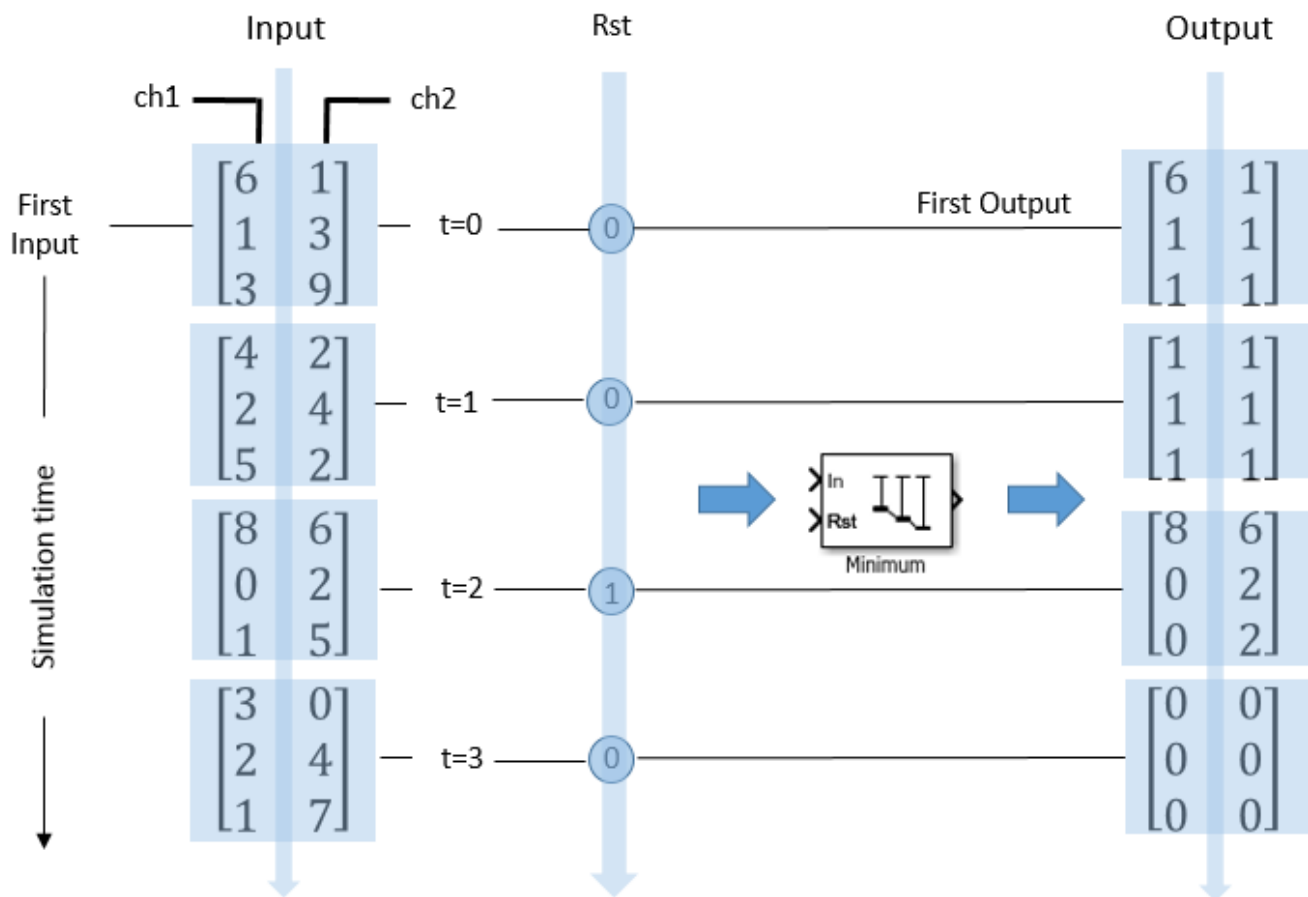
When you set **Mode** to one of Value, Index, or Value and Index, and specify a dimension, the block produces results identical to the MATLAB `min` function, when it is called as `[y, I] = min(u, [], D)`.

- `u` is the data input.
- `D` is the dimension.
- `y` is the minimum value.
- `I` is the index of the minimum value.

The minimum value along the entire input is identical to calling the `min` function as `[y, I] = min(u(:))`.

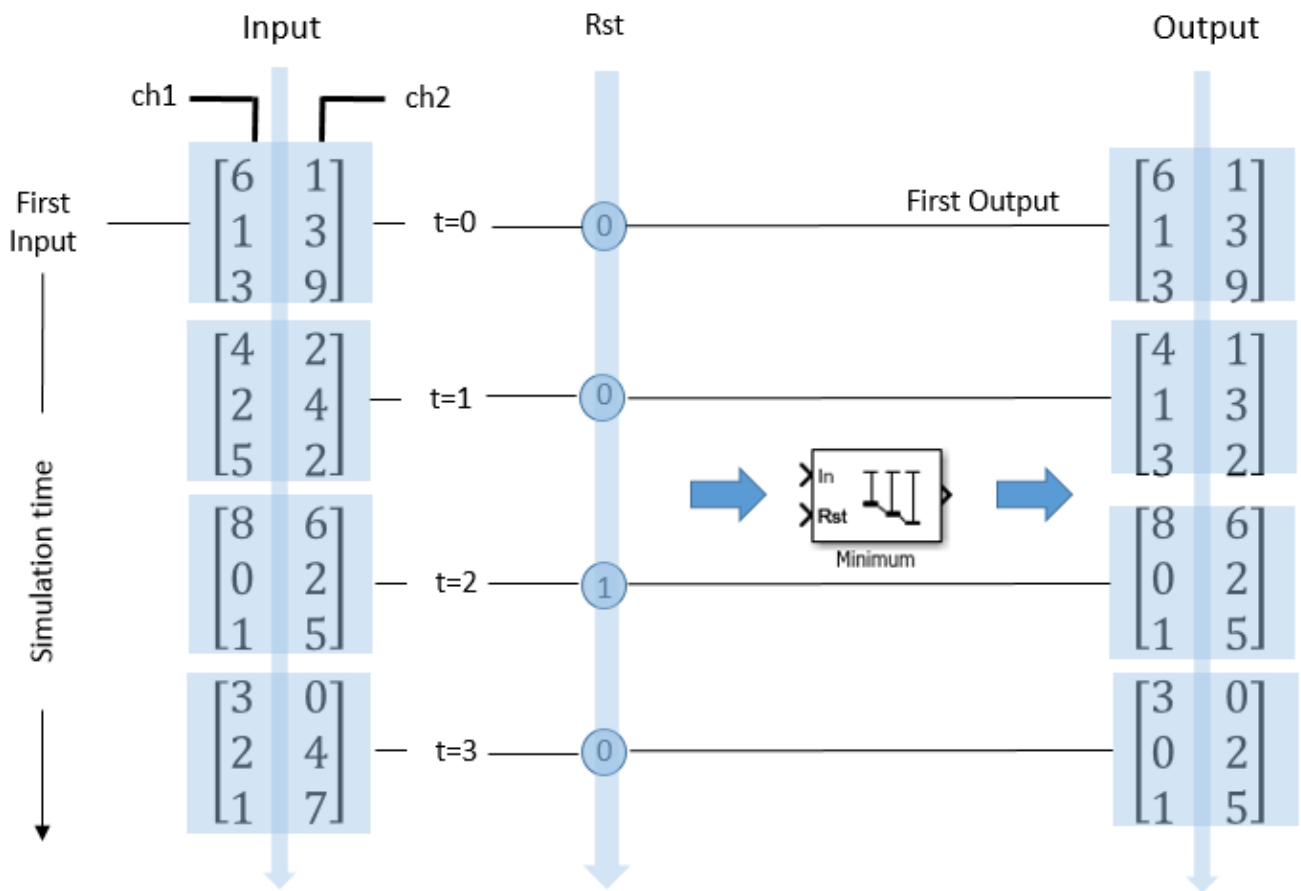
### Running Minimum

When you set **Mode** to Running, and **Input processing** to Columns as channels (frame based), the block treats each column of the input as a separate channel. In this example, the block processes a two-channel signal with a frame size of three under these settings.



The block outputs the minimum value over each channel since the last reset. At  $t = 2$ , the reset event occurs. The minimum value in the second column changes to 6, and then 2, even though these values are greater than 1, which was the minimum value since the previous reset event.

When you set **Mode** to Running, and **Input processing** to Elements as channels (sample based), the block treats each element of the input as a separate channel. In this example, the block processes a two-channel signal with a frame size of three under these settings.



Each element  $y_{ij}$  of the output contains the minimum value observed in element  $u_{ij}$  for all inputs since the last reset. The reset event occurs at  $t = 2$ . When a reset event occurs, the running minimum,  $y_{ij}$ , in the current frame is reset to the element  $u_{ij}$ .

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.



## HDL Architecture

This block has multi-cycle implementations that introduce additional latency in the generated code. To see the added latency, view the generated model or validation model. See “Generated Model and Validation Model” (HDL Coder).

| Architecture    | Additional cycles of latency                  | Description  |
|-----------------|---|--|
| default<br>Tree | 0   | Generates a tree structure of comparators.   |
| Cascade         | 1, when block has a single vector input port. | This implementation is optimized for latency * area, with medium speed. See “Cascade Architecture Best Practices” (HDL Coder). |

## HDL Block Properties

|                                  |  |
|----------------------------------|--|
| <b>ConstrainedOutputPipeline</b> | Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder). |
| <b>InputPipeline</b>             | Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).                 |
| <b>OutputPipeline</b>            | Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).               |
| <b>InstantiateStages</b>         | Generate a VHDL entity or Verilog module for each cascade stage. The default is off. See also “InstantiateStages” (HDL Coder).   |
| <b>SerialPartition</b>           | Specify partitions for Cascade-serial implementations as a vector of the lengths of each partition. The default setting uses the minimum number of stages. See also “SerialPartition” (HDL Coder).                                       |

## Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

The parameters on the **Data Types Tab** of the block are used only for complex fixed-point inputs. The sum of the squares of the real and imaginary parts of such an input are formed before a comparison is made, as described under the 'Mode' parameter in “Main Tab” on page 2-1008. The results of the squares of the real and imaginary parts are placed into the product output data type. The result of the sum of the squares is placed into the accumulator data type. These parameters are ignored for other types of inputs.

## See Also

### Functions

min | cummin

### Objects

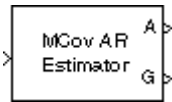
dsp.MovingMinimum | dsp.MovingMaximum

**Blocks**

Maximum | Moving Maximum | Moving Minimum | Mean

# Modified Covariance AR Estimator

Compute estimate of autoregressive (AR) model parameters using modified covariance method



## Libraries:

DSP System Toolbox / Estimation / Parametric Estimation

## Description

The Modified Covariance AR Estimator block uses the modified covariance method to fit an autoregressive (AR) model to the input data. This method minimizes the forward and backward prediction errors in the least squares sense.

## Ports

### Input

#### Input — Input

column vector | unoriented vector

Specify the input data as a column vector or an unoriented vector. The block assumes that the input data is the output of an AR system driven by white noise and represents a frame of consecutive time samples from a single-channel signal.

Data Types: single | double

### Output

#### A — Normalized estimate of the AR model polynomial coefficients

column vector

Normalized estimate of the AR model polynomial coefficients  $A(z)$ , returned as a column vector of length  $p+1$  in descending powers of  $z$ .

The block computes the estimate of these coefficients independently for each successive input frame.

$$H(z) = \frac{G}{A(z)} = \frac{G}{1 + a(2)z^{-1} + \dots + a(p+1)z^{-p}}$$

where,

- $H(z)$  -- Transfer function of the estimated AR model
- $G$  -- Scalar gain
- $A(z)$  -- Polynomial coefficients of the AR model

Data Types: single | double

#### G — Model gain

scalar

Gain of the estimated AR model, returned as a scalar.

$$H(z) = \frac{G}{A(z)} = \frac{G}{1 + a(2)z^{-1} + \dots + a(p+1)z^{-p}}$$

where,

- $H(z)$  -- Transfer function of the estimated AR model
- $G$  -- Scalar gain
- $A(z)$  -- Polynomial coefficients of the AR model

Data Types: `single` | `double`

## Parameters

**Estimation order** — Order of AR model

4 (default) | positive integer

Specify the estimation order  $p$  of the AR model (all-pole model) as a positive integer. To guarantee a nonsingular output, you must set the **Estimation order** parameter to be less than or equal to two thirds the input vector length. Otherwise, the output might be singular.

## Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | <code>double</code>   <code>single</code> |
| <b>Multidimensional Signals</b> | No  |
| <b>Variable-Size Signals</b>    | No  |

## More About

### AR(p) Model

In an AR model of order  $p$ , the current output is a linear combination of the past  $p$  outputs plus a white noise input. The weights on the  $p$  past outputs minimize the mean squared prediction error of the autoregression.

Let  $y(n)$  be a wide-sense stationary random process obtained by filtering white noise of variance  $e$  with the system function  $A(z)$ . If  $P_y(e^{j\omega})$  is the power spectral density of  $y(n)$ , then

$$P_y(e^{j\omega}) = \frac{e}{|A(e^{j\omega})|^2} = \frac{e}{\left|1 + \sum_{k=1}^p a(k)e^{-j\omega k}\right|^2}.$$

Because the modified covariance method characterizes the input data using an all-pole model, the correct choice of the model order  $p$  is important.

## Compare AR Model Parameter Estimation Methods

This table compares the features of the Burg AR Estimator block to the Covariance AR Estimator, Modified Covariance AR Estimator, and the Yule-Walker AR Estimator blocks.

The Yule-Walker AR Estimator and Burg AR Estimator blocks return similar results for large frame sizes.

|                                      | <b>Burg AR Estimator</b>   | <b>Covariance AR Estimator</b>                                    | <b>Modified Covariance AR Estimator</b>   | <b>Yule-Walker AR Estimator</b>  |
|--------------------------------------|--|---|---|--|
| <b>Characteristics</b>               | Does not apply window to data  | Does not apply window to data                                     | Does not apply window to data   | Applies window to data   |
|                                      | Minimizes the forward and backward prediction errors in the least squares sense, with the AR coefficients constrained to satisfy the L-D recursion | Minimizes the forward prediction error in the least squares sense | Minimizes the forward and backward prediction errors in the least squares sense | Minimizes the forward prediction error in the least squares sense (also called "autocorrelation method")         |
| <b>Advantages</b>                    | Always produces a stable model   |   |   | Always produces a stable model   |
| <b>Disadvantages</b>                 |  | Can produce unstable models                                       | Can produce unstable models   | Performs relatively poorly for short data records  |
| <b>Conditions for Nonsingularity</b> |  | Order must be less than or equal to 1/2 the input frame size      | Order must be less than or equal to 2/3 the input frame size                    | Because of the biased estimate, the autocorrelation matrix is guaranteed to positive-definite, hence nonsingular |

## Version History

Introduced before R2006a

## References

- [1] Kay, S. M. *Modern Spectral Estimation: Theory and Application*. Englewood Cliffs, NJ: Prentice-Hall, 1988.
- [2] Marple, S. L., Jr., *Digital Spectral Analysis with Applications*. Englewood Cliffs, NJ: Prentice-Hall, 1987.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Generated code relies on the memcpy or memset function (string.h) under certain conditions.

## **See Also**

### **Functions**

armcov

### **Blocks**

Burg AR Estimator | Covariance AR Estimator | Modified Covariance Method | Yule-Walker AR Estimator

# Modified Covariance Method

Power spectral density estimate using modified covariance method



## Libraries:

DSP System Toolbox / Estimation / Power Spectrum Estimation

## Description

The Modified Covariance Method block estimates the power spectral density (PSD) of the input using the modified covariance method. This method fits an autoregressive (AR) model to the signal by minimizing the forward and backward prediction errors in the least squares sense.

The block computes the spectrum from the FFT of the estimated AR model parameters.

## Ports

### Input

#### Input — Input

column vector | unoriented vector

Specify the input as a column vector or an unoriented vector. This input represents a frame of consecutive time samples from a single-channel signal.

Data Types: `single` | `double`

### Output

#### Output — Power spectral density estimate

column vector

Power spectral density estimate of the signal at  $N_{fft}$  equally spaced frequency points, returned as a column vector. The frequency points are in the range  $[0, F_s)$ , where  $F_s$  is the sampling rate of the signal.

Data Types: `single` | `double`

## Parameters

### Estimation order — Order of AR model

6 (default) | positive integer

Specify the estimation order of the AR model (all-pole model) as a positive integer. The block computes the spectrum from the FFT of the estimated AR model parameters. To guarantee a nonsingular output, the value of the **Estimation order** parameter must be less than or equal to 2/3 of the input vector length.

**Inherit FFT length from estimation order** — Inherit FFT length from estimation order

off (default) | on

When you select the **Inherit FFT length from estimation order** parameter, the FFT length  $N_{fft}$  is one greater than the estimation order. To specify the number of points on which to perform the FFT, clear the **Inherit FFT length from estimation order** parameter. You can then specify a power-of-two FFT length using the **FFT length** parameter. The block zero-pads or wraps the input to  $N_{fft}$  before computing the FFT.

**FFT length** — FFT length

256 (default) | positive integer greater than or equal to 2

Enter the number of data points  $N_{fft}$  on which to perform the FFT as a positive integer greater than or equal to 2. When  $N_{fft}$  is larger than the input frame size, the block zero-pads each frame as needed. When  $N_{fft}$  is smaller than the input frame size, the block wraps each frame as needed.

**Dependencies**

To enable this parameter, clear the **Inherit FFT length from estimation order** parameter.

**Inherit sample time from input** — Inherit sample time from input

on (default) | off

When you select the **Inherit sample time from input** parameter, the block computes the frequency data from the sample period of the input signal. For the block to produce valid output, the following conditions must hold:

- The input to the block is the original signal, with no samples added or deleted (by insertion of zeros, for example).
- The sample period of the time-domain signal in the simulation equals the sample period of the original time series.

If these conditions do not hold, clear the **Inherit sample time from input** check box. You can then specify a sample time using the **Sample time of original time series** parameter.

**Sample time of original time series** — Sample time of original time-domain signal

1 (default) | positive scalar

Specify the sample time of the original time-domain signal as a positive scalar.

**Dependencies**

To enable this parameter, clear the **Inherit sample time from input** parameter.

**Block Characteristics**

|                                 |                 |
|---------------------------------|-----------------|
| <b>Data Types</b>               | double   single |
| <b>Multidimensional Signals</b> | No              |



|                              |    |
|------------------------------|----|
| <b>Variable-Size Signals</b> | No |
|------------------------------|----|

## More About

### Compare Power Spectral Density Estimation Methods

The Burg Method and Yule-Walker Method blocks return similar results for large frame sizes.

This table compares the features of the Burg Method block to the Covariance Method, Modified Covariance Method, and the Yule-Walker Method blocks.

|                        | <b>Burg</b>  | <b>Covariance</b>   | <b>Modified Covariance</b>  | <b>Yule-Walker</b>   |
|------------------------|--|---|---|--|
| <b>Characteristics</b> | Does not apply window to data  | Does not apply window to data   | Does not apply window to data   | Applies window to data   |
|                        | Minimizes the forward and backward prediction errors in the least squares sense, with the AR coefficients constrained to satisfy the L-D recursion | Minimizes the forward prediction error in the least squares sense                   | Minimizes the forward and backward prediction errors in the least squares sense   | Minimizes the forward prediction error in the least squares sense (also called autocorrelation method) |
| <b>Advantages</b>      | High resolution for short data records   | Better resolution than Yule-Walker for short data records (more accurate estimates) | High resolution for short data records  | Performs as well as other methods for large data records   |
|                        | Always produces a stable model   | Able to extract frequencies from data consisting of $p$ or more pure sinusoids      | Able to extract frequencies from data consisting of $p$ or more pure sinusoids<br>Does not suffer spectral line-splitting | Always produces a stable model   |
| <b>Disadvantages</b>   | Peak locations highly dependent on initial phase   | Can produce unstable models   | Can produce unstable models   | Performs relatively poorly for short data records  |
|                        | Can suffer spectral line-splitting for sinusoids in noise, or when order is very large   | Frequency bias for estimates of sinusoids in noise                                  | Peak locations slightly dependent on initial phase  | Frequency bias for estimates of sinusoids in noise   |

|                                      | <b>Burg</b>  | <b>Covariance</b>  | <b>Modified Covariance</b>                                   | <b>Yule-Walker</b>  |
|--------------------------------------|--|--|--|---|
|                                      | Frequency bias for estimates of sinusoids in noise |  | Minor frequency bias for estimates of sinusoids in noise     |   |
| <b>Conditions for Nonsingularity</b> |  | Order must be less than or equal to 1/2 the input frame size | Order must be less than or equal to 2/3 the input frame size | Because of the biased estimate, the autocorrelation matrix is guaranteed to be positive-definite, hence nonsingular |

## Version History

Introduced before R2006a

## References

- [1] Kay, S. M. *Modern Spectral Estimation: Theory and Application*. Englewood Cliffs, NJ: Prentice-Hall, 1988.
- [2] Marple, S. L. Jr., *Digital Spectral Analysis with Applications*. Englewood Cliffs, NJ: Prentice-Hall, 1987.
- [3] Orfanidis, S. J. *Introduction to Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1995.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

## See Also

Burg Method | Covariance Method | Modified Covariance AR Estimator | Short-Time FFT | Yule-Walker Method

### Topics

“Spectral Analysis”

# Moving Average

Moving average



**Libraries:**  
DSP System Toolbox / Statistics

## Description

The Moving Average block computes the moving average of the input signal along each channel independently over time. The block uses the sliding window method or the exponential weighting method to compute the moving average. In the sliding window method, a window of specified length moves over the data sample by sample, and the block computes the average over the data in the window. In the exponential weighting method, the block multiplies the data samples with a set of weighting factors and then sums the weighted data to compute the average. For more details on these methods, see “Algorithms” on page 2-1031.

## Input/Output Ports

### Input

**x** — Data input  
vector | matrix

The block computes the moving average of the data specified at this input port. Specify real- or complex-valued multichannel inputs of the size  $m$ -by- $n$ , where  $m \geq 1$  and  $n \geq 1$ .

When the **Allow arbitrary frame length for fixed-size input signals** parameter appears and is not selected, and you input a fixed-size signal, the frame length must be a multiple of the hop size (window length – overlap length). In all other cases, the input frame length can be arbitrary.

The block accepts variable-size inputs (frame length changes during simulation). When you input a variable-size signal, the frame length of the signal can be arbitrary.

This port is unnamed until you set **Method** to Exponential weighting and select the **Specify forgetting factor from input port** parameter.

Data Types: single | double

**lambda** — Forgetting factor  
positive real scalar in the range (0,1]

The forgetting factor determines how much weight past data is given. A forgetting factor of 0.9 gives more weight to the older data than does a forgetting factor of 0.1. A forgetting factor of 1.0 indicates infinite memory – all previous samples are given an equal weight.

### Dependencies

This port appears when you set **Method** to Exponential weighting and select the **Specify forgetting factor from input port** parameter.

Data Types: `single` | `double`

## Output

**Port\_1** — Moving average output

`vector` | `matrix`

Moving average, returned as a vector or a matrix. The block computes the moving average based on the **Method** parameter settings using either the sliding window method or the exponential weighting method. For more details, see “Algorithms” on page 2-1031.

This table provides more details on the dimensions of the output signal.

| Input Signal         | Input Dimensions  | Output Dimensions When Allow arbitrary frame length for fixed-size input signals Appears  | Output Dimensions When Allow arbitrary frame length for fixed-size input signals Does Not Appear |
|----------------------|---|---|--|
| Fixed-size signal    | $m$ -by- $n$ , where $m$ is a multiple of the hop size (window length – overlap length)     | $(m/\text{hop size})$ -by- $n$  | $m$ -by- $n$   |
| Fixed-size signal    | $m$ -by- $n$ , where $m$ is not a multiple of the hop size (window length – overlap length) | $\text{ceil}(m/\text{hop size})$ -by- $n$ when you select <b>Allow arbitrary frame length for fixed-size input signals</b><br><br>If you do not select <b>Allow arbitrary frame length for fixed-size input signals</b> , the block errors. | $m$ -by- $n$   |
| Variable-size signal | $m$ -by- $n$  | $\text{ceil}(m/\text{hop size})$ -by- $n$   | $m$ -by- $n$   |

When the output has an upper bound size of  $\text{ceil}(m/\text{hop size})$ -by- $n$ , during simulation, the size of the first dimension varies within this bound and the size of the second dimension remains constant. For an example that shows this behavior, see “Compute Moving Average of Noisy Step Signal”.

Data Types: `single` | `double`

## Parameters

If a parameter is listed as tunable, then you can change its value during simulation.

**Method** — Averaging method

`Sliding window` (default) | `Exponential weighting`

- **Sliding window** — A window of length **Window length** moves over the input data along each channel. For every sample the window moves over, the block computes the average over the data in the window.

- **Exponential weighting** — The block multiplies the samples by a set of weighting factors. The magnitude of the weighting factors decreases exponentially as the age of the data increases, but the magnitude never reaches zero. To compute the average, the algorithm sums the weighted data.

**Specify window length** — Flag to specify window length

on (default) | off

When you select this check box, the length of the sliding window is equal to the value you specify in **Window length**. When you clear this check box, the length of the sliding window is infinite. In this mode, the block computes the average of the current sample and all previous samples in the channel.

#### Dependencies

To enable this parameter, set **Method** to `Sliding window`.

**Window length** — Length of sliding window

4 (default) | positive scalar integer

Specifies the length of the sliding window in samples.

#### Dependencies

To enable this parameter, set **Method** to `Sliding window` and select the **Specify window length** check box.

**Overlap length** — Overlap length between windows

3 (default) | nonnegative integer

Specify the overlap length between sliding windows as a nonnegative integer. The value of overlap length varies in the range  $[0, \mathbf{Window\ length} - 1]$ .

#### Dependencies

To enable this parameter, set **Method** to `Sliding window` and select the **Specify window length** check box.

**Allow arbitrary frame length for fixed-size input signals** — Allow arbitrary frame length for fixed-size input signals

off (default) | on

Specify whether fixed-size input signals (whose size does not change during simulation) can have an arbitrary frame length, where the frame length does not have to be a multiple of the hop size. Hop size is defined as **Window length** – **Overlap length**. The block uses this parameter setting only for fixed-size input signals and ignores this parameter if the input has a variable-size.

When the input signal is a variable-size signal, the signal can have arbitrary frame length, that is, the frame length does not have to be a multiple of the hop size.

For fixed-size input signals, if you:

- Select the **Allow arbitrary frame length for fixed-size input signals** parameter, the frame length of the signal does not have to be a multiple of the hop size. If the input is not a multiple of

the hop size, then the output is generally a variable-size signal. Therefore, to support arbitrary input size, the block must also support variable-size operations, which you can enable by selecting the **Allow arbitrary frame length for fixed-size input signals** parameter.

- Clear the **Allow arbitrary frame length for fixed-size input signals** parameter, the input frame length must be a multiple of the hop size.

#### Dependencies

To enable this parameter, set **Method** to `Sliding window` and select the **Specify window length** check box.

**Specify forgetting factor from input port** — Flag to specify forgetting factor  
off (default) | on

When you select this check box, the forgetting factor is input through the **lambda** port. When you clear this check box, the forgetting factor is specified on the block dialog through the **Forgetting factor** parameter.

#### Dependencies

To enable this parameter, set **Method** to `Exponential weighting`.

**Forgetting factor** — Exponential weighting factor  
0.9 (default) | positive real scalar in the range (0,1]

The forgetting factor determines how much weight past data is given. A forgetting factor of 0.9 gives more weight to the older data than does a forgetting factor of 0.1. A forgetting factor of 1.0 indicates infinite memory - all previous samples are given an equal weight.

**Tunable:** Yes

#### Dependencies

To enable this parameter, set **Method** to `Exponential weighting` and clear the **Specify forgetting factor from input port** check box.

**Simulate using** — Type of simulation to run  
Code generation (default) | Interpreted execution

Specify the type of simulation to run as one of the following:

- `Code generation` -- Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than `Interpreted execution`.
- `Interpreted execution` -- Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than `Code generation`.

## Block Characteristics

|                   |                 |
|-------------------|-----------------|
| <b>Data Types</b> | double   single |
|-------------------|-----------------|

|                                 |     |
|---------------------------------|-----|
| <b>Multidimensional Signals</b> | No  |
| <b>Variable-Size Signals</b>    | Yes |

## Algorithms

### Sliding Window Method

In the sliding window method, the output for each input sample is the average of the current sample and  $Len - 1$  previous samples.  $Len$  is the length of the window in samples. To compute the first output sample, the algorithm waits until it receives the hop size number of input samples. Hop size is defined as window length - overlap length. Remaining samples in the window are considered to be zero. As an example, if the window length is 5 and the overlap length is 2, then the algorithm waits until it receives 3 samples of input to compute the first sample of the output. After generating the first output, it generates the subsequent output samples for every hop size number of input samples.

When you do not specify the window length, the algorithm chooses an infinite window length. In this mode, the output is the moving average of the current sample and all the previous samples in the channel.

For an example, see “Sliding Window Method and Exponential Weighting Method”.

### Exponential Weighting Method

In the exponential weighting method, the moving average is computed recursively using these formulas:

$$w_{N,\lambda} = \lambda w_{N-1,\lambda} + 1$$

$$\bar{x}_{N,\lambda} = \left(1 - \frac{1}{w_{N,\lambda}}\right) \bar{x}_{N-1,\lambda} + \left(\frac{1}{w_{N,\lambda}}\right) x_N$$

- $\bar{x}_{N,\lambda}$  — Moving average at the current sample
- $x_N$  — Current data input sample
- $\bar{x}_{N-1,\lambda}$  — Moving average at the previous sample
- $\lambda$  — Forgetting factor
- $w_{N,\lambda}$  — Weighting factor applied to the current data sample
- $\left(1 - \frac{1}{w_{N,\lambda}}\right) \bar{x}_{N-1,\lambda}$  — Effect of the previous data on the average

For the first sample, where  $N = 1$ , the algorithm chooses  $w_{N,\lambda} = 1$ . For the next sample, the weighting factor is updated and used to compute the average, as per the recursive equation. As the age of the data increases, the magnitude of the weighting factor decreases exponentially and never reaches zero. In other words, the recent data has more influence on the current average than the older data.

The value of the forgetting factor determines the rate of change of the weighting factors. A forgetting factor of 0.9 gives more weight to the older data than does a forgetting factor of 0.1. A forgetting factor of 1.0 indicates infinite memory. All the previous samples are given an equal weight.

For an example, see “Sliding Window Method and Exponential Weighting Method”.

## Version History

Introduced in R2016b

### R2022b: New Overlap length parameter

Starting in R2022b, you can specify the overlap length between sliding windows using the **Overlap length** parameter.

### R2022b: Support for arbitrary input frame length

Moving Average block supports input signals with arbitrary frame lengths when the:

- Input signal is a fixed-size signal (frame length does not change during simulation). If **Allow arbitrary frame length for fixed-size input signals** is enabled, select the parameter.
- Input signal is a variable-size signal (frame length changes during simulation).

When this block supports an arbitrary frame length input signal, the input frame length does not have to be a multiple of the hop size.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

## See Also

### Blocks

Moving RMS | Moving Maximum | Moving Minimum | Moving Standard Deviation | Moving Variance | Median Filter

### Objects

dsp.MovingAverage | dsp.MovingRMS | dsp.MovingMaximum | dsp.MovingMinimum | dsp.MovingStandardDeviation | dsp.MovingVariance | dsp.MedianFilter

### Topics

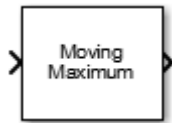
“What Are Moving Statistics?”

“Sliding Window Method and Exponential Weighting Method”



# Moving Maximum

Moving maximum



**Libraries:**  
DSP System Toolbox / Statistics

## Description

The Moving Maximum block determines the moving maximum of the input signal along each channel independently over time. The block uses the sliding window method to determine the moving maximum. In this method, a window of specified length moves over each channel sample by sample, and the block determines the maximum over the data in the window. For more details, see “Algorithms” on page 2-1034.

## Input/Output Ports

### Input

**Port\_1** — Data input

column vector | row vector | matrix

Data over which the moving maximum is determined using the sliding window method. The block accepts real-valued or complex-valued multichannel inputs, that is,  $m$ -by- $n$  size inputs, where  $m \geq 1$  and  $n \geq 1$ . The block also accepts variable-size inputs. During simulation, you can change the size of each input channel. However, the number of channels cannot change.

Data Types: single | double

### Output

**Port\_1** — Moving maximum output

column vector | row vector | matrix

Moving maximum output, determined using the sliding window method. The size of the output matches the size of the input. The window slides column-wise along each channel, and the block determines the maximum of the data in the window. For more details, see “Algorithms” on page 2-1034.

Data Types: single | double

## Parameters

**Specify window length** — Flag to specify window length

on (default) | off

When you select this check box, the length of the sliding window is equal to the value you specify through the **Window length** parameter. When you clear this check box, the length of the sliding

window is infinite. In this mode, the block determines the maximum of the current sample and all previous samples in the channel.

**Window length** — Length of the sliding window

4 (default) | positive scalar integer

**Window length** specifies the length of the sliding window in samples.

#### Dependencies

This parameter appears when you select the **Specify window length** check box.

**Simulate using** — Type of simulation to run

Code generation (default) | Interpreted execution

Specify the type of simulation to run as one of the following:

- **Code generation** -- Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than **Interpreted execution**.
- **Interpreted execution** -- Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than **Code generation**.

## Block Characteristics

|                                 |                           |
|---------------------------------|---------------------------|
| <b>Data Types</b>               | double   integer   single |
| <b>Direct Feedthrough</b>       | no                        |
| <b>Multidimensional Signals</b> | no                        |
| <b>Variable-Size Signals</b>    | yes                       |
| <b>Zero-Crossing Detection</b>  | no                        |

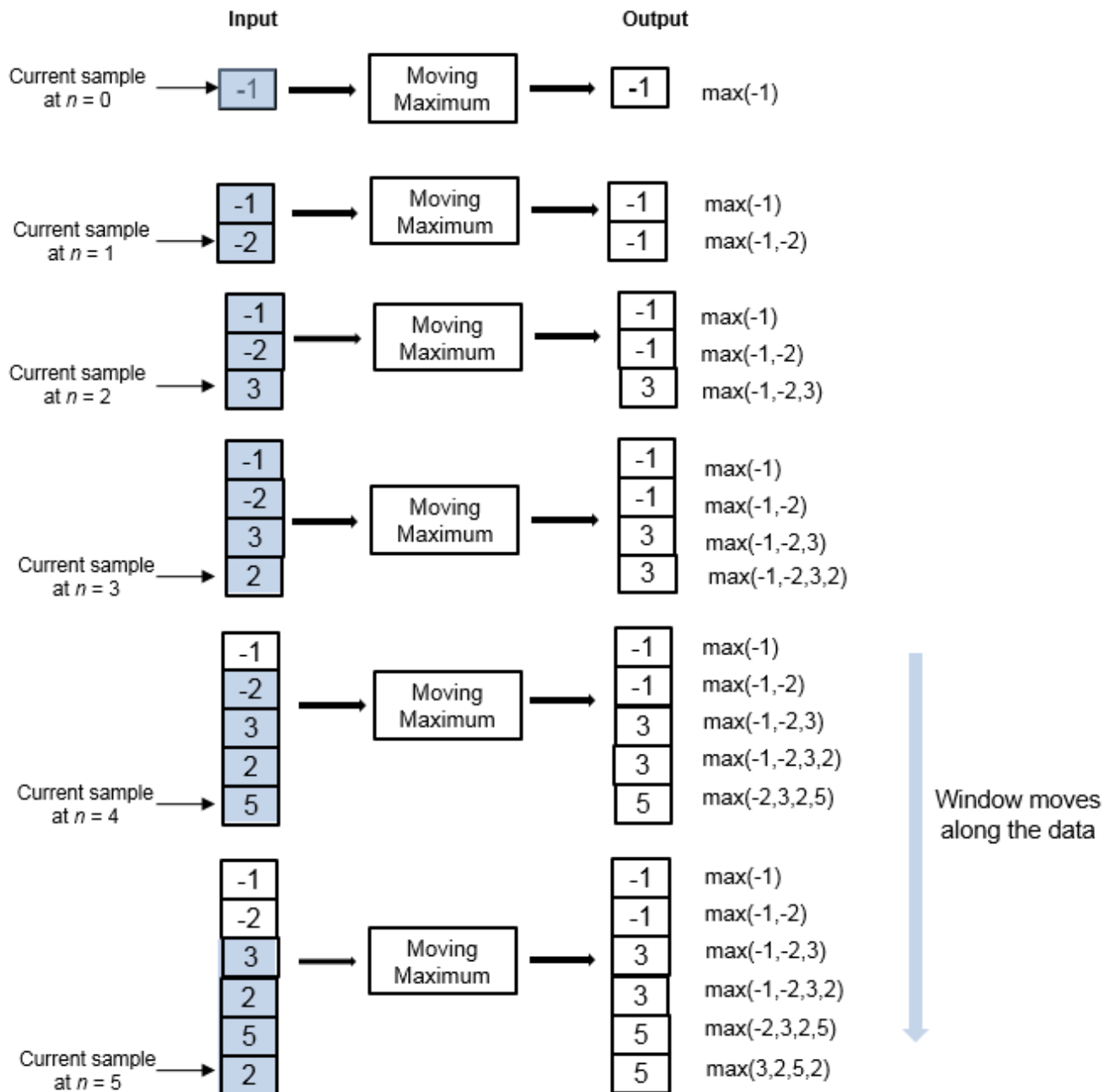
## Algorithms

### Sliding Window Method

In the sliding window method, the output for each input sample is the maximum of the current sample and the  $Len - 1$  previous samples.  $Len$  is the length of the window in samples. When the algorithm computes the first  $Len - 1$  outputs, the length of the window is the length of the data that is available.

When you do not specify the window length, the algorithm chooses an infinite window length. In this mode, the output is the maximum of the current sample and all the previous samples in the channel.

Consider an example of computing the moving maximum of a streaming input data using the sliding window method. The algorithm uses a window length of 4. With each input sample that comes in, the window of length 4 moves along the data.



## Version History

Introduced in R2016b

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

## **See Also**

### **Blocks**

Maximum | Moving Minimum | Moving Average | Moving RMS | Moving Standard Deviation | Moving Variance | Median Filter

### **Objects**

`dsp.MovingMaximum` | `dsp.MovingMinimum` | `dsp.MovingAverage` | `dsp.MovingRMS` | `dsp.MovingStandardDeviation` | `dsp.MovingVariance` | `dsp.MedianFilter`

### **Topics**

“Streaming Signal Statistics”  
“What Are Moving Statistics?”

# Moving Minimum

Moving minimum



**Libraries:**  
DSP System Toolbox / Statistics

## Description

The Moving Minimum block determines the moving minimum of the input signal along each channel independently over time. The block uses the sliding window method to determine the moving minimum. In this method, a window of specified length moves over each channel sample by sample, and the block determines the minimum over the data in the window. For more details, see “Algorithms” on page 2-1038.

## Input/Output Ports

### Input

**Port\_1** — Data input  
column vector | row vector | matrix

Data over which the moving minimum is determined using the sliding window method. The block accepts real-valued or complex-valued multichannel inputs, that is,  $m$ -by- $n$  size inputs, where  $m \geq 1$  and  $n \geq 1$ . The block also accepts variable-size inputs. During simulation, you can change the size of each input channel. However, the number of channels cannot change.

Data Types: `single` | `double`

### Output

**Port\_1** — Moving minimum output  
column vector | row vector | matrix

Moving minimum output, determined using the sliding window method. The size of the output matches the size of the input. The window slides column-wise along each channel, and the block determines the minimum of the data in the window. For more details, see “Algorithms” on page 2-1038.

Data Types: `single` | `double`

## Parameters

**Specify window length** — Flag to specify window length  
on (default) | off

When you select this check box, the length of the sliding window is equal to the value you specify through the **Window length** parameter. When you clear this check box, the length of the sliding

window is infinite. In this mode, the block determines the minimum of the current sample and all the previous samples in the channel.

**Window length** — Length of the sliding window  
4 (default) | positive scalar integer

**Window length** specifies the length of the sliding window in samples.

#### Dependencies

This parameter appears when you select the **Specify window length** check box.

**Simulate using** — Type of simulation to run  
Code generation (default) | Interpreted execution

Specify the type of simulation to run as one of the following:

- **Code generation** -- Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than **Interpreted execution**.
- **Interpreted execution** -- Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than **Code generation**.

## Block Characteristics

|                                 |                           |
|---------------------------------|---------------------------|
| <b>Data Types</b>               | double   integer   single |
| <b>Direct Feedthrough</b>       | no                        |
| <b>Multidimensional Signals</b> | no                        |
| <b>Variable-Size Signals</b>    | yes                       |
| <b>Zero-Crossing Detection</b>  | no                        |

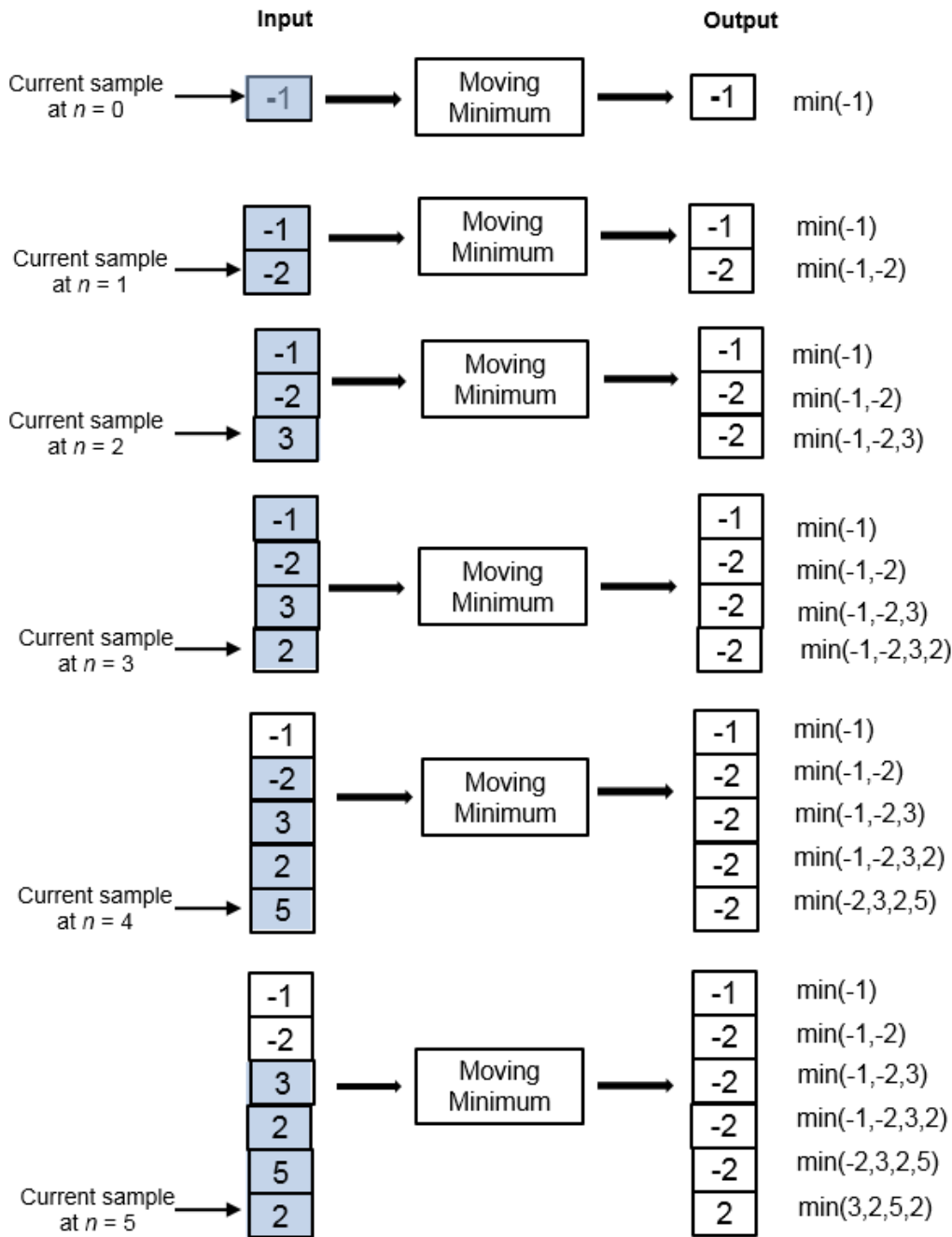
## Algorithms

### Sliding Window Method

In the sliding window method, the output for each input sample is the minimum of the current sample and the  $Len - 1$  previous samples.  $Len$  is the length of the window in samples. When the algorithm computes the first  $Len - 1$  outputs, the length of the window is the length of the data that is available.

When you do not specify the window length, the algorithm chooses an infinite window length. In this mode, the output is the minimum of the current sample and all the previous samples in the channel.

Consider an example of computing the moving minimum of a streaming input data using the sliding window method. The algorithm uses a window length of 4. With each input sample that comes in, the window of length 4 moves along the data.



Window moves along the data

## Version History

Introduced in R2016b

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

## **See Also**

### **Blocks**

[Minimum](#) | [Moving Maximum](#) | [Moving Average](#) | [Moving RMS](#) | [Moving Standard Deviation](#) | [Moving Variance](#) | [Median Filter](#)

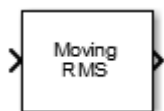
### **Objects**

[dsp.MovingMinimum](#) | [dsp.MovingMaximum](#) | [dsp.MovingAverage](#) | [dsp.MovingRMS](#) | [dsp.MovingStandardDeviation](#) | [dsp.MovingVariance](#) | [dsp.MedianFilter](#)



# Moving RMS

Moving RMS



## Libraries:

DSP System Toolbox / Statistics

## Description

The Moving RMS block computes the moving root mean square (RMS) of the input signal along each channel independently over time. The block uses either the sliding window method or the exponential weighting method to compute the moving RMS. In the sliding window method, a window of specified length moves over the data sample by sample, and the block computes the RMS over the data in the window. In the exponential weighting method, the block squares the data samples, multiplies them with a set of weighting factors, and sums the weighed data. The block then computes the RMS by taking the square root of the sum. For more details on these methods, see “Algorithms” on page 2-1045.

## Input/Output Ports

### Input

**x** — Data input  
vector | matrix

The block computes the moving RMS of the data specified at this input port. Specify real- or complex-valued multichannel inputs of the size  $m$ -by- $n$ , where  $m \geq 1$  and  $n \geq 1$ .

When the **Allow arbitrary frame length for fixed-size input signals** parameter appears and is not selected, and you input a fixed-size signal, the frame length must be a multiple of the hop size (window length – overlap length). In all other cases, the input frame length can be arbitrary.

The block accepts variable-size inputs (frame length changes during simulation). When you input a variable-size signal, the frame length of the signal can be arbitrary.

This port is unnamed until you set **Method** to Exponential weighting and select the **Specify forgetting factor from input port** parameter.

Data Types: single | double

**lambda** — Forgetting factor  
positive real scalar in the range (0,1]

The forgetting factor determines how much weight past data is given. A forgetting factor of 0.9 gives more weight to the older data than does a forgetting factor of 0.1. A forgetting factor of 1.0 indicates infinite memory – all previous samples are given an equal weight.

## Dependencies

This port appears when you set **Method** to Exponential weighting and select the **Specify forgetting factor from input port** parameter.

Data Types: single | double

## Output

**Port\_1** — Moving RMS output  
vector | matrix

Moving RMS output, returned as a vector or a matrix. The block computes the moving RMS based on the **Method** parameter settings using either the sliding window method or the exponential weighting method. For more details, see “Algorithms” on page 2-1045.

This table provides more details on the dimensions of the output signal.

| Input Signal         | Input Dimensions  | Output Dimensions When Allow arbitrary frame length for fixed-size input signals Appears  | Output Dimensions When Allow arbitrary frame length for fixed-size input signals Does Not Appear |
|----------------------|---|---|--|
| Fixed-size signal    | $m$ -by- $n$ , where $m$ is a multiple of the hop size (window length – overlap length)     | $(m/\text{hop size})$ -by- $n$  | $m$ -by- $n$   |
| Fixed-size signal    | $m$ -by- $n$ , where $m$ is not a multiple of the hop size (window length – overlap length) | $\text{ceil}(m/\text{hop size})$ -by- $n$ when you select <b>Allow arbitrary frame length for fixed-size input signals</b><br><br>If you do not select <b>Allow arbitrary frame length for fixed-size input signals</b> , the block errors. | $m$ -by- $n$   |
| Variable-size signal | $m$ -by- $n$  | $\text{ceil}(m/\text{hop size})$ -by- $n$   | $m$ -by- $n$   |

When the output has an upper bound size of  $\text{ceil}(m/\text{hop size})$ -by- $n$ , during simulation, the size of the first dimension varies within this bound and the size of the second dimension remains constant. For an example that shows this behavior, see “Compute Moving RMS of Noisy Step Signal”.

Data Types: single | double

## Parameters

If a parameter is listed as tunable, then you can change its value during simulation.

**Method** — Moving RMS method  
Sliding window (default) | Exponential weighting

- **Sliding window** — A window of length **Window length** moves over the input data along each channel. For every sample the window moves over, the block computes the RMS over the data in the window.
- **Exponential weighting** — The block multiplies the squares of the samples by a set of weighting factors. The magnitude of the weighting factors decreases exponentially as the age of the data increases, but the magnitude never reaches zero. To compute the RMS, the algorithm sums the weighted data and takes a square root of the sum.

**Specify window length** — Flag to specify window length

on (default) | off

When you select this check box, the length of the sliding window is equal to the value you specify in **Window length**. When you clear this check box, the length of the sliding window is infinite. In this mode, the block computes the RMS of the current sample and all the previous samples in the channel.

#### Dependencies

To enable this parameter, set **Method** to **Sliding window**.

**Window length** — Length of sliding window

4 (default) | positive scalar integer

Specifies the length of the sliding window in samples.

#### Dependencies

To enable this parameter, set **Method** to **Sliding window** and select the **Specify window length** check box.

**Overlap length** — Overlap length between windows

3 (default) | nonnegative integer

Specify the overlap length between sliding windows as a nonnegative integer. The value of overlap length varies in the range  $[0, \mathbf{Window\ length} - 1]$ .

#### Dependencies

To enable this parameter, set **Method** to **Sliding window** and select the **Specify window length** check box.

**Allow arbitrary frame length for fixed-size input signals** — Allow arbitrary frame length for fixed-size input signals

off (default) | on

Specify whether fixed-size input signals (whose size does not change during simulation) can have an arbitrary frame length, where the frame length does not have to be a multiple of the hop size. Hop size is defined as **Window length** – **Overlap length**. The block uses this parameter setting only for fixed-size input signals and ignores this parameter if the input has a variable-size.

When the input signal is a variable-size signal, the signal can have arbitrary frame length, that is, the frame length does not have to be a multiple of the hop size.

For fixed-size input signals, if you:

- Select the **Allow arbitrary frame length for fixed-size input signals** parameter, the frame length of the signal does not have to be a multiple of the hop size. If the input is not a multiple of the hop size, then the output is generally a variable-size signal. Therefore, to support arbitrary input size, the block must also support variable-size operations, which you can enable by selecting the **Allow arbitrary frame length for fixed-size input signals** parameter.
- Clear the **Allow arbitrary frame length for fixed-size input signals** parameter, the input frame length must be a multiple of the hop size.

### Dependencies

To enable this parameter, set **Method** to `Sliding window` and select the **Specify window length** check box.

**Specify forgetting factor from input port** — Flag to specify forgetting factor  
off (default) | on

When you select this check box, the forgetting factor is input through the **lambda** port. When you clear this check box, the forgetting factor is specified on the block dialog through the **Forgetting factor** parameter.

### Dependencies

To enable this parameter, set **Method** to `Exponential weighting`.

**Forgetting factor** — Exponential weighting factor  
0.9 (default) | positive real scalar in the range (0,1]

The forgetting factor determines how much weight past data is given. A forgetting factor of 0.9 gives more weight to the older data than does a forgetting factor of 0.1. A forgetting factor of 1.0 indicates infinite memory - all previous samples are given an equal weight.

**Tunable:** Yes

### Dependencies

To enable this parameter, set **Method** to `Exponential weighting` and clear the **Specify forgetting factor from input port** check box.

**Simulate using** — Type of simulation to run  
`Code generation` (default) | `Interpreted execution`

Specify the type of simulation to run as one of the following:

- `Code generation` -- Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than `Interpreted execution`.

- **Interpreted execution** -- Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than Code generation.

## Block Characteristics

|                                 |                 |
|---------------------------------|-----------------|
| <b>Data Types</b>               | double   single |
| <b>Multidimensional Signals</b> | No              |
| <b>Variable-Size Signals</b>    | Yes             |

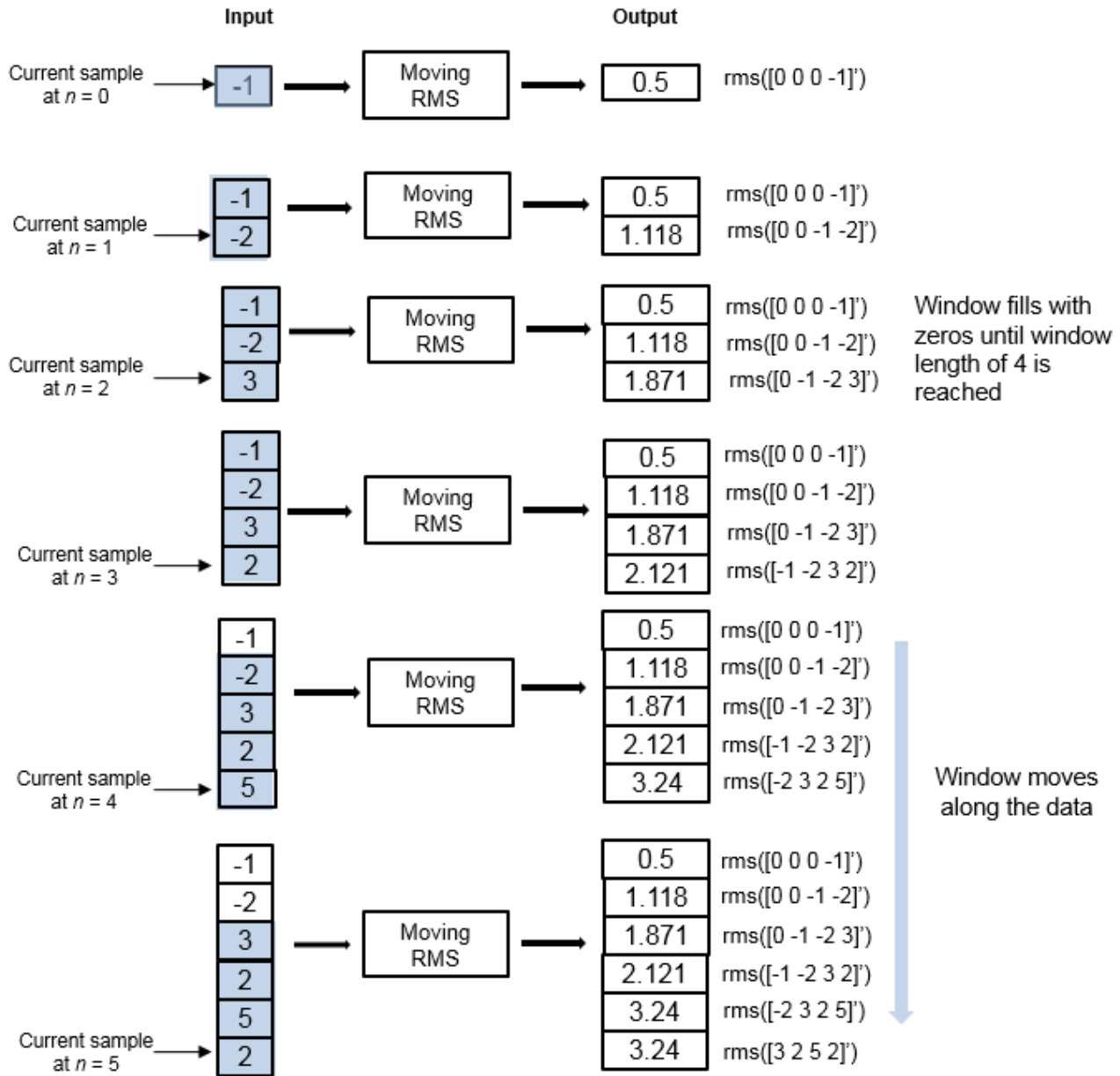
## Algorithms

### Sliding Window Method

In the sliding window method, the output for each input sample is the RMS of the current sample and  $Len - 1$  previous samples.  $Len$  is the length of the window in samples. To compute the first output sample, the algorithm waits until it receives the hop size number of input samples. Hop size is defined as window length - overlap length. Remaining samples in the window are considered to be zero. As an example, if the window length is 5 and the overlap length is 2, then the algorithm waits until it receives 3 samples of input to compute the first sample of the output. After generating the first output, it generates the subsequent output samples for every hop size number of input samples.

When you do not specify the window length, the algorithm chooses an infinite window length. In this mode, the output is the moving RMS of the current sample and all the previous samples in the channel.

Consider an example of computing the moving RMS of a streaming input data using the sliding window method. The algorithm uses a window length of 4 and an overlap length of 3. With each input sample that comes in, the window of length 4 moves along the data.



### Exponential Weighting Method

In the exponential weighting method, the moving RMS is computed recursively using these formulas:

$$w_{N,\lambda} = \lambda w_{N-1,\lambda} + 1$$

$$x_{rmsN,\lambda} = \sqrt{\left(1 - \frac{1}{w_{N,\lambda}}\right)x_{rmsN-1,\lambda} + \left(\frac{1}{w_{N,\lambda}}\right)x_N^2}$$

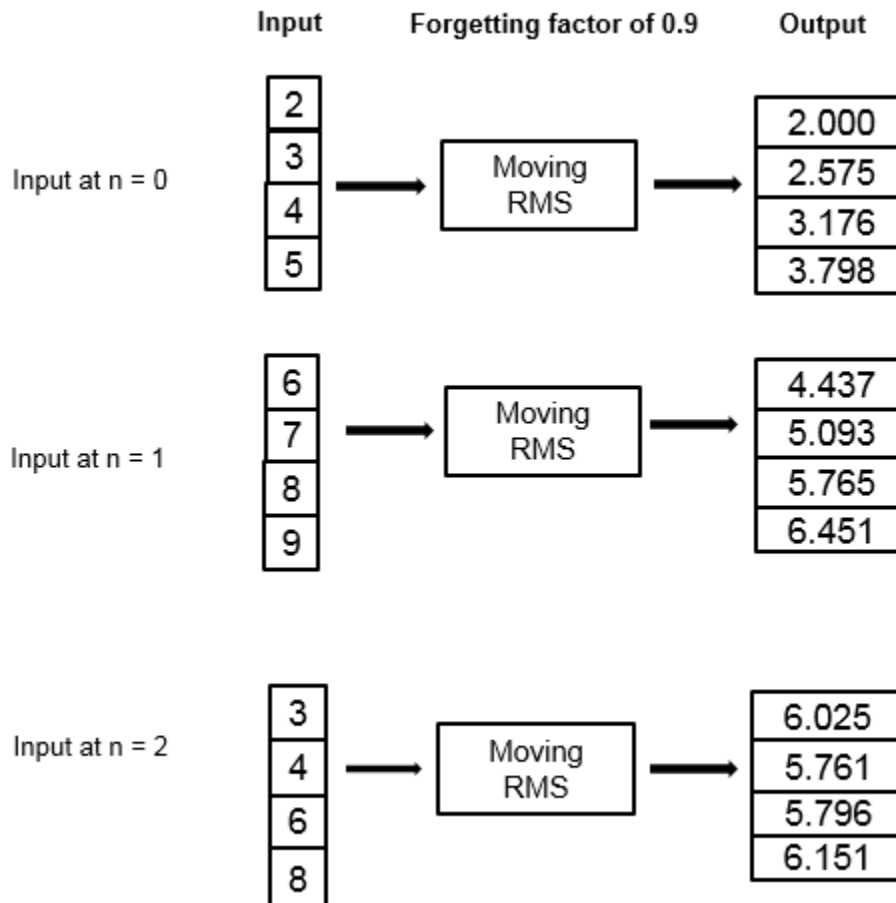
- $x_{rmsN,\lambda}$  – Moving RMS at the current sample
- $x_N^2$  – Square of the current input data sample

- $x_{rms_{N-1,\lambda}}$  – Moving RMS at the previous sample
- $\lambda$  – Forgetting factor
- $w_{N,\lambda}$  – Weighting factor applied to the current data sample
- $\left(1 - \frac{1}{w_{N,\lambda}}\right)x_{rms_{N-1,\lambda}}$  – Effect of the previous data on the RMS

For the first sample, where  $N = 1$ , the algorithm chooses  $w_{N,\lambda} = 1$ . For the next sample, the weighting factor is updated and used to compute the RMS, as per the recursive equation. As the age of the data increases, the magnitude of the weighting factor decreases exponentially and never reaches zero. In other words, the recent data has more influence on the current RMS than the older data.

The value of the forgetting factor determines the rate of change of the weighting factors. A forgetting factor of 0.9 gives more weight to the older data than does a forgetting factor of 0.1. A forgetting factor of 1.0 indicates infinite memory. All the previous samples are given an equal weight.

Here is an example of computing the moving RMS using the exponential weighting method. The forgetting factor is 0.9.



## Version History

Introduced in R2016b

### R2022b: New Overlap length parameter

Starting in R2022b, you can specify the overlap length between sliding windows using the **Overlap length** parameter.

### R2022b: Support for arbitrary input frame length

Moving RMS block supports input signals with arbitrary frame lengths when the:

- Input signal is a fixed-size signal (frame length does not change during simulation) and you select the **Allow arbitrary frame length for fixed-size input signals** parameter (if enabled).
- Input signal is a variable-size signal (frame length changes during simulation).

When this block supports an arbitrary frame length input signal, the input frame length does not have to be a multiple of the hop size.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

## See Also

### Blocks

RMS | Moving Average | Moving Maximum | Moving Minimum | Moving Standard Deviation | Moving Variance | Median Filter

### Objects

`dsp.MovingRMS` | `dsp.MovingAverage` | `dsp.MovingMaximum` | `dsp.MovingMinimum` | `dsp.MovingStandardDeviation` | `dsp.MovingVariance` | `dsp.MedianFilter`

### Topics

“What Are Moving Statistics?”

“Sliding Window Method and Exponential Weighting Method”

“Streaming Signal Statistics”

“Energy Detection in the Time Domain”



# Moving Variance

Moving variance



## Libraries:

DSP System Toolbox / Statistics

## Description

The Moving Variance block computes the moving variance of the input signal along each channel independently over time. The block uses either the sliding window method or the exponential weighting method to compute the moving variance. In the sliding window method, a window of specified length moves over the data sample by sample, and the block computes the variance over the data in the window. In the exponential weighting method, the block subtracts each sample of the data from the average, squares the difference, and multiplies the squared result by a weighting factor. The block then computes the variance by adding all the weighted data. For more details on these methods, see “Algorithms” on page 2-1053.

## Input/Output Ports

### Input

**x** — Data input  
vector | matrix

The block computes the moving variance of the data specified at this input port. Specify real- or complex-valued multichannel inputs of the size  $m$ -by- $n$ , where  $m \geq 1$  and  $n \geq 1$ .

When the **Allow arbitrary frame length for fixed-size input signals** parameter appears and is not selected, and you input a fixed-size signal, the frame length must be a multiple of the hop size (window length – overlap length). In all other cases, the input frame length can be arbitrary.

The block accepts variable-size inputs (frame length changes during simulation). When you input a variable-size signal, the frame length of the signal can be arbitrary.

This port is unnamed until you set **Method** to `Exponential weighting` and select the **Specify forgetting factor from input port** parameter.

Data Types: `single` | `double`

**lambda** — Forgetting factor  
positive real scalar in the range (0,1]

The forgetting factor determines how much weight past data is given. A forgetting factor of 0.9 gives more weight to the older data than does a forgetting factor of 0.1. A forgetting factor of 1.0 indicates infinite memory – all previous samples are given an equal weight.

## Dependencies

This port appears when you set **Method** to Exponential weighting and select the **Specify forgetting factor from input port** parameter.

Data Types: single | double

## Output

**Port\_1** — Moving variance output  
column vector | row vector | matrix

Moving variance output, returned as a vector or a matrix. The block computes the moving variance based on the **Method** parameter settings using either the sliding window method or the exponential weighting method. For more details, see “Algorithms” on page 2-1053.

This table provides more details on the dimensions of the output signal.

| Input Signal         | Input Dimensions  | Output Dimensions When Allow arbitrary frame length for fixed-size input signals Appears  | Output Dimensions When Allow arbitrary frame length for fixed-size input signals Does Not Appear |
|----------------------|---|---|--|
| Fixed-size signal    | $m$ -by- $n$ , where $m$ is a multiple of the hop size (window length – overlap length)     | $(m/\text{hop size})$ -by- $n$  | $m$ -by- $n$   |
| Fixed-size signal    | $m$ -by- $n$ , where $m$ is not a multiple of the hop size (window length – overlap length) | $\text{ceil}(m/\text{hop size})$ -by- $n$ when you select <b>Allow arbitrary frame length for fixed-size input signals</b><br><br>If you do not select <b>Allow arbitrary frame length for fixed-size input signals</b> , the block errors. | $m$ -by- $n$   |
| Variable-size signal | $m$ -by- $n$  | $\text{ceil}(m/\text{hop size})$ -by- $n$   | $m$ -by- $n$   |

When the output has an upper bound size of  $\text{ceil}(m/\text{hop size})$ -by- $n$ , during simulation, the size of the first dimension varies within this bound and the size of the second dimension remains constant. For an example that shows this behavior, see “Compute Moving Variance of Noisy Square Wave Signal”.

Data Types: single | double

## Parameters

If a parameter is listed as tunable, then you can change its value during simulation.

**Method** — Moving variance method  
Sliding window (default) | Exponential weighting

- **Sliding window** — A window of length **Window length** moves over the input data along each channel. For every sample the window moves over, the block computes the variance over the data in the window.
- **Exponential weighting** — The block subtracts each sample of the data from the average, squares the difference, and multiplies the squared result by a weighting factor. The block then computes the variance by adding all the weighted data. The magnitude of the weighting factors decreases exponentially as the age of the data increases, but the magnitude never reaches zero.

For more details on these methods, see “Algorithms” on page 2-1053.

**Specify window length** — Flag to specify window length

on (default) | off

When you select this check box, the length of the sliding window is equal to the value you specify in **Window length**. When you clear this check box, the length of the sliding window is infinite. In this mode, the block computes the variance of the current sample with respect to all the previous samples in the channel.

#### Dependencies

To enable this parameter, set **Method** to **Sliding window**.

**Window length** — Length of sliding window

4 (default) | positive scalar integer

Specifies the length of the sliding window in samples.

#### Dependencies

To enable this parameter, set **Method** to **Sliding window** and select the **Specify window length** check box.

**Overlap length** — Overlap length between windows

3 (default) | nonnegative integer

Specify the overlap length between sliding windows as a nonnegative integer. The value of overlap length varies in the range [0, **Window length** – 1].

#### Dependencies

To enable this parameter, set **Method** to **Sliding window** and select the **Specify window length** check box.

**Allow arbitrary frame length for fixed-size input signals** — Allow arbitrary frame length for fixed-size input signals

off (default) | on

Specify whether fixed-size input signals (whose size does not change during simulation) can have an arbitrary frame length, where the frame length does not have to be a multiple of the hop size. Hop

size is defined as **Window length – Overlap length**. The block uses this parameter setting only for fixed-size input signals and ignores this parameter if the input has a variable-size.

When the input signal is a variable-size signal, the signal can have arbitrary frame length, that is, the frame length does not have to be a multiple of the hop size.

For fixed-size input signals, if you:

- Select the **Allow arbitrary frame length for fixed-size input signals** parameter, the frame length of the signal does not have to be a multiple of the hop size. If the input is not a multiple of the hop size, then the output is generally a variable-size signal. Therefore, to support arbitrary input size, the block must also support variable-size operations, which you can enable by selecting the **Allow arbitrary frame length for fixed-size input signals** parameter.
- Clear the **Allow arbitrary frame length for fixed-size input signals** parameter, the input frame length must be a multiple of the hop size.

#### Dependencies

To enable this parameter, set **Method** to `Sliding window` and select the **Specify window length** check box.

**Specify forgetting factor from input port** — Flag to specify forgetting factor  
off (default) | on

When you select this check box, the forgetting factor is input through the **lambda** port. When you clear this check box, the forgetting factor is specified on the block dialog through the **Forgetting factor** parameter.

#### Dependencies

To enable this parameter, set **Method** to `Exponential weighting`.

**Forgetting factor** — Exponential weighting factor  
0.9 (default) | positive real scalar in the range (0,1]

The forgetting factor determines how much weight past data is given. A forgetting factor of 0.9 gives more weight to the older data than does a forgetting factor of 0.1. A forgetting factor of 1.0 indicates infinite memory - all previous samples are given an equal weight.

**Tunable:** Yes

#### Dependencies

To enable this parameter, set **Method** to `Exponential weighting` and clear the **Specify forgetting factor from input port** check box.

**Simulate using** — Type of simulation to run  
`Code generation` (default) | `Interpreted execution`

Specify the type of simulation to run as one of the following:

- `Code generation` -- Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent

simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than Interpreted execution.

- Interpreted execution -- Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than Code generation.

## Block Characteristics

|                                 |                 |
|---------------------------------|-----------------|
| <b>Data Types</b>               | double   single |
| <b>Multidimensional Signals</b> | No              |
| <b>Variable-Size Signals</b>    | Yes             |

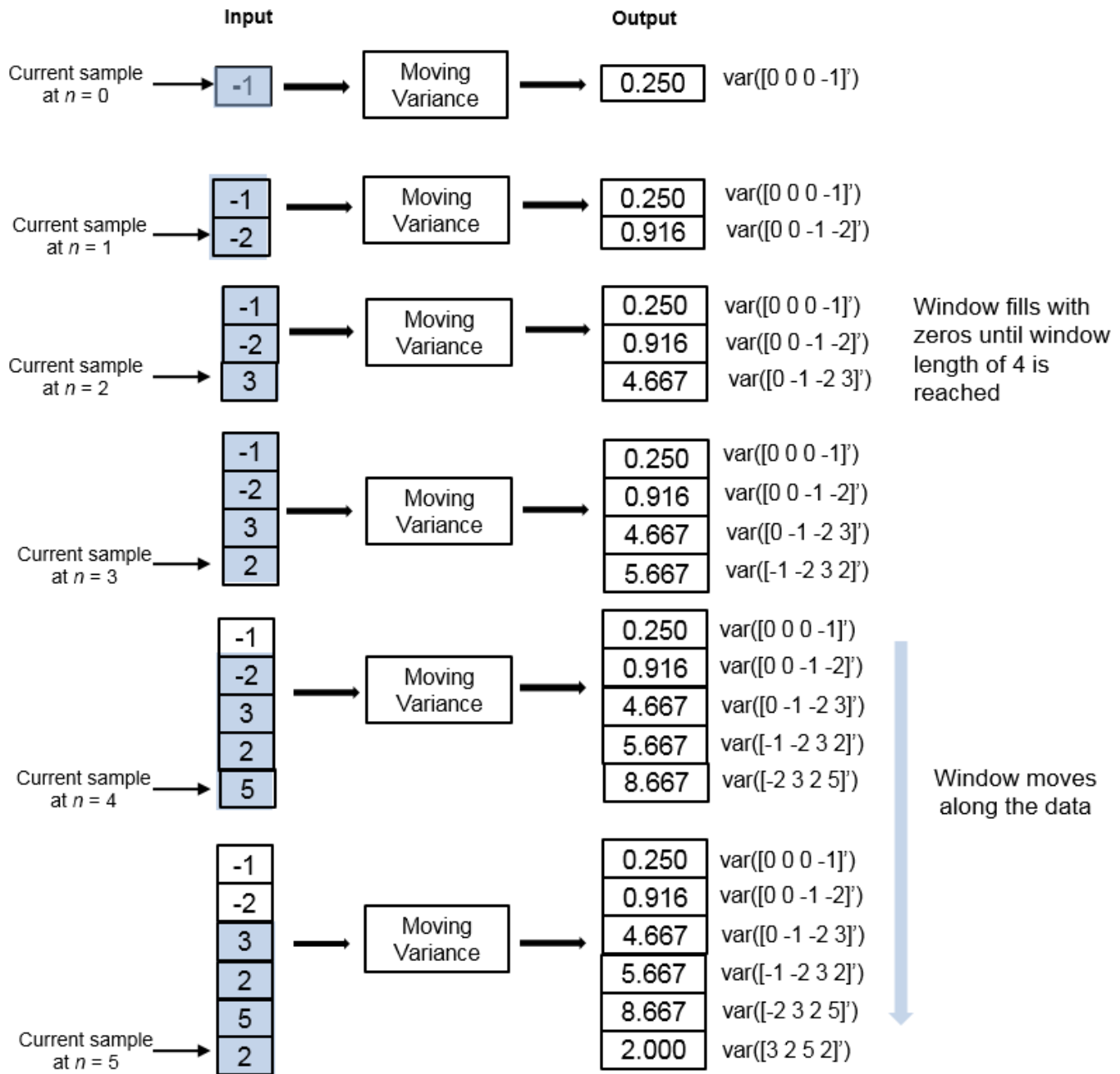
## Algorithms

### Sliding Window Method

In the sliding window method, the output at the current sample is the variance of the current sample with respect to the data in the window. To compute the first output sample, the algorithm waits until it receives the hop size number of input samples. Hop size is defined as window length - overlap length. Remaining samples in the window are considered to be zero. As an example, if the window length is 5 and the overlap length is 2, then the algorithm waits until it receives 3 samples of input to compute the first sample of the output. After generating the first output, it generates the subsequent output samples for every hop size number of input samples.

When you do not specify the window length, the algorithm chooses an infinite window length. In this mode, the output is the moving variance of the current sample with respect to all previous samples in the channel.

Consider an example of computing the moving variance of a streaming input data using the sliding window method. The algorithm uses a window length of 4 and an overlap length of 3. With each input sample that comes in, the window of length 4 moves along the data.



### Exponential Weighting Method

In the exponential weighting method, the moving variance is computed recursively using these formulas:

$$s^2_{N,\lambda} = \frac{1}{v_{N,\lambda}} \sum_{k=1}^N \lambda^{N-k} [x_k - \bar{x}_{N,\lambda}]^2$$

$$v_{N,\lambda} = \frac{2\lambda(1 - \lambda^{N-1})}{(1 - \lambda)(1 + \lambda)}$$

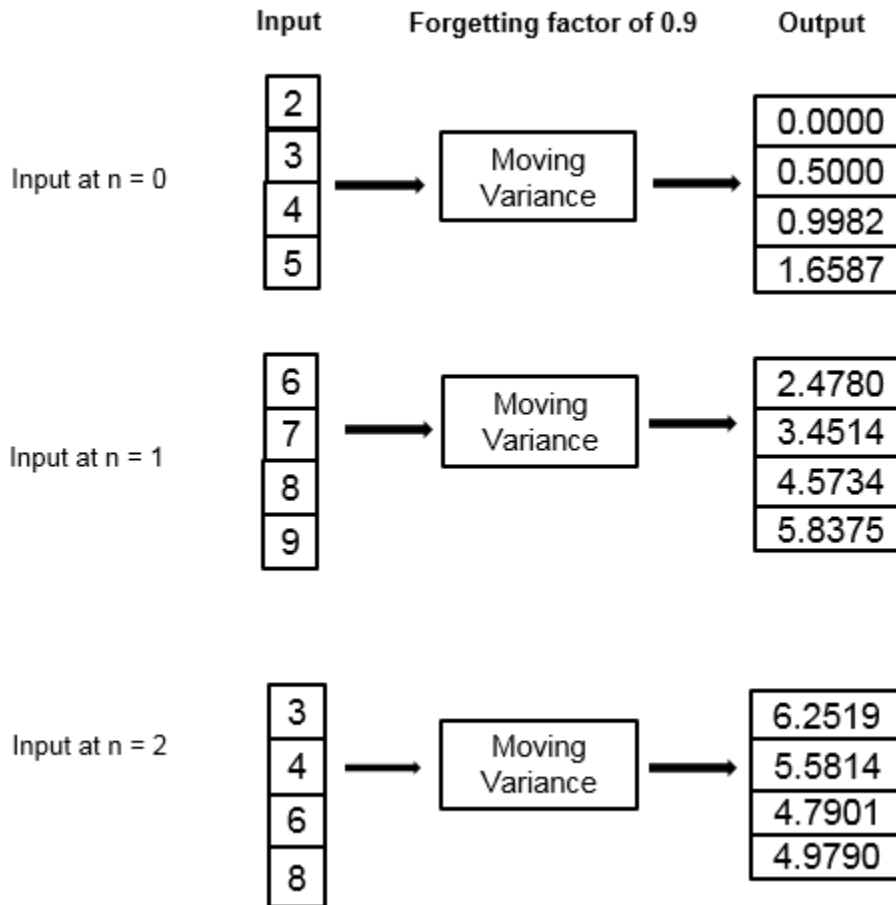
To compute the moving variance, the algorithm implements these equations recursively.

- $s_{N,\lambda}^2$  — Moving variance of the current data sample with respect to the rest of the data in the channel.
- $\bar{x}_{N,\lambda}$  — Moving average at the current sample. For details on computing the moving average, see `dsp.MovingAverage`.
- $[x_k - \bar{x}_{N,\lambda}]^2$  — Difference between each data sample and the average of the data, squared.
- $\sum_{k=1}^N \lambda^{N-k} [x_k - \bar{x}_{N,\lambda}]^2$  — Difference between each data sample and the average of the data, squared and multiplied with the forgetting factor. All the squared terms are added.
- $\frac{1}{v_{N,\lambda}}$  — Weighting factor applied to the sum.
- $\lambda$  — Forgetting factor you can specify through the `ForgettingFactor` property.

As the age of the data increases, the magnitude of the weighting factor decreases exponentially, and never reaches zero. In other words, the recent data has more influence on the current variance, than the older data.

The value of the forgetting factor determines the rate of change of the weighting factors. A forgetting factor of 0.9 gives more weight to the older data than does a forgetting factor of 0.1. A forgetting factor of 1.0 indicates infinite memory. All the past samples are given an equal weight.

Consider an example of computing the moving variance using the exponential weighting method. The forgetting factor is 0.9.



## Version History

Introduced in R2016b

### R2022b: New Overlap length parameter

Starting in R2022b, you can specify the overlap length between sliding windows using the **Overlap length** parameter.

### R2022b: Support for arbitrary input frame length

Moving Variance block supports input signals with arbitrary frame lengths when the:

- Input signal is a fixed-size signal (frame length does not change during simulation) and you select the **Allow arbitrary frame length for fixed-size input signals** parameter (if enabled).
- Input signal is a variable-size signal (frame length changes during simulation).

When this block supports an arbitrary frame length input signal, the input frame length does not have to be a multiple of the hop size.



## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

## See Also

### Blocks

Variance | Moving Average | Moving Maximum | Moving Minimum | Moving Standard Deviation | Moving RMS | Median Filter

### Objects

`dsp.MovingVariance` | `dsp.MovingAverage` | `dsp.MovingMaximum` | `dsp.MovingMinimum` | `dsp.MovingStandardDeviation` | `dsp.MovingRMS` | `dsp.MedianFilter`

### Topics

“What Are Moving Statistics?”

“Sliding Window Method and Exponential Weighting Method”

“Streaming Signal Statistics”

# Moving Standard Deviation

Moving standard deviation



**Libraries:**  
DSP System Toolbox / Statistics

## Description

The Moving Standard Deviation block computes the moving standard deviation of the input signal along each channel independently over time. The block uses either the sliding window method or the exponential weighting method to compute the moving standard deviation. In the sliding window method, a window of specified length moves over the data sample by sample, and the block computes the standard deviation over the data in the window. In the exponential weighting method, the block computes the exponentially weighted moving variance and takes the square root. For more details on these methods, see “Algorithms” on page 2-1062.

## Input/Output Ports

### Input

**x** — Data input  
vector | matrix

The block computes the moving standard deviation of the data specified at this input port. Specify real- or complex-valued multichannel inputs of the size  $m$ -by- $n$ , where  $m \geq 1$  and  $n \geq 1$ .

When the **Allow arbitrary frame length for fixed-size input signals** parameter appears and is not selected, and you input a fixed-size signal, the frame length must be a multiple of the hop size (window length – overlap length). In all other cases, the input frame length can be arbitrary.

The block accepts variable-size inputs (frame length changes during simulation). When you input a variable-size signal, the frame length of the signal can be arbitrary.

This port is unnamed until you set **Method** to Exponential weighting and select the **Specify forgetting factor from input port** parameter.

Data Types: single | double

**lambda** — Forgetting factor  
positive real scalar in the range (0,1]

The forgetting factor determines how much weight past data is given. A forgetting factor of 0.9 gives more weight to the older data than does a forgetting factor of 0.1. A forgetting factor of 1.0 indicates infinite memory – all previous samples are given an equal weight.

### Dependencies

This port appears when you set **Method** to Exponential weighting and select the **Specify forgetting factor from input port** parameter.

Data Types: `single` | `double`

## Output

**Port\_1** — Moving standard deviation output  
vector | matrix

Moving standard deviation output, returned as a vector or a matrix. The block computes the moving standard deviation based on the **Method** parameter settings using either the sliding window method or the exponential weighting method. For more details, see “Algorithms” on page 2-1062.

This table provides more details on the dimensions of the output signal.

| Input Signal         | Input Dimensions  | Output Dimensions When Allow arbitrary frame length for fixed-size input signals Appears  | Output Dimensions When Allow arbitrary frame length for fixed-size input signals Does Not Appear |
|----------------------|---|---|--|
| Fixed-size signal    | $m$ -by- $n$ , where $m$ is a multiple of the hop size (window length – overlap length)     | $(m/\text{hop size})$ -by- $n$  | $m$ -by- $n$   |
| Fixed-size signal    | $m$ -by- $n$ , where $m$ is not a multiple of the hop size (window length – overlap length) | $\text{ceil}(m/\text{hop size})$ -by- $n$ when you select <b>Allow arbitrary frame length for fixed-size input signals</b><br><br>If you do not select <b>Allow arbitrary frame length for fixed-size input signals</b> , the block errors. | $m$ -by- $n$   |
| Variable-size signal | $m$ -by- $n$  | $\text{ceil}(m/\text{hop size})$ -by- $n$   | $m$ -by- $n$   |

When the output has an upper bound size of  $\text{ceil}(m/\text{hop size})$ -by- $n$ , during simulation, the size of the first dimension varies within this bound and the size of the second dimension remains constant. For an example that shows this behavior, see “Compute Moving Standard Deviation of Noisy Square Wave Signal”.

Data Types: `single` | `double`

## Parameters

If a parameter is listed as tunable, then you can change its value during simulation.

**Method** — Moving standard deviation method  
Sliding window (default) | Exponential weighting

- **Sliding window** — A window of length **Window length** moves over the input data along each channel. For every sample the window moves by, the block computes the standard deviation over the data in the window.
- **Exponential weighting** — The block computes the exponentially weighted moving standard deviation and takes the square root. The magnitude of the weighting factors decreases exponentially as the age of the data increases, but the magnitude never reaches zero.

For more details on these methods, see “Algorithms” on page 2-1062.

**Specify window length** — Flag to specify window length  
on (default) | off

When you select this check box, the length of the sliding window is equal to the value you specify in **Window length**. When you clear this check box, the length of the sliding window is infinite. In this mode, the block computes the standard deviation of the current sample with respect to all the previous samples in the channel.

#### Dependencies

To enable this parameter, set **Method** to **Sliding window**.

**Window length** — Length of sliding window  
4 (default) | positive scalar integer

Specifies the length of the sliding window in samples.

#### Dependencies

To enable this parameter, set **Method** to **Sliding window** and select the **Specify window length** check box.

**Overlap length** — Overlap length between windows  
3 (default) | nonnegative integer

Specify the overlap length between sliding windows as a nonnegative integer. The value of overlap length varies in the range  $[0, \mathbf{Window\ length} - 1]$ .

#### Dependencies

To enable this parameter, set **Method** to **Sliding window** and select the **Specify window length** check box.

**Allow arbitrary frame length for fixed-size input signals** — Allow arbitrary frame length for fixed-size input signals

off (default) | on

Specify whether fixed-size input signals (whose size does not change during simulation) can have an arbitrary frame length, where the frame length does not have to be a multiple of the hop size. Hop size is defined as **Window length** – **Overlap length**. The block uses this parameter setting only for fixed-size input signals and ignores this parameter if the input has a variable-size.

When the input signal is a variable-size signal, the signal can have arbitrary frame length, that is, the frame length does not have to be a multiple of the hop size.

For fixed-size input signals, if you:

- Select the **Allow arbitrary frame length for fixed-size input signals** parameter, the frame length of the signal does not have to be a multiple of the hop size. If the input is not a multiple of the hop size, then the output is generally a variable-size signal. Therefore, to support arbitrary input size, the block must also support variable-size operations, which you can enable by selecting the **Allow arbitrary frame length for fixed-size input signals** parameter.
- Clear the **Allow arbitrary frame length for fixed-size input signals** parameter, the input frame length must be a multiple of the hop size.

### Dependencies

To enable this parameter, set **Method** to `Sliding window` and select the **Specify window length** check box.

**Specify forgetting factor from input port** — Flag to specify forgetting factor

off (default) | on

When you select this check box, the forgetting factor is input through the **lambda** port. When you clear this check box, the forgetting factor is specified on the block dialog through the **Forgetting factor** parameter.

### Dependencies

To enable this parameter, set **Method** to `Exponential weighting`.

**Forgetting factor** — Exponential weighting factor

0.9 (default) | positive real scalar in the range (0,1]

The forgetting factor determines how much weight past data is given. A forgetting factor of 0.9 gives more weight to the older data than does a forgetting factor of 0.1. A forgetting factor of 1.0 indicates infinite memory - all previous samples are given an equal weight.

**Tunable:** Yes

### Dependencies

To enable this parameter, set **Method** to `Exponential weighting` and clear the **Specify forgetting factor from input port** check box.

**Simulate using** — Type of simulation to run

Code generation (default) | Interpreted execution

Specify the type of simulation to run as one of the following:

- `Code generation` -- Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than `Interpreted execution`.

- **Interpreted execution** -- Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than Code generation.

## Block Characteristics

|                                 |                 |
|---------------------------------|-----------------|
| <b>Data Types</b>               | double   single |
| <b>Multidimensional Signals</b> | No              |
| <b>Variable-Size Signals</b>    | Yes             |

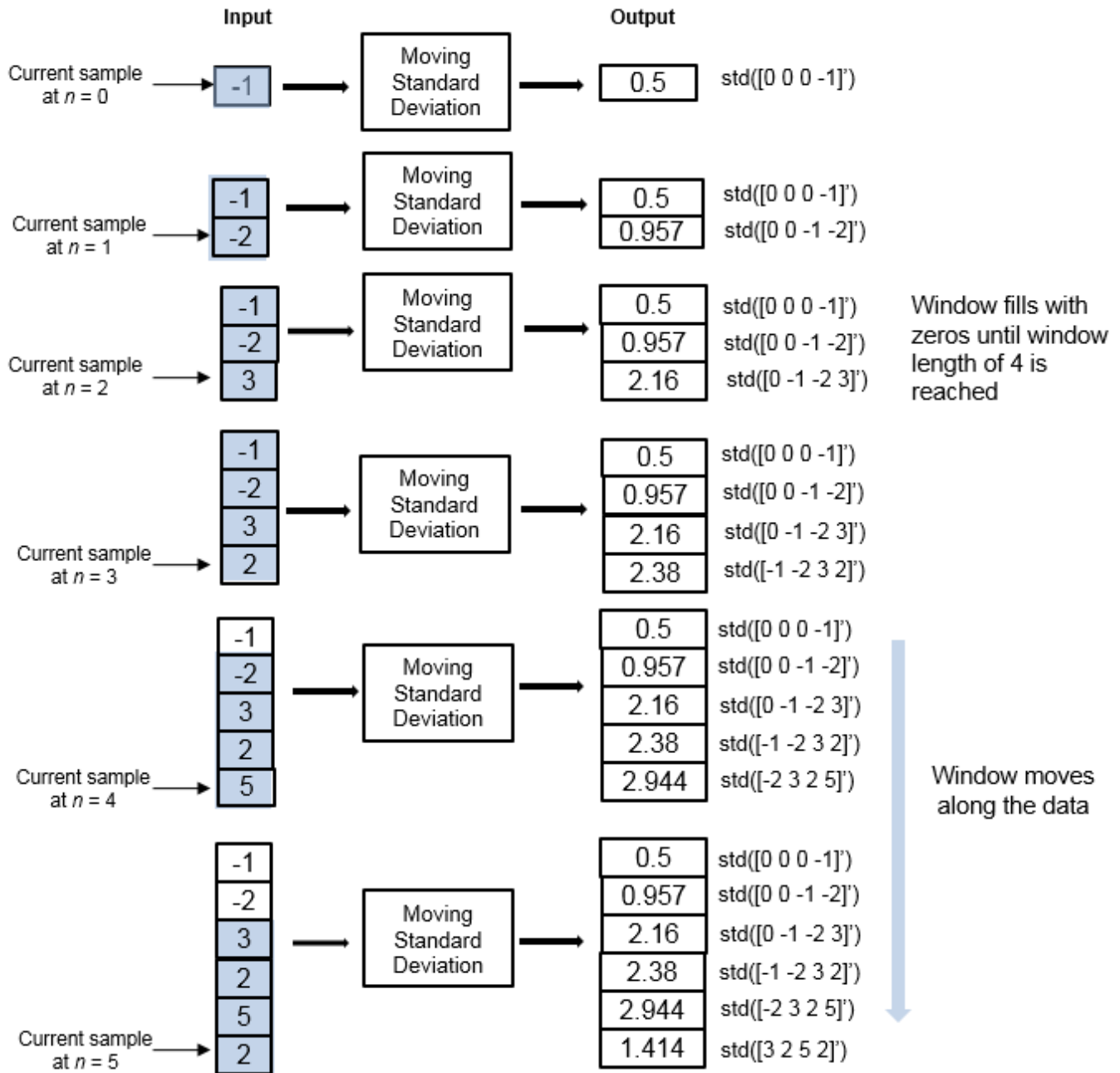
## Algorithms

### Sliding Window Method

In the sliding window method, the output at the current sample is the standard deviation of the current sample with respect to the data in the window. To compute the first output sample, the algorithm waits until it receives the hop size number of input samples. Hop size is defined as window length - overlap length. Remaining samples in the window are considered to be zero. As an example, if the window length is 5 and the overlap length is 2, then the algorithm waits until it receives 3 samples of input to compute the first sample of the output. After generating the first output, it generates the subsequent output samples for every hop size number of input samples.

When you do not specify the window length, the algorithm chooses an infinite window length. In this mode, the output is the moving standard deviation of the current sample with respect to all the previous samples in the channel.

Consider an example of computing the moving standard deviation of a streaming input data using the sliding window method. The algorithm uses a window length of 4 and an overlap length of 3. With each input sample that comes in, the window of length 4 moves along the data.



### Exponential Weighting Method

In the exponential weighting method, the moving standard deviation is computed recursively using these formulas:

$$s_{N,\lambda} = \sqrt{\frac{1}{v_{N,\lambda}} \sum_{k=1}^N \lambda^{N-k} [x_k - \bar{x}_{N,\lambda}]^2}$$

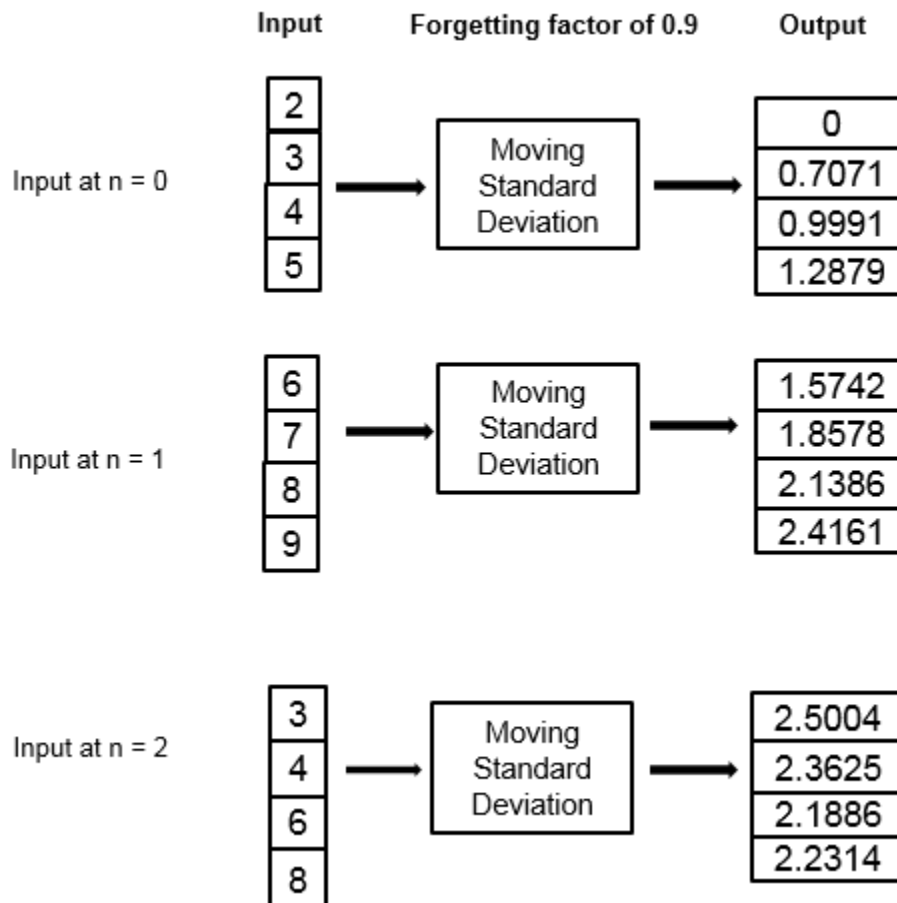
$$v_{N,\lambda} = \frac{2\lambda(1 - \lambda^{N-1})}{(1 - \lambda)(1 + \lambda)}$$

- $s_{N,\lambda}$  — Moving standard deviation of the current data sample with respect to the rest of the data.
- $[x_k - \bar{x}_{N,\lambda}]^2$  — Difference between each data sample and the average of the data, squared.
- $\sum_{k=1}^N \lambda^{N-k} [x_k - \bar{x}_{N,\lambda}]^2$  — Difference between each data sample and the average of the data, squared and multiplied with the forgetting factor. All the squared terms are added.
- $\frac{1}{v_{N,\lambda}}$  — Weighting factor applied to the sum.
- $\lambda$  — Forgetting factor.

As the age of the data increases, the magnitude of the weighting factor decreases exponentially and never reaches zero. In other words, the recent data has more influence on the current standard deviation than the older data.

The value of the forgetting factor determines the rate of change of the weighting factors. A forgetting factor of 0.9 gives more weight to the older data than does a forgetting factor of 0.1. A forgetting factor of 1.0 indicates infinite memory. All previous samples are given an equal weight.

Consider an example of computing the moving standard deviation using the exponential weighting method. The forgetting factor is 0.9.





## Version History

Introduced in R2016b

### R2022b: New Overlap length parameter

Starting in R2022b, you can specify the overlap length between sliding windows using the **Overlap length** parameter.

### R2022b: Support for arbitrary input frame length

Moving Standard Deviation block supports input signals with arbitrary frame lengths when the:

- Input signal is a fixed-size signal (frame length does not change during simulation) and you select the **Allow arbitrary frame length for fixed-size input signals** parameter (if enabled).
- Input signal is a variable-size signal (frame length changes during simulation).

When this block supports an arbitrary frame length input signal, the input frame length does not have to be a multiple of the hop size.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

## See Also

### Blocks

Standard Deviation | Moving Average | Moving Maximum | Moving Minimum | Moving Variance | Moving RMS | Median Filter

### Objects

`dsp.MovingStandardDeviation` | `dsp.MovingAverage` | `dsp.MovingMaximum` | `dsp.MovingMinimum` | `dsp.MovingVariance` | `dsp.MovingRMS` | `dsp.MedianFilter`

### Topics

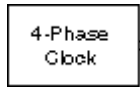
“What Are Moving Statistics?”

“Sliding Window Method and Exponential Weighting Method”

“Streaming Signal Statistics”

## Multiphase Clock

Generate multiple binary clock signals



### Libraries:

DSP System Toolbox / Signal Management / Switches and Counters  
 DSP System Toolbox / Sources

### Description

The Multiphase Clock block generates a 1-by- $N$  vector of clock signals, where you specify the integer  $N$  in the **Number of phases** parameter. Each of the  $N$  phases has the same frequency,  $f$ , specified in hertz by the **Clock frequency** parameter.

The clock signal indexed by the **Starting phase** parameter is the first to become active, at  $t=0$ . The other signals in the output vector become active in turn, each one lagging the preceding signal's activation by  $1/(Nf)$  seconds, the phase interval. The period of the output is therefore  $1/(Nf)$  seconds.

The active level can be either high (1) or low (0), as specified by the **Active level (polarity)** parameter. You specify the duration of the active level,  $D$ , as an integer between 1 and  $N-1$  using the **Number of phase intervals over which the clock is active** parameter. This value specifies the number of phase intervals that each signal remains in the active state after becoming active. The active duty cycle of the signal is  $D/N$ .

### Ports

#### Output

**Port\_1** — Vector of clock signals  
 vector

1-by- $N$  vector of clock signals, where you specify  $N$  using the **Number of phases** parameter. For more information, see “Description” on page 2-1066.

Data Types: single | double | Boolean

### Parameters

**Clock frequency (Hz)** — Frequency of all clock signals

1 (default) | positive scalar

The frequency of all output clock signals, specified as a positive scalar.

**Number of phases, N** — Number of phases in output vector

4 (default) | positive integer

The number of different phases,  $N$ , in the output vector, specified as a positive integer scalar.

**Starting phase (1 to N)** — Vector index for starting phase

1 (default) | integer from 1 to  $N$

The vector index of the output signal to first become active, specified as a scalar integer from 1 to  $N$ .

**Number of phase intervals over which clock is active (1 to N-1)** — Duration of active level for each output

3 (default) | integer from 1 to  $N-1$

The duration of the active level,  $D$ , for every output signal specified as a scalar integer from 1 to  $N-1$ . The value you specify determines the number of phase intervals that each signal remains in the active state after becoming active. The active duty cycle of the signal is  $D/N$ .

**Active level (polarity)** — Active level

High (1) (default) | Low (0)

The active level of the output, specified as High (1) or Low (0).

**Output data type** — Output data type

Logical (default) | Boolean

The output data type, specified as Logical or Boolean.

## Block Characteristics

|                                 |                           |
|---------------------------------|---------------------------|
| <b>Data Types</b>               | double   single   Boolean |
| <b>Multidimensional Signals</b> | No                        |
| <b>Variable-Size Signals</b>    | No                        |

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

## See Also

### Blocks

Clock | Counter | Pulse Generator | Event-Count Comparator

## Multiport Selector

Distribute arbitrary subsets of input rows or columns to multiple output ports



## Library

Signal Management / Indexing

dspindex

## Description

The Multiport Selector block extracts multiple subsets of rows or columns from  $M$ -by- $N$  input matrix  $u$ , and propagates each new submatrix to a distinct output port. The block treats an unoriented length- $M$  vector input as an  $M$ -by-1 matrix.

The **Indices to output** parameter is a cell array whose  $k$ th cell contains a one-dimensional indexing expression specifying the subset of input rows or columns to be propagated to the  $k$ th output port. The total number of cells in the array determines the number of output ports on the block.

When you set the **Select** parameter to **Rows**, the block uses the one-dimensional indices you specify to select matrix rows, and all elements on the chosen rows are included. When you set the **Select** parameter to **Columns**, the block uses the one-dimensional indices you specify to select matrix columns, and all elements on the chosen columns are included. A given input row or column can appear any number of times in any of the outputs, or not at all.

When an index references a nonexistent row or column of the input, the block reacts with the action you specify using the **Invalid index** parameter.

## Examples

### Example 1

Consider the following **Indices to output** cell array:

```
{4, [1:2 5], [7;8], 10:-1:6}
```

This is a four-cell array, which requires the block to generate four independent outputs (each at a distinct port). The table below shows the dimensions of these outputs when **Select** = **Rows** and the input dimension is  $M$ -by- $N$ .

| Cell | Expression | Description               | Output Size |
|------|------------|---------------------------|-------------|
| 1    | 4          | Row 4 of input            | 1-by- $N$   |
| 2    | [1:2 5]    | Rows 1, 2, and 5 of input | 3-by- $N$   |
| 3    | [7;8]      | Rows 7 and 8 of input     | 2-by- $N$   |

| Cell | Expression | Description                      | Output Size    |
|------|------------|----------------------------------|----------------|
| 4    | 10:-1:6    | Rows 10, 9, 8, 7, and 6 of input | 5-by- <i>N</i> |

## Parameters

### Select

Specify the dimension of the input to select, Rows or Columns.

### Indices to output

A cell array specifying the row- or column-subsets to propagate to each of the output ports. The number of cells in the array determines the number of output ports on the block.

### Invalid index

Specify how the block handles an invalid index value. You can select one of the following options:

- `Clip index` — Clip the index to the nearest valid value, and do not issue an alert.

For example, if the block receives a 64-by-4 input and the **Select** parameter is set to Rows, the block clips an index of 72 to 64. For the same input, if the **Select** parameter is set to Columns, the block clips an index of 72 to 4. In both cases, the block clips an index of -2 to 1.

- `Clip and warn` — Clip the index to the nearest valid value and display a warning message at the MATLAB command line.
- `Generate error` — Display an error dialog box and terminate the simulation.

## Supported Data Types

| Port    | Supported Data Types   |
|---------|--|
| Input   | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed and unsigned)</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> <li>• Enumerated</li> </ul> |
| Outputs | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed and unsigned)</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> <li>• Enumerated</li> </ul> |

## See Also

Permute Matrix

DSP System Toolbox

|                   |                    |
|-------------------|--------------------|
| Selector          | Simulink           |
| Submatrix         | DSP System Toolbox |
| Variable Selector | DSP System Toolbox |

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Generated code relies on the memcopy or memset function (`string.h`) under certain conditions.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

### HDL Architecture

This block has one default HDL architecture.

### HDL Block Properties

|                                  |  |
|----------------------------------|--|
| <b>ConstrainedOutputPipeline</b> | Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder). |
| <b>InputPipeline</b>             | Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).                 |
| <b>OutputPipeline</b>            | Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).               |

### Complex Data Support

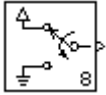
This block supports code generation for complex signals.

### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

# N-Sample Enable

Output ones or zeros for specified number of sample times



## Libraries:

DSP System Toolbox / Sources

DSP System Toolbox / Signal Management / Switches and Counters

## Description

The N-Sample Enable block outputs the inactive value (0 or 1, whichever is not selected in the **Active level** parameter) during the first  $N$  sample times, where  $N$  is the **Trigger count** value. Beginning with output sample  $N+1$ , the block outputs the active value (1 or 0, whichever you select in the **Active level** parameter) until a reset event occurs or the simulation terminates.

The output of the block is always a scalar.

The **Reset input** check box enables the **Rst** input port. At any time during the count, a trigger event at the input port resets the counter to its initial state. You specify the type of trigger event using the **Trigger type** parameter. This block supports triggered subsystems when you select the **Reset input** check box.

## Ports

### Input

**Rst** — Reset signal  
scalar

Reset input signal, specified as a scalar. At any time during the count, a trigger event at the input port resets the counter to its initial state.

---

**Tip** The N-Sample Enable block supports triggered subsystems when you select the **Reset input** check box to enable the **Rst** input port.

---

### Dependencies

To enable this input port, select the **Reset input** check box.

Data Types: Boolean

### Output

**Port\_1** — Output signal  
scalar

Scalar output containing the inactive value (0 or 1, whichever is not selected in the **Active level** parameter) during the first  $N$  sample times, where  $N$  is the **Trigger count** value. Beginning with output sample  $N+1$ , the block outputs the active value (1 or 0, whichever you select in the **Active level** parameter) until a reset event occurs or the simulation terminates.

Data Types: Boolean

## Parameters

**Trigger count, N** — Number of samples to output the active value

8 (default) | scalar integer

Specify the number of samples for which the block outputs the active value as a scalar integer, greater than or equal to zero.

**Tunable:** Yes

**Active level** — Active level

Low (0) (default) | High (1)

Specify the value to output after the first *N* sample times as 0 or 1.

**Tunable:** Yes

**Reset input** — Enable reset input port

off (default) | on

To enable the reset (**Rst**) input port, select this check box. When you clear this check box, the **Rst** input port is disabled.

---

**Tip** When you select the **Reset input** check box, the N-Sample Enable block supports triggered subsystems.

---

**Trigger type** — Type of triggering event

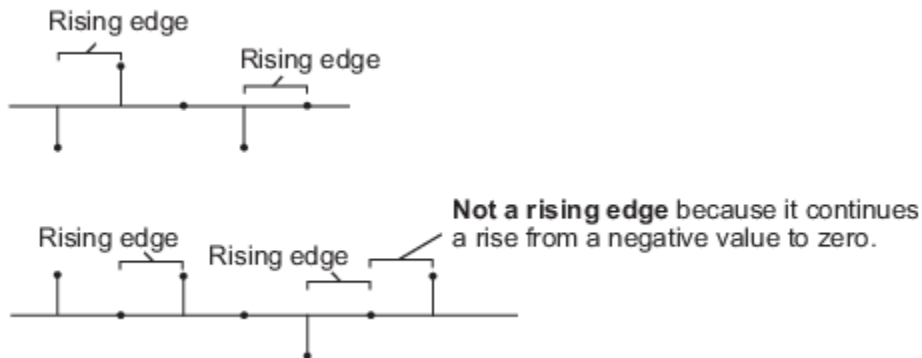
Rising edge (default) | Falling edge | Either edge | Non-zero sample

Select type of event that triggers a reset when the **Rst** port is enabled.

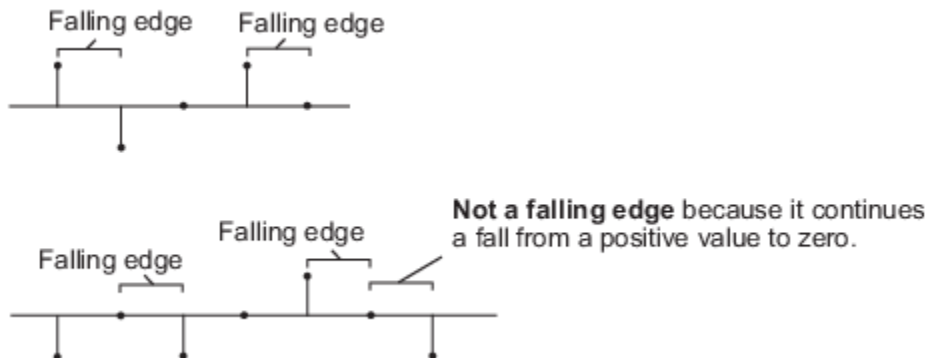
You can specify the triggering event as:

- **Rising edge** — Triggers a reset operation when the **Rst** input does one of the following:
  - Rises from a negative value to a positive value or zero
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure)





- **Falling edge** — Triggers a reset operation when the **Rst** input does one of the following:
  - Falls from a positive value to a negative value or zero
  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero (see the following figure)



- **Either edge** — Triggers a reset operation when the **Rst** input is a Rising edge or Falling edge.
- **Non-zero sample** — Triggers a reset operation at each sample time that the **Rst** input is not zero.

### Dependencies

To enable this parameter, select the **Reset input** check box.

### Sample time — Output sample period

1 (default) | positive scalar

Specify the sample period,  $T_s$ , for the block's counter as a positive finite scalar. The block switches from the active value to the inactive value at  $t=T_s(N+1)$ .

### Output data type — Output data type

Logical (default) | Boolean

Specify the output data type as Logical or Boolean.

## Block Characteristics

|                                 |                  |
|---------------------------------|------------------|
| <b>Data Types</b>               | Boolean   double |
| <b>Direct Feedthrough</b>       | no               |
| <b>Multidimensional Signals</b> | no               |
| <b>Variable-Size Signals</b>    | no               |
| <b>Zero-Crossing Detection</b>  | no               |

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

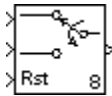
## See Also

### Blocks

Counter | N-Sample Switch

# N-Sample Switch

Switch between two inputs after specified number of sample periods



## Library

Signal Management / Switches and Counters

dspswit3

## Description

The N-Sample Switch block outputs the signal connected to the top input port during the first  $N$  sample times after the simulation begins or the block is reset, where you specify  $N$  in the **Switch count** parameter. Beginning with output sample  $N+1$ , the block outputs the signal connected to the bottom input until the next reset event or the end of the simulation.

You specify the sample period of the output in the **Sample time** parameter (that is, the output sample period is not inherited from the sample period of either input). The block applies a zero-order hold at the input ports, so the value the block reads from a given port between input sample times is the value of the most recent input to that port.

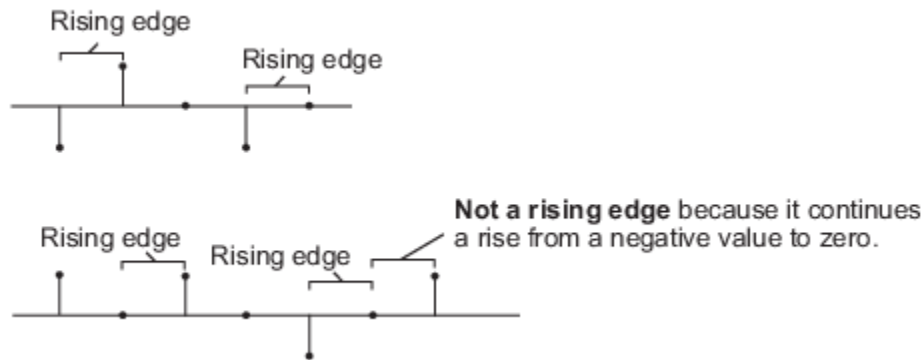
Both inputs must have the same dimension, except in the following two cases:

- When one input is a scalar, the block expands the scalar input to match the size of the other input.
- When one input is an unoriented vector and the other input is a row or column vector with the same number of elements, the block reshapes the unoriented vector to match the dimension of the other input.

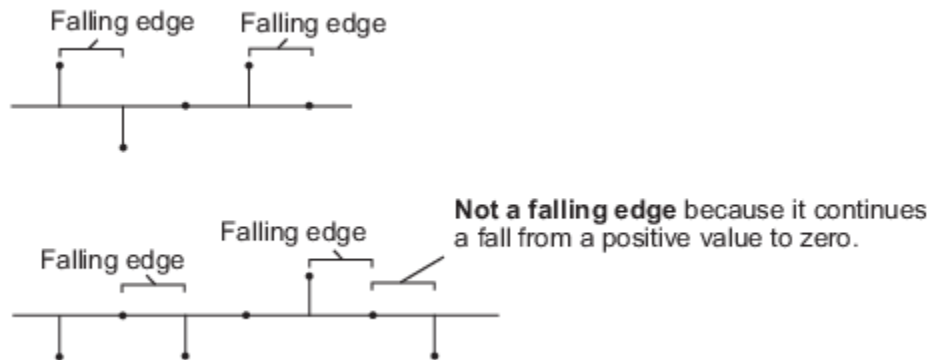
The **Reset input** check box enables the Rst input port. At any time during the count, a trigger event at the Rst port resets the counter to zero. The reset sample time must be a positive integer multiple of the input sample time. This block supports triggered subsystems when you select the **Reset input** check box.

You specify the triggering event in the **Trigger type** pop-up menu, and can be one of the following:

- **Rising edge** — Triggers a reset operation when the Rst input does one of the following:
  - Rises from a negative value to a positive value or zero
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure)



- **Falling edge** — Triggers a reset operation when the Rst input does one of the following:
  - Falls from a positive value to a negative value or zero
  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero (see the following figure)



- **Either edge** — Triggers a reset operation when the Rst input is a Rising edge or Falling edge (as described above).
- **Non-zero sample** — Triggers a reset operation at each sample time that the Rst input is not zero.

## Parameters

### Switch count

The number of sample periods,  $N$ , for which the output is connected to the top input before switching to the bottom input. Tunable (Simulink).

### Reset input

Enables the Rst input port when selected. The rate of the reset signal must be a positive integer multiple of the rate of the data signal input.

### Trigger type

The type of event at the Rst port that resets the block's counter. This parameter is enabled when you select **Reset input**. Tunable (Simulink).

### Sample time

The sample period,  $T_s$ , for the block's counter. The block switches inputs at  $t=T_s*(N+1)$ .

## Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point (signed and unsigned)
- Boolean — The block accepts Boolean inputs to the Rst port, which is enabled when you set the **Reset input** parameter.
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

## See Also

Counter

DSP System Toolbox

N-Sample Enable

DSP System Toolbox

## Version History

**Introduced before R2006a**

## Extended Capabilities

### C/C++ Code Generation

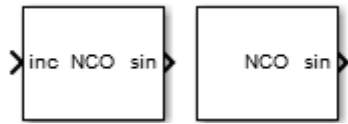
Generate C and C++ code using Simulink® Coder™.

### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

## NCO

Generate real or complex sinusoidal signals



### Libraries:

DSP System Toolbox / Signal Operations  
 DSP System Toolbox / Sources

## Description

The NCO block generates a multichannel real or complex sinusoidal signal, with independent frequency and phase in each output channel. The amplitude of the created signal is always 1. The NCO block supports real inputs only. All outputs are real except for the output signal in **Complex exponential** mode. For more information on how the block computes the output, see “Algorithms” on page 2-1084.

To produce a multichannel output, specify a vector quantity for the **Phase increment** and **Phase offset** parameters. Both parameters must have the same length, which defines the number of output channels. Each element of each vector is applied to a different output channel.

## Ports

### Input

**inc** — Phase increment  
 scalar | vector

Phase increment signal, specified as a real-valued scalar or vector. The input must have an integer data type, or a fixed-point data type with zero fraction length. The dimensions of the phase increment signal depend on how you choose to specify the **Phase offset** parameter:

- When you specify the **Phase offset** on the block dialog box, the **Phase increment** must be a scalar or a vector with the same length as the **Phase offset** value. The block applies each element of the vector to a different channel, and therefore the vector length defines the number of output channels.
- When you specify the **Phase offset** via the an input port, the **offset** port treats each column of the input as an independent channel. The **Phase increment** length must equal the number of columns in the input to the **offset** port.

### Dependencies

To enable this port, set **Phase increment source** to **Input** port.

Data Types: `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fixed point`

**offset** — Phase offset  
 scalar | vector | matrix

Phase offset signal, specified as a real-valued scalar, vector, or a full matrix. The input must have an integer data type, or a fixed-point data type with zero fraction length. The block treats each column of

the input to the offset port as an independent channel. The number of channels in the phase offset must match the number of channels in the data input. For each frame of the input, the block can apply different phase offsets to each sample and channel.

### Dependencies

To enable this port, set **Phase offset source** to `Input port`.

Data Types: `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fixed point`

### Output

**sin** — Sine output

`scalar` | `vector` | `matrix`

Sinusoidal output signal specified as a scalar, vector, or matrix. You can specify the data type of the signal using the **Output** data type parameter.

### Dependencies

To enable this port, set **Output signal** to `Sine` or `Sine and cosine`.

Data Types: `single` | `double` | `fixed point`

**cos** — Cosine output

`scalar` | `vector` | `matrix`

Cosinusoidal output signal specified as a scalar, vector, or matrix. You can specify the data type of the signal using the **Output Data Type** parameter.

### Dependencies

To enable this port, set **Output signal** to `Cosine` or `Sine and cosine`.

Data Types: `single` | `double` | `fixed point`

**exp** — Complex exponential

`scalar` | `vector` | `matrix`

Complex exponential output, specified as a scalar, vector, or matrix. You can specify the data type of the signal using the **Output Data Type** parameter.

### Dependencies

To enable this port, set **Output signal** to `Complex exponential`.

Data Types: `single` | `double` | `fixed point`

Complex Number Support: Yes

**Qerr** — Phase quantization error

`scalar` | `vector` | `matrix`

Phase quantization error specified as a scalar, vector, or matrix.

### Dependencies

To enable this port, select the **Show phase quantization error port** check box.

Data Types: `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fixed point`

## Parameters

### Main

#### Phase adder parameters

**Phase increment source** — Source of phase increment value

`Specify via dialog` | `Input port`

Choose how you specify the phase increment. The phase increment can come from an input port or from the dialog box parameter.

- If you select `Input port`, the **inc** input port appears on the block icon.
- If you select `Specify via dialog`, the **Phase increment** parameter appears.

#### Dependencies

When the block comes from the Signal Operations library, the default value of the **Phase increment Source** parameter is `Input port`.

When the block comes from the Sources library, the default value of the **Phase increment Source** parameter is `Specify via dialog`.

**Phase increment** — Phase increment value

100 (default) | scalar | vector

Specify the phase increment as an integer-valued scalar or vector. Only integer data types, including fixed-point data types with zero fraction length, are allowed. The dimensions of the phase increment depend on those of the phase offset:

- When you specify the phase offset on the block dialog box, the phase increment must be a scalar or a vector with the same length as the phase offset. The block applies each element of the vector to a different channel, and therefore the vector length defines the number of output channels.
- When you specify the phase offset via an input port, the offset port treats each column of the input as an independent channel. The phase increment length must equal the number of columns in the input to the offset port.

#### Dependencies

To enable this parameter, set **Phase increment source** to `Specify via dialog`.

**Phase offset source** — Source of phase offset

`Specify via dialog (default)` | `Input port`

Choose how you specify the phase offset. The phase offset can come from an input port or from the dialog box.

- If you select `Input port`, the **offset** port appears on the block icon.
- If you select `Specify via dialog`, the **Phase offset** parameter appears.

**Phase offset** — Phase offset value



0 (default) | scalar | vector

Specify the phase offset as an integer-valued scalar or vector. Only integer data types, including fixed-point data types with zero fraction length, are allowed. When you specify the phase offset using this dialog box parameter, it must be a scalar or vector with the same length as the phase increment. Scalars are expanded to a vector with the same length as the phase increment. Each element of the phase offset vector is applied to a different channel of the input, and therefore the vector length defines the number of output channels.

#### Dependencies

To enable this parameter, set **Phase offset source** to *Specify via dialog*.

#### Add internal dither — Add internal dithering

on (default) | off

Select to add internal dithering to the NCO algorithm. Dithering is added using the PN Sequence Generator from the Communications Toolbox™ product.

#### Number of dither bits — Number of dither bits

4 (default) | positive integer

Specify the number of dither bits as a positive integer.

#### Dependencies

To enable this port, select the **Add internal dither** check box.

#### Quantize phase — Enable quantization of accumulated phase

on (default) | off

To enable quantization of the accumulated phase, select this check box.

#### Number of quantized accumulator bits — Number of quantized accumulator bits

12 (default) | integer scalar

Specify the number of quantized accumulator bits as a scalar integer greater than one, and less than the accumulator word length. This value determines the number of entries in the lookup table.

#### Dependencies

To enable this port, select the **Quantize phase** check box.

#### Show phase quantization error port — Output quantization error

off (default) | on

Select to output the phase quantization error. When you select this check box, the **Qerr** port appears on the block icon.

#### Dependencies

To enable this parameter, select the **Quantize phase** check box.

**Output Parameters****Output signal** — Output signal

Sine (default) | Cosine | Complex exponential | Sine and cosine

Choose whether the block outputs a Sine, Cosine, Complex exponential, or both Sine and cosine signals. If you select Sine and cosine, the two signals output on different ports.

**Sample time** — Output sample period

1 (default) | positive scalar

When the block is acting as a source, specify the sample time in seconds as a positive scalar.

**Dependencies**

To enable this parameter, both **Phase increment source** and **Phase offset source** must be set to **Specify via dialog**. When either the phase increment or phase offset come in via a block input port, the sample time is inherited and this parameter is not visible.

**Samples per frame** — Samples per frame

1 (default) | positive integer

Specify the number of samples per frame as a positive integer. When the value is greater than one, the phase increment and phase offset can vary from channel to channel and from frame to frame, but they are constant along each channel in a given frame.

When the phase offset input port exists, it has the same frame status as any output port present. When the phase increment input port exists, it does not support frames.

**Dependencies**

To enable this parameter set **Phase increment source** and/or **Phase offset source** to **Specify via dialog**.

**Data Types****Rounding mode** — Rounding method

Floor (default)

This parameter is read-only.

When the input is fixed point, the NCO block always uses the rounding mode Floor.

**Overflow mode** — Overflow method

Wrap (default)

This parameter is read-only.

When the input is fixed point, the NCO block always uses the overflow mode Wrap.

**Accumulator Word Length** — Accumulator word length

16 (default) | integer from 2 to 128

Specify a **Word length** for the **Accumulator** as a positive integer from 2 to 128. The **Data Type** is always `Binary point scaling`, and the **Fraction length** is always 0.

**Output Data Type** — Output data type  
`double (default) | single | Binary point scaling`

Specify a **Data Type** for the block **Output**.

- Choose `double` or `single` for a floating-point implementation.
- When you select `Binary point scaling`, you can enter the word length and the fraction length of the output, in bits.

---

**Note** The lookup table for this block is constructed from double-precision floating-point values. Thus, the maximum amount of precision you can achieve in your output is 53 bits. Setting the word length of the **Output** data type to values greater than 53 bits does not improve the precision of your output.

---

**Output Word Length** — Output word length  
`16 (default) | integer from 2 to 128`

Specify a **Word length** for the block **Output** as a positive integer from 2 to 128.

#### Dependencies

To enable this parameter, set the **Data Type** for the **Output** to `Binary point scaling`.

**Output Fraction Length** — Output fraction length  
`14 (default) | integer`

Specify a **Fraction length** for the block **Output** as a scalar integer.

#### Dependencies

To enable this parameter, set the **Data Type** for the **Output** to `Binary point scaling`.

#### NCO Characterization

The **NCO Characterization** pane provides you with read-only details on the NCO signal currently being implemented by the block:

**Number of data points for lookup table** — Number of points in lookup table  
`no default`

The lookup table is implemented as a quarter-wave sine table. The number of lookup table data points is defined by

$$2^{\text{number of quantized accumulator bits}-2} + 1$$

Example: 1025

**Quarter wave sine lookup table size** — Size of quarter wave sine lookup table  
`no default`

The quarter wave sine lookup table size is defined by

$$\frac{(\text{number of data points for lookup table}) \cdot (\text{output word length})}{8} \text{ bytes}$$

Example: 2050 bytes

**Theoretical spurious free dynamic range** — Spurious free dynamic range  
no default

The spurious free dynamic range (SFDR) is calculated as follows for a lookup table with  $2^P$  entries:

$$SFDR = (6P) \text{ dB} \quad \text{without dither}$$

$$SFDR = (6P + 12) \text{ dB} \quad \text{with dither}$$

Example: 84 dBc

**Frequency resolution** — Frequency resolution  
no default

The frequency resolution is the smallest possible incremental change in frequency and is defined by:

$$\Delta f = \frac{1}{T_s \cdot 2^N} \text{ Hz}$$

Example: 15.2588 uHz

### Dependencies

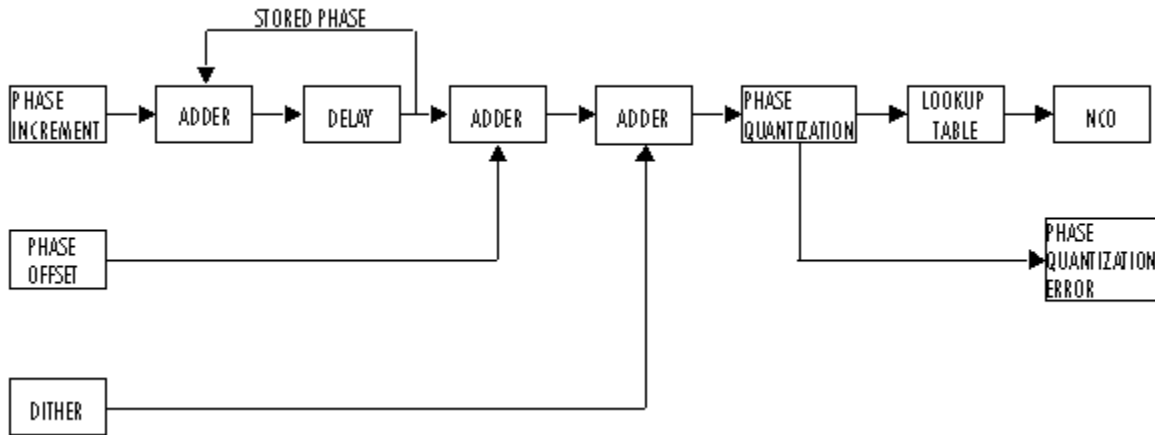
The frequency resolution only appears when you set **Phase increment source** and **Phase offset source** to Specify via dialog.

## Block Characteristics

|                                 |                       |
|---------------------------------|-----------------------|
| <b>Data Types</b>               | fixed point   integer |
| <b>Direct Feedthrough</b>       | no                    |
| <b>Multidimensional Signals</b> | no                    |
| <b>Variable-Size Signals</b>    | no                    |
| <b>Zero-Crossing Detection</b>  | no                    |

## Algorithms

The block implements the algorithm as shown in the following diagram:



The implementation of a numerically controlled oscillator (NCO) has two distinct parts. First, a phase accumulator accumulates the phase increment and adds in the phase offset. In this stage, an optional internal dither signal can also be added. The NCO output is then calculated by quantizing the results of the phase accumulator section and using them to select values from a lookup table. Since the lookup table contains a finite set of entries, in its normal mode of operation, the NCO block allows the adder's numeric values to overflow and wrap around. The Fixed-Point infrastructure then causes overflow warnings to appear on the command line. This overflow is of no consequence.

Given a desired output frequency  $F_0$ , calculate the value of the **Phase increment** block parameter with

$$phaseincrement = \left( \frac{F_0 \cdot 2^N}{F_s} \right)$$

where  $N$  is the accumulator word length and

$$F_s = \frac{1}{T_s} = \frac{1}{sampletime}$$

The frequency resolution of an NCO is defined by

$$\Delta f = \frac{1}{T_s \cdot 2^N} \text{Hz}$$

Given a desired phase offset (in radians), calculate the **Phase offset** block parameter with

$$phaseoffset = \frac{2^N \cdot desiredphaseoffset}{2\pi}$$

The spurious free dynamic range (SFDR) is estimated as follows for a lookup table with  $2^P$  entries, where  $P$  is the number of quantized accumulator bits:

$$SFDR = (6P) \text{ dB} \quad \text{without dither}$$

$$SFDR = (6P + 12) \text{ dB} \quad \text{with dither}$$

The NCO block uses a quarter-wave lookup table technique that stores table values from 0 to  $\pi/2$ . The block calculates other values on demand using the accumulator data type, then casts them into the

output data type. This can lead to quantization effects at the range limits of a given data type. For example, consider a case where you would expect the value of the sine wave to be  $-1$  at  $\pi$ . Because the lookup table value at that point must be calculated, the block might not yield exactly  $-1$ , depending on the precision of the accumulator and output data types.

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Generated code relies on the `memcpy` or `memset` function (`string.h`) under certain conditions.

### HDL Code Generation

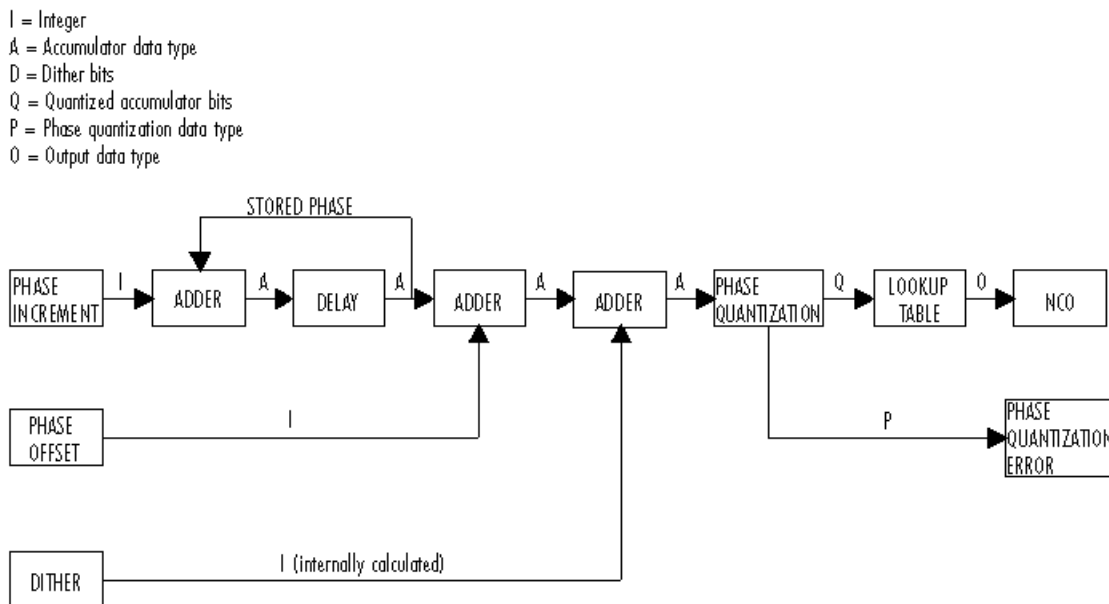
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL support for the NCO block will be removed in a future release. Use the NCO block instead. The DSP HDL Toolbox block provides an HDL-optimized algorithm with hardware-friendly control signals and simulates the latency of the HDL algorithm in Simulink.

### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

The following diagram shows the data types used within the NCO block.



- You can set the accumulator and output data types in the block dialog box as discussed in “Data Types” on page 2-1082.

**Note** The lookup table for this block is constructed from double-precision floating-point values. Thus, the maximum amount of precision you can achieve in your output is 53 bits. Setting the word length of the **Output** data type to values greater than 53 bits does not improve the precision of your output.

- The phase increment and phase offset inputs must be integers or fixed-point data types with zero fraction length.
- You specify the number of quantized accumulator bits in the **Number of quantized accumulator bits** parameter.
- The phase quantization error word length is equal to the accumulator word length minus the number of quantized accumulator bits, and the fraction length is zero.

## See Also

### Blocks

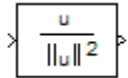
PN Sequence Generator | Sine Wave | Digital Down-Converter | Digital Up-Converter

### Objects

`dsp.NCO`

## Normalization

Perform vector normalization along rows, columns, or specified dimension



## Library

Math Functions / Math Operations

dspmathops

## Description

The Normalization block independently normalizes each row, column, or vector of the specified dimension of the input. The block accepts both fixed- and floating-point signals in the squared 2-norm mode, but only floating-point signals in the 2-norm mode. The output always has the same dimensions as the input.

This block treats an arbitrarily dimensioned input  $U$  as a collection of vectors oriented along the specified dimension. The block normalizes these vectors by either their norm or the square of their norm.

For example, consider a 3-dimensional input  $U(i,j,k)$  and assume that you want to normalize along the second dimension. First, define the 2-dimensional intermediate quantity  $V(i,k)$  such that each element of  $V$  is the norm of one of the vectors in  $U$ :

$$V(i, k) = \left( \sum_{j=1}^J U^2(i, j, k) \right)^{1/2}$$

Given  $V$ , the output of the block  $Y(i, j, k)$  in 2-norm mode is

$$Y(i, j, k) = \frac{U(i, j, k)}{V(i, k) + b}$$

In squared 2-norm mode, the block output is

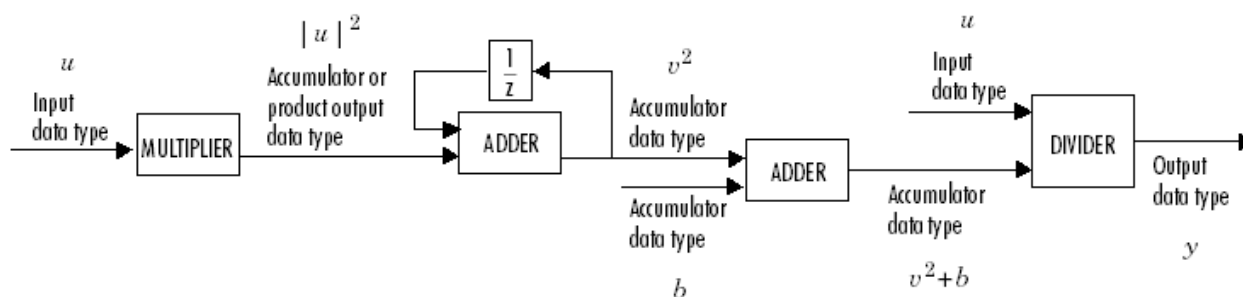
$$Y(i, j, k) = \frac{U(i, j, k)}{V(i, k)^2 + b}$$

The normalization bias,  $b$ , is typically chosen to be a small positive constant (for example, 1e-10) that prevents potential division by zero.

### Fixed-Point Data Types

The following diagram shows the data types used within the Normalization block for fixed-point signals (squared 2-norm mode only).





The output of the multiplier is in the product output data type when the input is real. When the input is complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication performed, see “Multiplication Data Types”. You can set the accumulator, product output, and output data types in the block dialog as discussed in “Parameters” on page 2-1089.

## Examples

See “Zero Algorithmic Delay” in the *DSP System Toolbox User's Guide* for an example.

## Parameters

### Main Tab

#### Norm

Specify the type of normalization to perform, 2-norm or Squared 2-norm. 2-norm mode supports floating-point signals only. Squared 2-norm supports both fixed-point and floating-point signals.

#### Normalization bias

Specify the real value  $b$  to be added in the denominator to avoid division by zero. Tunable (Simulink).

#### Normalize over

Specify whether to normalize along rows, columns, or the dimension specified in the **Dimension** parameter.

#### Dimension

Specify the one-based value of the dimension over which to normalize. The value of this parameter cannot exceed the number of dimensions in the input signal. This parameter is only visible if Specified dimension is selected for the **Normalize over** parameter.

### Data Types Tab

---

**Note** The parameters on this pane are only applicable to fixed-point signals when the block is in squared 2-norm mode. See “Fixed-Point Data Types” on page 2-1088 for a diagram of how the product output, accumulator, and output data types are used in this case.

---

### Rounding mode

Specify the rounding mode for fixed-point operations as one of the following:

- Floor
- Ceiling
- Convergent
- Nearest
- Round
- Simplest
- Zero

For more details, see rounding mode.


### Saturate on integer overflow

When you select this parameter, the block saturates the result of its fixed-point operation. When you clear this parameter, the block wraps the result of its fixed-point operation. For details on saturate and wrap, see overflow mode for fixed-point operations.

### Product output

Specify the product output data type. See “Fixed-Point Data Types” on page 2-1088 and “Multiplication Data Types” for illustrations depicting the use of the product output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as input`
- An expression that evaluates to a valid data type, for example, `fixdt([], 16, 0)`


Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Product output** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in *Simulink User's Guide* (Simulink) for more information.

### Accumulator

Specify the accumulator data type. See “Fixed-Point Data Types” on page 2-1088 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Same as product output`
- A rule that inherits a data type, for example, `Inherit: Same as input`
- An expression that evaluates to a valid data type, for example, `fixdt([], 16, 0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Accumulator** parameter.


See “Specify Data Types Using Data Type Assistant” (Simulink) in *Simulink User's Guide* (Simulink) for more information.

### Output

Specify the output data type. See “Fixed-Point Data Types” on page 2-1088 for illustrations depicting the use of the output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as product output`
- A rule that inherits a data type, for example, `Inherit: Same as input`

- A rule that inherits a data type, for example, `Inherit: Same as accumulator`
- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Output** parameter.

See “Control Data Types of Signals” (Simulink) in *Simulink User's Guide* (Simulink) for more information.

### Output Minimum

Specify the minimum value that the block should output. The default value is [] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Specify Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

### Output Maximum

Specify the maximum value that the block should output. The default value is [] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Specify Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

### Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

### Product Output Data Type Assistant Parameters

#### Mode

Select how you would like to specify the data type properties of the **Product output** data type. You can choose:

- **Inherit** — Lets you specify a rule for inheriting a data type, for example, `Inherit: Inherit via internal rule`
- **Fixed point** — Lets you specify the fixed-point attributes of the data type.
- **Expression** — Lets you specify an expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

#### Signedness

Specify the **Signedness** for the **Product output** data type.

#### Scaling

Specify the **Scaling** for the **Product output** data type.

For more information see “Scaling” in the DSP System Toolbox User's Guide.

**Word length**

Specify the **Word length** for the **Product output** data type.

**Fraction length**

Specify the **Fraction length** for the **Product output** data type.

**Data type override**

Specify the data type override mode. You can select one of the following options:

- **Inherit** — Inherits the data type override setting specified for the model.
- **Off** — Ignores the data type override setting specified for the model and uses the fixed-point data type you specify

This parameter appears only when you set the **Mode** parameter to **Built in** or **Fixed Point**. For more information, see “Specify Data Types Using Data Type Assistant” (Simulink).

**Accumulator Data Type Assistant Parameters****Mode**

Select how you would like to specify the data type properties of the **Accumulator** data type. You can choose:

- **Inherit** — Lets you specify a rule for inheriting a data type, for example, **Inherit via internal rule**
- **Fixed point** — Lets you specify the fixed-point attributes of the data type.
- **Expression** — Lets you specify an expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

**Signedness**

Specify the **Signedness** for the **Accumulator** data type.

**Scaling**

Specify the **Scaling** for the **Accumulator** data type.

For more information see “Scaling” in the DSP System Toolbox User's Guide.

**Word length**

Specify the **Word length** for the **Accumulator** data type.

**Fraction length**

Specify the **Fraction length** for the **Accumulator** data type.

**Data type override**

Specify the data type override mode. You can select one of the following options:

- **Inherit** — Inherits the data type override setting specified for the model.
- **Off** — Ignores the data type override setting specified for the model and uses the fixed-point data type you specify

This parameter appears only when you set the **Mode** parameter to **Built in** or **Fixed Point**. For more information, see “Specify Data Types Using Data Type Assistant” (Simulink).

**Output Data Type Assistant Parameters****Mode**

Select how you would like to specify the data type properties of the **Output** data type. You can choose:

- **Inherit** — Lets you specify a rule for inheriting a data type, for example, `Inherit: Inherit via internal rule`
- **Fixed point** — Lets you specify the fixed-point attributes of the data type.
- **Expression** — Lets you specify an expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

**Signedness**

Specify the **Signedness** for the **Output** data type.

**Scaling**

Specify the **Scaling** for the **Output** data type.

For more information see “Scaling” in the DSP System Toolbox User's Guide.

**Word length**

Specify the **Word length** for the **Output** data type.

**Fraction length**

Specify the **Fraction length** for the **Output** data type.

**Data type override**

Specify the data type override mode. You can select one of the following options:

- **Inherit** — Inherits the data type override setting specified for the model.
- **Off** — Ignores the data type override setting specified for the model and uses the fixed-point data type you specify

This parameter appears only when you set the **Mode** parameter to **Built in** or **Fixed Point**. For more information, see “Specify Data Types Using Data Type Assistant” (Simulink).

## Supported Data Types

| Port   | Supported Data Types  |
|--------|---|
| Input  | <ul style="list-style-type: none"><li>• Double-precision floating point</li><li>• Single-precision floating point</li><li>• Fixed point (signed and unsigned)</li><li>• 8-, 16-, and 32-bit signed integers</li><li>• 8-, 16-, and 32-bit unsigned integers</li></ul> |
| Output | <ul style="list-style-type: none"><li>• Double-precision floating point</li><li>• Single-precision floating point</li><li>• Fixed point (signed and unsigned)</li><li>• 8-, 16-, and 32-bit signed integers</li><li>• 8-, 16-, and 32-bit unsigned integers</li></ul> |

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

## See Also

### Functions

norm

### Blocks

Array-Vector Multiply | Reciprocal Condition

# Notch-Peak Filter

Design second-order tunable notching and peaking IIR filter



## Library

Filtering / Filter Designs

dspfdesign

## Description

The Notch-Peak Filter block filters each channel of the input signal over time using a specified center frequency and 3 dB bandwidth. This block offers tunable filter design parameters, which enable you to tune the filter characteristics while the simulation is running.

The block designs the filter according to the filter parameters set in the block dialog box. The output port properties such as datatype, complexity, and dimension, are identical to the input port properties.

Each column of the input signal is treated as a separate channel. If the input is a two-dimensional signal, the first dimension represents the channel length (or frame size) and the second dimension represents the number of channels. If the input is a one-dimensional signal, then it is interpreted as a single channel.

This block supports variable-size input, enabling you to change the channel length during simulation. To enable variable-size input, clear the **Inherit sample rate from input** check box. The number of channels must remain constant.

## Algorithms

This block brings the capabilities of the `dsp.NotchPeakFilter` System object to the Simulink environment.

The filter uses a coupled allpass structure to optimize joint computation of the peak and notch response. For information on the algorithms used by the Notch-Peak filter block, see the “Algorithms” on page 4-908 section of `dsp.NotchPeakFilter`.

## Parameters

### Filter specification

Parameters or coefficients used to design the filter, specified as one of the following:

- Bandwidth and center frequency (default) — Design the filter using **3 dB bandwidth (Hz)** and **Notch/Peak center frequency (Hz)**.

- **Coefficients** — Design the filter using **Bandwidth coefficient** and **Center frequency coefficient**.
- **Quality factor and center frequency** — Design the filter using **Quality factor** and **Notch/Peak center frequency (Hz)**.

This parameter is nontunable.

#### **Specify bandwidth from input port**

When you select this check box, the 3 dB bandwidth is input through the **BW** port. When you clear this check box, the 3 dB bandwidth is specified on the block dialog through the **3 dB bandwidth (Hz)** parameter.

This parameter applies when you set **Filter specification** to Bandwidth and center frequency.

#### **3 dB bandwidth (Hz)**

3 dB bandwidth of the filter, specified as a finite positive numeric scalar that is less than half the sample rate of the input signal. This parameter applies when you set **Filter specification** to Bandwidth and center frequency, and clear the **Specify bandwidth from input port** parameter. The default is 2205. This parameter is tunable.

#### **Specify center frequency from input port**

When you select this check box, the center frequency is input through the **Fc** port. When you clear this check box, the center frequency is specified on the block dialog through the **Notch/Peak center frequency (Hz)** parameter.

This parameter applies when you set **Filter specification** to Bandwidth and center frequency or Quality factor and center frequency.

#### **Notch/Peak center frequency (Hz)**

Center frequency of the notch and peak of the filter, specified as a finite positive numeric scalar that is less than half the sample rate of the input signal. This parameter applies when you set **Filter specification** to Bandwidth and center frequency or Quality factor and center frequency, and clear the **Specify center frequency from input port** parameter. The default is 11025. This parameter is tunable.

#### **Specify bandwidth coefficient from input port**

When you select this check box, the bandwidth coefficient is input through the port, **BWCoeff**. When you clear this check box, the bandwidth coefficient is specified on the block dialog through the **Bandwidth coefficient** parameter.

This parameter applies when you set **Filter specification** to Coefficients.

#### **Bandwidth coefficient**

Coefficient that determines the 3 dB bandwidth of the filter, specified as a finite numeric scalar in the range  $[-1 \ 1]$ .

- -1 corresponds to the maximum 3 dB bandwidth (one-fourth the sample rate of the input signal).
- 1 corresponds to the minimum bandwidth (0 Hz, that is, an allpass filter).

This parameter applies when you set **Filter specification** to Coefficients and clear the **Specify bandwidth coefficient from input port** parameter. The default is 0.72654. This parameter is tunable.



**Specify center frequency coefficient from input port**

When you select this check box, the center frequency coefficient is input through the **FcCoeff** port. When you clear this check box, the center frequency coefficient is specified on the block dialog through the **Center frequency coefficient** parameter.

This parameter applies when you set **Filter specification** to Coefficients.

**Center frequency coefficient**

Coefficient that determines the center frequency of the filter, specified as a finite numeric scalar in the range [-1 1].

- -1 corresponds to the minimum center frequency (0 Hz).
- 1 corresponds to the maximum center frequency (half the sample rate of the input signal).

This parameter applies when you set **Filter specification** to Coefficients and clear the **Specify center frequency coefficient from input port** parameter. The default is 0, which corresponds to quarter the sample rate of the input signal. This parameter is tunable.

**Specify quality factor from input port**

When you select this check box, the quality factor is input through the **Q** port. When you clear this check box, the quality factor is specified on the block dialog through the **Quality factor** parameter.

This parameter applies when you set **Filter specification** to Quality factor and center frequency.

**Quality factor**

Quality factor of the notch and peak filter, specified as a real positive scalar. The quality factor is defined as **Notch/Peak center frequency (Hz) / 3 dB bandwidth (Hz)**. A higher quality factor corresponds to a narrower peak or dip. This parameter applies when you set **Filter specification** to Quality factor and center frequency and clear the **Specify quality factor from input port** parameter. The default is 5. This parameter is tunable.

**Output**

Output of the filter block, specified as one of the following:

- Notch and Peak (default) — The block outputs the notch and peak responses of the filter.
- Notch — The block outputs the notch response of the filter.
- Peak — The block outputs the peak response of the filter.

This parameter is nontunable.

**Inherit sample rate from input**

When you select this check box, the block sample rate is computed as  $N/T_s$ , where  $N$  is the frame size of the input signal, and  $T_s$  is the sample time of the input signal. When you clear this check box, the block's sample rate is the value specified in **Input sample rate (Hz)**. By default, this check box is selected.

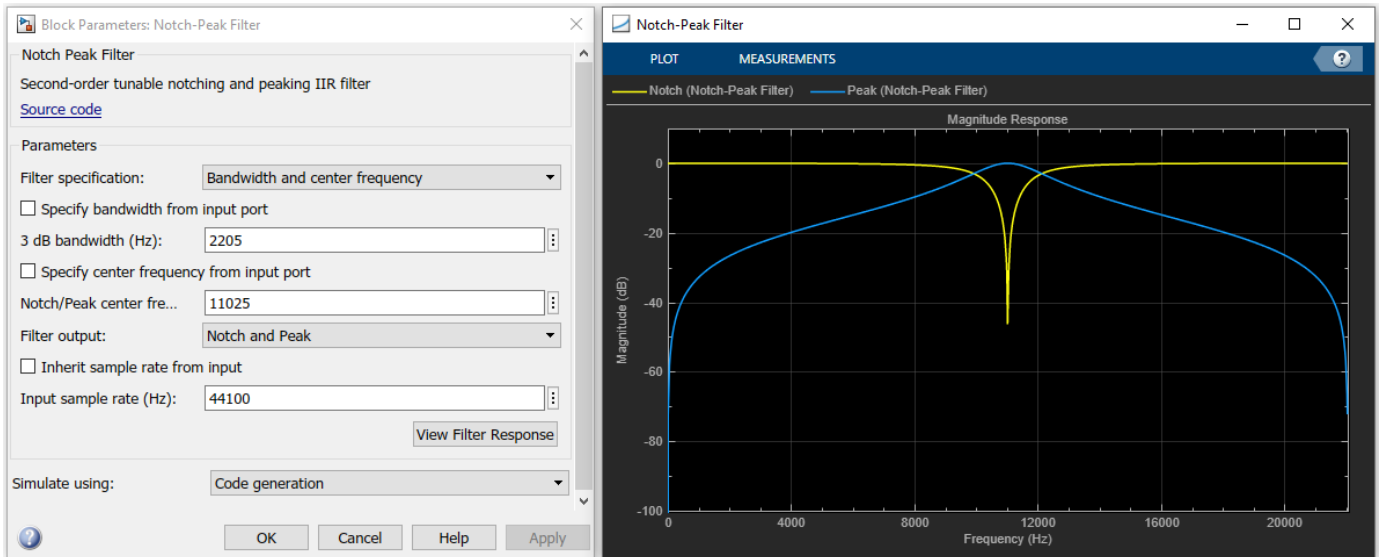
**Input sample rate (Hz)**

Sample rate of the input signal, specified as a positive scalar value. The default is 44100. This parameter applies when you clear the **Inherit sample rate from input** check box. This parameter is nontunable.

**View Filter Response**

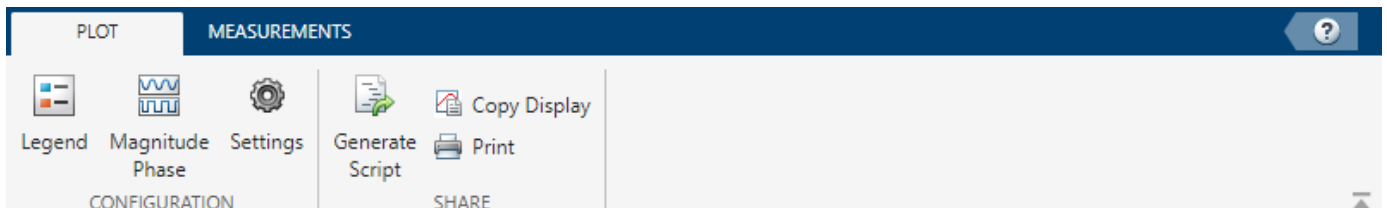
Opens the Dynamic Filter Visualizer window and displays the magnitude and phase response of the notch-peak filter. The response is based on the parameters you select in the Block Parameters

dialog box. To update the response while the visualizer is running, modify the parameters in the dialog box and click **Apply**.

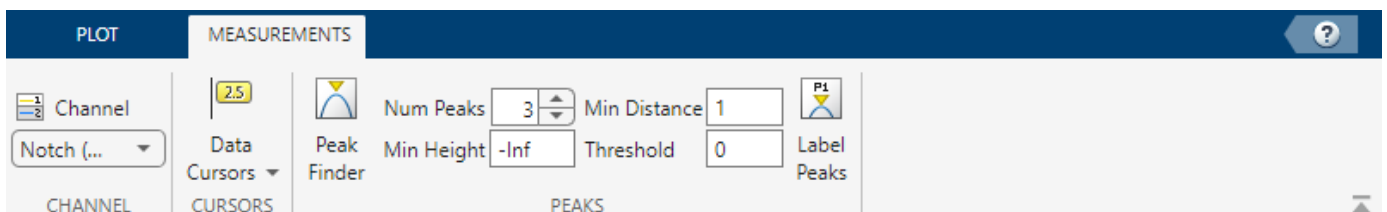


You can configure the plot settings and the frequency response measurements from the interface of the visualizer.

On the **Plot** tab, you can enable the legend, specify to plot both magnitude and phase responses of the filter, modify the plot settings, generate a script to re-create the plot, and even save or share the settings.



On the **Measurements** tab, you can enable data cursors, and display the peak values of the filter response.



For more details on the Dynamic Filter Visualizer interface and its tools, see “Configure Filter Visualizer”.

### Simulate using

Type of simulation to run. You can set this parameter to:

- Code generation (default)

Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than Interpreted execution.

- Interpreted execution

Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than Code generation.

## Supported Data Types

| Port   | Supported Data Types   |
|--------|--|
| Input  | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> </ul> |
| Output | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> </ul> |

## References

- [1] Orfanidis, Sophocles J. *Introduction to Signal Processing*. Upper Saddle River, NJ: Prentice-Hall, 1996.

## Version History

Introduced in R2015a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

## See Also

### Objects

`dsp.NotchPeakFilter`

### Blocks

Parametric EQ Filter

# Nyquist Filter

Design Nyquist filter



## Library

Filtering / Filter Designs

dspfdesign

## Description

This block brings the filter design capabilities of the `filterBuilder` function to the Simulink environment.

This block supports SIMD code generation. For details, see “Code Generation” on page 2-1104.

## Dialog Box

See “Nyquist Filter Design — Main Pane” on page 5-634 for more information about the parameters of this block. The **Data Types** and **Code Generation** panes are not available for blocks in the DSP System Toolbox Filter Designs library.

Parameters of this block that do not change filter order or structure are tunable.

### View filter response

This button opens the Filter Visualization Tool (**FVTool**) from the Signal Processing Toolbox product. You can use the tool to display:

- Magnitude response, phase response, and group delay in the frequency domain.
- Impulse response and step response in the time domain.
- Pole-zero information.

The tool also helps you evaluate filter performance by providing information about filter order, stability, and phase linearity. For more information on FVTool, see the Signal Processing Toolbox documentation.

### Filter Specifications

In this group, you specify your filter format, such as the impulse response and the filter order.

#### Band

Specifies the location of the center of the transition region between the passband and the stopband. The center of the transition region,  $F_c$ , is calculated using the value for **Band**:

$$F_c = F_s / (2 \cdot \mathbf{Band}).$$

The default value, 2, corresponds to a halfband filter.

### Impulse response

Select either FIR or IIR from the drop-down list. FIR is the default. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly. These options are both available only when **Band** is 2. For values of **Band** greater than 2, only FIR designs are supported.

---

**Note** The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

---

### Filter order mode

Select either Minimum (the default) or Specify from the drop-down list. Selecting Specify enables the **Order** option (see the following sections) so you can enter the filter order.

### Filter type

Select Single-rate, Decimator, Interpolator, or Sample-rate converter. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, the block specifies a single-rate filter.

- Selecting Decimator or Interpolator activates the **Decimation Factor** or the **Interpolation Factor** options respectively.
- Selecting Sample-rate converter activates both factors.

### Order

Enter the filter order. This option is enabled only if Specify was selected for **Filter order mode**.

### Decimation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to Decimator or Sample-rate converter. The default value is 2.

### Interpolation Factor

Enter the interpolation factor. This option is enabled only if the **Filter type** is set to Interpolator or Sample-rate converter. The default value is 2.

## Frequency Specifications

The parameters in this group allow you to specify your filter response curve.

### Frequency constraints

Select the filter features that the block uses to define the frequency response characteristics.

### Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select Normalized (0–1) to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—Hz, kHz, MHz, or GHz. Selecting one of the unit options enables the **Input sample rate** parameter.

### Input sample rate

F<sub>s</sub>, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the

specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

**Transition width**

Specify the width of the transition between the end of the passband and the edge of the stopband. Specify the value in normalized frequency units or the absolute units you select in **Frequency units**.

**Magnitude Specifications**

Parameters in this group specify the filter response in the passbands and stopbands.

**Magnitude units**

Specify the units for any parameter you provide in magnitude specifications. From the drop-down list, select one of the following options:

- Linear — Specify the magnitude in linear units.
- dB — Specify the magnitude in decibels (default)
- Squared — Specify the magnitude in squared units.

**Stopband attenuation**

Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels.

**Algorithm**

The parameters in this group allow you to specify the design method and structure of your filter.

**Design Method**

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is Butterworth, and the default FIR method is Kaiser window.

**Scale SOS filter coefficients to reduce chance of overflow**

Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

**Design Options**

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

**Density factor**

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 20 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

### Minimum order

When you select this parameter, the design method determines and design the minimum order filter to meet your specifications. Some filters do not provide this parameter. Select **Any**, **Even**, or **Odd** from the drop-down list to direct the design to be any minimum order, or minimum even order, or minimum odd order.

### Stopband Shape

Stopband shape lets you specify how the stopband changes with increasing frequency. Choose one of the following options:

- **Flat** — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.
- **Linear** — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.

### Stopband Decay

When you set Stopband shape, Stopband decay specifies the amount of decay applied to the stopband. the following conditions apply to Stopband decay based on the value of Stopband Shape:

- When you set **Stopband shape** to **Flat**, **Stopband decay** has no affect on the stopband.
- When you set **Stopband shape** to **Linear**, enter the slope of the stopband in units of dB/rad/s. The block applies that slope to the stopband.
- When you set **Stopband shape** to  $1/f$ , enter a value for the exponent  $n$  in the relation  $(1/f)^n$  to define the stopband decay. The block applies the  $(1/f)^n$  relation to the stopband to result in an exponentially decreasing stopband attenuation.

## Filter Implementation

### Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter.

### Use basic elements to enable filter customization

Select this check box to implement the filter as a subsystem of basic Simulink blocks. Clear the check box to implement the filter as a high-level subsystem. By default, this check box is cleared.

The high-level implementation provides better compatibility across various filter structures, especially filters that would contain algebraic loops when constructed using basic elements. On the other hand, using basic elements enables the following optimization parameters:

- **Optimize for zero gains** — Terminate chains that contain Gain blocks with a gain of zero.
- **Optimize for unit gains** — Remove Gain blocks that scale by a factor of one.
- **Optimize for delay chains** — Substitute delay chains made up of  $n$  unit delays with a single delay by  $n$ .
- **Optimize for negative gains** — Use subtraction in Sum blocks instead of negative gains in Gain blocks.

### Input processing

Specify how the block should process the input. The available options may vary depending on he settings of the **Filter Structure** and **Use basic elements for filter customization** parameters. You can set this parameter to one of the following options:

- **Columns as channels (frame based)** — When you select this option, the block treats each column of the input as a separate channel.
- **Elements as channels (sample based)** — When you select this option, the block treats each element of the input as a separate channel.

### Rate options

When the **Filter type** parameter specifies a multirate filter, select the rate processing rule for the block from following options:

- **Enforce single-rate processing** — When you select this option, the block maintains the sample rate of the input.
- **Allow multirate processing** — When you select this option, the block adjusts the rate at the output to accommodate an increased or reduced number of samples. To select this option, you must set the **Input processing** parameter to **Elements as channels (sample based)**.

### Use variable names for coefficients

Select this check box to enable the specification of coefficients using MATLAB variables. The available coefficient names differ depending on the filter structure. Using symbolic names allows tuning of filter coefficients in generated code. By default, this check box is cleared.

## Supported Data Types

| Port   | Supported Data Types   |
|--------|--|
| Input  | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> </ul> |
| Output | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> </ul> |

## Version History

Introduced in R2006b

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

The Nyquist Filter block supports SIMD code generation using Intel AVX2 technology under these conditions:

- **Filter type** is set to **Single-rate**, **Decimator**, or **Interpolator**.
- For **Filter type** that is set to **Single-rate**, **Structure** is set to **Direct-form FIR** or **Direct-form FIR transposed**.
- For **Filter type** that is set to **Decimator**, **Structure** is set to **Direct-form FIR polyphase decimator** and **Rate options** is set to **Enforce single-rate processing**.
- For **Filter type** that is set to **Interpolator**:



- **Interpolation Factor** cannot be equal to 1.
- **Rate options** is set to Enforce single-rate processing.
- **Input processing** is set to Columns as channels (frame based).
- Input signal has a data type of single or double.
- Input port dimensions cannot be equal to [1 1].

The SIMD technology significantly improves the performance of the generated code.

## Octave Filter

Design octave filter



## Compatibility

---

**Note** The Octave Filter block will be removed from DSP System Toolbox in a future release. Existing instances of the block continue to run. For new code, use the Octave Filter block from Audio Toolbox instead.

---

## Library

Filtering / Filter Designs

dspfdesign

## Description

This block brings the filter design capabilities of the `filterBuilder` function to the Simulink environment.

## Dialog Box

See “Octave Filter Design — Main Pane” on page 5-638 for more information about the parameters of this block. The **Data Types** and **Code Generation** panes are not available for blocks in the DSP System Toolbox Filter Designs library.

Parameters of this block that do not change filter order or structure are tunable.

### View filter response

This button opens the Filter Visualization Tool (**FVTool**) from the Signal Processing Toolbox product. You can use the tool to display:

- Magnitude response, phase response, and group delay in the frequency domain.
- Impulse response and step response in the time domain.
- Pole-zero information.

The tool also helps you evaluate filter performance by providing information about filter order, stability, and phase linearity. For more information on FVTool, see the Signal Processing Toolbox documentation.

## Filter Specifications

### Order

Specify filter order. Possible values are: 4, 6, 8, 10.

### Bands per octave

Specify the number of bands per octave. Possible values are: 1, 3, 6, 12, 24.

### Frequency units

Specify frequency units as Hz or kHz.

### Input sample rate

Specify the input sampling frequency in the frequency units specified previously.

### Center Frequency

Select from the drop-down list of available center frequency values.

## Algorithm

### Design Method

Butterworth is the design method used for this type of filter.

### Scale SOS filter coefficients to reduce chance of overflow

Select the check box to scale the filter coefficients.

## Filter Implementation

### Structure

Specify filter structure. Choose from:

- Direct-form I SOS
- Direct-form II SOS
- Direct-form I transposed SOS
- Direct-form II transposed SOS

### Use basic elements to enable filter customization

Select this check box to implement the filter as a subsystem of basic Simulink blocks. Clear the check box to implement the filter as a high-level subsystem. By default, this check box is cleared.

The high-level implementation provides better compatibility across various filter structures, especially filters that would contain algebraic loops when constructed using basic elements. On the other hand, using basic elements enables the following optimization parameters:

- **Optimize for zero gains** — Terminate chains that contain Gain blocks with a gain of zero.
- **Optimize for unit gains** — Remove Gain blocks that scale by a factor of one.
- **Optimize for delay chains** — Substitute delay chains made up of  $n$  unit delays with a single delay by  $n$ .
- **Optimize for negative gains** — Use subtraction in Sum blocks instead of negative gains in Gain blocks.

### Optimize for unit-scale values

Select this check box to scale unit gains between sections in SOS filters. This parameter is available only for SOS filters.

### Input processing

Specify how the block should process the input. The available options may vary depending on the settings of the **Filter Structure** and **Use basic elements for filter customization** parameters. You can set this parameter to one of the following options:

- **Columns as channels (frame based)** — When you select this option, the block treats each column of the input as a separate channel.
- **Elements as channels (sample based)** — When you select this option, the block treats each element of the input as a separate channel.

### Rate options

When the **Filter type** parameter specifies a multirate filter, select the rate processing rule for the block from following options:

- **Enforce single-rate processing** — When you select this option, the block maintains the sample rate of the input.
- **Allow multirate processing** — When you select this option, the block adjusts the rate at the output to accommodate an increased or reduced number of samples. To select this option, you must set the **Input processing** parameter to **Elements as channels (sample based)**.

### Use variable names for coefficients

Select this check box to enable the specification of coefficients using MATLAB variables. The available coefficient names differ depending on the filter structure. Using symbolic names allows tuning of filter coefficients in generated code. By default, this check box is cleared.

## Supported Data Types

| Port   | Supported Data Types   |
|--------|--|
| Input  | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> </ul> |
| Output | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> </ul> |

## Version History

Introduced in R2007a

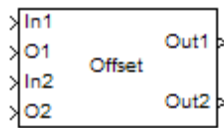
## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

# Offset

Truncate vectors by removing or keeping beginning or ending values



## Libraries:

DSP System Toolbox / Signal Operations

## Description

The Offset block removes or keeps values from the beginning or end of the input vectors. The number of values to remove or keep is determined by the offset value that you specify through the **O** ports (**O1**, **O2**, ...). The length of the truncated vector is determined by the **Output port length** and **Output length** parameters. The **Number of input data-offset pairs** parameter determines the number of input data-offset pairs.

## Ports

### Input

**In1** — First input data

scalar | vector

Specify the first input data as a scalar or a vector. The inputs to the **In** ports (**In1**, **In2**, ...) must be of the same size and data type.

When the input is fixed-point, it is signed only.

Data Types: single | double | int8 | int16 | int32 | fixed point

**O1** — First offset value

integer

Specify the offset value that the block uses to truncate the first input data as an integer. The offset value determines the number of samples the block removes or keeps at the beginning or at the end of the first data.

The inputs to the **O** ports (**O1**, **O2**, ...) must have the same data type. The block rounds any offset value that is a noninteger value to the nearest integer value. If the offset value is out of the  $[0 N]$  range, then the **Action for out of range offset value** parameter determines how the block behaves.  $N$  is the input vector length.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32

**In2** — Second input data

scalar | vector

Specify the second input data as a scalar or a vector. The inputs to the **In** ports (**In1**, **In2**, ...) must be of the same size and data type.

When the input is fixed-point, it is signed only.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `fixed point`

### **O2** — Second offset value

integer

Specify the offset value that the block uses to truncate the second input data as an integer. The offset value determines the number of samples the block removes or keeps at the beginning or at the end of the second data.

The inputs to the **O** ports (**O1**, **O2**, ...) must have the same data type. The block rounds any offset value that is a noninteger value to the nearest integer value. If the offset value is out of the  $[0 N]$  range, then the **Action for out of range offset value** parameter determines how the block behaves.  $N$  is the input vector length.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

### **Output**

#### **Out1** — First truncated data

scalar | vector

The block returns the truncated data as a scalar or a vector. The length of the truncated data depends on the **Output port length** and the **Output length** parameters. If **Output port length** is set to `Same as input`, the truncated data has the same size as the input data. If **Output port length** is set to `User-defined`, then the **Output length** parameter determines the length of the output data.

There is one output port for each pair of **In** and **O** ports.

The truncated output has the same data type as the input data. When the output is fixed-point, it is signed only.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `fixed point`

#### **Out2** — Second truncated data

scalar | vector

The block returns the truncated data as a scalar or a vector. The length of the truncated data depends on the **Output port length** and the **Output length** parameters. If **Output port length** is set to `Same as input`, the truncated data has the same size as the input data. If **Output port length** is set to `User-defined`, then the **Output length** parameter determines the length of the output data.

There is one output port for each pair of **In** and **O** ports.

The truncated output has the same data type as the input data. When the output is fixed-point, it is signed only.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `fixed point`

## **Parameters**

**Mode** — Specify which values to discard and which values to keep

`Remove beginning samples (default)` | `Remove ending samples` | `Keep beginning samples` | `Keep ending samples`

Use this parameter to specify which values the block discards and which values the block keeps from the input vector.

Select from the following:

- **Remove beginning samples** -- Discard the initial values of a vector.
- **Remove ending samples** -- Discard the final values of a vector.
- **Keep beginning samples** -- Retain the initial values of a vector.
- **Keep ending samples** -- Retain the final values of a vector.

**Number of input data-offset pairs** — Number of input data-offset pairs

2 (default) | positive integer

Use the **Number of input data-offset pairs** parameter to specify the number of inputs to the block. The number of input ports is twice the scalar value you specify in this parameter. For example, if you enter 3, input ports **In1**, **O1**, **In2**, **O2**, **In3**, and **O3** appear on the block.

**Output port length** — Mode to specify output port length

Same as input (default) | User-defined

Use this parameter to specify the length of the output vectors. If you select **Same as input**, the output vectors have the same length as the input vectors. If you select **User-defined**, you can enter the desired length in the **Output length** parameter.

**Output length** — Desired length of output vectors

9 (default) | positive integer

Enter a scalar that represents the desired length of the output vectors. If the desired output length that you specify is greater than the number of values that the block extracts from the input vector, the block zero-pads the end of the vector to reach the length that you specify.

### Dependencies

This parameter is visible only if you set the **Output port length** parameter to **User-defined**.

**Action for out of range offset value** — Action for out of range offset value

Clip (default) | Clip and warn | Error

Use this parameter to determine how the block behaves when an offset value is not in the range such that  $0 \leq \text{offset value} \leq N$ , where  $N$  is the input vector length.

- **Clip** -- Offset values less than 0 are set to 0 and offset values greater than  $N$  are set to  $N$ .
- **Clip and warn** -- Offset values less than 0 are set to 0 and offset values greater than  $N$  are set to  $N$ . In addition, the block displays a warning message.
- **Error** -- Simulation stops and the block displays an error when the offset values are out of range.

## Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | double   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no                                      |
| <b>Multidimensional Signals</b> | no                                      |
| <b>Variable-Size Signals</b>    | no                                      |
| <b>Zero-Crossing Detection</b>  | no                                      |

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Generated code relies on the memcpy or memset function (string.h) under certain conditions.

### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

If the input is fixed point, it must be a signed integer or a signed fixed point value with a power-of-two slope and zero bias.

## See Also

Detrend | Pad | DC Blocker | Unwrap



## Overlap-Add FFT Filter (Obsolete)

Implement overlap-add method of frequency-domain filtering



### Library

Filtering / Filter Implementations

dsparch4

### Description

---

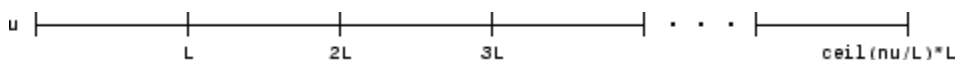
**Note** The Overlap-Add FFT Filter block has been replaced with the Frequency-Domain FIR Filter block. Existing instances of the Overlap-Add FFT Filter block continue to run.

---

The Overlap-Add FFT Filter block uses an FFT to implement the *overlap-add method*, a technique that combines successive frequency-domain filtered sections of an input sequence.

The block accepts vector or matrix inputs, and treats each column of the input as an individual channel. The block unbuffers the input data into row vectors such that the length of the output vector is equal to the number of channels in the input. The data output rate of the block is  $M$  times faster than its data input rate, where  $M$  is the length of the columns in the input (frame-size).

The block breaks the scalar input sequence  $u$ , of length  $nu$ , into length- $L$  nonoverlapping data sections,



which it linearly convolves with the filter's FIR coefficients,

$$H(z) = B(z) = b_1 + b_2z^{-1} + \dots + b_{n+1}z^{-n}$$

The numerator coefficients for  $H(z)$  are specified as a vector by the **FIR coefficients** parameter. The coefficient vector,  $\mathbf{b} = [b(1) \ b(2) \ \dots \ b(n+1)]$ , can be generated by one of the filter design functions in the Signal Processing Toolbox product, such as `fir1`. All filter states are internally initialized to zero.

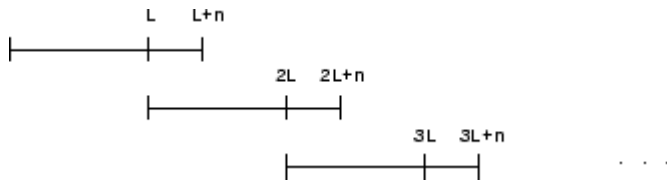
When either the filter coefficients or the inputs to the block are complex, the **Output** parameter should be set to `Complex`. Otherwise, the default **Output** setting, `Real`, instructs the block to take only the real part of the solution.

The block's overlap-add operation is equivalent to

$$y = \text{ifft}(\text{fft}(u(i:i+L-1), \text{nfft}) .* \text{fft}(b, \text{nfft}))$$

where you specify `nfft` in the **FFT size** parameter as a power-of-two value greater (typically *much* greater) than  $n+1$ . Values for **FFT size** that are not powers of two are rounded upwards to the nearest power-of-two value to obtain `nfft`.

The block overlaps successive output sections by  $n$  points and sums them.



The first  $L$  samples of each summation are output in sequence. The block chooses the parameter  $L$  based on the filter order and the FFT size.

$$L = \text{nfft} - n$$

### Latency

In *single-tasking* operation, the Overlap-Add FFT Filter block has a latency of  $\text{nfft} - n + 1$  samples. The first  $\text{nfft} - n + 1$  consecutive outputs from the block are zero; the first filtered input value appears at the output as sample  $\text{nfft} - n + 2$ .

In *multitasking* operation, the Overlap-Add FFT Filter block has a latency of  $2 * (\text{nfft} - n) + 1$  samples. The first  $2 * (\text{nfft} - n) + 1$  consecutive outputs from the block are zero; the first filtered input value appears at the output as sample  $2 * (\text{nfft} - n) + 3$ .

---

**Note** For more information on latency and the Simulink software tasking modes, see “Excess Algorithmic Delay (Tasking Latency)” and “Time-Based Scheduling and Code Generation” (Simulink Coder).

---

## Parameters

### FFT size

The size of the FFT, which should be a power-of-two value greater than the length of the specified FIR filter.

### FIR coefficients

The filter numerator coefficients.

### Output

The complexity of the output; `Real` or `Complex`. When the input signal or the filter coefficients are complex, this should be set to `Complex`.

## References

Oppenheim, A. V. and R. W. Schaffer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989.

Proakis, J. and D. Manolakis. *Digital Signal Processing*. 3rd ed. Englewood Cliffs, NJ: Prentice-Hall, 1996.

## Supported Data Types

- Double-precision floating point
- Single-precision floating point

## See Also

Overlap-Save FFT Filter

DSP System Toolbox product

## Version History

Introduced before R2006a

## Overlap-Save FFT Filter (Obsolete)

Implement overlap-save method of frequency-domain filtering



### Library

Filtering / Filter Implementations

dsparch4

### Description

---

**Note** The Overlap-Save FFT Filter block has been replaced with the Frequency-Domain FIR Filter block. Existing instances of the Overlap-Save FFT Filter block continue to run.

---

The Overlap-Save FFT Filter block uses an FFT to implement the *overlap-save method*, a technique that combines successive frequency-domain filtered sections of an input sequence.

The block accepts vector or matrix inputs, and treats each column of the input as an individual channel. The block unbuffers the input data into row vectors such that the length of the output vector is equal to the number of channels in the input. The data output rate of the block is  $M$  times faster than its data input rate, where  $M$  is the length of the columns in the input (frame-size).

Overlapping sections of input  $u$  are circularly convolved with the FIR filter coefficients

$$H(z) = B(z) = b_1 + b_2z^{-1} + \dots + b_{n+1}z^{-n}$$

The numerator coefficients for  $H(z)$  are specified as a vector by the **FIR coefficients** parameter. The coefficient vector,  $\mathbf{b} = [b(1) \ b(2) \ \dots \ b(n+1)]$ , can be generated by one of the filter design functions in the Signal Processing Toolbox product, such as `fir1`. All filter states are internally initialized to zero.

When either the filter coefficients or the inputs to the block are complex, the **Output** parameter should be set to **Complex**. Otherwise, the default **Output** setting, **Real**, instructs the block to take only the real part of the solution.

The circular convolution of each section is computed by multiplying the FFTs of the input section and filter coefficients, and computing the inverse FFT of the product.

```
y = ifft(fft(u(i:i+(L-1))),nfft) .* fft(b,nfft)
```

where you specify `nfft` in the **FFT size** parameter as a power of two value greater (typically *much* greater) than  $n+1$ . Values for **FFT size** that are not powers of two are rounded upwards to the nearest power-of-two value to obtain `nfft`.

The first  $n$  points of the circular convolution are invalid and are discarded. The Overlap-Save FFT Filter block outputs the remaining  $n_{\text{fft}} - n$  points, which are equivalent to the linear convolution.

### Latency

In *single-tasking* operation, the Overlap-Save FFT Filter block has a latency of  $n_{\text{fft}} - n + 1$  samples. The first  $n_{\text{fft}} - n + 1$  consecutive outputs from the block are zero; the first filtered input value appears at the output as sample  $n_{\text{fft}} - n + 2$ .

In *multitasking* operation, the Overlap-Save FFT Filter block has a latency of  $2 * (n_{\text{fft}} - n + 1)$  samples. The first  $2 * (n_{\text{fft}} - n + 1)$  consecutive outputs from the block are zero; the first filtered input value appears at the output as sample  $2 * (n_{\text{fft}} - n) + 3$ .

---

**Note** For more information on latency and the Simulink environment tasking modes, see “Excess Algorithmic Delay (Tasking Latency)” and “Time-Based Scheduling and Code Generation” (Simulink Coder).

---

## Parameters

### FFT size

The size of the FFT, which should be a power of two value greater than the length of the specified FIR filter.

### FIR coefficients

The filter numerator coefficients.

### Output

The complexity of the output; `Real` or `Complex`. When the input signal or the filter coefficients are complex, this should be set to `Complex`.

## References

Oppenheim, A. V. and R. W. Schaffer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989.

Proakis, J. and D. Manolakis. *Digital Signal Processing*. 3rd ed. Englewood Cliffs, NJ: Prentice-Hall, 1996.

## Supported Data Types

- Double-precision floating point
- Single-precision floating point

## See Also

Overlap-Add FFT Filter

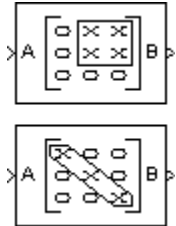
DSP System Toolbox

## **Version History**

**Introduced before R2006a**

## Overwrite Values

Overwrite submatrix or subdiagonal of input



## Library

- Math Functions / Matrices and Linear Algebra / Matrix Operations

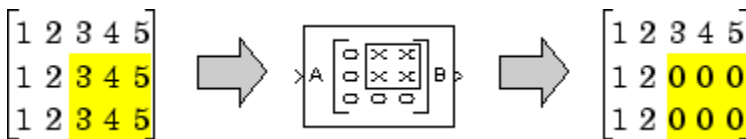
`dspmtx3`

- Signal Management / Indexing

`dspindex`

## Description

The Overwrite Values block overwrites a contiguous submatrix or subdiagonal of an input matrix. You can provide the overwriting values by typing them in a block parameter, or through an additional input port, which is useful for providing overwriting values that change at each time step.



The block accepts scalars, vectors and matrices. The output always has the same size as the original input signal, not necessarily the same size as the signal containing the overwriting values. The input(s) and output of this block must have the same data type.

### Specifying the Overwriting Values

The **Source of overwriting value(s)** parameter determines how you must provide the overwriting values, and has the following settings.

- Specify via dialog** — You must provide the overwriting value(s) in the **Overwrite with** parameter. The block uses the same overwriting values to overwrite the specified portion of the input at each time step. To learn how to specify valid overwriting values, see “Valid Overwriting Values” on page 2-1120.
- Second input port** — You must provide overwriting values through a second block input port, *V*. Use this setting to provide different overwriting values at each time step. The output inherits its size and rate from the input signal, *not* the overwriting values.

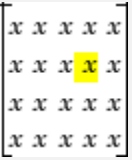



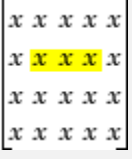
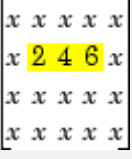
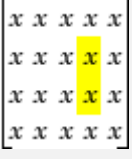
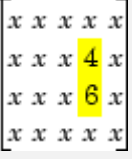
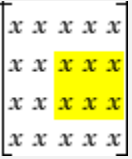

The rate at which you provide the overwriting values through input port V must match the rate at which the block receives each input matrix at input port A. In other words, the input signals must have the same Simulink sample time.

**Valid Overwriting Values**

The overwriting values can be a single constant, vector, or matrix, depending on the portion of the input you are overwriting, regardless of whether you provide the overwriting values through an input port or by providing them in the **Overwrite with** parameter.



**Valid Overwriting Values**

| Portion of Input to Overwrite  | Valid Overwriting Values  | Example   |
|--|---|---|
| <p>A single element in the input</p>                    | <p>Any constant value, <math>v</math></p>                             | <p><math>v = 9</math></p>   |
| <p>A length-<math>k</math> portion of the diagonal</p>  | <p>Any length-<math>k</math> column or row vector, <math>v</math></p> | <p><math>k = 3 \quad v = [2 \ 4 \ 6] \quad \text{or} \quad \begin{bmatrix} 2 \\ 4 \\ 6 \end{bmatrix}</math></p>         |
| <p>A length-<math>k</math> portion of a row</p>        | <p>Any length-<math>k</math> row vector, <math>v</math></p>           | <p><math>k = 3 \quad v = [2 \ 4 \ 6]</math></p>    |
| <p>A length-<math>k</math> portion of a column</p>    | <p>Any length-<math>k</math> column vector, <math>v</math></p>        | <p><math>k = 2 \quad v = \begin{bmatrix} 4 \\ 6 \end{bmatrix}</math></p>    |
| <p>An <math>m</math>-by-<math>n</math> submatrix</p>  | <p>Any <math>m</math>-by-<math>n</math> matrix, <math>v</math></p>    | <p><math>m = 2 \quad n = 3 \quad v = \begin{bmatrix} 4 &amp; 5 &amp; 6 \\ 7 &amp; 8 &amp; 9 \end{bmatrix}</math></p>  |

This block supports Simulink virtual buses.

## Parameters

---

**Note** Only some of the following parameters are visible in the dialog box at any one time.

---

### Overwrite

Determines whether to overwrite a specified submatrix or a specified portion of the diagonal.

### Source of overwriting value(s)

Determines where you must provide the overwriting values: either through an input port, or by providing them in the **Overwrite with** parameter. For more information, see “Specifying the Overwriting Values” on page 2-1119.

### Overwrite with

The value(s) with which to overwrite the specified portion of the input matrix. Enabled only when **Source of overwriting value(s)** is set to *Specify via dialog*. To learn how to specify valid overwriting values, see “Valid Overwriting Values” on page 2-1120.

### Row span

The range of input rows to be overwritten. Options are *All rows*, *One row*, or *Range of rows*. For descriptions of these options, see “Parameters” on page 2-1122.

### Row/Starting row

The input row that is the first row of the submatrix that the block overwrites. For a description of the options for the **Row** and **Starting row** parameters, see Settings for Row, Column, Starting Row, and Starting Column Parameters. **Row** is enabled when **Row span** is set to *One row*, and **Starting row** when **Row span** is set to *Range of rows*.

### Row index/Starting row index

Index of the input row that is the first row of the submatrix that the block overwrites. See how to use these parameters in Settings for Row, Column, Starting Row, and Starting Column Parameters. **Row index** is enabled when **Row** is set to *Index*, and **Starting row index** when **Starting row** is set to *Index*.

### Row offset/Starting row offset

The offset of the input row that is the first row of the submatrix that the block overwrites. See how to use these parameters in Settings for Row, Column, Starting Row, and Starting Column Parameters. **Row offset** is enabled when **Row** is set to *Offset from middle* or *Offset from last*, and **Starting row offset** is enabled when **Starting row** is set to *Offset from middle* or *Offset from last*.

### Ending row

The input row that is the last row of the submatrix that the block overwrites. For a description of this parameter's options, see Settings for Ending Row and Ending Column Parameters. This parameter is enabled when **Row span** is set to *Range of rows*, and **Starting row** is set to any option but *Last*.

### Ending row index

Index of the input row that is the last row of the submatrix that the block overwrites. See how to use this parameter in Settings for Ending Row and Ending Column Parameters. Enabled when **Ending row** is set to *Index*.

**Ending row offset**

The offset of the input row that is the last row of the submatrix that the block overwrites. See how to use this parameter in Settings for Ending Row and Ending Column Parameters. Enabled when **Ending row** is set to `Offset from middle` or `Offset from last`.

**Column span**

The range of input columns to be overwritten. Options are `All columns`, `One column`, or `Range of columns`. For descriptions of the analogous row options, see “Parameters” on page 2-1122.

**Column/Starting column**

The input column that is the first column of the submatrix that the block overwrites. For a description of the options for the **Column** and **Starting column** parameters, see Settings for Row, Column, Starting Row, and Starting Column Parameters. **Column** is enabled when **Column span** is set to `One column`, and **Starting column** when **Column span** is set to `Range of columns`.

**Column index/Starting column index**

Index of the input column that is the first column of the submatrix that the block overwrites. See how to use these parameters in Settings for Row, Column, Starting Row, and Starting Column Parameters. **Column index** is enabled when **Column** is set to `Index`, and **Starting column index** when **Starting column** is set to `Index`.

**Column offset/Starting column offset**

The offset of the input column that is the first column of the submatrix that the block overwrites. See how to use these parameters in Settings for Row, Column, Starting Row, and Starting Column Parameters. **Column offset** is enabled when **Column** is set to `Offset from middle` or `Offset from last`, and **Starting column offset** is enabled when **Starting column** is set to `Offset from middle` or `Offset from last`.

**Ending column**

The input column that is the last column of the submatrix that the block overwrites. For a description of this parameter's options, see Settings for Ending Row and Ending Column Parameters. This parameter is enabled when **Column span** is set to `Range of columns`, and **Starting column** is set to any option but `Last`.

**Ending column index**

Index of the input column that is the last column of the submatrix that the block overwrites. See how to use this parameter in Settings for Ending Row and Ending Column Parameters. This parameter is enabled when **Ending column** is set to `Index`.

**Ending column offset**

The offset of the input column that is the last column of the submatrix that the block overwrites. See how to use this parameter in Settings for Ending Row and Ending Column Parameters. This parameter is enabled when **Ending column** is set to `Offset from middle` or `Offset from last`.

**Diagonal span**

The range of diagonal elements to be overwritten. Options are `All elements`, `One element`, or `Range of elements`. For descriptions of these options, see “Overwriting a Subdiagonal” on page 2-1127.

**Element/Starting element**

The input diagonal element that is the first element in the subdiagonal that the block overwrites. For a description of the options for the **Element** and **Starting element** parameters, see Element

and Starting Element Parameters. **Element** is enabled when **Element span** is set to One element, and **Starting element** when **Element span** is set to Range of elements.

#### **Element index/Starting element index**

Index of the input diagonal element that is the first element of the subdiagonal that the block overwrites. See how to use these parameters in Element and Starting Element Parameters.

**Element index** is enabled when **Element** is set to Index, and **Starting element index** when **Starting element** is set to Index.

#### **Element offset/Starting element offset**

The offset of the input diagonal element that is the first element of the subdiagonal that the block overwrites. See how to use these parameters in Element and Starting Element Parameters.

**Element offset** is enabled when **Element** is set to Offset from middle or Offset from last, and **Starting element offset** is enabled when **Starting element** is set to Offset from middle or Offset from last.

#### **Ending element**

The input diagonal element that is the last element of the subdiagonal that the block overwrites. For a description of this parameter's options, see Ending Element Parameters. This parameter is enabled when **Element span** is set to Range of elements, and **Starting element** is set to any option but Last.

#### **Ending element index**

Index of the input diagonal element that is the last element of the subdiagonal that the block overwrites. See how to use this parameter in Ending Element Parameters. This parameter is enabled when **Ending element** is set to Index.

#### **Ending element offset**

The offset of the input diagonal element that is the last element of the subdiagonal that the block overwrites. See how to use this parameter in Ending Element Parameters. This parameter is enabled when **Ending element** is set to Offset from middle or Offset from last.

## **Examples**

### **Overwriting a Submatrix**

To overwrite a submatrix, follow these steps:

- 1 Set the **Overwrite** parameter to Submatrix.
- 2 Specify the overwriting values as described in “Specifying the Overwriting Values” on page 2-1119.
- 3 Specify which rows and columns of the input matrix are contained in the submatrix that you want to overwrite by setting the **Row span** parameter to one of the following options and the **Column span** to the analogous column-related options:
  - All rows — The submatrix contains all rows of the input matrix.
  - One row — The submatrix contains only one row of the input matrix, which you must specify in the **Row** parameter, as described in the following table.
  - Range of rows — The submatrix contains one or more rows of the input, which you must specify in the **Starting Row** and **Ending row** parameters, as described in the following tables.
- 4 When you set **Row span** to One row or Range of rows, you need to further specify the row(s) contained in the submatrix by setting the **Row** or **Starting row** and **Ending row** parameters.

Likewise, when you set **Column span** to One column or Range of columns, you must further specify the column(s) contained in the submatrix by setting the **Column** or **Starting column** and **Ending column** parameters. For descriptions of the settings for these parameters, see the following tables.

#### Settings for Row, Column, Starting Row, and Starting Column Parameters

| Settings for Specifying the Submatrix's First Row or Column | First Row of Submatrix<br>(Only row for Row span = One row)   | First Column of Submatrix<br>(Only row for Row span = One row)   |
|---|---|--|
| First   | First row of the input  | First column of the input  |
| Index   | Input row specified in the <b>Row index</b> parameter   | Input column specified in the <b>Column index</b> parameter  |
| Offset from last  | Input row with the index $M - \text{rowOffset}$ where $M$ is the number of input rows, and $\text{rowOffset}$ is the value of the <b>Row offset</b> or <b>Starting row offset</b> parameter                     | Input column with the index $N - \text{colOffset}$ where $N$ is the number of input columns, and $\text{colOffset}$ is the value of the <b>Column offset</b> or <b>Starting column offset</b> parameter                        |
| Last  | Last row of the input   | Last column of the input   |
| Offset from middle  | Input row with the index $\text{floor}(M/2 + 1 - \text{rowOffset})$ where $M$ is the number of input rows, and $\text{rowOffset}$ is the value of the <b>Row offset</b> or <b>Starting row offset</b> parameter | Input column with the index $\text{floor}(N/2 + 1 - \text{colOffset})$ where $N$ is the number of input columns, and $\text{colOffset}$ is the value of the or <b>Column offset</b> or <b>Starting column offset</b> parameter |
| Middle  | Input row with the index $\text{floor}(M/2 + 1)$ where $M$ is the number of input rows  | Input columns with the index $\text{floor}(N/2 + 1)$ where $N$ is the number of input columns  |

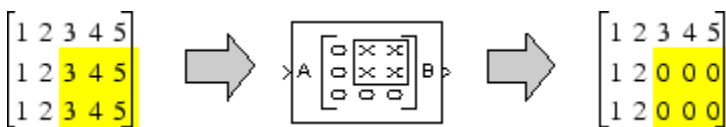
## Settings for Ending Row and Ending Column Parameters

| Settings for Specifying the Submatrix's Last Row or Column | Last Row of Submatrix  | Last Column of Submatrix  |
|--|--|---|
| Index  | Input row specified in the <b>Ending row index</b> parameter   | Input column specified in the <b>Ending column index</b> parameter  |
| Offset from last   | Input row with the index $M - \text{rowOffset}$ where $M$ is the number of input rows, and $\text{rowOffset}$ is the value of the <b>Ending row offset</b> parameter                     | Input column with the index $N - \text{colOffset}$ where $N$ is the number of input columns, and $\text{colOffset}$ is the value of the <b>Ending column offset</b> parameter                     |
| Last   | Last row of the input  | Last column of the input  |
| Offset from middle   | Input row with the index $\text{floor}(M/2 + 1 - \text{rowOffset})$ where $M$ is the number of input rows, and $\text{rowOffset}$ is the value of the <b>Ending row offset</b> parameter | Input column with the index $\text{floor}(N/2 + 1 - \text{colOffset})$ where $N$ is the number of input columns, and $\text{colOffset}$ is the value of the <b>Ending column offset</b> parameter |
| Middle   | Input row with the index $\text{floor}(M/2 + 1)$ where $M$ is the number of input rows   | Input columns with the index $\text{floor}(N/2 + 1)$ where $N$ is the number of input columns   |

For example, to overwrite the lower-right 2-by-3 submatrix of a 3-by-5 input matrix with all zeros, enter the following set of parameters:

- **Overwrite** = Submatrix
- **Source of overwriting value(s)** = Specify via dialog
- **Overwrite with** = 0
- **Row span** = Range of rows
- **Starting row** = Index
- **Starting row index** = 2
- **Ending row** = Last
- **Column span** = Range of columns
- **Starting column** = Offset from last
- **Starting column offset** = 2
- **Ending column** = Last

The following figure shows the block with the above settings overwriting a portion of a 3-by-5 input matrix.



There are often several possible parameter combinations that select the *same* submatrix from the input. For example, instead of specifying Last for **Ending column**, you could select the same submatrix by specifying

- **Ending column** = Index
- **Ending column index** = 5

### Overwriting a Subdiagonal

To overwrite a subdiagonal, follow these steps:

- 1 Set the **Overwrite** parameter to **Diagonal**.
- 2 Specify the overwriting values as described in “Specifying the Overwriting Values” on page 2-1119.
- 3 Specify the subdiagonal that you want to overwrite by setting the **Diagonal span** parameter to one of the following options:
  - All elements — Overwrite the entire input diagonal.
  - One element — Overwrite one element in the diagonal, which you must specify in the **Element** parameter (described below).
  - Range of elements — Overwrite a portion of the input diagonal, which you must specify in the **Starting element** and **Ending element** parameters, as described in the following table.
- 4 When you set **Diagonal span** to One element or Range of elements, you need to further specify which diagonal element(s) to overwrite by setting the **Element** or **Starting element** and **Ending element** parameters. See the following tables.

### Element and Starting Element Parameters

| Settings for Element and Starting Element Parameters | First Element in Subdiagonal<br>(Only element when Diagonal span = One element)  |
|--|--|
| First  | Diagonal element in first row of the input   |
| Index  | $k$ th diagonal element, where $k$ is the value of the <b>Element index</b> or <b>Starting element index</b> parameter   |
| Offset from last                                     | Diagonal element in the row with the index $M - \text{offset}$<br>where $M$ is the number of input rows, and $\text{offset}$ is the value of the <b>Element offset</b> or <b>Starting element offset</b> parameter                           |
| Last   | Diagonal element in the last row of the input  |
| Offset from middle                                   | Diagonal element in the input row with the index $\text{floor}(M/2 + 1 - \text{offset})$<br>where $M$ is the number of input rows, and $\text{offset}$ is the value of the <b>Element offset</b> or <b>Starting element offset</b> parameter |
| Middle   | Diagonal element in the input row with the index $\text{floor}(M/2 + 1)$<br>where $M$ is the number of input rows  |

### Ending Element Parameters

| Settings for Ending Element Parameter | Last Element in Subdiagonal  |
|---------------------------------------|--|
| Index                                 | $k$ th diagonal element, where $k$ is the value of the <b>Ending element index</b> parameter   |
| Offset from last                      | Diagonal element in the row with the index $M - \text{offset}$ where $M$ is the number of input rows, and $\text{offset}$ is the value of the <b>Ending element offset</b> parameter                           |
| Last                                  | Diagonal element in the last row of the input  |
| Offset from middle                    | Diagonal element in the input row with the index $\text{floor}(M/2 + 1 - \text{offset})$ where $M$ is the number of input rows, and $\text{offset}$ is the value of the <b>Ending element offset</b> parameter |
| Middle                                | Diagonal element in the input row with the index $\text{floor}(M/2 + 1)$ where $M$ is the number of input rows   |

### Supported Data Types

The input(s) and output of this block must have the same data type.

| Port | Supported Data Types  |
|------|---|
| A    | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed and unsigned)</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> <li>• Bus</li> </ul> |
| V    | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed and unsigned)</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> <li>• Bus</li> </ul> |



| Port | Supported Data Types  |
|------|---|
| B    | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed and unsigned)</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> <li>• Bus</li> </ul> |

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Generated code relies on the memcpy or memset function (`string.h`) under certain conditions.

### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

## See Also

### Functions

reshape

### Blocks

Reshape | Selector | Submatrix | Variable Selector

## Pad

Pad or truncate specified dimension(s)



### Libraries:

DSP System Toolbox / Signal Operations

## Description

The Pad block extends or crops the dimensions of an input signal by padding or truncating along its columns, rows, columns and rows, or along any other dimension(s) that you specify. The block truncates the input when you specify output dimensions that are shorter than the corresponding input dimensions. If the input and output lengths are the same, the block acts as a pass-through.

You can enter the pad value in the block mask or via an input port. You can enter output sizes in the block mask, or have the block pad the specified dimensions until their length is the next highest power of two. The **Pad signal at** parameter controls whether the block pads or truncates the input signal dimensions at the beginning, end, or both. For odd pad or truncation lengths, the extra pad or truncation value is applied at the end of the signal. When the block is in the **Specified dimensions** mode, you can specify either the output size or the pad size.

You can configure the block to warn or error when an input signal is truncated using the **Action when truncation occurs** parameter.

## Ports

### Input

#### Input 1 — Data input

scalar | vector | matrix | *N*-D array

Specify the input as a scalar, vector, matrix, or an *N*-D array.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | Boolean | fixed point | bus

### Output

#### Output 1 — Padded or truncated output

scalar | vector | matrix | *N*-D array

Output of the Pad block. The size and dimension of the output depends on whether the block has padded or truncated the input along the specified dimension. The data type and complexity of the output matches that of the input.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | Boolean | fixed point | bus

## Parameters

**Pad over** — Dimension over which to pad or truncate

Columns (default) | Rows | Columns and rows | None | Specified dimensions

Specify the dimensions over which to pad or truncate:

- Columns
- Rows
- Columns and rows
- None
- Specified dimensions

**Dimensions to pad** — Specify dimension over which to pad or truncate

1 (default) | scalar | vector

Specify the one-based dimension(s) over which to pad or truncate. The value for this parameter can be a scalar or a vector. For example, specify 1 to pad columns. Specify [1 2] to pad columns and rows. Specify [1 3 5] to pad the first, third, and fifth dimensions.

### Dependencies

This parameter appears only when you set **Pad over** to Specified dimensions.

**Pad value source** — Source of pad value

Specify via dialog (default) | Input port

Choose how you specify the pad value. The pad value can come from the dialog or from an input port:

- If you select Specify via dialog, the **Pad value** parameter appears.
- If you select Input port, the PVal port appears on the block icon.

**Pad value** — Pad value

0 (default) | scalar

Specify the constant scalar value with which to pad the input.

**Tunable:** Yes

### Dependencies

This parameter appears only when you set **Pad value source** to Specify via dialog.

**Output column mode** — Mode of specifying column length

User-specified | Next power of two

Choose how you specify the column length of the output:

- If you select User-specified, the **Column size** parameter appears.

- If you select `Next power of two`, the block pads the output columns until their length is the next highest power of two. If the column length is already a power of two, the block does not pad the columns.

**Dependencies**

This parameter appears only when you set **Pad over** to `Columns` or `Columns and rows`.

**Column size** — Column length of output

1 | scalar

Specify the column length of the output. If the specified column length is longer than the input column length, the block pads the columns. If the specified column length is shorter than the input column length, the block truncates the columns.

**Dependencies**

This parameter appears only when you set **Output column mode** to `User-specified`.

**Output row mode** — Mode of specifying row length

`User-specified (default)` | `Next power of two`

Choose how you specify the row length of the output:

- If you select `User-specified`, the **Row size** parameter appears.
- If you select `Next power of two`, the block pads the output rows until their length is the next highest power of two. If the row length is already a power of two, the block does not pad the rows.

**Dependencies**

This parameter appears only when you set **Pad over** to `Rows` or `Columns and rows`.

**Row size** — Row length of output

1 | scalar

Specify the row length of the output. If the specified row length is longer than the input row length, the block pads the rows. If the specified row length is shorter than the input row length, the block truncates the rows.

**Dependencies**

This parameter appears only when you set **Output row mode** to `User-specified`.

**Specify** — Specify output size or pad size

`Output size (default)` | `Pad size`

Choose whether you want to control the output length of the specified dimensions by specifying the output size or the pad size.

**Dependencies**

This parameter appears only when you set **Pad over** to `Specified dimensions`.

**Output size mode** — Mode of specifying output length

User-specified | Next power of two

Choose how you specify the output length of the specified dimensions:

- If you select **User-specified**, the **Output size** parameter appears.
- If you select **Next power of two**, the block pads the specified dimensions until their length is the next highest power of two. If the dimension length is already a power of two, the block does not pad in that dimension.

**Dependencies**

This parameter appears only when you set **Pad over** to **Specified dimensions** and the **Specify** parameter to **Output size**.

**Output size** — Output length along specified dimension

1 | scalar | vector

Specify the output length of the specified dimension(s). This parameter must be a scalar or a vector with the same number of elements as the **Dimensions to pad** parameter. Each element in the **Output size** vector gives the output length for the corresponding dimension in the **Dimensions to pad** vector. If the specified length is longer than the input length for a given dimension, the block pads that dimension. If the specified length is shorter than the input length for a given dimension, the block truncates that dimension.

**Dependencies**

This parameter appears only when you set **Pad over** to **Specified dimensions** and the **Specify** parameter to **Output size**.

**Pad size at beginning** — Specify number of values to add to beginning of input signal

0 | scalar | vector

Specify how many values to add to the beginning of the input signal along the specified dimension(s). This parameter must be a scalar or a vector with the same number of elements as the **Dimensions to pad** parameter. Each element in the **Pad size at beginning** parameter gives the pad length at the beginning of the corresponding dimension in the **Dimensions to pad** parameter. Values of this parameter must be nonnegative integers.

**Dependencies**

This parameter appears only when you set **Pad over** to **Specified dimensions** and the **Specify** parameter to **Pad size**.

**Pad size at end** — Specify number of values to add to end of input signal

0 | scalar | vector

Specify how many values to add to the end of the input signal along the specified dimension(s). This parameter must be a scalar or a vector with the same number of elements as the **Dimensions to pad** parameter. Each element in the **Pad size at end** parameter gives the pad length at the end of the corresponding dimension in the **Dimensions to pad** parameter. Values of this parameter must be nonnegative integers.

**Dependencies**

This parameter appears only when you set **Pad over** to `Specified dimensions` and the **Specify** parameter to `Pad size`.

**Pad signal at** — Specify where to pad or truncate signal

End (default) | Beginning | Beginning and end

Specify whether to pad or truncate the signal at the `Beginning`, `End`, or `Beginning and end` of the specified dimension(s). When you select `Beginning and end`, half the pad length is added to the beginning of the signal and the other half is added to the end of the signal. For an odd pad length, the extra value is added to the end of the signal. When you specify to truncate the signal in this mode, an equal number of values are truncated from the beginning and the end of the signal. In the case of an odd truncation length, the extra value is removed from the end of the signal.

**Dependencies**

This parameter appears only when you set **Pad over** to `Columns`, `Rows`, and `Columns and rows`.

**Action when truncation occurs** — Action to take when truncation occurs

None | Warning | Error

Specify the action the block takes when it truncates the input.

- `None` -- The block does not notify when the input is truncated.
- `Warning` -- The block displays a warning when the input is truncated.
- `Error` -- The block displays an error message and terminates the simulation when the input is truncated.

**Dependencies**

This parameter appears only when you set **Pad over** to `Columns`, `Rows`, and `Columns and rows`.

**Block Characteristics**

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | Boolean   double   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no  |
| <b>Multidimensional Signals</b> | no  |
| <b>Variable-Size Signals</b>    | no  |
| <b>Zero-Crossing Detection</b>  | no  |

**Version History**

Introduced before R2006a

---

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Generated code relies on the `memcpy` or `memset` function (`string.h`) under certain conditions.

### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

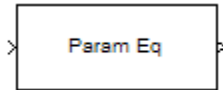
## See Also

### Blocks

Concatenate | Variable Selector | Repeat | Submatrix | Upsample

## Parametric Equalizer

Design parametric equalizer



## Compatibility

---

**Note** The Parametric Equalizer block has been replaced by the Parametric EQ Filter block. Existing instances of the Parametric Equalizer block will continue to operate. For new models, use the Parametric EQ Filter block.

---

## Library

Filtering / Filter Designs

dspfdesign

## Description

This block brings the filter design capabilities of the `filterBuilder` function to the Simulink environment.

## Dialog Box

See “Parametric Equalizer Filter Design — Main Pane” on page 5-639 for more information about the parameters of this block. The **Data Types** and **Code Generation** panes are not available for blocks in the DSP System Toolbox Filter Designs library.

Parameters of this block that do not change filter order or structure are tunable.



Function Block Parameters: Parametric Equalizer

Parametric Equalizer

Design a parametric equalizer.

[View Filter Response](#)

Filter specifications

Order mode:  Order:

Frequency specifications

Frequency constraints:

Frequency units:  Input Fs:

Center frequency:  Bandwidth

Passband width:

Gain specifications

Gain constraints:

Gain units:

Reference gain:  Center frequency gain:

Bandwidth gain:  Passband gain:

Algorithm

Design method:

Scale SOS filter coefficients to reduce chance of overflow

Filter Implementation

Structure:

Use basic elements to enable filter customization

Optimize for unit scale values

Input processing:

Use symbolic names for coefficients

**View filter response**

This button opens the Filter Visualization Tool (**FVTool**) from the Signal Processing Toolbox product. You can use the tool to display:

- Magnitude response, phase response, and group delay in the frequency domain.
- Impulse response and step response in the time domain.
- Pole-zero information.

The tool also helps you evaluate filter performance by providing information about filter order, stability, and phase linearity. For more information on FVTool, see the Signal Processing Toolbox documentation.

**Filter Specifications****Order mode**

Select **Minimum** to design a minimum order filter that meets the design specifications, or **Specify** to enter a specific filter order. The order mode also affects the possible frequency constraints, which in turn limit the gain specifications. For example, if you specify a **Minimum** order filter, the available frequency constraints are:

- Center frequency, bandwidth, passband width
- Center frequency, bandwidth, stopband width

If you select **Specify**, the available frequency constraints are:

- Center frequency, bandwidth
- Center frequency, quality factor
- Shelf type, cutoff frequency, quality factor
- Shelf type, cutoff frequency, shelf slope parameter
- Low frequency, high frequency

**Order**

Specify the filter order. This parameter is enabled only if the **Order mode** is set to **Specify**.

**Frequency specifications**

Depending on the filter order, the possible frequency constraints change. Once you choose the frequency constraints, the input boxes in this area change to reflect the selection.

**Frequency constraints**

Select the specification to represent the frequency constraints. The following options are available:

- Center frequency, bandwidth, passband width (available for minimum order only)
- Center frequency, bandwidth, stopband width (available for minimum order only)
- Center frequency, bandwidth (available for a specified order only)
- Center frequency, quality factor (available for a specified order only)
- Shelf type, cutoff frequency, quality factor (available for a specified order only)
- Shelf type, cutoff frequency, shelf slope parameter (available for a specified order only)

- Low frequency, high frequency (available for a specified order only)

### Frequency units

Select the frequency units from the available drop down list (Normalized, Hz, kHz, MHz, GHz). If Normalized is selected, then the **Input Fs** box is disabled for input.

### Input Fs

Enter the input sampling frequency. This input box is disabled for input if Normalized is selected in the **Frequency units** input box.

### Center frequency

Enter the center frequency in the units specified by the value in **Frequency units**.

### Bandwidth

The bandwidth determines the frequency points at which the filter magnitude is attenuated by the value specified as the **Bandwidth gain** in the **Gain specifications** section. By default, the **Bandwidth gain** defaults to  $\text{db}(\text{sqrt}(.5))$ , or -3 dB relative to the center frequency. The **Bandwidth** property only applies when the **Frequency constraints** are: Center frequency, bandwidth, passband width, Center frequency, bandwidth, stopband width, or Center frequency, bandwidth.

### Passband width

The passband width determines the frequency points at which the filter magnitude is attenuated by the value specified as the **Passband gain** in the **Gain specifications** section. This option is enabled only if the filter is of minimum order, and the frequency constraint selected is Center frequency, bandwidth, passband width.

### Stopband width

The stopband width determines the frequency points at which the filter magnitude is attenuated by the value specified as the **Stopband gain** in the **Gain specifications** section. This option is enabled only if the filter is of minimum order, and the frequency constraint selected is Center frequency, bandwidth, stopband width.

### Low frequency

Enter the low frequency cutoff. This option is enabled only if the filter order is user specified and the frequency constraint selected is Low frequency, high frequency. The filter magnitude is attenuated by the amount specified in **Bandwidth gain**.

### High frequency

Enter the high frequency cutoff. This option is enabled only if the filter order is user specified and the frequency constraint selected is Low frequency, high frequency. The filter magnitude is attenuated by the amount specified in **Bandwidth gain**.

## Gain Specifications

Depending on the filter order and frequency constraints, the possible gain constraints change. Also, once you choose the gain constraints the input boxes in this area change to reflect the selection.

### Gain constraints

Select the specification array to represent gain constraints, and remember that not all of these options are available for all configurations. The following is a list of all available options:

- Reference, center frequency, bandwidth, passband
- Reference, center frequency, bandwidth, stopband

- Reference, center frequency, bandwidth, passband, stopband
- Reference, center frequency, bandwidth

**Gain units**

Specify the gain units either dB or squared. These units are used for all gain specifications in the dialog box.

**Reference gain**

The reference gain determines the level to which the filter magnitude attenuates in **Gain units**. The reference gain is a *floor* gain for the filter magnitude response. For example, you may use the reference gain together with the **Center frequency gain** to leave certain frequencies unattenuated (reference gain of 0 dB) while boosting other frequencies.

**Bandwidth gain**

Specifies the gain in **Gain units** at which the bandwidth is defined. This property applies only when the **Frequency constraints** specification contains a bandwidth parameter, or is Low frequency, high frequency.

**Center frequency gain**

Specify the center frequency in **Gain units**

**Passband gain**

The passband gain determines the level in **Gain units** at which the passband is defined. The passband is determined either by the **Passband width** value, or the **Low frequency** and **High frequency** values in the **Frequency specifications** section.

**Stopband gain**

The stopband gain is the level in **Gain units** at which the stopband is defined. This property applies only when the **Order mode** is minimum and the **Frequency constraints** are Center frequency, bandwidth, stopband width.

**Boost/cut gain**

The boost/cut gain applies only when the designing a shelving filter. Shelving filters include the **Shelf** type parameter in the **Frequency constraints** specification. The gain in the passband of the shelving filter is increased by **Boost/cut gain** dB from a *floor* gain of 0 dB.

**Algorithm****Design method**

Select the design method from the drop-down list. Different methods are available depending on the chosen filter constraints.

**Scale SOS filter coefficients to reduce chance of overflow**

Select the check box to scale the filter coefficients.

**Filter Implementation****Structure**

Specify filter structure. Choose from:

- Direct-form I SOS
- Direct-form II SOS
- Direct-form I transposed SOS

- Direct-form II transposed SOS

### Use basic elements to enable filter customization

Select this check box to implement the filter as a subsystem of basic Simulink blocks. Clear the check box to implement the filter as a high-level subsystem. By default, this check box is cleared.

The high-level implementation provides better compatibility across various filter structures, especially filters that would contain algebraic loops when constructed using basic elements. On the other hand, using basic elements enables the following optimization parameters:

- **Optimize for zero gains** — Terminate chains that contain Gain blocks with a gain of zero.
- **Optimize for unit gains** — Remove Gain blocks that scale by a factor of one.
- **Optimize for delay chains** — Substitute delay chains made up of  $n$  unit delays with a single delay by  $n$ .
- **Optimize for negative gains** — Use subtraction in Sum blocks instead of negative gains in Gain blocks.

### Optimize for unit-scale values

Select this check box to scale unit gains between sections in SOS filters. This parameter is available only for SOS filters.

### Input processing

Specify how the block should process the input. The available options may vary depending on the settings of the **Filter Structure** and **Use basic elements for filter customization** parameters. You can set this parameter to one of the following options:

- **Columns as channels (frame based)** — When you select this option, the block treats each column of the input as a separate channel.
- **Elements as channels (sample based)** — When you select this option, the block treats each element of the input as a separate channel.

---

**Note** The Inherited (this choice will be removed – see release notes) option will be removed in a future release. See “Frame-Based Processing” in the *DSP System Toolbox Release Notes* for more information.

---

### Rate options

When the **Filter type** parameter specifies a multirate filter, select the rate processing rule for the block from following options:

- **Enforce single-rate processing** — When you select this option, the block maintains the sample rate of the input.
- **Allow multirate processing** — When you select this option, the block adjusts the rate at the output to accommodate an increased or reduced number of samples. To select this option, you must set the **Input processing** parameter to **Elements as channels (sample based)**.

### Use symbolic names for coefficients

Select this check box to enable the specification of coefficients using MATLAB variables. The available coefficient names differ depending on the filter structure. Using symbolic names allows tuning of filter coefficients in generated code. By default, this check box is cleared.

## Supported Data Types

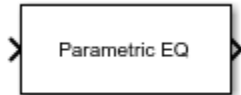
| Port   | Supported Data Types  |
|--------|---|
| Input  | <ul style="list-style-type: none"><li>• Double-precision floating point</li><li>• Single-precision floating point</li></ul> |
| Output | <ul style="list-style-type: none"><li>• Double-precision floating point</li><li>• Single-precision floating point</li></ul> |

## Version History

Introduced in R2007a

## Parametric EQ Filter (Obsolete)

(Removed) Model second-order parametric equalizer filter




---

**Note** The Parametric EQ Filter block requires Audio Toolbox. Existing models using the Parametric EQ Filter block continue to run. For new models, use the Parametric EQ block from Audio Toolbox.

---

### Library

dspobslib

### Description

The Parametric EQ Filter block filters each channel of the input signal over time using a specified center frequency, bandwidth, and peak (dip) gain. This block offers tunable filter design parameters, which enable you to tune the filter characteristics while the simulation is running.

The block designs the filter according to the filter parameters set in the block dialog box. The output port properties, such as datatype, complexity, and dimension, are identical to the input port properties.

Each column of the input signal is treated as a separate channel. If the input is a two-dimensional signal, the first dimension represents the channel length (or frame size) and the second dimension represents the number of channels. If the input is a one-dimensional signal, then it is interpreted as a single channel.

This block supports variable-size input, enabling you to change the channel length during simulation. To enable variable-size input, clear the **Inherit sample rate from input** check box. The number of channels must remain constant.

### Parameters

#### Filter specification

Parameters or coefficients used to design the filter, specified as one of the following:

- Bandwidth and center frequency (default) — Design the filter using **Filter bandwidth (Hz)**, **Equalizer center frequency (Hz)**, and **Gain (dB)**.
- Coefficients — Design the filter using **Bandwidth coefficient**, **Center frequency coefficient**, and **Gain (Linear Units)**.
- Quality factor and center frequency — Design the filter using **Equalizer center frequency (Hz)**, **Gain (dB)**, and **Quality factor**.

This parameter is nontunable.

**Specify bandwidth from input port**

When you select this check box, the filter bandwidth is input through the **BW** port. When you clear this check box, the filter bandwidth is specified on the block dialog through the **Filter bandwidth (Hz)** parameter.

This parameter applies when you set **Filter specification** to Bandwidth and center frequency.

**Filter bandwidth (Hz)**

Bandwidth of the filter, specified as a finite positive numeric scalar that is less than half the sample rate of the input signal. This parameter applies when you set **Filter specification** to Bandwidth and center frequency and clear the **Specify bandwidth from input port** parameter. The default is 2205. This parameter is tunable.

**Specify center frequency from input port**

When you select this check box, the equalizer center frequency is input through the **Fc** port. When you clear this check box, the equalizer center frequency is specified on the block dialog through the **Equalizer center frequency (Hz)** parameter.

This parameter applies when you set **Filter specification** to Bandwidth and center frequency or Quality factor and center frequency.

**Equalizer center frequency (Hz)**

Center frequency of the filter, specified as a finite positive scalar that is less than half the sample rate of the input signal. This parameter applies when you set **Filter specification** to Bandwidth and center frequency or Quality factor and center frequency, and clear the **Specify center frequency from input port** parameter. The default is 11025. This parameter is tunable.

**Specify gain (dB) from input port**

When you select this check box, the peak or dip gain of the filter in dB is input through the **PGaindB** port. When you clear this check box, the filter gain is specified on the block dialog through the **Gain (dB)** parameter.

This parameter applies when you set **Filter specification** to Bandwidth and center frequency or Quality factor and center frequency.

**Gain (dB)**

Peak or dip gain of the filter, specified as a real scalar in dB. A value greater than zero corresponds to a peak. A value less than zero corresponds to a dip. This parameter applies when you set **Filter specification** to Bandwidth and center frequency or Quality factor and center frequency, and clear the **Specify gain (dB) from input port** parameter. The default is 6.0206. This parameter is tunable.

**Specify bandwidth coefficient from input port**

When you select this check box, the bandwidth coefficient is input through the **BWCoeff** port. When you clear this check box, the bandwidth coefficient is specified on the block dialog through the **Bandwidth coefficient** parameter.

This parameter applies when you set **Filter specification** to Coefficients.

**Bandwidth coefficient**

Coefficient that determines the filter bandwidth, specified as a finite numeric scalar in the range [-1 1].



- -1 corresponds to the maximum bandwidth (one-fourth the sample rate of the input signal).
- 1 corresponds to the minimum bandwidth (0 Hz, that is, an allpass filter).

This parameter applies when you set **Filter specification** to **Coefficients** and clear the **Specify bandwidth coefficient from input port** parameter. The default is 0.72654. This parameter is tunable.

#### **Specify center frequency coefficient from input port**

When you select this check box, the center frequency coefficient is input through the **FcCoeff** port. When you clear this check box, the center frequency coefficient is specified on the block dialog through the **Center frequency coefficient** parameter.

This parameter applies when you set **Filter specification** to **Coefficients**.

#### **Center frequency coefficient**

Coefficient that determines the center frequency of the filter, specified as a finite numeric scalar in the range [-1 1].

- -1 corresponds to the minimum center frequency (0 Hz).
- 1 corresponds to the maximum center frequency (half the sample rate of the input signal).

This parameter applies when you set **Filter specification** to **Coefficients** and clear the **Specify center frequency coefficient from input port** parameter. The default is 0, which corresponds to one-fourth the sample rate of the input signal. This parameter is tunable.

#### **Specify gain from input port**

When you select this check box, the peak or dip gain of the filter in linear units is input through the **PGain** port. When you clear this check box, the filter gain is specified on the block dialog through the **Gain (Linear Units)** parameter.

This parameter applies when you set **Filter specification** to **Coefficients**.

#### **Gain (Linear Units)**

Peak or dip gain of the filter, specified as a real positive scalar in linear units. A value greater than one boosts the input signal. A value less than one attenuates the input signal. This parameter applies when you set **Filter specification** to **Coefficients** and clear the **Specify gain from input port** parameter. The default is 2. This parameter is tunable.

#### **Specify quality factor from input port**

When you select this check box, the quality factor is input through the **Q** port. When you clear this check box, the quality factor is specified on the block dialog through the **Quality factor** parameter.

This parameter applies when you set **Filter specification** to **Quality factor** and center frequency.

#### **Quality factor**

Quality factor of the filter, specified as a real positive scalar. The quality factor is defined as **Equalizer center frequency (Hz) / Filter bandwidth (Hz)**. A higher quality factor corresponds to a narrower peak or dip. This parameter applies when you set **Filter specification** to **Quality factor** and center frequency and clear the **Specify quality factor from input port** parameter. The default is 5. This parameter is tunable.

#### **Inherit sample rate from input**

When you select this check box, the block's sample rate is computed as  $N/T_s$ , where  $N$  is the frame size of the input signal, and  $T_s$  is the sample time of the input signal. When you clear this

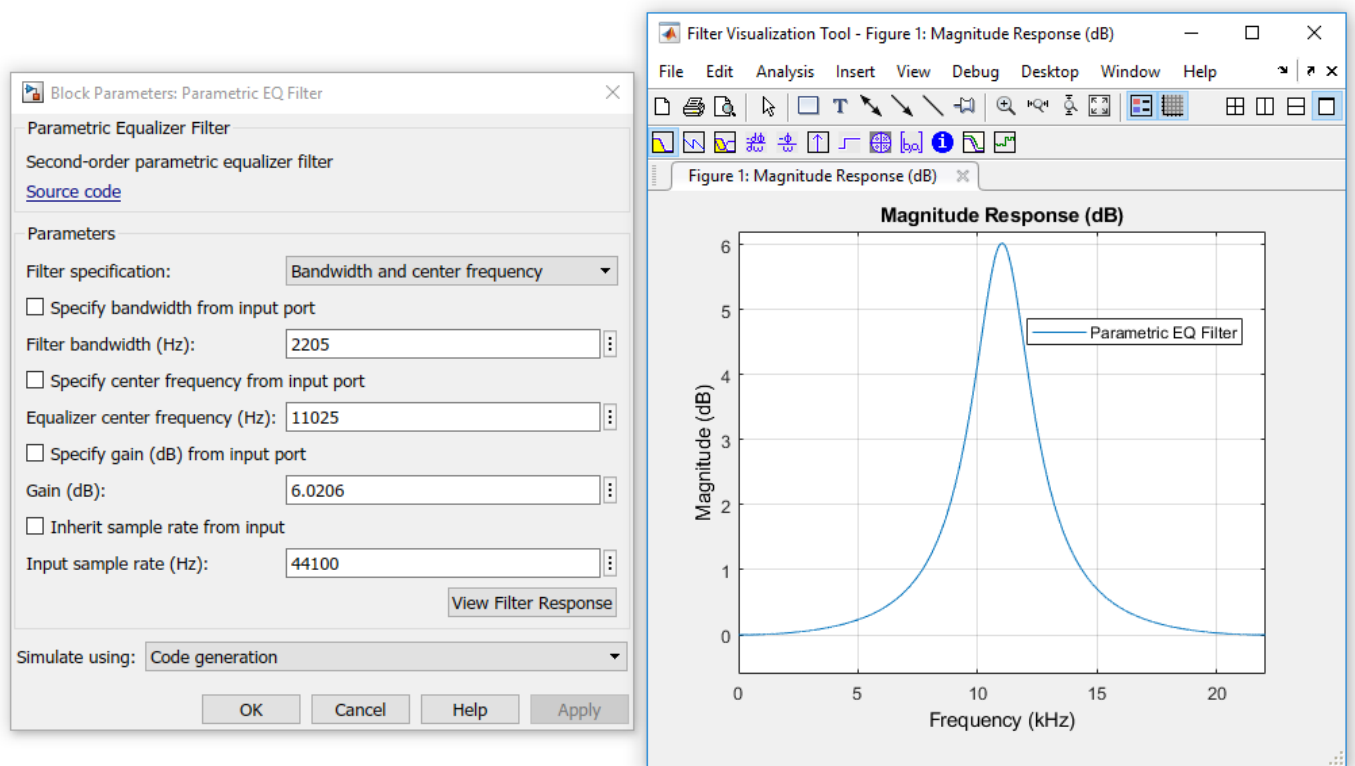
check box, the block sample rate is the value specified in **Input sample rate (Hz)**. By default, this check box is selected.

### Input sample rate (Hz)

Sample rate of the input signal, specified as a positive scalar value. The default is 44100. This parameter applies when you clear the **Inherit sample rate from input** check box. This parameter is nontunable.

### View Filter Response

Opens the Filter Visualization Tool FVTool and displays the magnitude/phase response of the Parametric EQ Filter. The response is based on the block dialog box parameters. Changes made to these parameters update FVTool.



To update the magnitude response while FVTool is running, modify the dialog box parameters and click **Apply**.

### Simulate using

Type of simulation to run. You can set this parameter to:

- Code generation (default)

Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than Interpreted execution.

- Interpreted execution

Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than Code generation.

## Supported Data Types

| Port   | Supported Data Types   |
|--------|--|
| Input  | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> </ul> |
| Output | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> </ul> |

## References

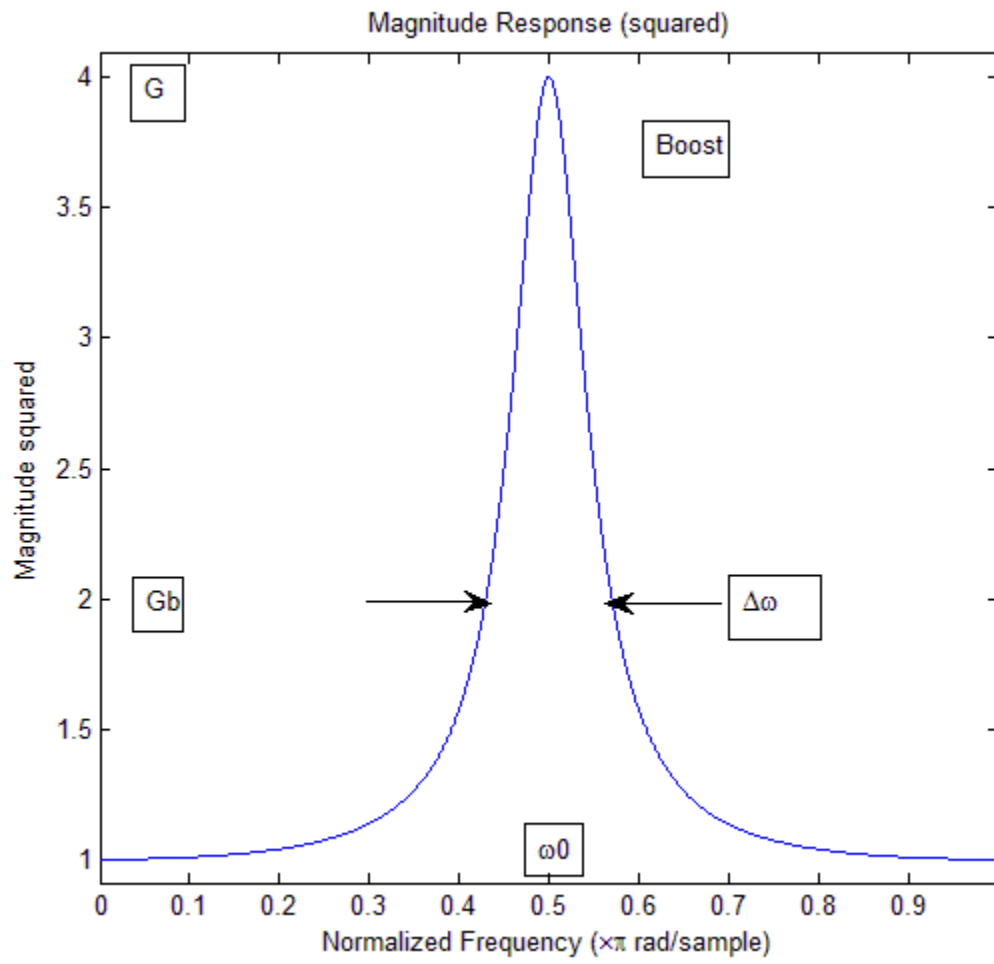
[1] Orfanidis, Sophocles J. *Introduction to Signal Processing*. Upper Saddle River, NJ: Prentice-Hall, 1996.

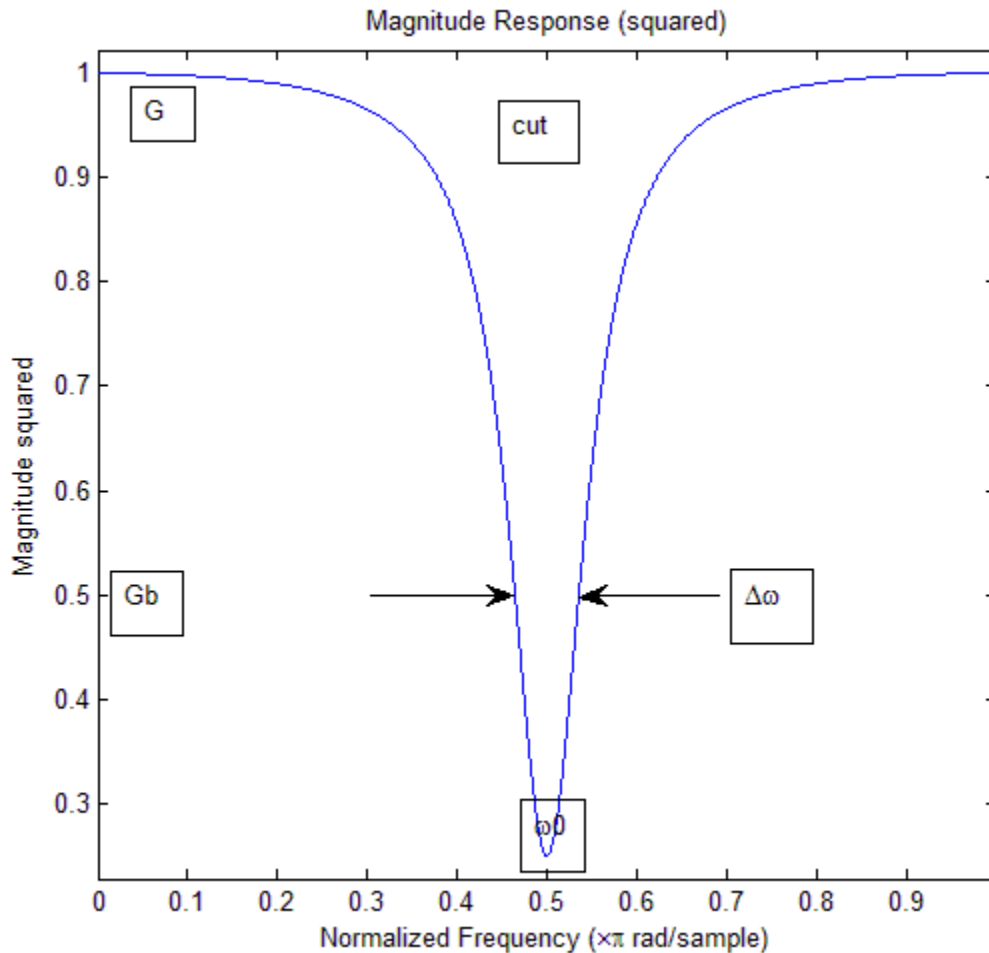
## Algorithms

The parametric equalizer is formed by a linear combination of a peak and a notch filter. See the **Algorithm** section of `dsp.NotchPeakFilter` for details.

$$H(z) = H_{notch}(z) + GH_{peak}(z)$$

Here is a graph of the two cases (boost and cut) of the magnitude squared of the transfer functions:





The transfer function can be written as:

$$H(z) = \frac{\left(\frac{1+G\gamma}{1+\gamma}\right) - 2\left(\frac{\cos\omega_0}{1+\gamma}\right)z^{-1} + \left(\frac{1-G\gamma}{1+\gamma}\right)z^{-2}}{1 - 2\left(\frac{\cos\omega_0}{1+\gamma}\right)z^{-1} + \left(\frac{1-\gamma}{1+\gamma}\right)z^{-2}}$$

where

$$\gamma = \tan\left(\frac{\Delta\omega}{2}\right)$$

and

$$G_B^2 = \frac{1+G^2}{2}$$

$G$  is the parametric equalizer gain, and  $G_B$  is the bandwidth gain, that is, the gain level at which the bandwidth  $\Delta\omega$  is measured.

The `dsp.NotchPeakFilter` that does most of the work is implemented in a decoupled way so that the center frequency can be tuned independently from the bandwidth. Note that the  $Q$  factor is defined as center frequency/bandwidth.

## **Version History**

**Introduced in R2015a**

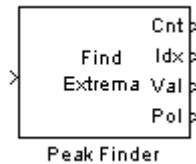
### **Extended Capabilities**

#### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

# Peak Finder

Determine whether each value of input signal is local minimum or maximum



## Libraries:

DSP System Toolbox / Signal Operations

## Description

The Peak Finder block counts the number of local extrema in each column of a real-valued input signal. The block outputs the number of local extrema at the **Cnt** port. You can also configure the block to output the extrema indices, the extrema values, and a binary value indicating if the extrema are maxima or minima. To qualify as an extremum, a point has to be larger (or smaller) than both its neighboring points, so the block does not consider end points when counting extrema.

## Ports

### Input

#### Port\_1 — Data input

column vector of at least three rows | matrix with at least three rows

The Peak Finder blocks accepts real-valued vector or matrix inputs with at least three rows.

Example: [9 6 10 3 5 5 0 12; 9 6 1 13 4 1 0 12]'

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

### Output

#### Cnt — Number of peak values

scalar | row vector

Number of peak values the block detects in the input signal, returned as a scalar or a row vector of length equal to the number of channels in the data input.

Example: [4 3]

Data Types: uint32

#### Idx — Indices of the peak values

column vector | matrix

Indices of the peak values in the input signal, returned as a column vector or matrix. The size of the peak indices output is the same as that of the input. All nonzero elements represent peak indices.

If you set **Index base** to Zero, the block outputs zero-based extrema indices. If you set **Index base** to One, the block outputs one-based extrema indices.

Example: [1 2 3 6 0 0 0 0 0 0; 2 3 6 0 0 0 0 0 0 0]'

### Dependencies

To enable this output port, select the **Output peak indices** parameter.

Data Types: uint32

### Val — Peak values

column vector | matrix

Peak values in the input signal, returned as a column vector or matrix.

Example: [6 10 3 0 0 0 0 0 0 0; 1 13 0 0 0 0 0 0 0 0]'

### Dependencies

To enable this output port, select the **Output peak values** parameter.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

### Pol — Peak value polarity

column vector | matrix

Peak value polarity in input signal, returned as a column vector or a matrix of logical 1s and 0s. If the signal value is a maximum, the block outputs 1 at this port. If the signal value is a minimum, the block outputs 0 at this port.

Example: [0 1 0 0 0 0 0 0 0 0; 0 1 0 0 0 0 0 0 0 0]'

### Dependencies

To enable this port, select either **Output peak indices** or **Output peak values**, and set the **Peak type(s)** parameter to Maxima and Minima.

Data Types: Boolean

## Parameters

**Peak type(s)** — Type of peaks to identify

Maxima and Minima (default) | Maxima | Minima

Specify the type of peaks to identify as one of these:

- **Maxima** -- If you select **Ignore peaks within threshold of neighboring values**, the block identifies the current value as a maximum if  $(current - previous) > threshold$  and  $(current - next) > threshold$ . Specify *threshold* in the **Threshold** parameter. If you do not select **Ignore peaks within threshold of neighboring values**, the block identifies a maxima if the current value is larger than both its neighboring points.
- **Minima** -- If you select **Ignore peaks within threshold of neighboring values**, the block identifies the current value as a minimum if  $(current - previous) < -threshold$  and  $(current - next) < -threshold$ . If you do not select **Ignore peaks within threshold of neighboring values**, the block identifies a minima if the current value is smaller than both its neighboring points.
- **Maxima and Minima** -- The block identifies the maxima and the minima points.



**Index base** — Index base

Zero (default) | One

Specify the base of the extrema indices as either:

- Zero -- The first index starts with 0. For example, consider an input vector  $[-1.5, 0.5, 0]$ . The peak value of this vector is 0.5 and the index of this peak value is 1.
- One -- In this setting, the first index starts with 1. The index of the peak value in  $[-1.5, 0.5, 0]$  is 2.

**Output peak indices** — Enable output of peak indices

off (default) | on

Select this check box if you want the block to output the extrema indices at the **Idx** port.**Output peak values** — Enable output of peak values

off (default) | on

Select this parameter if you want the block to output the extrema values at the **Val** port.**Maximum number of peaks to find** — Maximum number of peak values to identify

10 (default) | integer greater than or equal to 1

Enter the number of extrema to look for in each input signal as an integer greater than or equal to 1. The block stops searching the input signal for extrema once the maximum number of extrema has been found.

**Ignore peaks within threshold of neighboring values** — Ignore peaks below a threshold

off (default) | on

Select this parameter if you want to eliminate the detection of peaks whose amplitudes are within a specified threshold of neighboring values. This feature allows the block to ignore noise by focusing on values above the specified threshold.

When you clear this parameter, the block identifies the current value as a peak if the current value is larger (or smaller) than both its neighboring points. When you select this parameter, the block identifies the current value as a maximum if  $(current - previous) > threshold$  and  $(current - next) > threshold$ . The block identifies the current value as a minimum if  $(current - previous) < -threshold$  and  $(current - next) < -threshold$ .

**Threshold** — Threshold below which peaks are ignored

0 (default) | positive real scalar | vector

Specify the threshold below which peaks are ignored as a real scalar greater than or equal to 0 or a vector with all elements greater than or equal to 0. The length of the vector must be equal to the number of channels.

This property identifies the current input value to be a maximum if  $(current - previous) > threshold$  and  $(current - next) > threshold$ . The current value is a minimum if  $(current - previous) < -threshold$  and  $(current - next) < -threshold$ .

### Dependencies

To enable this parameter, select the **Ignore peaks within threshold of neighboring values** parameter.

**Saturate on integer overflow** — Method of overflow action

off (default) | on

When you select this parameter, the block saturates the result of its fixed-point operations. When you clear this parameter, the block wraps the result of its fixed-point operations. For details on `saturate` and `wrap`, see `overflow mode for fixed-point operations`.

### Dependencies

To enable this parameter, select the **Ignore peaks within threshold of neighboring values** parameter.

## Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | double   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no                                      |
| <b>Multidimensional Signals</b> | no                                      |
| <b>Variable-Size Signals</b>    | no                                      |
| <b>Zero-Crossing Detection</b>  | no                                      |

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Generated code relies on the `memcpy` or `memset` function (`string.h`) under certain conditions.

### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

## See Also

### Blocks

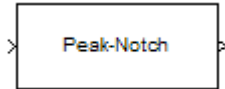
Maximum | Minimum

**Topics**

“Find Peak Values in a Signal”

## Peak-Notch Filter

Design peak or notch filter



## Compatibility

---

**Note** The Peak-Notch Filter block has been replaced by the Notch-Peak Filter block. Existing instances of the Peak-Notch Filter block will continue to operate. For new models, use the Notch-Peak Filter block.

---

## Library

Filtering / Filter Designs

dspfdesign

## Description

This block brings the filter design capabilities of the `filterBuilder` function to the Simulink environment.

## Dialog Box

See “Peak/Notch Filter Design — Main Pane” on page 5-642 for more information about the parameters of this block. The **Data Types** and **Code Generation** panes are not available for blocks in the DSP System Toolbox Filter Designs library.

Parameters of this block that do not change filter order or structure are tunable.

Function Block Parameters: Peak-Notch Filter

Peak/Notch Filter

Design a peak or notch filter.

View Filter Response

Filter specifications

Response: Peak Order: 6

Frequency specifications

Frequency constraints: Center frequency and quality factor

Frequency units: Normalized (0 to 1) Input Fs: 2

Center frequency: 0.5 Quality factor: 2.5

Magnitude specifications

Magnitude constraints: Unconstrained

Algorithm

Design method: Butterworth

Scale SOS filter coefficients to reduce chance of overflow

Filter Implementation

Structure: Direct-form II SOS

Use basic elements to enable filter customization

Optimize for unit scale values

Input processing: Columns as channels (frame based)

Use symbolic names for coefficients

OK Cancel Help Apply

### View filter response

This button opens the Filter Visualization Tool (**FVTool**) from the Signal Processing Toolbox product. You can use the tool to display:

- Magnitude response, phase response, and group delay in the frequency domain.

- Impulse response and step response in the time domain.
- Pole-zero information.

The tool also helps you evaluate filter performance by providing information about filter order, stability, and phase linearity. For more information on FVTool, see the Signal Processing Toolbox documentation.

### **Filter Specifications**

In this area you can specify whether you want to design a peaking filter or a notching filter, as well as the order of the filter.

#### **Response**

Select **Peak** or **Notch** from the drop-down list. The rest of the parameters that specify are equivalent for either filter type.

#### **Order**

Enter the filter order. The order must be even.

### **Frequency Specifications**

This group of parameters allows you to specify frequency constraints and units.

#### **Frequency Constraints**

Select the frequency constraints for filter specification. There are two choices as follows:

- Center frequency and quality factor
- Center frequency and bandwidth

#### **Frequency units**

The frequency units are normalized by default. If you specify units other than normalized, the block assumes that you wish to specify an input sampling frequency, and enables this input box. The choice of frequency units are: Normalized (0 to 1), Hz, kHz, MHz, GHz.

#### **Input Fs**

This input box is enabled if **Frequency units** other than Normalized (0 to 1) are specified. Enter the input sampling frequency.

#### **Center frequency**

Enter the center frequency in the units specified previously.

#### **Quality Factor**

This input box is enabled only when Center frequency and quality factor is chosen for the **Frequency Constraints**. Enter the quality factor.

#### **Bandwidth**

This input box is enabled only when Center frequency and bandwidth is chosen for the **Frequency Constraints**. Enter the bandwidth.

### **Magnitude Specifications**

This group of parameters allows you to specify the magnitude constraints, as well as their values and units.

### Magnitude Constraints

Depending on the choice of constraints, the other input boxes are enabled or disabled. Select from four magnitude constraints available:

- Unconstrained
- Passband ripple
- Stopband attenuation
- Passband ripple and stopband attenuation

### Magnitude units

Select the magnitude units: either dB or squared.

### Apass

This input box is enabled if the magnitude constraints selected are Passband ripple or Passband ripple and stopband attenuation. Enter the passband ripple.

### Astop

This input box is enabled if the magnitude constraints selected are Stopband attenuation or Passband ripple and stopband attenuation. Enter the stopband attenuation.

### Algorithm

The parameters in this group allow you to specify the design method and structure of your filter.

### Design Method

Lists all design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter the methods available to design filters changes as well.

### Scale SOS filter coefficients to reduce chance of overflow

Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

### Filter Implementation

#### Structure

Specify filter structure. Choose from:

- Direct-form I SOS
- Direct-form II SOS
- Direct-form I transposed SOS
- Direct-form II transposed SOS

#### Use basic elements to enable filter customization

Select this check box to implement the filter as a subsystem of basic Simulink blocks. Clear the check box to implement the filter as a high-level subsystem. By default, this check box is cleared.

The high-level implementation provides better compatibility across various filter structures, especially filters that would contain algebraic loops when constructed using basic elements. On the other hand, using basic elements enables the following optimization parameters:

- **Optimize for zero gains** — Terminate chains that contain Gain blocks with a gain of zero.
- **Optimize for unit gains** — Remove Gain blocks that scale by a factor of one.
- **Optimize for delay chains** — Substitute delay chains made up of  $n$  unit delays with a single delay by  $n$ .
- **Optimize for negative gains** — Use subtraction in Sum blocks instead of negative gains in Gain blocks.

### Optimize for unit-scale values

Select this check box to scale unit gains between sections in SOS filters. This parameter is available only for SOS filters.

### Input processing

Specify how the block should process the input. The available options may vary depending on the settings of the **Filter Structure** and **Use basic elements for filter customization** parameters. You can set this parameter to one of the following options:

- **Columns as channels (frame based)** — When you select this option, the block treats each column of the input as a separate channel.
- **Elements as channels (sample based)** — When you select this option, the block treats each element of the input as a separate channel.

---

**Note** The **Inherited (this choice will be removed – see release notes)** option will be removed in a future release. See “Frame-Based Processing” in the *DSP System Toolbox Release Notes* for more information.

---

### Rate options

When the **Filter type** parameter specifies a multirate filter, select the rate processing rule for the block from following options:

- **Enforce single-rate processing** — When you select this option, the block maintains the sample rate of the input.
- **Allow multirate processing** — When you select this option, the block adjusts the rate at the output to accommodate an increased or reduced number of samples. To select this option, you must set the **Input processing** parameter to **Elements as channels (sample based)**.

### Use symbolic names for coefficients

Select this check box to enable the specification of coefficients using MATLAB variables. The available coefficient names differ depending on the filter structure. Using symbolic names allows tuning of filter coefficients in generated code. By default, this check box is cleared.

## Supported Data Types

| Port  | Supported Data Types   |
|-------|--|
| Input | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> </ul> |



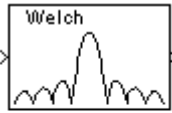
| Port   | Supported Data Types  |
|--------|---|
| Output | <ul style="list-style-type: none"><li data-bbox="418 300 834 331">• Double-precision floating point</li><li data-bbox="418 338 824 369">• Single-precision floating point</li></ul> |

## Version History

Introduced in R2007a

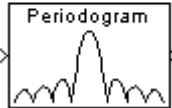
## Periodogram

Power spectral density or mean-square spectrum estimate using periodogram method



### Libraries:

DSP System Toolbox / Estimation / Power Spectrum Estimation



## Description

The Periodogram block estimates the power spectral density (PSD) or mean-square spectrum (MSS) of the input. The block uses the periodogram method and Welch's averaged, modified periodogram method. The block averages the squared magnitude of the FFT function computed over windowed sections of the input. It then normalizes the spectral average by the square of the sum of the window samples. For more information, see “Periodogram” and “Welch's Method”.

## Ports

### Input

**Port\_1** — Input signal

vector | matrix

Input signal, specified as a vector or a matrix.

The block treats  $M$ -by- $N$  matrix input as  $M$  sequential time samples from  $N$  independent channels.

Data Types: single | double

### Output

**Port\_1** — Output signal

vector | matrix

Output signal, returned as a vector or a matrix.

Each column of the output matrix contains the estimate of the power spectral density of the corresponding input column at  $N_{fft}$  equally spaced frequency points. The frequency points are in the range  $[0, F_s)$ , where  $F_s$  is the sampling frequency of the signal.

The block computes a separate estimate for each of the  $N$  independent channels and generates an  $N_{fft}$ -by- $N$  matrix output.

Data Types: single | double

## Parameters

### Measurement — Measurement type

Power spectral density (default) | Mean-square spectrum

Specify the type of measurement for the block to perform: Power spectral density or Mean-square spectrum.

**Tunable:** No

### Window — Window type

Hamming (default) | Bartlett | Blackman | Boxcar | Chebyshev | Hann | Hanning | Kaiser | Triang

Select the type of window to apply.

For details, see the Window Function block reference page.

**Tunable:** Yes

### Beta — Beta

5 (default) | scalar

Enter the  $\beta$  parameter for the Kaiser window. Increasing **Beta** widens the mainlobe and decreases the amplitude of the sidelobes in the displayed frequency magnitude response.

For more details, see the Window Function block reference page.

**Tunable:** Yes

### Dependencies

This parameter becomes visible if, for the Window parameter, you chose Kaiser.

### Window sampling — Window sampling

Symmetric (default) | Periodic

From the list, choose Symmetric or Periodic.

See the Window Function block reference page for more details.

**Tunable:** Yes

### Dependencies

This parameter becomes visible only if you set the Window parameter to one of Blackman, Hamming, Hann, or Hanning.

### FFT implementation — FFT implementation

Auto (default) | Radix-2 | FFTW

Set this parameter to `FFTW` to support an arbitrary length input signal. The block restricts generated code with an FFTW implementation to MATLAB host computers.

Set this parameter to `Radix-2` for bit-reversed processing, fixed or floating-point data, or for portable C-code generation by using Simulink Coder. The first dimension  $M$  of the input matrix must be a power of two. To work with other input sizes, use the Pad block to pad or truncate these dimensions to powers of two, or if possible, choose the FFTW implementation.

Set this parameter to `Auto` to enable the block choose the FFT implementation. For non-power-of-two transform lengths, the block restricts generated code to MATLAB host computers.

**Tunable:** No

**Stopband attenuation in dB** — Stopband attenuation in dB

50 (default) | scalar

Enter the level, in decibels (dB), of stopband attenuation,  $R_s$ , for the Chebyshev window.

**Tunable:** Yes

**Dependencies**

This parameter becomes visible if, for the `Window` parameter, you choose `Chebyshev`.

**Inherit FFT length from input dimensions** — Inherit FFT length

off (default) | on

When you select this check box, the block uses the input frame size as the number of data points,  $N_{fft}$ , on which to perform the FFT. To specify the number of points on which to perform the FFT, clear the **Inherit FFT length from input dimensions** check box. You can then specify a power of two FFT length by using the “FFT length” on page 2-0 parameter.

**Tunable:** No

**FFT length** — Number of data points

256 (default) | scalar

Enter the number of data points on which to perform the FFT,  $N_{fft}$ . When  $N_{fft}$  is larger than the input frame size, the block zero-pads each frame as needed. When  $N_{fft}$  is smaller than the input frame size, the block wraps each frame as needed.

When you set the `FFT implementation` parameter to `Radix-2`, this value must be a power of two.

**Tunable:** No

**Dependencies**

This parameter becomes visible only when you clear the “Inherit FFT length from input dimensions” on page 2-0 check box.

**Number of spectral averages** — Number of spectral averages

4 (default) | scalar

Specify the number of spectral to average. When you set this value to 1, the block computes the periodogram of the input. When you set this value greater 1, the block implements “Welch's Method” to compute a modified periodogram of the input.

**Tunable:** No

**Inherit sample time from input** — Inherit sample time from input

on (default) | off

If you select the **Inherit sample time from input** check box, the block computes the frequency data from the sample period of the input signal. For the block to produce valid output, the following conditions must hold:

- The input to the block is the original signal, with no samples added or deleted (by insertion of zeros, for example).
- The sample period of the time-domain signal in the simulation equals the sample period of the original time series.

If these conditions do not hold, clear the **Inherit sample time from input** check box. You can then specify a sample time by using the `Sample time of original time series` parameter.

#### Dependencies

This parameter becomes visible only if you set the `Measurement` parameter to `Power spectral density`.

**Sample time of original time series** — Sample time of original time series

1 (default) | scalar

Specify the sample time of the original time-domain signal.

**Tunable:** No

#### Dependencies

This parameter becomes visible only when you clear the `Inherit sample time from input` check box.

### Block Characteristics

|                                 |                 |
|---------------------------------|-----------------|
| <b>Data Types</b>               | double   single |
| <b>Multidimensional Signals</b> | No              |
| <b>Variable-Size Signals</b>    | No              |

### Version History

Introduced before R2006a

## References

- [1] FFTW ( <https://www.fftw.org> )
- [2] Frigo, M. and S. G. Johnson, "FFTW: An Adaptive Software Architecture for the FFT," *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing* , Vol. 3, 1998, pp. 1381-1384.
- [3] Oppenheim, A. V. and R. W. Schaffer. *Discrete-Time Signal Processing* . Englewood Cliffs, NJ: Prentice Hall, 1989.
- [4] Orfanidis, S. J. *Introduction to Signal Processing* . Englewood Cliffs, NJ: Prentice-Hall, 1995.
- [5] Proakis, J. and D. Manolakis. *Digital Signal Processing*. 3rd ed. Englewood Cliffs, NJ: Prentice-Hall, 1996.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Usage notes and limitations:

- Generated code relies on the `memcpy` or `memset` function (`string.h`) under certain conditions.
- When the following conditions apply, the executable generated from this block relies on prebuilt dynamic library files (`.dll` files) included with MATLAB:
  - **FFT implementation** is set to FFTW.
  - **Inherit FFT length from input dimensions** is cleared and **FFT length** is set to a value that is not a power of two.

Use the `packNGo` function to package the code generated from this block and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see "How To Run a Generated Executable Outside MATLAB".

- When the FFT length is a power of two, you can generate standalone C and C++ code from this block.

## See Also

### Blocks

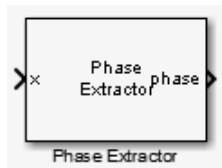
Burg Method | Inverse Short-Time FFT | Magnitude FFT | Short-Time FFT | Spectrum Analyzer | Window Function | Yule-Walker Method

### Topics

"Spectral Analysis"  
"Welch's Algorithm of Averaging Modified Periodograms"

# Phase Extractor

Extract the unwrapped phase of a complex input



## Libraries:

DSP System Toolbox / Signal Operations

## Description

The Phase Extractor block extracts the unwrapped phase of a complex input.

## Ports

### Input

**Input 1** — Input signal  
vector | matrix

Specify the input signal as a vector or a matrix. When the input is a matrix, the block treats each column of the signal as an independent channel. The first dimension is the length of the channel. The second dimension is the number of channels. The block treats one dimensional inputs as one channel.

Data Types: single | double

### Output

**Output 1** — Unwrapped phase  
vector | matrix

The block returns the unwrapped phase of the signal.

The block preserves the input size and dimension, and the output port rate equals the input port rate.

Data Types: single | double

## Parameters

**Unwrap phase only within the frame** — Unwrap phase only within the frame

off (default) | on

When you clear this check box, the block ignores boundaries between the input frames. When you select this check box, the block treats each frame of input data independently, and resets the initial phase value for each new input frame.

**Simulate using** — Type of simulation to run  
Code generation (default) | Interpreted execution

Specify the type of simulation to run as one of the following:

- **Code generation** -- Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than **Interpreted execution**.
- **Interpreted execution** -- Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than **Code generation**.

## Block Characteristics

|                                 |                 |
|---------------------------------|-----------------|
| <b>Data Types</b>               | double   single |
| <b>Direct Feedthrough</b>       | no              |
| <b>Multidimensional Signals</b> | no              |
| <b>Variable-Size Signals</b>    | yes             |
| <b>Zero-Crossing Detection</b>  | no              |

## Algorithms

Consider an input frame of length  $N$ :

$$\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{pmatrix}$$

The `step` method acts on this frame and produces this output:

$$\begin{pmatrix} \Phi_1 \\ \Phi_2 \\ \vdots \\ \Phi_N \end{pmatrix}$$

where:

$$\Phi_i = \Phi_{i-1} + \text{angle}(x_{i-1}^* x_i)$$

Here,  $i$  runs from 1 to  $N$ . The `angle` function returns the phase angle in radians.

If the input signal consists of multiple frames:

- If you set `TreatFramesIndependently` to `true`, the `step` method treats each frame independently. Therefore, in each frame, the `step` method calculates the phase using the preceding formula where:
  - $\Phi_0$  is 0.



- $x_0$  is 1.
- If you set `TreatFramesIndependently` to `false`, the `step` method ignores boundaries between frames. Therefore, in each frame, the `step` method calculates the phase using the preceding formula where:
  - $\Phi_0$  is the last unwrapped phase from the previous frame.
  - $x_0$  is the last sample from the previous frame.

## Version History

Introduced in R2014b

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

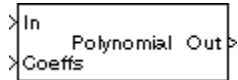
## See Also

### Objects

`dsp.PhaseExtractor`

# Polynomial Evaluation

Evaluate polynomial expression



## Library

Math Functions / Polynomial Functions

dsppolyfun

## Description

The Polynomial Evaluation block applies a polynomial function to the real or complex input at the In port.

`y = polyval(u)` % Equivalent MATLAB code

The Polynomial Evaluation block performs these types of operation more efficiently than the equivalent construction using Simulink Sum and Math Function blocks.

When you select the **Use constant coefficients** check box, you specify the polynomial expression in the **Constant coefficients** parameter. When you do not select **Use constant coefficients**, a variable polynomial expression is specified by the input to the **Coeffs** port. In both cases, the polynomial is specified as a vector of real or complex coefficients in order of descending exponents.

The table below shows some examples of the block's operation for various coefficient vectors.

| Coefficient Vector | Equivalent Polynomial Expression               |
|--------------------|--|
| [1 2 3 4 5]        | $y = u^4 + 2u^3 + 3u^2 + 4u + 5$               |
| [1 0 3 0 5]        | $y = u^4 + 3u^2 + 5$                           |
| [1 2+i 3 4-3i 5i]  | $y = u^4 + (2 + i)u^3 + 3u^2 + (4 - 3i)u + 5i$ |

Each element of a vector or matrix input to the In port is processed independently, and the output size is the same as the input.

## Parameters

### Use constant coefficients

Select to enable the **Constant coefficients** parameter and disable the **Coeffs** input port.

### Constant coefficients

Specify the vector of polynomial coefficients to apply to the input, in order of descending exponents. This parameter is enabled when you select the **Use constant coefficients** check box.

## Supported Data Types

- Double-precision floating point
- Single-precision floating point

## See Also

Least Squares Polynomial Fit

Math Function

Sum

polyval

DSP System Toolbox

Simulink

Simulink

MATLAB

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

## Polynomial Stability Test

Use Schur-Cohn algorithm to determine whether all roots of input polynomial are inside unit circle



### Library

Math Functions / Polynomial Functions

dsppolyfun

### Description

The Polynomial Stability Test block uses the Schur-Cohn algorithm to determine whether all roots of a polynomial are within the unit circle.

```
y = all(abs(roots(u)) < 1) % Equivalent MATLAB code
```

Each column of the M-by-N input matrix  $u$  contains M coefficients from a distinct polynomial,

$$f(x) = u_1x^{M-1} + u_2x^{M-2} + \dots + u_M$$

arranged in order of descending exponents,  $u_1, u_2, \dots, u_M$ . The polynomial has order M-1 and positive integer exponents.

Inputs to the block represent the polynomial coefficients as shown in the previous equation. The block always treats length-M unoriented vector input as an M-by-1 matrix.

The output is a 1-by-N matrix with each column containing the value 1 or 0. The value 1 indicates that the polynomial in the corresponding column of the input is stable; that is, the magnitudes of all solutions to  $f(x) = 0$  are less than 1. The value 0 indicates that the polynomial in the corresponding column of the input might be unstable; that is, the magnitude of at least one solution to  $f(x) = 0$  is greater than or equal to 1.

### Applications

This block is most commonly used to check the pole locations of the denominator polynomial,  $A(z)$ , of a transfer function,  $H(z)$ .

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_1 + b_2z^{-1} + \dots + b_mz^{-(m-1)}}{a_1 + a_2z^{-1} + \dots + a_nz^{-(n-1)}}$$

The poles are the  $n-1$  roots of the denominator polynomial,  $A(z)$ . When any poles are located outside the unit circle, the transfer function  $H(z)$  is unstable. As is typical in DSP applications, the transfer function above is specified in descending powers of  $z^{-1}$  rather than  $z$ .

## Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Boolean — Block outputs are always Boolean.

## See Also

Least Squares Polynomial Fit  
Polynomial Evaluation  
`polyfit`

DSP System Toolbox  
DSP System Toolbox  
MATLAB

## Version History

Introduced before R2006a

## Extended Capabilities

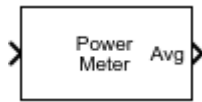
### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Generated code relies on the `memcpy` or `memset` function (`string.h`) under certain conditions.

## Power Meter

Measure power and the CCDF of the power of voltage signal in Simulink



**Libraries:**  
DSP System Toolbox / Statistics

### Description

The Power Meter block computes the power measurements of a voltage signal. When you select the **Compute CCDF** parameter, the block also calculates the complementary cumulative distribution function (CCDF) of the power of a voltage signal. The CCDF measurements that the block outputs are relative power and probability (in percentage). The power measurements include average power, peak power, and peak-to-average power ratio.

For more details on how the block computes the power measurements and the CCDF measurements, see “Algorithms” on page 2-1182.

### Ports

#### Input

**Port\_1** — Input signal  
vector | matrix

Specify the input signal in volts as a vector or a matrix. If  $x$  is a matrix, the block treats each column as an independent channel and computes the power measurement along each channel.

When the **Allow arbitrary frame length for fixed-size input signals** parameter appears and is not selected, and you input a fixed-size signal, the frame length must be a multiple of the hop size (window length – overlap length). In all other cases, the input frame length can be arbitrary.

The block accepts variable-size inputs (frame length changes during simulation). When you input a variable-size signal, the frame length of the signal can be arbitrary.

Data Types: single | double

**Rst** — Reset port  
scalar

Specify the boolean reset signal as a scalar. If the input on the **Rst** port is `true`, the block clears the internal histograms and statistics before processing the current input. If the input on the **Rst** port is `false`, the block computes the measurements from the start of the simulation or from the last reset.

#### Dependencies

To enable this port, select the **Compute CCDF** and **Reset port** parameters.

Data Types: Boolean

## Output

### Avg — Average power

scalar | vector | matrix

Average power of the voltage signal from the start of the simulation or from the last reset, returned as a scalar, vector, or a matrix. The block computes the average power along each channel in the units specified in the **Output power units** parameter. For details on how the block computes the average power, see “Average Power” on page 2-1183.

If you clear the **Compute CCDF** parameter, the block computes the moving average power using the “Sliding Window Method” on page 2-1182.

See this table for details on the output signal dimensions.

| Input Signal         | Input Dimensions  | Output Dimensions When Allow arbitrary frame length for fixed-size input signals Appears  | Output Dimensions When Allow arbitrary frame length for fixed-size input signals Does Not Appear |
|----------------------|---|---|--|
| Fixed-size signal    | $P$ -by- $M$ , where $M$ is a multiple of the hop size (window length – overlap length)     | $(P/\text{hop size})$ -by- $M$  | $P$ -by- $M$   |
| Fixed-size signal    | $P$ -by- $M$ , where $M$ is not a multiple of the hop size (window length – overlap length) | $\text{ceil}(P/\text{hop size})$ -by- $M$ when you select <b>Allow arbitrary frame length for fixed-size input signals</b><br><br>If you do not select <b>Allow arbitrary frame length for fixed-size input signals</b> , the block errors. | $P$ -by- $M$   |
| Variable-size signal | $P$ -by- $M$  | $\text{ceil}(P/\text{hop size})$ -by- $M$   | $P$ -by- $M$   |

When the output has an upper bound size of  $\text{ceil}(P/\text{hop size})$ -by- $M$ , during simulation, the size of the first dimension varies within this bound, while the size of the second dimension remains constant.

If you select the **Compute CCDF** parameter, the block computes the stationary average power of the entire signal along each channel. In this case, the size of the output is 1-by- $M$ , where  $M$  is the number of channels (columns) in the input signal.

### Dependencies

To enable this port, set **Measurement** to Average power or All.

Data Types: single | double

### Peak — Peak power

scalar | vector | matrix

Peak power of the voltage signal from the start of the simulation or from the last reset, returned as a scalar, vector, or a matrix. The block computes the peak power along each channel in the units specified in the **Output power units** parameter. For details on how the block computes the peak power, see “Peak Power” on page 2-1183.

If you clear the **Compute CCDF** parameter, the block computes the moving peak power using the “Sliding Window Method” on page 2-1182.

See this table for details on the output signal dimensions.

| Input Signal         | Input Dimensions  | Output Dimensions When Allow arbitrary frame length for fixed-size input signals Appears  | Output Dimensions When Allow arbitrary frame length for fixed-size input signals Does Not Appear |
|----------------------|---|---|--|
| Fixed-size signal    | $P$ -by- $M$ , where $M$ is a multiple of the hop size (window length – overlap length)     | $(P/\text{hop size})$ -by- $M$  | $P$ -by- $M$   |
| Fixed-size signal    | $P$ -by- $M$ , where $M$ is not a multiple of the hop size (window length – overlap length) | $\text{ceil}(P/\text{hop size})$ -by- $M$ when you select <b>Allow arbitrary frame length for fixed-size input signals</b><br><br>If you do not select <b>Allow arbitrary frame length for fixed-size input signals</b> , the block errors. | $P$ -by- $M$   |
| Variable-size signal | $P$ -by- $M$  | $\text{ceil}(P/\text{hop size})$ -by- $M$   | $P$ -by- $M$   |

When the output has an upper bound size of  $\text{ceil}(P/\text{hop size})$ -by- $M$ , during simulation, the size of the first dimension varies within this bound, while the size of the second dimension remains constant.

If you select the **Compute CCDF** parameter, the block computes the stationary peak power of the entire signal along each channel. In this case, the size of the output is 1-by- $M$ , where  $M$  is the number of channels (columns) in the input signal.

#### Dependencies

To enable this port, set **Measurement** to Peak power or All.

Data Types: single | double

**PAPR** — Peak-to-average power ratio  
scalar | vector | matrix

Ratio of the peak power to the average power (PAPR) of the voltage signal, returned as a scalar, vector, or a matrix. The block computes the peak-to-average power ratio along each channel. For details on how the block computes this measurement, see “Peak-to-Average Power Ratio” on page 2-1184.



If you clear the **Compute CCDF** parameter, the block computes the moving peak-to-average power ratio using the “Sliding Window Method” on page 2-1182.

See this table for details on the output signal dimensions.

| Input Signal         | Input Dimensions  | Output Dimensions When Allow arbitrary frame length for fixed-size input signals Appears  | Output Dimensions When Allow arbitrary frame length for fixed-size input signals Does Not Appear |
|----------------------|---|---|--|
| Fixed-size signal    | $P$ -by- $M$ , where $M$ is a multiple of the hop size (window length – overlap length)     | $(P/\text{hop size})$ -by- $M$  | $P$ -by- $M$   |
| Fixed-size signal    | $P$ -by- $M$ , where $M$ is not a multiple of the hop size (window length – overlap length) | $\text{ceil}(P/\text{hop size})$ -by- $M$ when you select <b>Allow arbitrary frame length for fixed-size input signals</b><br><br>If you do not select <b>Allow arbitrary frame length for fixed-size input signals</b> , the block errors. | $P$ -by- $M$   |
| Variable-size signal | $P$ -by- $M$  | $\text{ceil}(P/\text{hop size})$ -by- $M$   | $P$ -by- $M$   |

When the output has an upper bound size of  $\text{ceil}(P/\text{hop size})$ -by- $M$ , during simulation, the size of the first dimension varies within this bound, while the size of the second dimension remains constant.

If you select the **Compute CCDF** parameter, the block computes the stationary peak-to-average power ratio of the entire signal along each channel. In this case, the size of the output is 1-by- $M$ , where  $M$  is the number of channels (columns) in the input signal.

PAPR should be less than 100 dB to obtain accurate CCDF measurements. If PAPR is above 100 dB, the block shows only the highest 100 dB power levels.

### Dependencies

To enable this port, set **Measurement** to Peak-to-average power ratio or All.

Data Types: single | double

**RelPwr** — Relative power

$N$ -by- $M$  matrix

Relative power (dB above average power), returned as an  $N$ -by- $M$  matrix, where:

- $M$  is the number of channels (columns) in the input signal.
- When you set **CCDF output** to Relative power (dB above average power),  $N$  equals the length of the column vector that you specify in the **Specify probability (%) for relative power output** parameter.

- When you set **CCDF output** to Relative power and probability (whole CCDF curve),  $N$  equals  $\text{ceil}(\text{Power range (dB)}/\text{Power resolution (dB)}) + 1$ .

Relative power is the power in dB by which the instantaneous signal power is above the average signal power with a probability of **Prob**, expressed as a percentage. For details on how the block computes relative power, see “Relative Power” on page 2-1184.

### Dependencies

To enable this port, select the **Compute CCDF** parameter and then set the **CCDF output** parameter to Relative power (dB above average power) or Relative power and probability (whole CCDF curve).

Data Types: single | double

**Prob** — Probability in percentage  
*N*-by-*M* matrix

Probability in percentage, returned as an *N*-by-*M* matrix, where:

- $M$  is the number of channels (columns) in the input signal.
- When you set **CCDF output** to Probability (%),  $N$  equals the length of the column vector that you specify in the **Specify relative power (dB) for probability output** parameter.
- When you set **CCDF output** to Relative power and probability (whole CCDF curve),  $N$  equals  $\text{ceil}(\text{Power range (dB)}/\text{Power resolution (dB)}) + 1$ .

**Prob**(*i*)/100 is the probability that the instantaneous signal power of the  $j^{\text{th}}$  channel is above its average signal power by **RelPwr**(*i,j*) in dB.

### Dependencies

To enable this port, select the **Compute CCDF** parameter and then set the **CCDF output** parameter to Probability (%) or Relative power and probability (whole CCDF curve).

Data Types: single | double

## Parameters

**Measurement** — Desired power measurement  
 Average power (default) | Peak power | Peak-to-average power ratio | All

Specify the desired power measurement as one of the following:

- Average power (default)
- Peak power
- Peak-to-average power ratio
- All

For details on how the block computes these measurements, see “Algorithms” on page 2-1182.

**Reference load (ohms)** — Reference load in ohms  
 1 (default) | positive scalar in ohms

Specify the reference load that the power meter uses to compute power values as a real, positive scalar in ohms.

**Tunable:** Yes

Data Types: single | double

**Output power units** — Units of measured power values  
dBm (default) | dBW | Watts

Specify the units of the measured power values as one of the following:

- dBm
- dBW
- Watts

**Compute CCDF** — Enable CCDF computations  
off (default) | on

Select the **Compute CCDF** parameter to enable the block to compute the CCDF of the power of voltage signals. CCDF measurements include the relative power and the probability (in percentage). Relative power is the amount of power in dB by which the instantaneous signal power is above the average signal power. Probability in percentage refers to the probability that the instantaneous signal power is above the average signal power by the relative power in dB.

These measurements provide a qualitative way to view the power distribution between the amplitude values of an input signal over time. The CCDF curves (plot of probability in percentage against relative power in dB) are useful indicators of the dynamic range of a signal.

When you select **Compute CCDF**:

- The block uses all the input samples from the start of the simulation or from a reset signal on the **Rst** port to compute the statistics. The power measurements are stationary.
- You can specify variable-size input signals.

When you clear **Compute CCDF**,

- The block does not calculate the CCDF measurements and computes the power measurements using the “Sliding Window Method” on page 2-1182.
- The block accepts input signals of arbitrary frame length only when you select the **Allow arbitrary frame length for fixed-size input signals** parameter.

**Window length** — Window length  
256 (default) | positive integer

Specify the window length over which the block computes the measurement as a positive integer.

#### Dependencies

To enable this parameter, clear the **Compute CCDF** check box. When you select **Compute CCDF**, **Window length** becomes Inf.

Data Types: `single` | `double`

**Overlap length** — Overlap length between windows  
255 (default) | nonnegative integer

Specify the overlap length between sliding windows as a nonnegative integer. The value of overlap length varies in the range  $[0, \mathbf{Window\ length} - 1]$ .

#### Dependencies

To enable this parameter, clear the **Compute CCDF** check box. When you select **Compute CCDF**, **Overlap length** becomes 0.

Data Types: `single` | `double`

**Allow arbitrary frame length for fixed-size input signals** — Allow arbitrary frame length for fixed-size input signals

off (default) | on

Specify whether fixed-size input signals (whose size does not change during simulation) can have an arbitrary frame length, where the frame length does not have to be multiple of the hop size. Hop size is defined as  $\mathbf{Window\ length} - \mathbf{Overlap\ length}$ . The block uses this parameter setting only for fixed-size input signals and ignores this parameter if the input has a variable-size.

When the input signal is a variable-size signal, the signal can have arbitrary frame length, that is, the frame length does not have to be a multiple of the hop size.

For fixed-size input signals, if you:

- Select the **Allow arbitrary frame length for fixed-size input signals** parameter, the frame length of the signal does not have to be a multiple of the hop size. If the input is not a multiple of the hop size, then the output is generally a variable-size signal. Therefore, to support arbitrary input size, the block must also support variable-size operations, which you can enable by selecting the **Allow arbitrary frame length for fixed-size input signals** parameter.
- Clear the **Allow arbitrary frame length for fixed-size input signals** parameter, the input frame length must be a multiple of the hop size.

#### Dependencies

To enable this parameter, clear the **Compute CCDF** parameter.

**Power range (dB)** — Power range in dB  
50 (default) | positive scalar

Specify the x-axis range of the CCDF curves as a positive scalar in dB. The CCDF curves end at the maximum relative power, namely, the PAPR of the signal, and start at  $\text{PAPR} - \mathbf{Power\ range\ (dB)}$ . In the CCDF capability of the Power Meter block, relative power is the power in dB by which the instantaneous signal power is above the average signal power.

#### Dependencies

To enable this parameter, select the **Compute CCDF** check box.

Data Types: `single` | `double`

**Power resolution (dB)** — Power Resolution in dB

0.1 (default) | positive scalar

Specify the x-axis resolution of the CCDF curves as a positive scalar in dB.

**Dependencies**

To enable this parameter, select the **Compute CCDF** check box.

Data Types: single | double

**CCDF output** — CCDF output

Relative power (dB above average power) (default) | Probability (%) | Relative power and probability (whole CCDF curve)

Specify the block to output:

- **RelPwr** -- Relative power (dB above average power). This is the amount of power in dB by which the instantaneous signal power is above the average power.
- **Prob** -- Probability in percentage. This is the probability that the instantaneous signal power is above the average signal power by the value of the relative power in dB.
- Both the **RelPwr** and the **Prob**.

**Dependencies**

To enable this parameter, select the **Compute CCDF** check box.

**Specify probability (%) for relative power output** — Specify probability for relative power output

[10;1;0.1;0.01;0.001;0.0001] (default) | column vector

Specify the probability (in percentage) for the relative power output as a column vector. The length of this vector determines the number of rows in the relative power output at the **RelPwr** port.

**Dependencies**

To enable this parameter, select the **Compute CCDF** check box and set the **CCDF output** parameter to Relative power (dB above average power).

This also enables the **RelPwr** output port.

Data Types: single | double

**Specify relative power (dB) for probability output** — Specify relative power for probability output

[0;3;6;9;12] (default) | column vector

Specify the relative power (in dB) for the probability output as a column vector. The length of this vector determines the number of rows in the probability output at the **Prob** port.

**Dependencies**

To enable this parameter, select the **Compute CCDF** check box and set the **CCDF output** parameter to Probability (%).

This also enables the **Prob** output port.

Data Types: `single` | `double`

**Reset port** — Enable reset port  
`off` (default) | `on`

Select this parameter to enable the input port **Rst** that accepts a boolean signal. If the value at the **Rst** port is `true`, then the block clears the internal histograms and statistics before processing the current input.

### Dependencies

To enable this parameter, select the **Compute CCDF** check box.

**Simulate using** — Type of simulation to run  
`Code generation` (default) | `Interpreted execution`

Specify the type of simulation to run as one of the following:

- `Code generation` -- Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than `Interpreted execution`.
- `Interpreted execution` -- Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than `Code generation`.

## Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | <code>double</code>   <code>single</code> |
| <b>Direct Feedthrough</b>       | <code>no</code>                           |
| <b>Multidimensional Signals</b> | <code>no</code>                           |
| <b>Variable-Size Signals</b>    | <code>yes</code>                          |
| <b>Zero-Crossing Detection</b>  | <code>no</code>                           |

## Algorithms

### Sliding Window Method

When you do not configure the power meter to compute the CCDF measurements, the power meter computes the moving power measurements using the sliding window method.

In the sliding window method, the block computes the power measurement over a finite duration of the signal. The window length defines the length of the data over which the algorithm computes the power value. The window moves as new data comes in. The output for each input sample is the measurement over the current sample and  $Len - 1$  previous samples.  $Len$  is the length of the sliding window in samples. To compute the first output sample, the algorithm waits until it receives the hop size number of input samples. Hop size is defined as window length - overlap length. Remaining

samples in the window are considered to be zero. As an example, if the window length is 5 and the overlap length is 2, then the algorithm waits until it receives 3 samples of input to compute the first sample of the output. After generating the first output, it generates the subsequent output samples for every hop size number of input samples.

For a more detailed example, see “Sliding Window Method”.

If the window is large, the power that the block computes is closer to the stationary power of the data. For data that does not change rapidly, use a long window to get a smoother measurement. For data that changes fast, use a smaller window.

When you configure the power meter to compute the CCDF measurements, the algorithm computes the stationary power of the data. It sets the window length to `Inf` and the overlap length to `0`, making both the parameters read-only.

### Average Power

When you do not configure the power meter to compute the CCDF measurements, the power meter computes the moving average power of the voltage signal across each channel using the Sliding Window Method on page 2-1182. When you do configure the power meter to compute the CCDF measurements, the power meter computes the stationary average power of the voltage signal across each channel.

These equations give the average power in dBm, dBW, and in Watts units.

$$AvgPower_{dBm} = 10\log_{10}(Avg(|x|^2)/R) + 30$$

$$AvgPower_{dBW} = 10\log_{10}(Avg(|x|^2)/R)$$

$$AvgPower_{Watts} = Avg(|x|^2)/R$$

where,

- $x$  is the input voltage signal.
- $R$  is the reference load (in ohms) that the block uses to compute the power value.
- $Avg$  represents the moving average power when the power meter does not compute the CCDF measurements. The `powermeter` object in MATLAB uses the `dsp.MovingAverage` object and the Power Meter block in Simulink uses the Moving Average block.

When the power meter does compute the CCDF measurements,  $Avg$  represents the stationary average power across each channel.

### Peak Power

When you do not configure the power meter to compute the CCDF measurements, the power meter computes the moving peak power of the voltage signal across each channel using the Sliding Window Method on page 2-1182. When you do configure the power meter to compute the CCDF measurements, the power meter computes the stationary peak power of the voltage signal across each channel.

These equations give the peak power in dBm, dBW, and in Watts units.

$$PeakPower_{dBm} = 10\log_{10}(Max(|x|^2)/R) + 30$$

$$PeakPower_{dBW} = 10\log_{10}(Max(|x|^2)/R)$$

$$PeakPower_{Watts} = Max(|x|^2)/R$$

where,

- $x$  is the input voltage signal.
- $R$  is the reference load (in ohms) that the block uses to compute the power value.
- $Max$  represents the moving peak power when the power meter does not compute the CCDF measurements. The `powermeter` object in MATLAB uses the `dsp.MovingMaximum` object and the Power Meter block in Simulink uses the Moving Maximum block.

When the power meter does compute the CCDF measurements,  $Max$  represents the stationary peak power across each channel.

### Peak-to-Average Power Ratio

When you do not configure the power meter to compute the CCDF measurements, the power meter computes the moving peak-to-average power ratio of the voltage signal across each channel using the Sliding Window Method on page 2-1182. When you do configure the power meter to compute the CCDF measurements, the power meter computes the stationary peak-to-average power ratio of the voltage signal across each channel.

These equations give the peak-to-average power ratio in dBm, dBW, and in Watts units.

$$pkAvgPwr_{dBm} = 10\log_{10}(Max(|x|^2)/Avg(|x|^2))$$

$$pkAvgPwr_{dBW} = 10\log_{10}(Max(|x|^2)/Avg(|x|^2))$$

$$pkAvgPwr_{Watts} = Max(|x|^2)/Avg(|x|^2)$$

where,

- $x$  is the input voltage signal.
- $Avg$  represents the moving average power when the power meter does not compute the CCDF measurements. The `powermeter` object in MATLAB uses the `dsp.MovingAverage` object and the Power Meter block in Simulink uses the Moving Average block.

When the power meter does compute the CCDF measurements,  $Avg$  represents the stationary average power across each channel.

- $Max$  represents the moving peak power when the power meter does not compute the CCDF measurements. The `powermeter` object in MATLAB uses the `dsp.MovingMaximum` object and the Power Meter block in Simulink uses the Moving Maximum block.

When the power meter does compute the CCDF measurements,  $Max$  represents the stationary peak power across each channel.

### Relative Power

The power meter computes the relative power only when you configure the algorithm to compute the CCDF measurements. The algorithm uses a window of infinite duration to compute the relative power.



Relative power is the amount of power in dB by which the instantaneous signal power is above the average signal power. The block calculates the relative power using these equations:

If output power is in dBm,

$$relPower = 10\log_{10}(|x|^2/R) + 30 - AvgPower_{dBm}$$

If output power is in dBW,

$$relPower = 10\log_{10}(|x|^2/R) - AvgPower_{dBW}$$

If output power is in Watts,

$$relPower = (|x|^2/R) - AvgPower_{Watts}$$

where,

- $x$  is the input voltage signal.
- $R$  is the reference load (in ohms) that the block uses to compute the power value.
- $|x|^2/R$  is the instantaneous signal power in Watts.
- $AvgPower$  is the average power of the voltage signal. For more details on how the algorithm computes this measurement, see “Average Power” on page 2-1183.

### Probability

The power meter computes the probability only when you configure the algorithm to compute the CCDF measurements. Probability in percentage refers to the probability that the instantaneous signal power is above the average signal power by the relative power in dB.

## Version History

### Introduced in R2021a

#### R2022b: New Overlap length parameter

Starting in R2022b, when the block is not computing the CCDF measurements, you can specify the overlap length between sliding windows using the **Overlap length** parameter.

#### R2022b: Support for arbitrary input frame length

Power Meter block supports input signals with arbitrary frame lengths when the:

- Input signal is a fixed-size signal (frame length does not change during simulation) and you select the **Allow arbitrary frame length for fixed-size input signals** parameter (if enabled).
- Input signal is a variable-size signal (frame length changes during simulation).

When this block supports an arbitrary frame length input signal, the input frame length does not have to be a multiple of the hop size.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

## **See Also**

### **Objects**

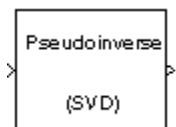
powermeter

### **Blocks**

Moving Average | Moving Maximum | Spectrum Estimator | Spectrum Analyzer

# Pseudoinverse

Compute Moore-Penrose pseudoinverse of matrix



## Library

Math Functions / Matrices and Linear Algebra / Matrix Inverses

dspinverses

## Description

The Pseudoinverse block computes the Moore-Penrose pseudoinverse of input matrix  $A$ .

`[U,S,V] = svd(A,0)` % Equivalent MATLAB code

The pseudoinverse of  $A$  is the matrix  $A^\dagger$  such that

$$A^\dagger = VS^\dagger U^*$$

where  $U$  and  $V$  are orthogonal matrices, and  $S$  is a diagonal matrix. The pseudoinverse has the following properties:

- $AA^\dagger = (AA^\dagger)^*$
- $A^\dagger A = (A^\dagger A)^*$
- $AA^\dagger A = A$
- $A^\dagger AA^\dagger = A^\dagger$

## Parameters

### Show error status port

Select to enable the E output port, which reports a failure to converge. The possible values you can receive on the port are:

- 0 — The pseudoinverse calculation converges.
- 1 — The pseudoinverse calculation does not converge.

If the pseudoinverse calculation fails to converge, the output at port X is an undefined matrix of the correct size.

### Simulate using

Type of simulation to run. You can set this parameter to:

- Interpreted execution (default)

Simulate model using the MATLAB interpreter. This option shortens startup time. For this block, the simulation speed in this mode is faster than in Code generation.

- Code generation

Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but the simulation speed increases with subsequent simulations.

## References

Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

## Supported Data Types

| Port | Supported Data Types   |
|------|--|
| A    | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> </ul> |
| X    | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> </ul> |
| E    | <ul style="list-style-type: none"> <li>• Boolean</li> </ul>  |

## See Also

|                              |                    |
|------------------------------|--------------------|
| Cholesky Inverse             | DSP System Toolbox |
| LDL Inverse                  | DSP System Toolbox |
| LU Inverse                   | DSP System Toolbox |
| Singular Value Decomposition | DSP System Toolbox |
| inv                          | MATLAB             |

See “Matrix Inverses” for related information.

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Generated code relies on the memcpy or memset function (string.h) under certain conditions.

## Pulse Shaping Filter (Obsolete)

(Removed) Design pulse shaping filter



### Compatibility

---

**Note** The Pulse Shaping Filter block has been removed from DSP System Toolbox block library. Existing instances of the Pulse Shaping Filter block will continue to operate. For new models, use the Raised Cosine Receive Filter and Raised Cosine Transmit Filter blocks from the Communications Toolbox library. These blocks replace the functionality of Pulse Shaping Filter block, when **Filter Type** is set to Decimator and Interpolator, respectively.

---

### Library

Filtering / Filter Designs

dspfdesign

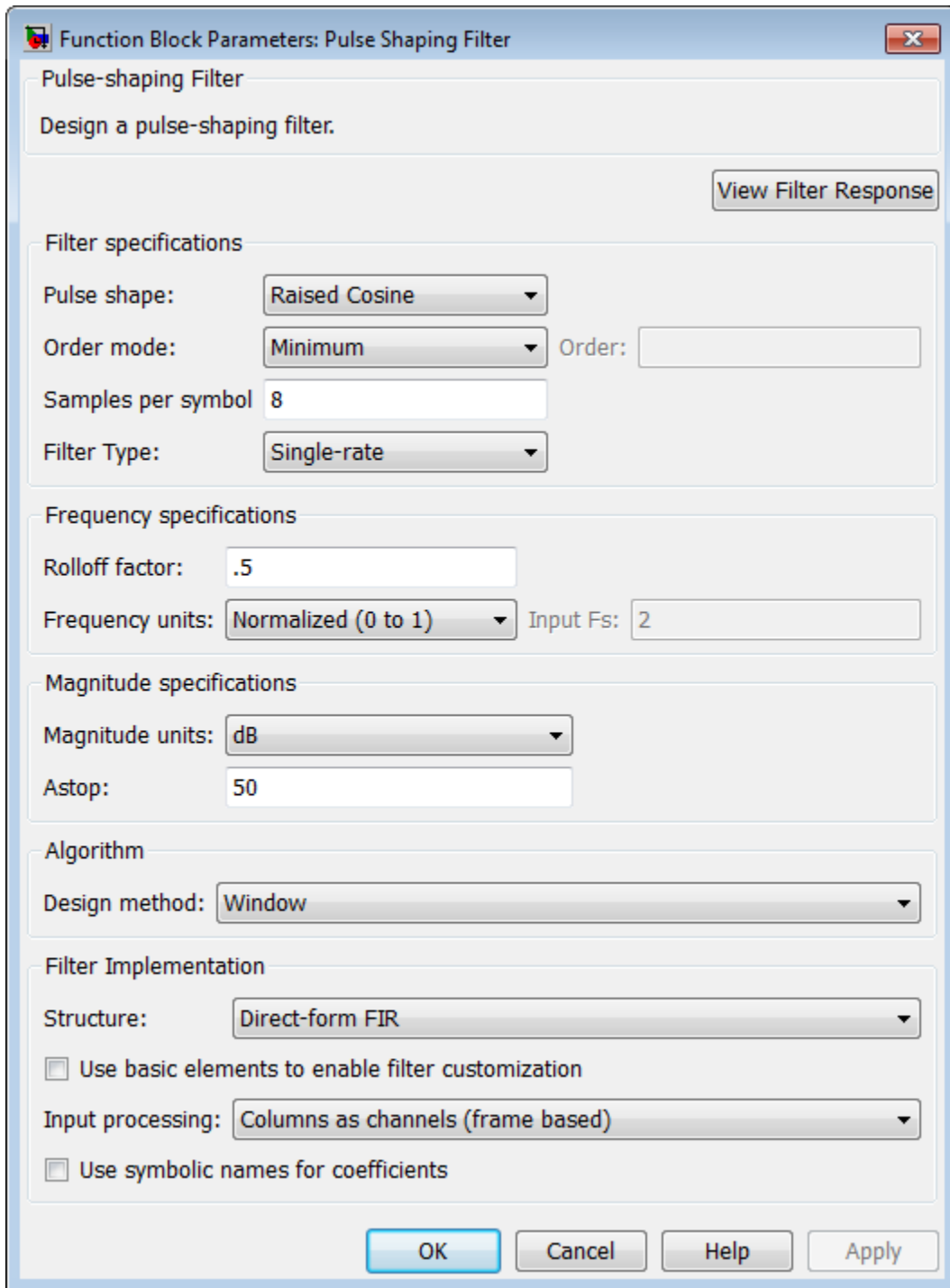
### Description

This block brings the filter design capabilities of the `filterBuilder` function to the Simulink environment.

### Dialog Box

See “Pulse-shaping Filter Design —Main Pane” on page 5-644 for more information about the parameters of this block. The **Data Types** and **Code** panes are not available for blocks in the DSP System Toolbox Filter Designs library.

Parameters of this block that do not change filter order or structure are tunable.



### View filter response

This button opens the Filter Visualization Tool (**FVTool**) from the Signal Processing Toolbox product. You can use the tool to display:

- Magnitude response, phase response, and group delay in the frequency domain.
- Impulse response and step response in the time domain.

- Pole-zero information.

The tool also helps you evaluate filter performance by providing information about filter order, stability, and phase linearity. For more information on FVTool, see the Signal Processing Toolbox documentation.

### Filter Specifications

In this group, you specify the shape and length of the filter.

#### Pulse shape

Select the shape of the impulse response from the following options:

- Raised Cosine
- Square Root Raised Cosine
- Gaussian

#### Order mode

This specification is only available for raised cosine and square root raised cosine filters. For these filters, select one of the following options:

- **Minimum**— This option will result in the minimum-length filter satisfying the user-specified **Frequency specifications**.
- **Specify order**—This option allows the user to construct a raised cosine or square root cosine filter of a specified order by entering an even number in the **Order** input box. The length of the impulse response will be  $\text{Order}+1$ .
- **Specify symbols**—This option enables the user to specify the length of the impulse response in an alternative manner. If **Specify symbols** is chosen, the **Order** input box changes to the **Number of symbols** input box.

#### Samples per symbol

Specify the oversampling factor. Increasing the oversampling factor guards against aliasing and improves the FIR filter approximation to the ideal frequency response. If **Order** is specified in **Number of symbols**, the filter length will be  $\text{Number of symbols} * \text{Samples per symbol} + 1$ . The product  $\text{Number of symbols} * \text{Samples per symbol}$  must be an even number.

If a Gaussian filter is specified, the filter length must be specified in **Number of symbols** and **Samples per symbol**. The product  $\text{Number of symbols} * \text{Samples per symbol}$  must be an even number. The filter length will be  $\text{Number of symbols} * \text{Samples per symbol} + 1$ .

### Frequency specifications

In this group, you specify the frequency response of the filter. For raised cosine and square root raised cosine filters, the frequency specifications include:

#### Rolloff factor

The rolloff factor takes values in the range [0,1]. The smaller the rolloff factor, the steeper the transition in the stopband.

#### Frequency units

The frequency units are normalized by default. If you specify units other than normalized, the block assumes that you wish to specify an input sampling frequency, and enables this input box. The choice of frequency units are: Normalized (0 to 1), Hz, kHz, MHz, GHz

For a Gaussian pulse shape, the available frequency specifications are:

#### Bandwidth-time product

This option allows the user to specify the width of the Gaussian filter. Note that this is independent of the length of the filter. The bandwidth-time product (BT) must be a positive real number. Smaller values of the bandwidth-time product result in larger pulse widths in time and steeper stopband transitions in the frequency response.

#### Frequency units

The frequency units are normalized by default. If you specify units other than normalized, the block assumes that you wish to specify an input sampling frequency, and enables this input box. The choice of frequency units are: Normalized (0 to 1), Hz, kHz, MHz, GHz

#### Magnitude specifications

If the **Order mode** is specified as minimum, the magnitude units may be selected from:

- dB — Specify the magnitude in decibels (default).
- Linear — Specify the magnitude in linear units.

#### Algorithm

The only design method available for FIR pulse-shaping filters is the window method.

#### Filter Implementation

##### Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. FIR filters use direct-form structure.

##### Use basic elements to enable filter customization

Select this check box to implement the filter as a subsystem of basic Simulink blocks. Clear the check box to implement the filter as a high-level subsystem. By default, this check box is cleared.

The high-level implementation provides better compatibility across various filter structures, especially filters that would contain algebraic loops when constructed using basic elements. On the other hand, using basic elements enables the following optimization parameters:

- **Optimize for zero gains** — Terminate chains that contain Gain blocks with a gain of zero.
- **Optimize for unit gains** — Remove Gain blocks that scale by a factor of one.
- **Optimize for delay chains** — Substitute delay chains made up of  $n$  unit delays with a single delay by  $n$ .
- **Optimize for negative gains** — Use subtraction in Sum blocks instead of negative gains in Gain blocks.

##### Optimize for unit-scale values

Select this check box to scale unit gains between sections in SOS filters. This parameter is available only for SOS filters.

##### Input processing

Specify how the block should process the input. The available options may vary depending on the settings of the **Filter Structure** and **Use basic elements for filter customization** parameters. You can set this parameter to one of the following options:



- **Columns as channels (frame based)** — When you select this option, the block treats each column of the input as a separate channel.
- **Elements as channels (sample based)** — When you select this option, the block treats each element of the input as a separate channel.

---

**Note** The **Inherited (this choice will be removed – see release notes)** option will be removed in a future release. See “Frame-Based Processing” in the *DSP System Toolbox Release Notes* for more information.

---

### Rate options

When the **Filter type** parameter specifies a multirate filter, select the rate processing rule for the block from following options:

- **Enforce single-rate processing** — When you select this option, the block maintains the sample rate of the input.
- **Allow multirate processing** — When you select this option, the block adjusts the rate at the output to accommodate an increased or reduced number of samples. To select this option, you must set the **Input processing** parameter to **Elements as channels (sample based)**.

### Use symbolic names for coefficients

Select this check box to enable the specification of coefficients using MATLAB variables. The available coefficient names differ depending on the filter structure. Using symbolic names allows tuning of filter coefficients in generated code. By default, this check box is cleared.

## Supported Data Types

| Port   | Supported Data Types   |
|--------|--|
| Input  | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> </ul> |
| Output | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> </ul> |

## Version History

**Introduced in R2009b**

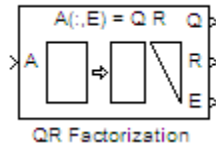
### See Also

#### Blocks

Raised Cosine Receive Filter | Raised Cosine Transmit Filter

## QR Factorization

Factor arbitrary matrix into unitary and upper triangular components



## Library

Math Functions / Matrices and Linear Algebra / Matrix Factorizations

dspfactors

## Description

The QR Factorization block uses a sequence of Householder transformations to triangularize the input matrix  $A$ . The block factors a column permutation of the  $M$ -by- $N$  input matrix  $A$  as

$$A_e = QR$$

The column-pivoted matrix  $A_e$  contains the columns of  $A$  permuted as indicated by the contents of length- $N$  permutation vector  $E$ .

```
A_e = A(:,E) % Equivalent MATLAB code
```

The block selects a column permutation vector  $E$ , which ensures that the diagonal elements of matrix  $R$  are arranged in order of decreasing magnitude.

$$|r_{i+1,j+1}| < |r_{i,j}| \quad i = j$$

The size of matrices  $Q$  and  $R$  depends on the setting of the **Output size** parameter:

- When you select **Economy** for the output size,  $Q$  is an  $M$ -by- $\min(M,N)$  unitary matrix, and  $R$  is a  $\min(M,N)$ -by- $N$  upper-triangular matrix.

```
[Q R E] = qr(A,0) % Equivalent MATLAB code
```

- When you select **Full** for the output size,  $Q$  is an  $M$ -by- $M$  unitary matrix, and  $R$  is a  $M$ -by- $N$  upper-triangular matrix.

```
[Q R E] = qr(A) % Equivalent MATLAB code
```

The block treats length- $M$  unoriented vector input as an  $M$ -by-1 matrix.

QR factorization is an important tool for solving linear systems of equations because of good error propagation properties and the invertibility of unitary matrices:

$$Q^{-1} = Q'$$

where  $Q'$  is the complex conjugate transpose of  $Q$ .

Unlike LU and Cholesky factorizations, the matrix  $A$  does not need to be square for QR factorization. However, QR factorization requires twice as many operations as LU Factorization (Gaussian elimination).

## Parameters

### Output size

Specify the size of output matrices  $Q$  and  $R$ :

- **Economy** — When this output size is selected, the block outputs an  $M$ -by- $\min(M,N)$  unitary matrix  $Q$  and a  $\min(M,N)$ -by- $N$  upper-triangular matrix  $R$ .
- **Full** — When this output size is selected, the block outputs an  $M$ -by- $M$  unitary matrix  $Q$  and a  $M$ -by- $N$  upper-triangular matrix  $R$ .

### Simulate using

- **Interpreted execution (default)**

Simulate model using the MATLAB interpreter. This option shortens startup time and has faster simulation speed compared to **Code generation**.

- **Code generation**

Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster subsequent simulations.

## References

Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

## Supported Data Types

| Port   | Supported Data Types   |
|--------|--|
| Input  | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> </ul> |
| Output | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> </ul> |

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Generated code relies on the `memcpy` or `memset` function (`string.h`) under certain conditions.

## **See Also**

### **Blocks**

Cholesky Factorization | LU Factorization | QR Solver | Singular Value Decomposition

### **Functions**

`qr`

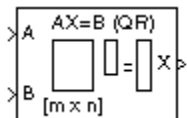
### **Topics**

“Matrix Factorizations”

“Factor Arbitrary Matrix Using QR Factorization Block”

# QR Solver

Find minimum-norm-residual solution to  $AX=B$



## Library

Math Functions / Matrices and Linear Algebra / Linear System Solvers

dsp solvers

## Description

The QR Solver block solves the linear system  $AX=B$ , which can be overdetermined, underdetermined, or exactly determined. The system is solved by applying QR factorization to the M-by-N matrix, A, at the A port. The input to the B port is the right side M-by-L matrix, B. The block treats length-M unoriented vector input as an M-by-1 matrix.

The output at the x port is the N-by-L matrix, X. X is chosen to minimize the sum of the squares of the elements of  $B-AX$ . When B is a vector, this solution minimizes the vector 2-norm of the residual ( $B-AX$  is the residual). When B is a matrix, this solution minimizes the matrix Frobenius norm of the residual. In this case, the columns of X are the solutions to the L corresponding systems  $AX_k=B_k$ , where  $B_k$  is the kth column of B, and  $X_k$  is the kth column of X.

X is known as the minimum-norm-residual solution to  $AX=B$ . The minimum-norm-residual solution is unique for overdetermined and exactly determined linear systems, but it is not unique for underdetermined linear systems. Thus when the QR Solver is applied to an underdetermined system, the output X is chosen such that the number of nonzero entries in X is minimized.

## Parameters

**Simulate using** — Type of simulation to run

Interpreted execution (default) | Code generation

- Interpreted execution

Simulate model using the MATLAB interpreter. This option shortens startup time and has faster simulation speed compared to Code generation.

- Code generation

Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster subsequent simulations.

## Algorithm

QR factorization factors a column-permuted variant ( $A_e$ ) of the M-by-N input matrix A as

$$A_e = QR$$

where Q is a M-by-min(M,N) unitary matrix, and R is a min(M,N)-by-N upper-triangular matrix.

The factored matrix is substituted for  $A_e$  in

$$A_e X = B_e$$

and

$$QRX = B_e$$

is solved for X by noting that  $Q^{-1} = Q^*$  and substituting  $Y = Q^* B_e$ . This requires computing a matrix multiplication for Y and solving a triangular system for X.

$$RX = Y$$

## Supported Data Types

- Double-precision floating point
- Single-precision floating point

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Generated code relies on the memcpy or memset function (string.h) under certain conditions.

## See Also

### Blocks

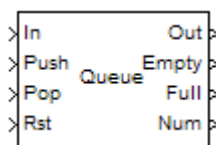
Levinson-Durbin | LDL Solver | LU Solver | QR Factorization | SVD Solver

### Topics

“Linear System Solvers”

# Queue

Store inputs in FIFO register



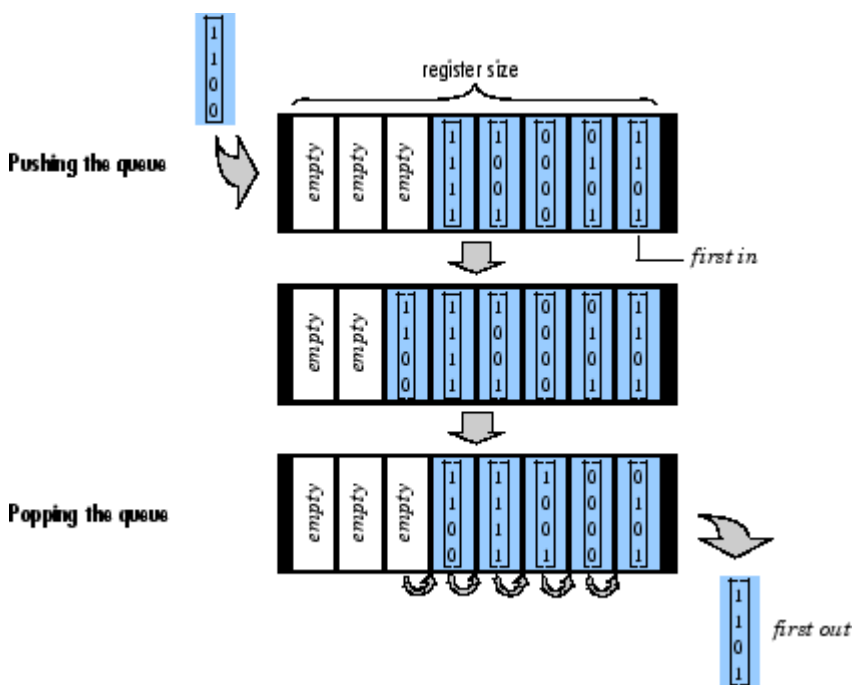
## Libraries:

DSP System Toolbox / Signal Management / Buffers

## Description

The Queue block stores a sequence of input samples in a first-in first-out (FIFO) register. Depending on the inputs at the ports, the block can push, pop, or empty the queue.

When the block receives a trigger event at the **Push** port, it pushes the input at the **In** port to the end of the queue. When the block receives a trigger event at the **Pop** port, it pops the first element off the queue and holds the **Out** port at that value. The first input to be pushed onto the queue is always the first to be popped off.



## Input/Output Ports

### Input

**In** — Input to FIFO register

scalar | vector | matrix

Specify the input to the FIFO register as a scalar, vector, or a matrix.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixed point`

**Push** — Trigger signal to push queue  
scalar

Specify the trigger signal to push the queue as a real scalar.

Input to the **Push** port must have the same data type as the input to the **Pop** and **Rst** ports.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean`

**Pop** — Trigger signal to pop queue  
scalar

Specify the trigger signal to pop the queue as a real scalar.

Input to the **Pop** port must have the same data type as the inputs to the **Push** and **Rst** ports.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean`

**Rst** — Trigger signal to empty queue  
scalar

Specify the trigger signal to empty the queue as a real scalar.

Input to the **Rst** port must have the same data type as the inputs to the **Push** and **Pop** ports.

### **Dependencies**

To enable this port, select **Show reset port (Rst) to clear internal stack buffer**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean`

### **Output**

**Out** — Output of FIFO register  
scalar | vector | matrix

Output of the FIFO register, returned as a scalar, vector, or a matrix.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixed point`

**Empty** — Indicates if queue is empty  
1 | 0

The block returns 1 at this port to indicate an empty queue and 0 otherwise.

### **Dependencies**

To enable this port, select **Show empty register indicator port (Empty)**.

Data Types: `double` | `Boolean`

**Full** — Indicates if queue is full  
1 | 0

The block returns 1 at this port to indicate a full queue and 0 otherwise.



If you select `Dynamic` reallocation in the **Push onto full register** parameter, the **Full** port remains 0.

#### Dependencies

To enable this port, select **Show full register indicator port (Full)**.

Data Types: `double` | `Boolean`

**Num** — Track number of elements in queue  
scalar

Track the number of elements in the queue at any given time.

If the data type at the **In** port is `double`, the data type at the **Num** port is `double`. Otherwise, the data type at the **Num** port is `uint32`.

#### Dependencies

To enable this port, select **Show number of register entries port (Num)**.

Data Types: `double` | `uint32`

## Parameters

**Register size** — Number of entries FIFO register can hold

8 (default) | positive integer

The number of entries that the FIFO register can hold, specified as a positive integer.

**Trigger type** — Type of trigger event for Push, Pop, and Rst ports

`Rising edge` (default) | `Falling edge` | `Either edge` | `Non-zero sample`

The type of event that triggers the execution of the block. Select one of these options:

| Trigger Type    | Description  |
|-----------------|--|
| Rising edge     | <p>Triggers execution of the block when the trigger input does one of the following:</p> <ul style="list-style-type: none"> <li>• Rises from a negative value to zero or a positive value.</li> <li>• Rises from zero to a positive value but is not a continuing rise.</li> </ul> |
| Falling edge    | <p>Triggers execution of the block when the trigger input does one of the following:</p> <ul style="list-style-type: none"> <li>• Falls from a positive value to zero or a negative value.</li> <li>• Falls from zero to a negative value but is not a continuing fall.</li> </ul> |
| Either edge     | Triggers execution of the block when the trigger input type is Rising edge or Falling edge (as described above).   |
| Non-zero sample | Triggers execution of the block at each sample time that the trigger input is not zero.  |

The rate of the trigger signal must be the same as the rate of the data signal input.

**Push onto full register** — Specify block behavior to push request when register is full

Dynamic reallocation (default) | Ignore | Warning | Error

Specify the block behavior when it receives a trigger at the **Push** port but the register is full. The available options are:

- **Dynamic reallocation** — Dynamically resize the register to accept as many additional inputs as memory permits.

To use this option, set the **System target file** parameter on the **Code Generation** pane of the Model Configuration Parameters dialog box to `grt_malloc.tlc – Generic Real-Time Target` with dynamic memory allocation.

---

**Note** If your model contains any referenced models that use a Queue block with the **Push onto full register** parameter set to **Dynamic reallocation**, you cannot simulate your top-level model in the accelerator mode in Simulink.

---

- **Ignore** — Ignore the trigger event and continue the simulation.
- **Warning** — Ignore the trigger event but display a warning message in the MATLAB Command Window.
- **Error** — Display an error dialog box and terminate the simulation.

---

**Note** The **Push onto full register** is a diagnostic parameter. Like all diagnostic parameters in the Configuration Parameters dialog box, this parameter is set to **Ignore** in the code generated for this block by the Simulink Coder code generation software.

---

**Pop empty register** — Specify block behavior to pop request when register is empty

Warning (default) | Ignore | Error

Specify the block behavior when it receives a trigger at the **Pop** port but the register is empty. The available options are:

- **Ignore** — Ignore the trigger event and continue the simulation.
- **Warning** — Ignore the trigger event but display a warning message in the MATLAB Command Window.
- **Error** — Display an error dialog box and terminate the simulation.

---

**Note** The **Pop empty register** is a diagnostic parameter. Like all diagnostic parameters in the Configuration Parameters dialog box, this parameter is set to **Ignore** in the code generated for this block by the Simulink Coder code generation software.

---

**Show empty register indicator port (Empty)** — Enable Empty output port

off (default) | on

Select to enable the **Empty** output port to indicate an empty register.

**Show full register indicator port (Full)** — Enable Full output port

off (default) | on

Select to enable the **Full** output port to indicate a full register.

**Show number of register entries port (Num)** — Enable Num output port

off (default) | on

Select to enable the **Num** output port to show the number of register entries.

**Show reset port (Rst) to clear internal stack buffer** — Enable Rst input port

off (default) | on

Select to enable the **Rst** input port. The block empties the queue when it receives a trigger at this port.

**Clear output port on reset** — Set Out port to 0 when clearing queue

off (default) | on

If you select **Clear output port on reset**, then a trigger event at the **Rst** port empties the queue and sets the value at the **Out** port to 0. This setting also applies when a disabled subsystem containing the Queue block is re-enabled.

#### Dependencies

To enable this parameter, select the **Show reset port (Rst) to clear internal stack buffer** parameter.

**Allow direct feedthrough** — Allow input data to be readily available at Out port

on (default) | off

Select to allow the input data to be available immediately at the **Out** port of the block. Clear to delay the input data by an extra frame.

When you select the **Allow direct feedthrough** parameter and trigger two or more of the control input ports at the same time step, the block executes the operations in the following order:

- 1 Reset
- 2 Push
- 3 Pop

When you clear the **Allow direct feedthrough** parameter and trigger two or more of the control input ports at the same time step, the block executes the operations in the following order:

- 1 Reset
- 2 Pop
- 3 Push

The rate of the trigger signal must be the same as the rate of the data signal input.

### Block Characteristics



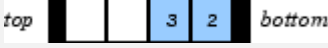






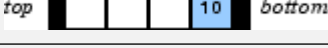


|                           |   |
|---------------------------|---|
| <b>Data Types</b>         | Boolean   double   fixed point   integer   single |
| <b>Direct Feedthrough</b> | no  |

|                                 |    |
|---------------------------------|----|
| <b>Multidimensional Signals</b> | no |
| <b>Variable-Size Signals</b>    | no |
| <b>Zero-Crossing Detection</b>  | no |

## More About

### Operation of Queue Block

This table shows how the Queue block operates when you set the **Register size** to 4, **Trigger type** to **Either edge**, and enable the **Clear output port on reset** parameter. Because the block triggers on both rising and falling edges, each transition from 1 to 0 or 0 to 1 in the **Push**, **Pop**, and **Rst** columns represents a distinct trigger event. 1 in the **Empty** column indicates an empty queue, while 1 in the **Full** column indicates a full queue.

| In | Push | Pop | Rst | Queue  | Out | Empty | Full | Num |
|----|------|-----|-----|--|-----|-------|------|-----|
| 1  | 0    | 0   | 0   | top  bottom   | 0   | 1     | 0    | 0   |
| 2  | 1    | 0   | 0   | top  bottom   | 0   | 0     | 0    | 1   |
| 3  | 0    | 0   | 0   | top  bottom  | 0   | 0     | 0    | 2   |
| 4  | 1    | 0   | 0   | top  bottom | 0   | 0     | 0    | 3   |
| 5  | 0    | 0   | 0   | top  bottom | 0   | 0     | 1    | 4   |
| 6  | 0    | 1   | 0   | top  bottom | 2   | 0     | 0    | 3   |
| 7  | 0    | 0   | 0   | top  bottom | 3   | 0     | 0    | 2   |
| 8  | 0    | 1   | 0   | top  bottom | 4   | 0     | 0    | 1   |
| 9  | 0    | 0   | 0   | top  bottom | 5   | 1     | 0    | 0   |
| 10 | 1    | 0   | 0   | top  bottom | 5   | 0     | 0    | 1   |
| 11 | 0    | 0   | 0   | top  bottom | 5   | 0     | 0    | 2   |
| 12 | 1    | 0   | 1   | top  bottom | 0   | 0     | 0    | 1   |

In the last time step, there are triggers at the **Push** and **Rst** ports simultaneously. The **Rst** trigger takes precedence: the block first clears the queue and then pushes the value at the In port into the queue.

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

The following limitations apply:

- Generated code relies on the `memcpy` or `memset` function (`string.h`) under certain conditions.
- When you select `Dynamic reallocation`, you must set the **System target file** parameter on the **Code Generation** pane of the Configuration Parameters dialog box to `grt_malloc.tlc - Generic Real-Time Target with dynamic memory allocation`.

### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

## See Also

### Blocks

Buffer | Delay Line | Stack

# Random Source

Generate randomly distributed values



**Libraries:**  
DSP System Toolbox / Sources

## Description

The Random Source block generates a frame of  $M$  values drawn from a uniform or Gaussian pseudorandom distribution. Specify  $M$  in the **Samples per frame** parameter.

## Ports

### Output

**Port\_1** — Signal of random values

scalar | vector | matrix

Signal of random values with uniform or Gaussian (normal) distribution.

Data Types: single | double

## Parameters

**Source type** — Uniform or Gaussian

Uniform (default) | Gaussian

The distribution from which to draw the random values, `Uniform` or `Gaussian`.

When you set the **Source type** parameter to `Uniform`, the output samples are drawn from a uniform distribution whose minimum and maximum values are specified by the **Minimum** and **Maximum** parameters, respectively. All values in this range are equally likely to be selected. A length- $N$  vector specified for one or both of these parameters generates an  $N$ -channel output ( $M$ -by- $N$  matrix) containing a unique random distribution in each channel.

For example, specify

- **Minimum** = [0 0 -3 -3]
- **Maximum** = [10 10 20 20]

to generate a four-channel output whose first and second columns contain random values in the range [0, 10], and whose third and fourth columns contain random values in the range [-3, 20]. When you specify only one of the **Minimum** and **Maximum** parameters as a vector, the block scalar expands the other parameter so it is the same length as the vector.

When you set the **Source type** parameter to `Gaussian`, you must also set the **Method** parameter, which determines the method by which the block computes the output.

**Method** — Method of computing Gaussian random values

Ziggurat (default) | Sum of uniform values

The method by which the block computes the Gaussian random values:

- **Ziggurat** — Produces Gaussian random values by using the ziggurat method.
- **Sum of uniform values** — Produces Gaussian random values by adding and scaling uniformly distributed random signals based on the central limit theorem. This theorem states that the probability distribution of the sum of a sufficiently high number of random variables approaches the Gaussian distribution. You must set the **Number of uniform values to sum** parameter, which determines the number of uniformly distributed random numbers to sum to produce a single Gaussian random value.

For both settings of the **Method** parameter, the output samples are drawn from the normal distribution defined by the **Mean** and **Variance** parameters. A length- $N$  vector specified for one or both of the **Mean** and **Variance** parameters generates an  $N$ -channel output ( $M$ -by- $N$  frame matrix) containing a distinct random distribution in each column. When you specify only one of these parameters as a vector, the block scalar expands the other parameter so it is the same length as the vector.

#### Dependencies

To enable this parameter, set **Source type** to Gaussian. For more information, see “Source type” on page 2-0 .

**Number of uniform values to sum** — Number of uniform values to sum

12 (default)

The number of uniformly distributed random values to sum to compute a single number in a Gaussian random distribution.

#### Dependencies

To enable this parameter, set **Source type** to Gaussian and **Method** to Sum of uniform values. For more information, see “Source type” on page 2-0 .

**Minimum** — Minimum value of uniform distribution

0 (default) | scalar | vector

The minimum value in the uniform distribution specified as a finite scalar or vector.

**Tunable:** Yes

#### Dependencies

To enable this parameter, set **Source type** to Uniform.

#### Limitations

Tunable (Simulink) in Simulation mode only.

**Maximum** — Maximum value of uniform distribution



1 (default) | scalar | vector

The maximum value in the uniform distribution specified as a finite scalar or vector.

**Tunable:** Yes

**Dependencies**

To enable this parameter, set **Source type** to Uniform.

**Limitations**

Tunable (Simulink) in Simulation mode only.

**Mean** — Mean value of Gaussian distribution

0 (default) | scalar | vector

The mean of the Gaussian (normal) distribution specified as a finite scalar or vector.

**Tunable:** Yes

**Dependencies**

To enable this parameter, set **Source type** to Gaussian.

**Limitations**

Tunable (Simulink) in Simulation mode only.

**Variance** — Variance of Gaussian distribution

1 (default) | scalar | vector

The variance of the Gaussian (normal) distribution.

**Dependencies**

To enable this parameter, set **Source type** to Gaussian.

**Limitations**

Tunable (Simulink) in Simulation mode only.

**Repeatability** — Repeatability of block output

Specify seed (default) | Repeatable | Not repeatable

The **Repeatability** parameter determines if the block outputs the same signal each time you run the simulation. You can set the parameter to one of the following options:

- **Repeatable** — Outputs the same signal each time you run the simulation. The first time you run the simulation, the block randomly selects an initial seed. The block reuses these same initial seeds every time you rerun the simulation.
- **Specify seed** — Outputs the same signal each time you run the simulation. Every time you run the simulation, the block uses the initial seeds specified in the **Initial seed** parameter. Also see “Initial seed” on page 2-0 .

- **Not repeatable** — Does not output the same signal each time you run the simulation. Every time you run the simulation, the block randomly selects an initial seed.

**Initial seed** — Initial seed for random number generator

1 (default) | scalar | vector

The initial seed(s) to use for the random number generator specified as a finite scalar or vector. The generator produces an identical sequence of pseudorandom numbers each time it is executed with a particular initial seed.

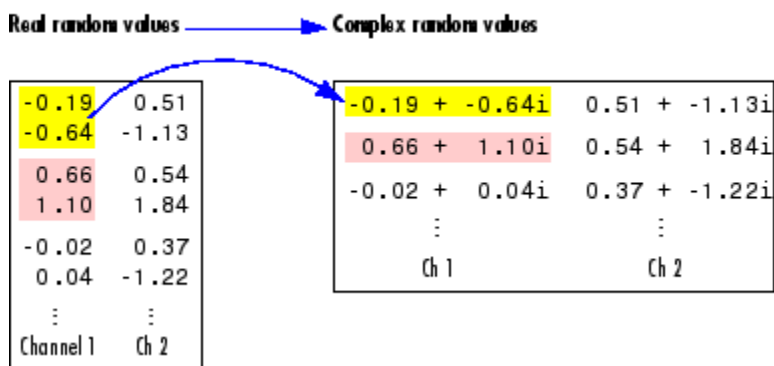
To specify the  $N$  initial seeds for an  $N$ -channel real-valued output, set the **Complexity** parameter to **Real** and provide one of the following in the **Initial seed** parameter:

- Length- $N$  vector of initial seeds — Uses each vector element as an initial seed for the corresponding channel in the  $N$ -channel output.
- Single scalar — Uses the scalar to generate  $N$  random values as the seeds for the  $N$ -channel output.

To specify the initial seeds for an  $N$ -channel complex-valued output, set the **Complexity** parameter to **Complex** and provide one of the following in the **Initial seed** parameter:

- Length- $N$  vector of initial seeds — Uses each vector element as an initial seed for generating  $N$  channels of *real* random values. The block uses pairs of adjacent values in each of these channels as the real and imaginary components of the final output, as illustrated in the following figure.
- Single scalar — Uses the scalar to generate  $N$  random values as the seeds for generating  $N$  channels of *real* random values. The block uses pairs of adjacent values in each of these channels as the real and imaginary components of the final output, as illustrated in the following figure.

Use  $N$  channels of real random values to create the  $N$ -channel complex random output.



**Tunable:** Yes

**Dependencies**

To enable this parameter, set **Repeatability** to **Specify seed**.

**Limitations**

Tunable (Simulink) in Simulation mode only.

**Inherit output port attributes** — Inherit output port parameters from downstream block

off (default) | on

When you select this check box, the block inherits the sample mode, sample time, output data type, complexity, and signal dimensions of the signal from the downstream block. When you select this check box, the **Sample mode**, **Sample time**, **Samples per frame**, **Output data type**, and **Complexity** parameters are disabled.

Suppose that you want to back propagate a 1-D vector. The output of the Random Source block is a 1-D vector of length  $M$ , where length  $M$  is inherited from the downstream block. When the **Minimum**, **Maximum**, **Mean**, or **Variance** parameter specifies  $N$  channels, the 1-D vector output contains  $M/N$  samples from each channel. An error occurs in this case when  $M$  is not an integer multiple of  $N$ .

Suppose that you want to back propagate a  $M$ -by- $N$  signal. When  $N > 1$ , your signal has  $N$  channels. When  $N = 1$ , your signal has  $M$  channels. The value of the **Minimum**, **Maximum**, **Mean**, or **Variance** parameter can be a scalar or a vector of length equal to the number of channels. You can specify these parameters as either row or column vectors, except when the signal is a row vector. In this case, the **Minimum**, **Maximum**, **Mean**, or **Variance** parameter must also be specified as a row vector.

**Sample mode** — Discrete or continuous

Discrete (default) | Continuous

The sample mode, specified as Continuous or Discrete.

When you set **Sample mode** to Discrete, the **Sample time** parameter value,  $T_s$ , specifies the random sequence sample period. In this mode, the block generates the number of samples specified by the **Samples per frame** parameter value,  $M$ , and outputs this frame with a period of  $MT_s$ .

When you set **Sample mode** to Continuous, the block is configured for continuous-time operation, and the **Sample time** and **Samples per frame** parameters are disabled. Note that many DSP System Toolbox blocks do not accept continuous-time inputs.

#### Dependencies

To enable this parameter, clear the **Inherit output port attributes** check box.

**Sample time** — Output sample period

1 (default) | scalar

The sample period,  $T_s$ , of the random output sequence when the **Sample mode** is Discrete, specified as a positive, finite, scalar. The output frame period is  $MT_s$ .

#### Dependencies

To enable this parameter, clear the **Inherit output port attributes** check box and set **Sample mode** to Discrete.

**Samples per frame** — Samples per output frame

1 (default) | positive integer

The number of samples,  $M$ , in each output frame, specified as a positive integer. The output frame period is  $MT_s$ .

**Dependencies**

To enable this parameter, clear the **Inherit output port attributes** check box and set **Sample mode** to **Discrete**.

**Output data type** — Output data type

Double (default) | Single

The data type of the output, specified as single-precision or double-precision.

**Dependencies**

To enable this parameter, clear the **Inherit output port attributes** check box.

**Complexity** — Complexity of output

Real (default) | Complex

The complexity of the output, specified as **Real** or **Complex**. These settings control all channels of the output, so real and complex data cannot be combined in the same output. For complex output with a **Uniform** distribution, the real and imaginary components in each channel are both drawn from the same uniform random distribution, defined by the **Minimum** and **Maximum** parameters for that channel.

For complex output with a **Gaussian** distribution, the real and imaginary components in each channel are drawn from normal distributions with different means. In this case, the **Mean** parameter for each channel should specify a complex value; the real component of the **Mean** parameter specifies the mean of the real components in the channel, while the imaginary component specifies the mean of the imaginary components in the channel. When either the real or imaginary component is omitted from the **Mean** parameter, a default value of 0 is used for the mean of that component.

For example, a **Mean** parameter setting of  $[5+2i \ 0.5 \ 3i]$  generates a three-channel output with the following means.

|                |                   |                      |
|----------------|-------------------|----------------------|
| Channel 1 mean | <i>real</i> = 5   | <i>imaginary</i> = 2 |
| Channel 2 mean | <i>real</i> = 0.5 | <i>imaginary</i> = 0 |
| Channel 3 mean | <i>real</i> = 0   | <i>imaginary</i> = 3 |

For complex output, the **Variance** parameter,  $\sigma^2$ , specifies the *total variance* for each output channel. This is the sum of the variances of the real and imaginary components in that channel.

$$\sigma^2 = \sigma_{\text{Re}}^2 + \sigma_{\text{Im}}^2$$

The specified variance is equally divided between the real and imaginary components, so that

$$\sigma_{\text{Re}}^2 = \frac{\sigma^2}{2}$$

$$\sigma_{\text{Im}}^2 = \frac{\sigma^2}{2}$$

**Dependencies**

To enable this parameter, clear the **Inherit output port attributes** check box.

## Block Characteristics

|                                 |                 |
|---------------------------------|-----------------|
| <b>Data Types</b>               | double   single |
| <b>Direct Feedthrough</b>       | no              |
| <b>Multidimensional Signals</b> | no              |
| <b>Variable-Size Signals</b>    | no              |
| <b>Zero-Crossing Detection</b>  | no              |

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Generated code relies on the memcpy or memset function (string.h) under certain conditions.

## See Also

### Blocks

Discrete Impulse | Maximum | Minimum | Signal From Workspace | Standard Deviation | Variance | Constant | Random Number | Signal Generator

### Functions

rand | randn | RandStream

### Topics

“Estimate the Transfer Function of an Unknown System”

## Real Cepstrum

Compute real cepstrum of input



**Libraries:**  
DSP System Toolbox / Transforms

### Description

The Real Cepstrum block computes the real cepstrum of the input.

### Ports

#### Input

**Input** — Input  
vector | matrix

Specify the input  $u$  as a vector or a matrix of size  $M$ -by- $N$ . The block treats each column of the input as an independent channel containing  $M$  consecutive samples. The block always processes unoriented vector inputs as a single channel, and returns the result as a length- $M$  column vector. The block does not accept complex-valued inputs.

Data Types: single | double

#### Output

**Output** — Real cepstrum output  
vector | matrix

Real cepstrum output, returned as an  $M_o$ -by- $N$  matrix.

When you clear the **Inherit FFT length from input port dimensions** parameter,  $M_o$  is the value you specify in the **FFT length** parameter. When you select the **Inherit FFT length from input port dimensions** parameter,  $M_o = M$ , where  $M$  is the frame size (number of rows) of the input signal.

Each output column contains the length- $M_o$  real cepstrum of the corresponding input column.

The output port rate is the same as the input port rate.

Data Types: single | double

### Parameters

**Inherit FFT length from input port dimensions** — Match the FFT length to the dimensions of the input port

on (default) | off

When you select this check box, the output frame size  $M_o$  matches the input frame size  $M$ .

**FFT length** — Compute FFT length

64 (default) | positive integer

The number of frequency points at which to compute the FFT, which is also the output frame size  $M_o$ .

**Dependency**

To enable this parameter, clear the **Inherit FFT length from input port dimensions** parameter.

## Block Characteristics

|                                 |                 |
|---------------------------------|-----------------|
| <b>Data Types</b>               | double   single |
| <b>Direct Feedthrough</b>       | no              |
| <b>Multidimensional Signals</b> | no              |
| <b>Variable-Size Signals</b>    | no              |
| <b>Zero-Crossing Detection</b>  | no              |

## More About

### Real Cepstrum

The *real cepstrum* is the inverse Fourier transform of the real logarithm of the magnitude of the Fourier transform of a sequence.

Here is the equivalent MATLAB code for computing the real cepstrum.

```
y = real(ifft(log(abs(fft(u,Mo))))))
```

or, more compactly,

```
y = rceps(u,Mo)
```

where,

- $u$  -- Input
- $y$  -- Real cepstrum output
- $M_o$  -- Output frame size

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Generated code relies on the memcopy or memset function (string.h) under certain conditions.

## **See Also**

### **Functions**

rceps

### **Blocks**

Complex Cepstrum | DCT | FFT

### **Topics**

“SSB Modulation”





## References

Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

## Supported Data Types

- Double-precision floating point
- Single-precision floating point

## See Also

|               |                    |
|---------------|--------------------|
| Matrix 1-Norm | DSP System Toolbox |
| Normalization | DSP System Toolbox |
| rcond         | MATLAB             |

## Version History

Introduced before R2006a

## Extended Capabilities

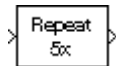
### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Generated code relies on the `memcpy` or `memset` function (`string.h`) under certain conditions.

# Repeat

Resample input at higher rate by repeating values



## Libraries:

DSP System Toolbox / Signal Operations  
 Communications Toolbox / Sequence Operations  
 DSP System Toolbox HDL Support / Signal Operations

## Description

The Repeat block upsamples each channel of the  $M_i$ -by- $N$  input to a rate  $L$  times higher than the input sample rate. To do so, the block repeats each consecutive input sample  $L$  times at the output. Specify the integer  $L$  in the **Repetition count**, **L** parameter or through the input port **L**.

You can use the Repeat block in triggered subsystems when you set the **Rate options** parameter to **Enforce single-rate processing**.

## Ports

### Input

**in** — Input signal

scalar | vector | matrix

Specify the input signal as a real or a complex-valued scalar, vector, or a matrix.

When you set **Input processing** to **Columns as channels (frame based)** and **Rate options** to **Enforce single-rate processing**, the input can be a variable-size signal. When the input is a variable-size signal, the frame size (number of rows) and the number of channels (columns) of the signal can change during simulation.

This port is unnamed until you set **Repetition count source** to **Input port**.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

**L** — Repetition count

positive integer  $\leq L_{max}$

Specify the repetition count  $L$  as a positive integer less than or equal to the value you specify in the **Maximum repetition count**, **L<sub>max</sub>** parameter  $L_{max}$ .

### Dependency

To enable this port, set the **Repetition count source** parameter to **Input port**.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32

### Output

**Port\_1** — Output signal

scalar | vector | matrix

The Repeat block outputs a signal that has the same data type and complexity as the input signal **in**. The size of the output depends on the value of the repetition count and the option you select in the **Rate options** parameter.

If you set **Repetition count, L** to  $L$  and **Rate options** to:

- **Enforce single-rate processing** -- The block maintains the input sample rate by increasing the output frame size by a factor of  $L$ .
- **Allow multirate processing** -- The block resamples the signal such that the output sample rate is  $L$  times faster than the input sample rate. The output signal has the same frame size as the input signal.

When you input the repetition count through the input port **L**, the **Rate options** parameter is automatically set to **Enforce single-rate processing**. In this case, the block maintains the input sample rate at the output by increasing the output frame size by a factor of  $L$ .

The output is a variable-size signal when one or both of these conditions are met:

- Input is a variable-size signal.
- Repetition count is input through the port **L**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixed point`

## Parameters

**Repetition count source** — Mode of specifying repetition count

Dialog parameter (default) | Input port

Specify the repetition count through one of these options:

- **Dialog parameter** -- The block enables the **Repetition count, L** parameter in the block dialog box.
- **Input port** -- The block enables the input port **L** through which you specify the repetition count.

**Repetition count, L** — Repetition count

5 (default) | positive integer

Specify the repetition count  $L$  as a positive integer. This value is the number of times that the block repeats the input value. This is the factor by which the block increases the output frame size or sample rate.

### Dependency

To enable this parameter, set **Repetition count source** to **Dialog parameter**.

**Maximum repetition factor, Lmax** — Maximum repetition count

16 (default) | positive integer  $\geq 1$

Specify the maximum repetition  $L_{max}$  as a positive integer greater than or equal to 1. The repetition count you specify through the input port **L** must be less than or equal to  $L_{max}$ .

**Dependency**

To enable this parameter, set **Repetition count source** to `Input port`.

**Input processing** — Method to process input signals

`Columns as channels (frame based) (default)` | `Elements as channels (sample based)`

Specify how the block should process the input by setting this parameter to one of these options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel. In this mode, the block can perform single-rate or multirate processing.
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel. In this mode, the block always performs multirate processing.

**Dependency**

To enable this parameter, set **Repetition count source** to `Dialog parameter`.

When you set **Repetition count source** to `Input port`, the **Input processing** parameter is automatically set to `Columns as channels (frame based)`.

**Rate options** — Method by which block upsamples input

`Enforce single-rate processing (default)` | `Allow multirate processing`

Specify the method by which the block upsamples the input by selecting one of these options:

- `Enforce single-rate processing` — When you select this option, the block maintains the input sample rate by increasing the output frame size by a factor of  $L$ . To select this option, you must set the **Input processing** parameter to `Columns as channels (frame based)`.
- `Allow multirate processing` — When you select this option, the block resamples the signal such that the output sample rate is  $L$  times faster than the input sample rate.

**Dependency**

To enable this parameter, set **Repetition count source** to `Dialog parameter`.

When you set **Repetition count source** to `Input port`, the **Rate options** parameter is automatically set to `Enforce single-rate processing`.

**Initial conditions** — Initial conditions

`0 (default)` | `scalar` | `matrix`

Specify the value with which the block is initialized for cases of nonzero latency as a scalar or a matrix.

**Dependencies**

To enable this parameter, set **Rate options** to `Allow multirate processing`.

## Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | Boolean   double   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no  |
| <b>Multidimensional Signals</b> | no  |
| <b>Variable-Size Signals</b>    | yes   |
| <b>Zero-Crossing Detection</b>  | no  |

## More About

### Frame-Based Processing

When you set the **Input processing** parameter to `Columns as channels (frame based)`, the block upsamples each column of the input over time. In this mode, the block can perform either single-rate or multirate processing. You can use the **Rate options** parameter to specify how the block upsamples the input.

- When you set the **Rate options** parameter to `Enforce single-rate processing`, the input and output of the block have the same sample rate. In this mode, the block outputs a signal with a proportionally larger frame size than the input. The block upsamples each channel independently by repeating each row of the input matrix  $L$  times at the output. For upsampling by a factor of  $L$ , the output frame size is  $L$  times larger than the input frame size ( $M_o = M_i * L$ ), but the input and output frame rates are equal.

For an example that shows how the Repeat block operates in the single-rate processing mode, see “Repeat Signal In Single-Rate Frame-Based Processing Mode”.

- When you set the **Rate options** parameter to `Allow multirate processing`, the block treats an  $M_i$ -by- $N$  matrix input as  $N$  independent channels. The block generates the output at the faster (upsampled) rate by using a proportionally shorter frame period at the output port than at the input port. For  $L$  repetitions of the input, the output frame period is  $L$  times shorter than the input frame period ( $T_{fo} = T_{fi}/L$ ). In this mode, the output always has the same frame size as the input.

For an example that shows how the Repeat block operates in the multirate frame-based processing mode, see “Repeat Signal In Multirate Frame-Based Processing Mode”.

### Sample-Based Processing

When you set the **Input processing** parameter to `Elements as channels (sample based)`, the block treats an  $M$ -by- $N$  matrix input as  $M*N$  independent channels, and upsamples each channel over time. The block upsamples each channel over time such that the output sample rate is  $L$  times higher than the input sample rate ( $T_{so} = T_{si}/L$ ). In this mode, the output is always the same size as the input.

### Nonzero Latency

The Repeat block has tasking latency for a multirate multitasking operation:

- In the multirate sample-based processing mode, the block repeats the initial condition for each channel for the first  $L$  output samples. The first element of the input channel appears as output sample  $L+1$ . The **Initial conditions** parameter can be an  $M_i$ -by- $N$  matrix containing one value for each channel, or a scalar value that the block applies to all signal channels.

- In the multirate frame-based processing mode, the block repeats the first row of the initial condition matrix for the first  $L$  output samples, the second row of the initial condition matrix for the next  $L$  output samples, and so on. The first row of the first input matrix appears in the output as sample  $M_1L+1$ . The **Initial conditions** parameter can be an  $M_1$ -by- $N$  matrix, or a scalar value that the block repeats across all elements of the  $M_1$ -by- $N$  matrix.

---

**Note** For more information on latency and Simulink tasking modes, see “Excess Algorithmic Delay (Tasking Latency)” and “Time-Based Scheduling and Code Generation” (Simulink Coder).

---

## Zero Latency

The Repeat block has *zero-tasking latency* for all single-rate operations. The block is in a single-rate mode if you specify a repetition count of 1 or if you set the **Input processing** parameter to `Columns as channels (frame based)` and the **Rate options** parameter to `Enforce single-rate processing`.

The Repeat block also has zero-tasking latency for multirate operations if you run your model in Simulink single-tasking mode.

Zero-tasking latency means that the block repeats the first input (received at  $t=0$ ) for the first  $L$  output samples, the second input for the next  $L$  output samples, and so on.

## Version History

### Introduced before R2006a

#### R2022b: Support for variable-size input signal

Starting in R2022b, when you set **Input processing** to `Columns as channels (frame based)` and **Rate options** to `Enforce single-rate processing`, the input can be a variable-size signal. That is, the frame size (number of rows) and the number of channels (columns) of the signal can change during simulation.

When the input is a variable-size signal, the output is also a variable-size signal.

#### R2023a: Tune repetition count through input port

You can now specify the repetition count through an input port when the Repeat block operates in the single-rate frame-based processing mode. When you specify the repetition count through the input port, you can change the value during simulation.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Generated code relies on the `memcpy` or `memset` function (`string.h`) under certain conditions.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

### Best Practices

The Repeat block uses fewer hardware resources than the Upsample block. If your algorithm does not require zero-padding upsampling, use the Repeat block.

### HDL Architecture

This block has one default HDL architecture.

### HDL Block Properties

|                                  |  |
|----------------------------------|--|
| <b>ConstrainedOutputPipeline</b> | Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder). |
| <b>InputPipeline</b>             | Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).                 |
| <b>OutputPipeline</b>            | Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).               |

### Restrictions

**Input processing** set to `Columns as channels (frame based)` is not supported.

### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

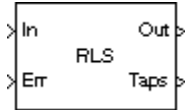
### See Also

FIR Interpolation | Upsample



## RLS Adaptive Filter (Obsolete)

Compute filter estimates for input using RLS adaptive filter algorithm



### Library

dspobslib

### Description

---

**Note** The RLS Adaptive Filter block is still supported but is likely to be obsoleted in a future release. We strongly recommend replacing this block with the RLS Filter block.

---

The RLS Adaptive Filter block recursively computes the recursive least squares (RLS) estimate of the FIR filter coefficients.

The corresponding RLS filter is expressed in matrix form as

$$k(n) = \frac{\lambda^{-1}P(n-1)u(n)}{1 + \lambda^{-1}u^H(n)P(n-1)u(n)}$$

$$y(n) = \widehat{w}^H(n-1)u(n)$$

$$e(n) = d(n) - y(n)$$

$$\widehat{w}(n) = \widehat{w}(n-1) + k(n)e^*(n)$$

$$P(n) = \lambda^{-1}P(n-1) - \lambda^{-1}k(n)u^H(n)P(n-1)$$

where  $\lambda^{-1}$  denotes the reciprocal of the exponential weighting factor. The variables are as follows

| Variable         | Description                                    |
|------------------|--|
| $n$              | The current algorithm iteration                |
| $u(n)$           | The buffered input samples at step $n$         |
| $P(n)$           | The inverse correlation matrix at step $n$     |
| $k(n)$           | The gain vector at step $n$                    |
| $\widehat{w}(n)$ | The vector of filter-tap estimates at step $n$ |
| $y(n)$           | The filtered output at step $n$                |
| $e(n)$           | The estimation error at step $n$               |
| $d(n)$           | The desired response at step $n$               |
| $\lambda$        | The exponential memory weighting factor        |

The block icon has port labels corresponding to the inputs and outputs of the RLS algorithm. Note that inputs to the **In** and **Err** ports must be sample-based scalars. The signal at the **Out** port is a scalar, while the signal at the **Taps** port is a sample-based vector.

| Block Ports | Corresponding Variables   |
|-------------|---|
| In          | $u$ , the scalar input, which is internally buffered into the vector $u(n)$ |
| Out         | $y(n)$ , the filtered scalar output   |
| Err         | $e(n)$ , the scalar estimation error  |
| Taps        | $\hat{w}(0)$ , the vector of filter-tap estimates                           |

An optional **Adapt** input port is added when you select the **Adapt input** check box in the dialog box. When this port is enabled, the block continuously adapts the filter coefficients while the **Adapt** input is nonzero. A zero-valued input to the **Adapt** port causes the block to stop adapting, and to hold the filter coefficients at their current values until the next nonzero **Adapt** input.

The implementation of the algorithm in the block is optimized by exploiting the symmetry of the inverse correlation matrix  $P(n)$ . This decreases the total number of computations by a factor of two.

The **FIR filter length** parameter specifies the length of the filter that the RLS algorithm estimates. The **Memory weighting factor** corresponds to  $\lambda$  in the equations, and specifies how quickly the filter “forgets” past sample information. Setting  $\lambda=1$  specifies an infinite memory; typically,  $0.95 \leq \lambda \leq 1$ .

The **Initial value of filter taps** specifies the initial value  $\hat{w}(0)$  as a vector, or as a scalar to be repeated for all vector elements. The initial value of  $P(n)$  is

$$I \frac{1}{\hat{\sigma}^2}$$

where you specify  $\hat{\sigma}^2$  in the **Initial input variance estimate** parameter.

## Parameters

### FIR filter length

The length of the FIR filter.

### Memory weighting factor

The exponential weighting factor, in the range  $[0, 1]$ . A value of 1 specifies an infinite memory. Tunable (Simulink).

### Initial value of filter taps

The initial FIR filter coefficients.

### Initial input variance estimate

The initial value of  $1/P(n)$ .

### Adapt input

Enables the **Adapt** port.

## References

Haykin, S. *Adaptive Filter Theory*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1996.

## Supported Data Types

- Double-precision floating point
- Single-precision floating point

## See Also

Kalman Adaptive Filter (Obsolete)      DSP System Toolbox

LMS Adaptive Filter (Obsolete)      DSP System Toolbox

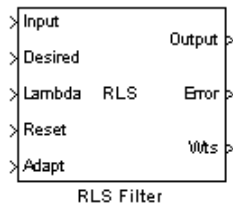
See “Noise Cancellation in Simulink Using Normalized LMS Adaptive Filter” for related information.

## Version History

**Introduced in R2008b**

## RLS Filter

Compute filtered output, filter error, and filter weights for given input and desired signal using RLS adaptive filter algorithm



## Library

Filtering / Adaptive Filters

dspadpt3

## Description

The RLS Filter block recursively computes the least squares estimate (RLS) of the FIR filter weights. The block estimates the filter weights, or coefficients, needed to convert the input signal into the desired signal. Connect the signal you want to filter to the Input port. The input signal can be a scalar or a column vector. Connect the signal you want to model to the Desired port. The desired signal must have the same data type, complexity, and dimensions as the input signal. The Output port outputs the filtered input signal. The Error port outputs the result of subtracting the output signal from the desired signal.

The corresponding RLS filter is expressed in matrix form as

$$\mathbf{k}(n) = \frac{\lambda^{-1} \mathbf{P}(n-1) \mathbf{u}(n)}{1 + \lambda^{-1} \mathbf{u}^H(n) \mathbf{P}(n-1) \mathbf{u}(n)}$$

$$y(n) = \mathbf{w}(n-1) \mathbf{u}(n)$$

$$e(n) = d(n) - y(n)$$

$$\mathbf{w}(n) = \mathbf{w}(n-1) + \mathbf{k}^H(n) e(n)$$

$$\mathbf{P}(n) = \lambda^{-1} \mathbf{P}(n-1) - \lambda^{-1} \mathbf{k}(n) \mathbf{u}^H(n) \mathbf{P}(n-1)$$

where  $\lambda^{-1}$  denotes the reciprocal of the exponential weighting factor. The variables are as follows

| Variable        | Description                                      |
|-----------------|--|
| $n$             | The current time index                           |
| $\mathbf{u}(n)$ | The vector of buffered input samples at step $n$ |
| $\mathbf{P}(n)$ | The inverse covariance matrix at step $n$        |
| $\mathbf{k}(n)$ | The gain vector at step $n$                      |
| $\mathbf{w}(n)$ | The vector of filter-tap estimates at step $n$   |

| Variable  | Description                      |
|-----------|----------------------------------|
| $y(n)$    | The filtered output at step $n$  |
| $e(n)$    | The estimation error at step $n$ |
| $d(n)$    | The desired response at step $n$ |
| $\lambda$ | The forgetting factor            |

The implementation of the algorithm in the block is optimized by exploiting the symmetry of the inverse covariance matrix  $P(n)$ . This decreases the total number of computations by a factor of two.

Use the **Filter length** parameter to specify the length of the filter weights vector.

The **Forgetting factor (0 to 1)** parameter corresponds to  $\lambda$  in the equations. It specifies how quickly the filter “forgets” past sample information. Setting  $\lambda=1$  specifies an infinite memory. Typically,  $1 - \frac{1}{2L} < \lambda < 1$ , where  $L$  is the filter length. You can specify a forgetting factor using the input port, Lambda, or enter a value in the **Forgetting factor (0 to 1)** parameter in the Block Parameters: RLS Filter dialog box.

Enter the initial filter weights,  $\hat{w}(0)$ , as a vector or a scalar for the **Initial value of filter weights** parameter. When you enter a scalar, the block uses the scalar value to create a vector of filter weights. This vector has length equal to the filter length and all of its values are equal to the scalar value.

The initial value of  $P(n)$  is

$$\frac{1}{\sigma^2}I$$

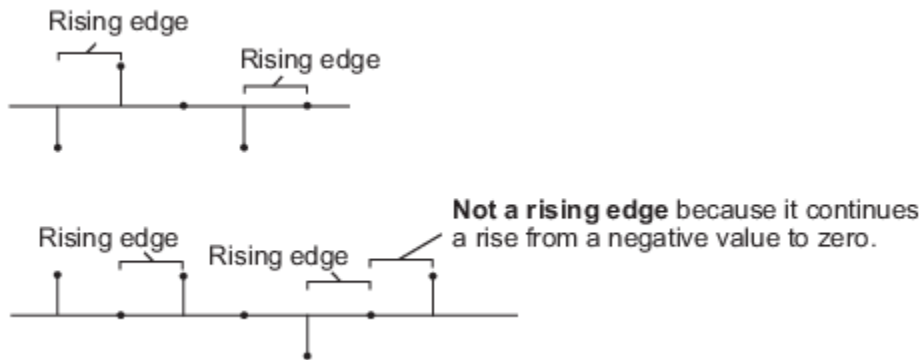
where you specify  $\sigma^2$  in the **Initial input variance estimate** parameter.

When you select the **Adapt port** check box, an Adapt port appears on the block. When the input to this port is nonzero, the block continuously updates the filter weights. When the input to this port is zero, the filter weights remain at their current values.

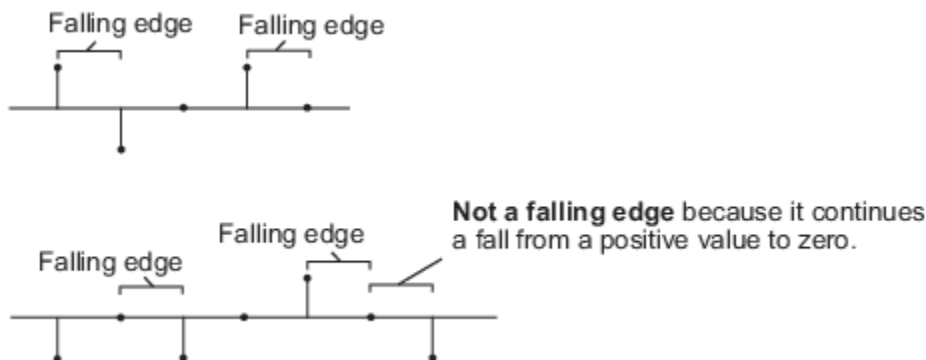
When you want to reset the value of the filter weights to their initial values, use the **Reset input** parameter. The block resets the filter weights whenever a reset event is detected at the Reset port. The reset signal rate must be the same rate as the data signal input.

From the **Reset input** list, select None to disable the Reset port. To enable the Reset port, select one of the following from the **Reset input** list:

- **Rising edge** — Triggers a reset operation when the Reset input does one of the following:
  - Rises from a negative value to a positive value or zero
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero; see the following figure



- **Falling edge** — Triggers a reset operation when the Reset input does one of the following:
  - Falls from a positive value to a negative value or zero
  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero; see the following figure



- **Either edge** — Triggers a reset operation when the Reset input is a **Rising edge** or **Falling edge**, as described above
- **Non-zero sample** — Triggers a reset operation at each sample time that the Reset input is not zero

Select the **Output filter weights** check box to create a Wts port on the block. For each iteration, the block outputs the current updated filter weights from this port.

## Examples

The `rlsdemo` example illustrates a noise cancellation system built around the RLS Filter block.

## Parameters

### Filter length

Enter the length of the FIR filter weights vector.

### Specify forgetting factor via

Select **Dialog** to enter a value for the forgetting factor in the Block parameters: RLS Filter dialog box. Select **Input port** to specify the forgetting factor using the Lambda input port.

**Forgetting factor (0 to 1)**

Enter the exponential weighting factor in the range  $0 \leq \lambda \leq 1$ . A value of 1 specifies an infinite memory. Tunable (Simulink).

**Initial value of filter weights**

Specify the initial values of the FIR filter weights.

**Initial input variance estimate**

The initial value of  $1/P(n)$ .

**Adapt port**

Select this check box to enable the Adapt input port.

**Reset input**

Select this check box to enable the Reset input port.

**Output filter weights**

Select this check box to export the filter weights from the Wts port.

**References**

Hayes, M.H. *Statistical Digital Signal Processing and Modeling*. New York: John Wiley & Sons, 1996.

**Supported Data Types**

- Double-precision floating point
- Single-precision floating point

**See Also**

|                                      |                    |
|--------------------------------------|--------------------|
| Kalman Adaptive Filter<br>(Obsolete) | DSP System Toolbox |
| LMS Filter                           | DSP System Toolbox |
| Block LMS Filter                     | DSP System Toolbox |
| Fast Block LMS Filter                | DSP System Toolbox |

See “Noise Cancellation in Simulink Using Normalized LMS Adaptive Filter” for related information.

**Version History**

Introduced before R2006a

**Extended Capabilities****C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

Generated code relies on the memcpy or memset function (string.h) under certain conditions.

## RMS

Root mean square value of input or sequence of inputs



**Libraries:**  
DSP System Toolbox / Statistics

### Description

The RMS block computes the root mean square (RMS) value of each row or column of the input, or along vectors of a specified dimension of the input. It can also compute the RMS value of the entire input. You can specify the dimension using the **Find the RMS value over** parameter. The RMS block can also track the RMS value in a sequence of inputs over a period of time. To track the RMS value in a sequence of inputs, select the **Running RMS** parameter.

---

**Note** The **Running** mode in the RMS block will be removed in a future release. To compute the running RMS in Simulink, use the Moving RMS block instead.

---

### Ports

#### Input

**In** — Data input  
vector | matrix |  $N$ -D array

The block accepts real-valued or complex-valued multichannel and multidimensional inputs.

This port is unnamed until you select the **Running RMS** parameter and set the **Reset port** parameter to any option other than **None**.

Data Types: single | double

**Rst** — Reset port  
scalar

Specify the reset event that causes the block to reset the running RMS. The sample time of the **Rst** input must be a positive integer multiple of the input sample time.

#### Dependencies

To enable this port, select the **Running RMS** parameter and set the **Reset port** parameter to any option other than **None**.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean

#### Output

**Port\_1** — RMS value along the specified dimension  
scalar | vector | matrix |  $N$ -D array



The data type of the output matches the data type of the input.

When you do not select the **Running RMS** parameter, the block computes the RMS value in each row or column of the input, or along vectors of a specified dimension of the input. It can also compute the RMS value of the entire input at each individual sample time. Each element in the output array  $y$  is the RMS value of the corresponding column, row, or entire input. The output array  $y$  depends on the setting of the **Find the RMS value over** parameter. Consider a three-dimensional input signal of size  $M$ -by- $N$ -by- $P$ . When you set **Find the RMS value over** to:

- **Entire input** — The output at each sample time is a scalar that contains the RMS value of the  $M$ -by- $N$ -by- $P$  input matrix.
- **Each row** — The output at each sample time consists of an  $M$ -by-1-by- $P$  array, where each element contains the RMS value of each vector over the second dimension of the input. For an  $M$ -by- $N$  matrix input, the output at each sample time is an  $M$ -by-1 column vector.
- **Each column** — The output at each sample time consists of a 1-by- $N$ -by- $P$  array, where each element contains the RMS value of each vector over the first dimension of the input. For an  $M$ -by- $N$  matrix input, the output at each sample time is a 1-by- $N$  row vector.

In this mode, the block treats length- $M$  unoriented vector inputs as  $M$ -by-1 column vectors.

- **Specified dimension** — The output at each sample time depends on the value of the **Dimension** parameter. If you set the **Dimension** to 1, the output is the same as when you select **Each column**. If you set the **Dimension** to 2, the output is the same as when you select **Each row**. If you set the **Dimension** to 3, the output at each sample time is an  $M$ -by- $N$  matrix containing the RMS value of each vector over the third dimension of the input.

When you select **Running RMS**, the block tracks the RMS value of each channel in a time sequence of inputs. In this mode, you must also specify a value for the **Input processing** parameter.

- **Elements as channels (sample based)** — The block treats each element of the input as a separate channel. For a three-dimensional input signal of size  $M$ -by- $N$ -by- $P$ , the block outputs an  $M$ -by- $N$ -by- $P$  array. Each element  $y_{ijk}$  of the output contains the RMS value of the element  $u_{ijk}$  for all inputs since the last reset.

When a reset event occurs, the running RMS  $y_{ijk}$  in the current frame is reset to the element  $u_{ijk}$ .

- **Columns as channels (frame based)** — The block treats each column of the input as a separate channel. This option does not support input signals with more than two dimensions. For a two-dimensional input signal of size  $M$ -by- $N$ , the block outputs an  $M$ -by- $N$  matrix. Each element  $y_{ij}$  of the output contains the RMS value of the elements in the  $j$ th column of all inputs since the last reset, up to and including the element  $u_{ij}$  of the current input.

When a reset event occurs, the running RMS for each channel becomes the RMS value of all the samples in the current input frame, up to and including the current input sample.

Data Types: `single` | `double`

## Parameters

### Main Tab

**Running RMS** — Option to select running RMS  
off (default) | on

When you select the **Running RMS** parameter, the block tracks the RMS value of each channel in a time sequence of inputs.

**Find the RMS value over** — Dimension over which the block computes the RMS value  
Each column (default) | Entire input | Each row | Specified dimension

- **Each column** — The block outputs the RMS value over each column.
- **Each row** — The block outputs the RMS value over each row.
- **Entire input** — The block outputs the RMS value over the entire input.
- **Specified dimension** — The block outputs the RMS value over the dimension specified in the **Dimension** parameter.

#### Dependencies

To enable this parameter, clear the **Running RMS** parameter.

**Dimension** — Custom dimension  
1 (default) | scalar

Specify the dimension (one-based value) of the input signal over which the RMS value is computed. The value of this parameter must be greater than 0 and less than the number of dimensions in the input signal.

#### Dependencies

To enable this parameter, set **Find the RMS value over** to Specified dimension.

**Input processing** — Method to process the input in running mode  
Columns as channels (frame based) (default) | Elements as channels (sample based)

- **Columns as channels (frame based)** — The block treats each column of the input as a separate channel. This option does not support input signals with more than two dimensions. For a two-dimensional input signal of size  $M$ -by- $N$ , the block outputs an  $M$ -by- $N$  matrix. Each element  $y_{ij}$  of the output contains the RMS value of the elements in the  $j$ th column of all inputs since the last reset, up to and including the element  $u_{ij}$  of the current input.

When a reset event occurs, the running RMS for each channel becomes the RMS value of all the samples in the current input frame, up to and including the current input sample.

- **Elements as channels (sample based)** — The block treats each element of the input as a separate channel. For a three-dimensional input signal of size  $M$ -by- $N$ -by- $P$ , the block outputs an  $M$ -by- $N$ -by- $P$  array. Each element  $y_{ijk}$  of the output contains the RMS value of the element  $u_{ijk}$  for all inputs since the last reset.

When a reset event occurs, the running RMS  $y_{ijk}$  in the current frame is reset to the element  $u_{ijk}$ .

#### Variable-Size Inputs

When your inputs are of variable size, and you select the **Running RMS** parameter, then:

- If you set the **Input processing** parameter to Elements as channels (sample based), the state is reset.

- If you set the **Input processing** parameter to `Columns as channels (frame based)`, then:
  - When the input size difference is in the number of channels (number of columns), the state is reset.
  - When the input size difference is in the length of channels (number of rows), there is no reset and the running operation is carried out as usual.

### Dependencies

To enable this parameter, select the **Running RMS** parameter.

### Reset port — Reset event

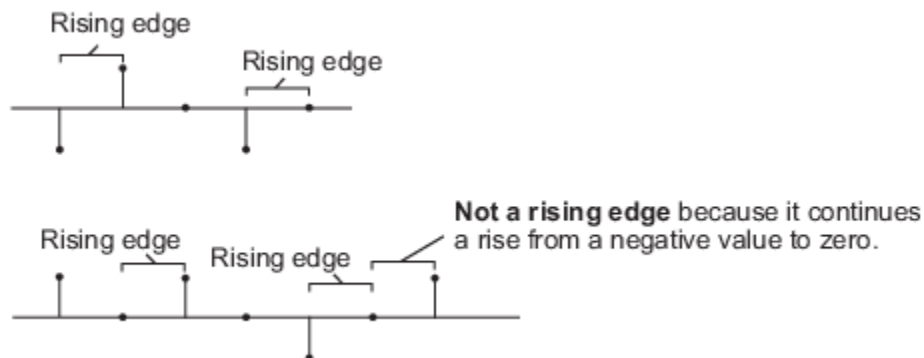
None (default) | Rising edge | Falling edge | Either edge | Non-zero sample

The block resets the running RMS whenever a reset event is detected at the optional **Rst** port. The reset sample time must be a positive integer multiple of the input sample time.

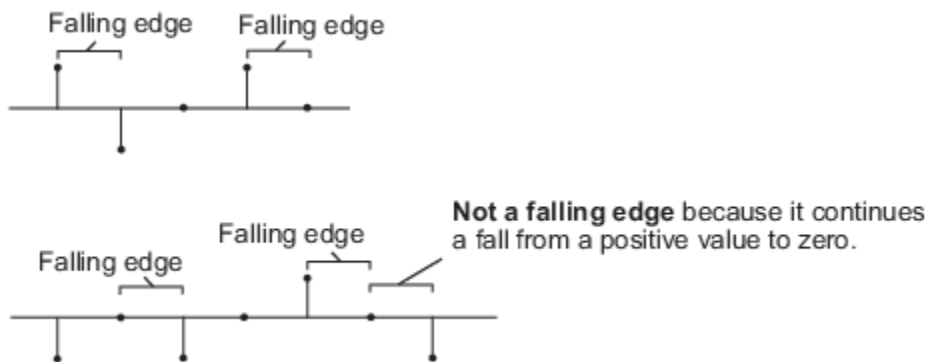
When a reset event occurs while the **Input processing** parameter is set to `Elements as channels (sample based)`, the running RMS for each channel is initialized to the value in the corresponding channel of the current input. Similarly, when the **Input processing** parameter is set to `Columns as channels (frame based)`, the running RMS for each channel becomes the RMS value of all the samples in the current input frame, up to and including the current input sample.

Use this parameter to specify the reset event.

- **None** — Disables the **Rst** port.
- **Rising edge** — Triggers a reset operation when the **Rst** input does one of the following:
  - Rises from a negative value to either a positive value or zero.
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero.



- **Falling edge** — Triggers a reset operation when the **Rst** input does one of the following:
  - Falls from a positive value to a negative value or zero.
  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero.



- **Either edge** — Triggers a reset operation when the **Rst** input is a Rising edge or Falling edge.
- **Non-zero sample** — Triggers a reset operation at each sample time, when the **Rst** input is not zero.

**Note** When running simulations in the Simulink multitasking mode, reset signals have a one-sample latency. Therefore, when the block detects a reset event, there is a one-sample delay at the reset port rate before the block applies the reset. For more information on latency and the Simulink tasking modes, see “Excess Algorithmic Delay (Tasking Latency)” and “Time-Based Scheduling and Code Generation” (Simulink Coder).

### Dependencies

To enable this parameter, select the **Running RMS** parameter.

### Block Characteristics

|                                 |                 |
|---------------------------------|-----------------|
| <b>Data Types</b>               | double   single |
| <b>Direct Feedthrough</b>       | no              |
| <b>Multidimensional Signals</b> | no              |
| <b>Variable-Size Signals</b>    | yes             |
| <b>Zero-Crossing Detection</b>  | no              |

### More About

#### Root Mean Square (RMS)

The RMS value of a discrete-time signal is the square root of the arithmetic mean of the squares of the signal sample values.

For an  $M$ -by- $N$  input matrix  $u$ , the RMS value of the  $j$ th column of the input is given by:

$$y_j = \sqrt{\frac{\sum_{i=1}^M |u_{ij}|^2}{M}} \quad 1 \leq j \leq N$$

## Algorithms

### Root Mean Square (RMS)

When you clear the **Running RMS** parameter in the block and specify a dimension, the block produces results identical to the MATLAB `rms` function, when it is called as `y = rms(u,D)`.

- `u` is the data input.
- `D` is the dimension.
- `y` is the RMS value.

The RMS value along the entire input is identical to calling the `rms` function as `y = rms(u(:))`.

When inputs are complex, the block computes the RMS value of the magnitude of the complex input.

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

## See Also

### Functions

`rms`

### Objects

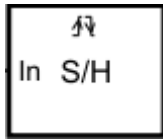
`dsp.MovingRMS`

### Blocks

Moving RMS | Mean

# Sample and Hold

Sample and hold input signal




## Libraries:

DSP System Toolbox / Signal Operations

DSP System Toolbox HDL Support / Signal Operations

## Description

The Sample and Hold block acquires the input at the signal port whenever it receives a trigger event at the trigger port (marked by ). The block then holds the output at the acquired input value until the next triggering event occurs.

## Ports

### Input

**In** — Signal port

scalar | vector | matrix

The signal port can accept data in the form of a scalar, vector, or matrix.

### Dependencies

This port is named **In<Lo>** when you select the **Latch (buffer) input** parameter.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | Boolean | fixed point

**Trigger** — Trigger port

scalar

The trigger input must be a sample-based scalar with sample rate equal to the input frame rate at the signal port. You specify the trigger event using the **Trigger type** parameter.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | Boolean | fixed point

### Output

**Port\_1** — Sample and hold value

scalar | vector | matrix

Sample and hold output, returned as a scalar, vector, or a matrix. The block acquires input at the signal port whenever it receives a trigger event at the trigger port. The block then holds the acquired data until the next triggering event occurs.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | Boolean | fixed point

## Parameters

**Trigger type** — Type of trigger

Rising edge (default) | Falling edge | Either edge

The type of event that triggers the block to acquire the input signal:

- **Rising edge** -- The trigger input rises from a negative value or zero to a positive value.
- **Falling edge** -- The trigger input falls from a positive value or zero to a negative value.
- **Either edge** -- The trigger input either rises from a negative value or zero to a positive value or falls from a positive value or zero to a negative value.

**Initial condition** — Block output prior to first trigger event

0 (default) | scalar | vector | matrix

Specify the block's output before the first trigger event using the **Initial condition** parameter. When the acquired input is an  $M$ -by- $N$  matrix, the **Initial condition** can be an  $M$ -by- $N$  matrix or a scalar repeated across all elements of the matrix. When the input is a length- $M$  unoriented vector, the **Initial condition** can be a length- $M$  row or column vector, or a scalar to be repeated across all elements of the vector.

**Latch (buffer) input** — Latch buffer input

off (default) | on

If you select the **Latch (buffer) input** check box, the block outputs the value of the input from the previous time step until the next triggering event occurs. To use this block in a loop, select this check box.

## Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | Boolean   double   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no  |
| <b>Multidimensional Signals</b> | no  |
| <b>Variable-Size Signals</b>    | no  |
| <b>Zero-Crossing Detection</b>  | no  |

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL code for the Sample and Hold block is generated as a Triggered Subsystem. Similar restrictions apply to both blocks. See “Using Triggered Subsystems for HDL Code Generation” (HDL Coder).

### **HDL Block Properties**

For HDL block property descriptions, see “HDL Block Properties: General” (HDL Coder).

### **Best Practices**

When using the Sample and Hold block in models targeted for HDL code generation, consider the following:

- For synthesis results to match Simulink results, drive the trigger port with registered logic (with a synchronous clock) on the FPGA.
- It is good practice to put a unit delay on the output signal. Doing so prevents the code generator from inserting extra bypass registers in the HDL code.
- The use of triggered subsystems, such as the Sample and Hold block, can affect synthesis results in the following ways:
  - In some cases, the system clock speed can drop by a small percentage.
  - Generated code uses more resources, scaling with the number of triggered subsystem instances.

### **Restrictions**

The Sample and Hold block must meet the following conditions:

- The DUT (i.e., the top-level subsystem for which code is generated) must not be the Sample and Hold block.
- The trigger signal must be a scalar.
- The data type of the trigger signal must be either `boolean` or `ufix1`.
- The output of the Sample and Hold block must have an initial value of 0.
- The input, output, and trigger signal of the Sample and Hold block must run at the same rate. If one of the input or the trigger signals is an output of a Signal Editor block, see “Using the Signal Editor Block” (HDL Coder) for how to match rates.

### **Fixed-Point Conversion**

Design and simulate fixed-point systems using Fixed-Point Designer™.

## **See Also**

### **Blocks**

Downsample | N-Sample Switch



# Sample-Rate Converter

Multistage sample-rate conversion



## Libraries:

DSP System Toolbox / Signal Operations

## Description

The Sample-Rate Converter block implements a multistage FIR sample-rate converter. The multistage FIR converter converts the rate of each channel of the input signal from the input sample rate to the output sample rate. Multistage implementations minimize the amount of computation required for the sample-rate conversions by first reducing the sample rate of the input signal. Next, the block determines the optimal number of decimators and interpolators based on the parameters you specify in the block dialog box. The block then accordingly designs filters in the individual stages.

This block supports SIMD code generation. For details, see “Code Generation” on page 2-1246.

## Ports

### Input

**Port 1** — Input signal

scalar | vector | matrix

Specify the input signal as a real or a complex-valued scalar, vector, or a matrix.

The block treats each column of a two-dimensional input signal as a separate channel. If the input is a two-dimensional signal, the first dimension represents the channel length (or frame size), and the second dimension represents the number of channels. If the input is a one-dimensional signal, then the block interprets the signal as having a single channel.

When you input a variable-size signal (frame length changes during simulation), the frame length of the signal can be arbitrary, that is, the input frame length does not have to be a multiple of the decimation factor. When you input a fixed-size signal (frame length does not change during simulation), the frame length can be arbitrary only when you select the **Allow arbitrary frame length for fixed-size input signals** parameter.

To determine the decimation factor, click the **View Info** button in the block dialog box.

Data Types: single | double

### Output

**Port 1** — Resampled signal

scalar | vector | matrix

The Sample-Rate Converter block outputs a signal that has the same data type and complexity as the input signal.

The output signal dimensions depend on the input signal dimensions, the overall interpolation and decimation factors of the Sample-Rate Converter block, and whether you select the **Allow arbitrary frame length for fixed-size input signals** parameter. You can determine the overall interpolation factor  $L$  and the decimator factor  $M$  by clicking the **View Info** button in the block dialog box.

This table provides more details on the dimensions of the output signal when you input a fixed-size input signal.

### Fixed-Size Input Signal

| Input Signal  | Output Signal Dimensions   |
|---|--|
| $P$ -by-1 column vector, where $P$ is a multiple of the decimation factor $M$     | $(LP/M)$ -by-1, where $L$ is the interpolation factor  |
| $P$ -by-1 column vector, where $P$ is not a multiple of the decimation factor $M$ | $\text{ceil}(LP/M)$ -by-1 when you select <b>Allow arbitrary frame length for fixed-size input signals</b> .<br><br>If you do not select <b>Allow arbitrary frame length for fixed-size input signals</b> , the block errors.    |
| $P$ -by- $Q$ matrix, where $P$ is a multiple of the decimation factor $M$         | $(LP/M)$ -by- $Q$  |
| $P$ -by- $Q$ matrix, where $P$ is not a multiple of the decimation factor $M$     | $\text{ceil}(LP/M)$ -by- $Q$ when you select <b>Allow arbitrary frame length for fixed-size input signals</b> .<br><br>If you do not select <b>Allow arbitrary frame length for fixed-size input signals</b> , the block errors. |

This table gives more details on the dimensions of the output signal when you input a variable-size input signal. When you input a variable-size signal (frame length changes during simulation), the **Allow arbitrary frame length for fixed-size input signals** parameter is visible in the block dialog box but does not have any impact on the input frame length. You can input a variable-size signal of any frame length even if you do not select the **Allow arbitrary frame length for fixed-size input signals** parameter.

### Variable-Size Input Signal

| Input Signal | Output Signal Dimensions     |
|--------------|------------------------------|
| $P$ -by- $Q$ | $\text{ceil}(LP/M)$ -by- $Q$ |

Data Types: `single` | `double`

## Parameters

**Sample rate of input signal (Hz)** — Sample rate of input signal in Hz

48e3 (default) | positive scalar

Specify the sample rate of the input signal as a positive scalar in Hz. The input sample rate must be greater than the bandwidth of interest.

**Sample rate of output signal (Hz)** — Sample rate of output signal in Hz

96e3 (default) | positive scalar

Specify the sample rate of the output signal as a positive scalar in Hz. The output sample rate must be greater than the bandwidth of interest.

**Tolerance for output sample rate** — Maximum tolerance for output sample rate

0 (default) | positive scalar in the range [0,0.5]

Specify the maximum tolerance for the output sample rate as a positive scalar in the range [0,0.5]. The default is 0.

The actual output sample rate varies but is within the specified range. For example, if you set the **Tolerance for output sample rate** to 0.01, then the actual output sample rate is in the range given by sample rate of output signal  $\pm 1\%$ . This flexibility allows for a simpler filter design.

**Two-sided bandwidth of interest (Hz)** — Two-sided bandwidth of interest in Hz

40e3 (default) | positive scalar

Specify the two-sided bandwidth of interest (after the rate of conversion) as a positive scalar in Hz.

**Stopband attenuation (dB)** — Minimum attenuation for aliased components in dB

80 (default) | positive scalar

Specify the minimum amount of attenuation for aliased components in the stopband as a positive scalar in dB. This parameter is the minimum amount by which the block attenuates any aliasing involved in the process.

**Allow arbitrary frame length for fixed-size input signals** — Allow arbitrary frame length for fixed-size input signals

off (default) | on

Specify whether fixed-size input signals (whose size does not change during simulation) can have an arbitrary frame length, where the frame length does not have to be a multiple of the decimation factor. The block uses this parameter only for fixed-size input signals and ignores it if the input data varies in size during simulation.

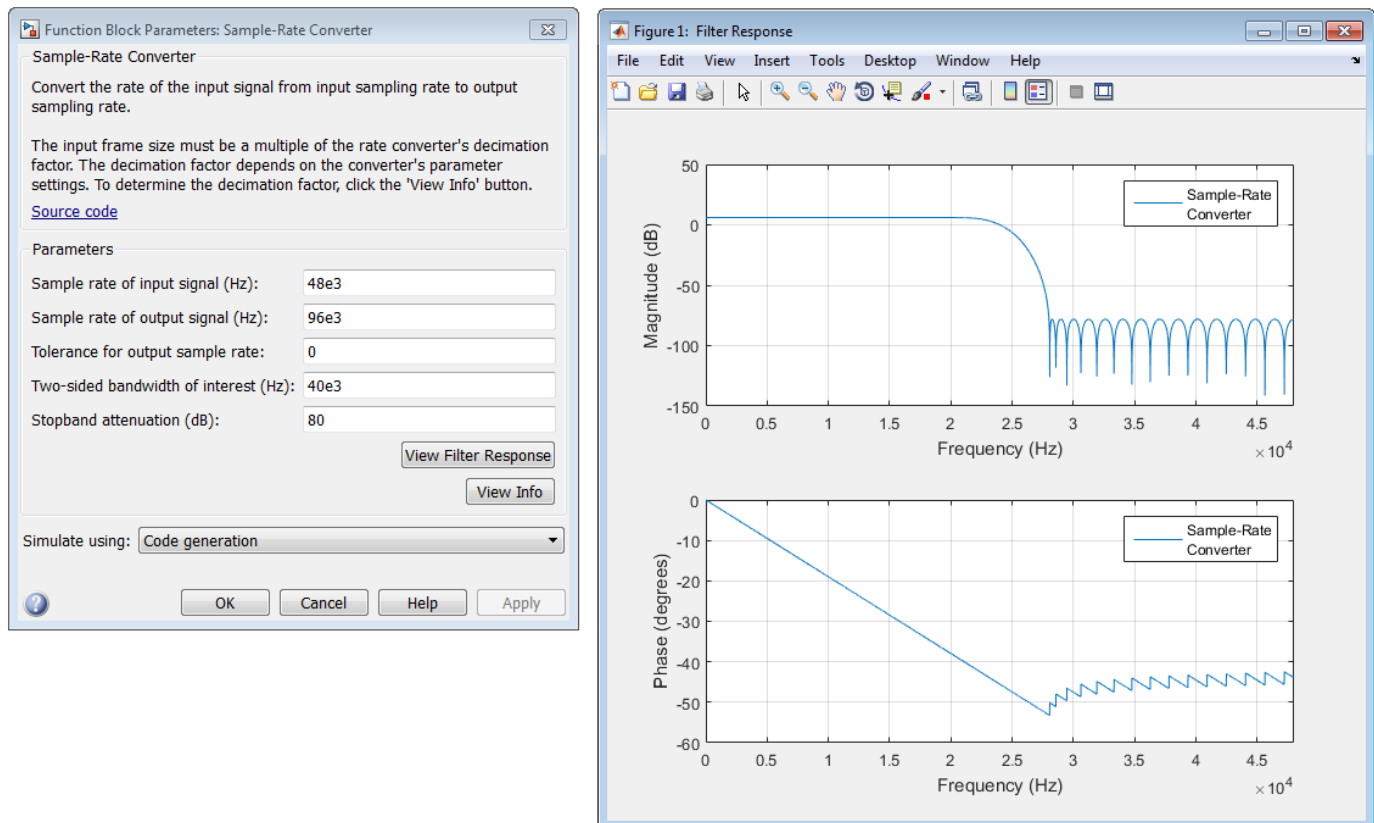
When the input signal is a variable-size signal, the signal can have arbitrary frame length, that is, the frame length does not have to be a multiple of the decimation factor.

For fixed-size input signals, if you:

- Select the **Allow arbitrary frame length for fixed-size input signals** parameter, the frame length of the signal does not have to be a multiple of the decimation factor. If the input is not a multiple of the decimation factor, then the output is generally a variable-size signal. Therefore, to support arbitrary input size, the block must also support variable-size operations, which you can enable by selecting the **Allow arbitrary frame length for fixed-size input signals** parameter.
- Clear the **Allow arbitrary frame length for fixed-size input signals** parameter, the input frame length must be a multiple of the decimation factor.

### View Filter Response — View Filter Response button

Opens the Filter Visualization Tool FVTool and displays the magnitude/phase response of the Sample-Rate Converter. The response is based on the block parameters. FVTool updates when you change the parameters.

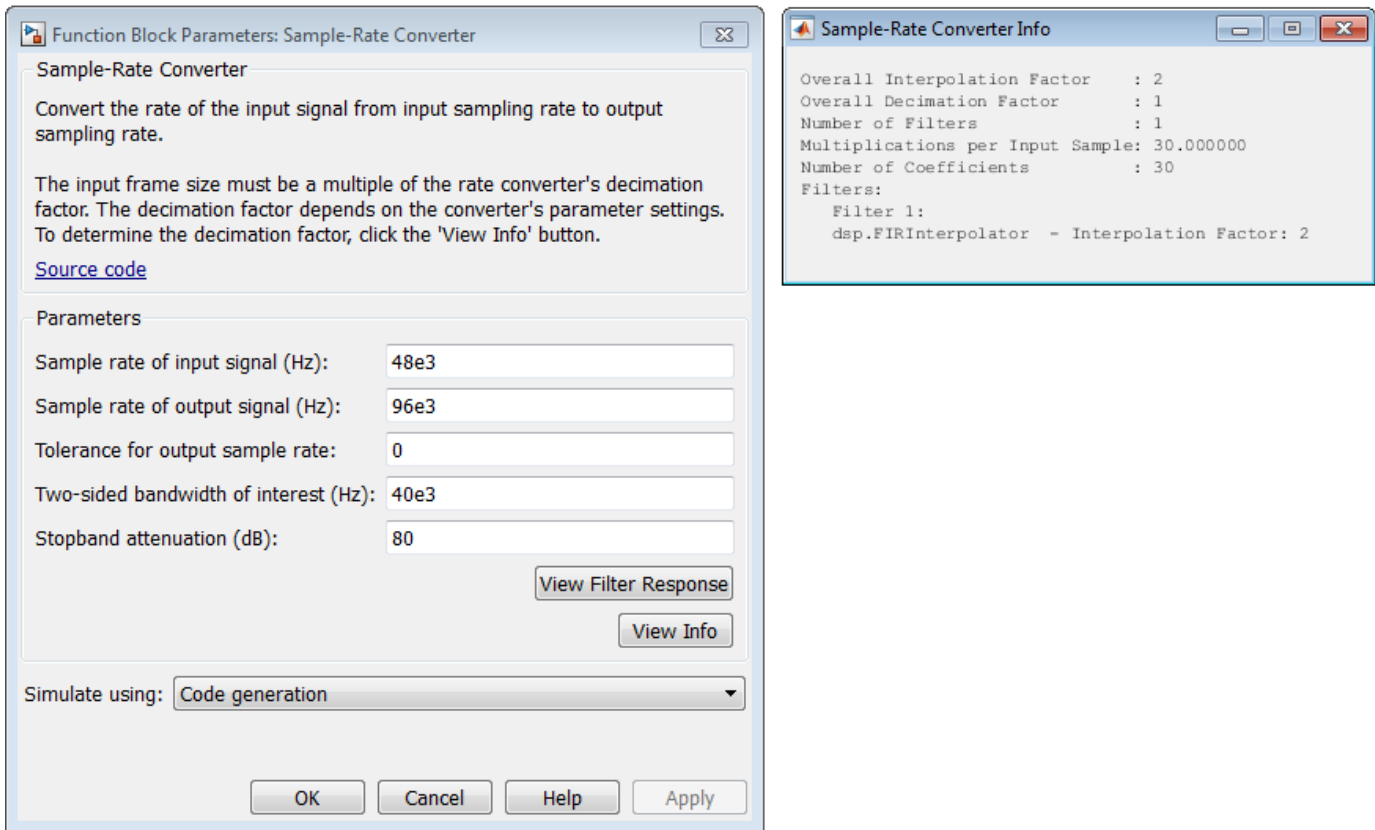


To update the magnitude response while FVTool is running, modify the parameters in the dialog box and click **Apply**.

### View Info — Display information about filter button

Click on the **View Info** button to display the filter information of the Sample-Rate Converter block:

- Overall Interpolation Factor
- Overall Decimation Factor
- Number of Filters
- Multiplication per Input Sample
- Number of Coefficients
- Filters



The button brings the functionality of the `info` function into the Simulink environment.

**Simulate using** — Type of simulation to run

Code generation (default) | Interpreted execution

Type of simulation to run. You can set this parameter to:

- Code generation (default)

Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than Interpreted execution.

- Interpreted execution

Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than Code generation.

## Block Characteristics

|                           |                 |
|---------------------------|-----------------|
| <b>Data Types</b>         | double   single |
| <b>Direct Feedthrough</b> | no              |

|                                 |     |
|---------------------------------|-----|
| <b>Multidimensional Signals</b> | no  |
| <b>Variable-Size Signals</b>    | yes |
| <b>Zero-Crossing Detection</b>  | no  |

## Algorithms

- The general multistage sample rate converter performs a multistage decimation, a single-stage sample rate conversion, and a multistage interpolation. Actual designs include at most two of these steps.
- The procedure automatically determines the optimal number of decimation or interpolation stages. In special cases, the algorithm can perform a decimation or interpolation in a single stage.
- The algorithm always attempts to start by reducing the sample rate. This decreases the amount of computation required. The decimation step ensures that the intermediate sample rate does not go below the bandwidth of interest and that the block does not filter out any information.
- Each individual stage uses halfband or Nyquist filters to minimize the number of nonzero coefficients.
- The algorithm allows transition-band aliasing because it decreases the implementation cost. It keeps the signal within the bandwidth of interest alias free up to the value specified by the `StopbandAttenuation` property.

## Version History

Introduced in R2015b

### R2023a: Support for arbitrary input frame length

This block supports input signals with arbitrary frame lengths when the:

- Input signal is a fixed-size signal (frame length does not change during simulation) and you select the **Allow arbitrary frame length for fixed-size input signals** parameter.
- Input signal is a variable-size signal (frame length changes during simulation).

When this block supports an input signal with an arbitrary frame length, the input frame length does not have to be a multiple of the decimation factor.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

The Sample-Rate Converter block supports SIMD code generation using Intel AVX2 technology under these conditions:

- For upsampling, the ratio of output sample rate to input sample rate must be an integer.
- For downsampling, the ratio of input sample rate to output sample rate must be an integer.

- Input signal has a data type of `single` or `double`.

The SIMD technology significantly improves the performance of the generated code.

## **See Also**

### **Objects**

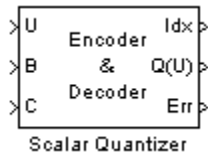
`dsp.SampleRateConverter`

### **Blocks**

Farrow Rate Converter

## Scalar Quantizer (Obsolete)

Convert input signal into set of quantized output values or index values, or convert set of index values into quantized output signal



## Library

dspobslib

## Description

---

**Note** The Scalar Quantizer block is still supported but is likely to be obsoleted in a future release. We strongly recommend replacing this block with the Scalar Quantizer Encoder block or the Scalar Quantizer Decoder block.

---

The Scalar Quantizer block has three modes of operation. In **Encoder** mode, the block maps each input value to a quantization region by comparing the input value to the quantizer boundary points defined in the **Boundary points** parameter. The block outputs the index of the associated region. In **Decoder** mode, the block transforms the input index values into quantized output values, defined in the **Codebook** parameter. In the **Encoder and Decoder** mode, the block performs both the encoding and decoding operations. The block outputs the index values and the quantized output values.

You can select how you want to enter the **Boundary points** and/or **Codebook** values using the **Source of quantizer** parameters. When you select **Specify via dialog**, type the parameters into the block parameters dialog box. Select **Input ports**, and port B and/or C appears on the block. In **Encoder** and **Encoder and decoder** mode, the input to port B is used as the **Boundary points**. In **Decoder** and **Encoder and decoder** mode, the input to port C is used as the **Codebook**.

In **Encoder** and **Encoder and decoder** mode, the **Boundary points** are the values used to break up the input signal into regions. Each region is specified by an index number. When your first boundary point is  $-\infty$  and your last boundary point is  $\infty$ , your quantizer is unbounded. When your first and last boundary point is finite, your quantizer is bounded. When only your first or last boundary point is  $-\infty$  or  $\infty$ , your quantizer is semi-bounded.

For instance, when your input signal ranges from 0 to 11, you can create a bounded quantizer using the following boundary points:

```
[0 0.5 3.7 5.8 6.0 11]
```

The boundary points can have equal or varied spacing. Any input values between 0 and 0.5 would correspond to index 0. Input values between 0.5 and 3.7 would correspond to index 1, and so on.



Suppose you wanted to create an unbounded quantizer with the following boundary points:

```
[-inf 0 2 5.5 7.1 10 inf]
```

When your input signal has values less than 0, these values would be assigned to index 0. When your input signal has values greater than 10, these values would be assigned to index 6.

When an input value is the same as a boundary point, the **Tie-breaking rule** parameter defines the index to which the value is assigned. When you want the input value to be assigned to the lower index value, select **Choose the lower index**. To assign the input value with the higher index, select **Choose the higher index**.

In **Decoder** and **Encoder** and **decoder** mode, the **Codebook** is a vector of quantized output values that correspond to each index value.

In **Encoder** and **Encoder** and **decoder** mode, the **Searching method** determines how the appropriate quantizer index is found. Select **Linear** and the Scalar Quantizer block compares the input value to the first region defined by the first two boundary points. When the input value does not fall within this region, the block then compares the input value to the next region. This process continues until the input value is determined to be within a region and is associated with the appropriate index value. The computational cost of this process is of the order  $P$ , where  $P$  is the number of boundary points.

Select **Binary** for the **Searching method** and the block compares the input value to the middle value of the boundary points vector. When the input value is larger than this boundary point, the block discards the boundary points that are lower than this middle value. The block then compares the input value to the middle boundary point of the new range, defined by the remaining boundary points. This process continues until the input value is associated with the appropriate index value. The computational cost of this process is of the order  $\log_2 P$ , where  $P$  is the number of boundary points. In most cases, the **Binary** option is faster than the **Linear** option.

In **Decoder** mode, the input to this block is a vector of index values, where  $0 \leq \text{index} < N$  and  $N$  is the length of the codebook vector. Use the **Action for out of range input** parameter to determine what happens when an input index value is out of this range. When you want any index values less than 0 to be set to 0 and any index values greater than or equal to  $N$  to be set to  $N - 1$ , select **Clip**. When you want to be warned when any index values less than 0 are set to 0 and any index values greater than or equal to  $N$  are set to  $N - 1$ , select **Clip and warn**. When you want the simulation to stop and display an error when the index values are out of range, select **Error**.

In **Encoder** and **decoder** mode, you can select the **Output the quantization error** check box. The quantization error is the difference between the input value and the quantized output value. Select this check box to output the quantization error for each input value from the **Err** port on this block.

### Data Type Support

In **Encoder** mode, the input data values and the boundary points can be the input to the block at ports **U** and **B**. Similarly, in **Encoder** and **decoder** mode, the codebook values can also be the input to the block at port **C**. The data type of the input data values, boundary points, and codebook values can be **double**, **single**, **uint8**, **uint16**, **uint32**, **int8**, **int16**, or **int32**. In **Decoder** mode, the input to the block can be the index values and the codebook values. The data type of the index input to the block at port **Idx** can be **uint8**, **uint16**, **uint32**, **int8**, **int16**, or **int32**. The data type of the codebook values can be **double**, **single**, **uint8**, **uint16**, **uint32**, **int8**, **int16**, or **int32**.

In Encoder mode, the output of the block is the index values. In Encoder and decoder mode, the output can also include the quantized output values and the quantization error. In Encoder and Encoder and decoder mode, use the **Output index data type** parameter to specify the data type of the index output from the block at port Idx. The data type of the index output can be uint8, uint16, uint32, int8, int16, or int32. The data type of the quantized output and the quantization error can be double, single, uint8, uint16, uint32, int8, int16, or int32. In Decoder mode, the output of the block is the quantized output values. Use the **Output data type** parameter to specify the data type of the quantized output values. The data type can be double, single, uint8, uint16, uint32, int8, int16, int32.

---

**Note** The input data, codebook values, boundary points, quantization error, and the quantized output values must have the same data type whenever they are present in any of the quantizer modes.

---

## Parameters

### Quantizer mode

Specify Encoder, Decoder, or Encoder and decoder as a mode of operation.

### Source of quantizer parameters

Choose Specify via dialog to type the parameters into the block parameters dialog box. Select Input ports to specify the parameters using the block's input ports. In Encoder and Encoder and decoder mode, input the **Boundary points** using port B. In Decoder and Encoder and decoder mode, input the **Codebook** values using port C.

### Boundary points

Enter a vector of values that represent the boundary points of the quantizer regions. Tunable (Simulink).

### Codebook

Enter a vector of quantized output values that correspond to each index value. Tunable (Simulink).

### Searching method

Select Linear and the block finds the region in which the input value is located using a linear search. Select Binary and the block finds the region in which the input value is located using a binary search.

### Tie-breaking rule

Set this parameter to determine the behavior of the block when the input value is the same as the boundary point. When you select Choose the lower index, the input value is assigned to lower index value. When you select Choose the higher index, the value is assigned to the higher index.

### Action for out of range input

Choose the block's behavior when an input index value is out of range, where  $0 \leq \text{index} < N$  and  $N$  is the length of the codebook vector. Select Clip, when you want any index values less than 0 to be set to 0 and any index values greater than or equal to  $N$  to be set to  $N - 1$ . Select Clip and warn, when you want to be warned when any index values less than 0 are set to 0 and any index values greater than or equal to  $N$  are set to  $N - 1$ . Select Error, when you want the simulation to stop and display an error when the index values are out of range.

**Output the quantization error**

In Encoder and decoder mode, select this check box to output the quantization error from the Err port on this block.

**Output index data type**

In Encoder and Encoder and decoder mode, specify the data type of the index output from the block at port Idx. The data type can be uint8, uint16, uint32, int8, int16, or int32.

This parameter becomes visible when you select the **Show additional parameters** check box.

**Output data type**

In Decoder mode, specify the data type of the quantized output. The data type can be uint8, uint16, uint32, int8, int16, int32, single, or double. This parameter becomes visible when you select Specify via dialog for the **Source of quantizer parameters** and you select the **Show additional parameters** check box.

**References**

Gersho, A. and R. Gray. *Vector Quantization and Signal Compression*. Boston: Kluwer Academic Publishers, 1992.

**Supported Data Types**

- Double-precision floating point
- Single-precision floating point
- Fixed point
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

For more information on what data types are supported for each quantizer mode, see “Data Type Support” on page 2-1249.

**See Also**

Quantizer

Scalar Quantizer Decoder

Scalar Quantizer Encoder

Uniform Encoder

Uniform Decoder

Simulink

DSP System Toolbox

DSP System Toolbox

DSP System Toolbox

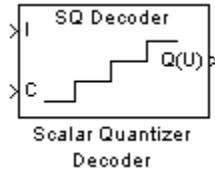
DSP System Toolbox

**Version History**

**Introduced in R2008b**

## Scalar Quantizer Decoder

Convert each index value into quantized output value



## Library

Quantizers

dspquant2

## Description

The Scalar Quantizer Decoder block transforms the zero-based input index values into quantized output values. The set of all possible quantized output values is defined by the **Codebook values** parameter.

Use the **Codebook values** parameter to specify a matrix containing all possible quantized output values. You can select how you want to enter the codebook values using the **Source of codebook** parameter. When you select *Specify via dialog*, type the codebook values into the block parameters dialog box. When you select *Input port*, port C appears on the block. The block uses the input to port C as the **Codebook values** parameter.

The input to this block is a vector of integer index values, where  $0 \leq \text{index} < N$  and  $N$  is the number of distinct codeword vectors in the codebook matrix. Use the **Action for out of range index value** parameter to determine what happens when an input index value is outside this range. When you want any index value less than 0 to be set to 0 and any index value greater than or equal to  $N$  to be set to  $N - 1$ , select *Clip*. When you want to be warned when clipping occurs, select *Clip and warn*. When you want the simulation to stop and the block to display an error when the index values are out of range, select *Error*.

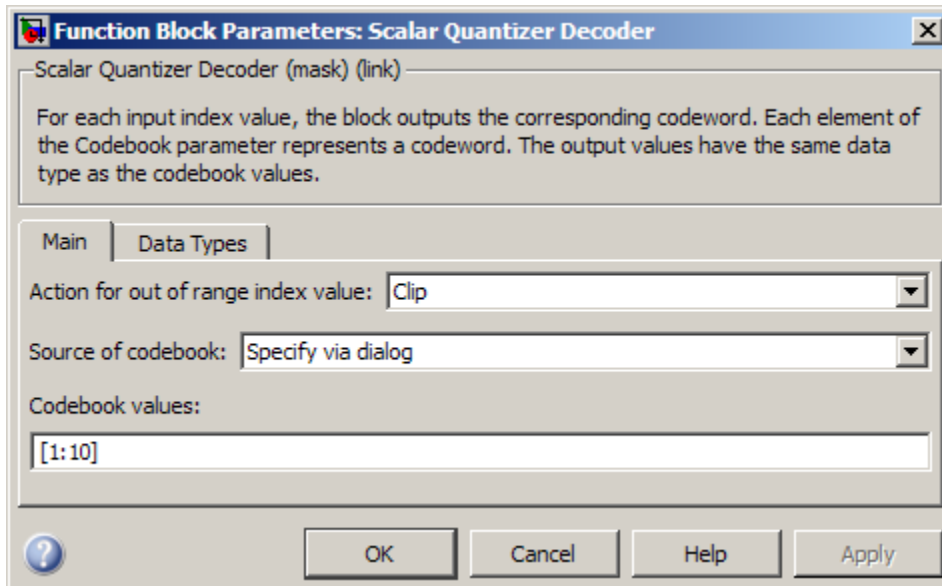
## Data Type Support

The data type of the index values input at port I can be `uint8`, `uint16`, `uint32`, `int8`, `int16`, or `int32`. The data type of the codebook values input at port C can be `double`, `single`, or `Fixed-point`.

The output of the block is the quantized output values. If, for the **Source of codebook** parameter, you select *Specify via dialog*, the **Codebook and output data type** parameter appears. You can use this parameter to specify the data type of the codebook and quantized output values. In this case, the data type of the output values can be *Same as input*, `double`, `single`, `Fixed-point`, or *User-defined*. If, for the **Source of codebook** parameter you select *Input port*, the quantized output values have the same data type as the codebook values input at port C.

## Dialog Box

The **Main** pane of the Scalar Quantizer Decoder block dialog appears as follows.



### Action for out of range index value

Use this parameter to determine the block's behavior when an input index value is out of range, where  $0 \leq \text{index} < N$  and  $N$  is the length of the codebook vector. Select **Clip**, when you want any index values less than 0 to be set to 0 and any index values greater than or equal to  $N$  to be set to  $N - 1$ . Select **Clip and warn**, when you want to be warned when clipping occurs. Select **Error**, when you want the simulation to stop and the block to display an error when the index values are outside the range.

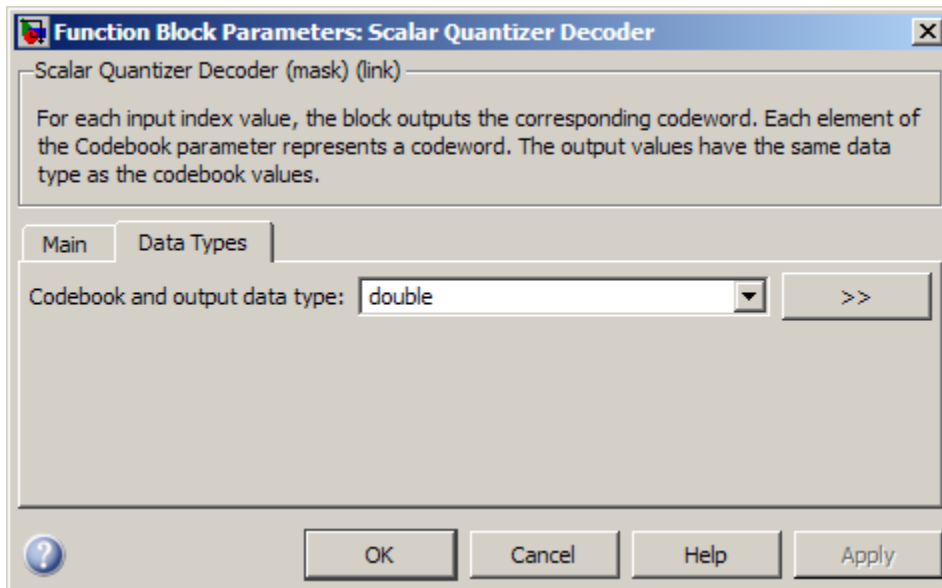
### Source of codebook

Choose **Specify via dialog** to type the codebook values into the block parameters dialog box. Select **Input port** to specify the codebook using input port C.

### Codebook values

Enter a vector of quantized output values that correspond to each index value. Tunable (Simulink).


The **Data Types** pane of the Scalar Quantizer Decoder block dialog appears as follows.



### Codebook and output data type

Specify the data type of the codebook and quantized output values. You can select one of the following:

- A rule that inherits a data type, for example, `Inherit: Same as input`.
- A built in data type, such as `double`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Control Data Types of Signals” (Simulink) in *Simulink User's Guide* (Simulink) for more information.

This parameter is available only when you set the **Source of codebook** parameter to `Specify via dialog`. If you set the **Source of codebook** parameter to `Input port`, the output values have the same data type as the input codebook values.

### Codebook and Output Data Type Assistant Parameters

#### Mode

Select how you would like to specify the data type properties of the **Codebook and output data type**. You can choose:

- `Inherit` — Lets you specify a rule for inheriting a data type, for example, `Inherit: Same as input`
- `Built in`— Lets you specify a built in data type, for example, `double`
- `Fixed point` — Lets you specify the fixed-point attributes of the data type.
- `Expression` — Lets you specify an expression that evaluates to a valid data type, for example, `fixdt(1,16)`

**Signedness**

Specify the **Signedness** for the **Codebook and output data type**.

**Scaling**

Specify the **Scaling** for the **Codebook and output data type**.

For more information see “Scaling” in the DSP System Toolbox User's Guide.

**Word length**

Specify the **Word length** for the **Codebook and output data type**.

**Fraction length**

Specify the **Fraction length** for the **Codebook and output data type**.

**Data type override**

Specify the data type override mode. You can select one of the following options:

- **Inherit** — Inherits the data type override setting specified for the model.
- **Off** — Ignores the data type override setting specified for the model and uses the fixed-point data type you specify

This parameter appears only when you set the **Mode** parameter to **Built in** or **Fixed Point**. For more information, see “Specify Data Types Using Data Type Assistant” (Simulink).

**References**

Gersho, A. and R. Gray. *Vector Quantization and Signal Compression*. Boston: Kluwer Academic Publishers, 1992.

**Supported Data Types**

| <b>Port</b> | <b>Supported Data Types</b>  |
|-------------|--|
| I           | <ul style="list-style-type: none"> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul>   |
| C           | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point</li> <li>• 8-, 16-, and 32-bit signed integers</li> </ul> |

| Port | Supported Data Types  |
|------|---|
| Q(U) | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul> |

For more information on what data types are supported for each quantizer mode, see “Data Type Support” on page 2-1252.

## See Also

Quantizer

Scalar Quantizer Encoder

Uniform Encoder

Uniform Decoder

Simulink

DSP System Toolbox

DSP System Toolbox

DSP System Toolbox

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.



# Scalar Quantizer Design

(Removed) Start Scalar Quantizer Design Tool (SQDTool) to design scalar quantizer using Lloyd algorithm

---

**Note** The Scalar Quantizer Design block and the associated SQDTool have been removed. For more information, see “Compatibility Considerations”.

---

## Library

Quantizers

dspquant2

## Description

Double-click on the Scalar Quantizer Design block to start SQDTool, a GUI that allows you to design and implement a scalar quantizer. Based on your input values, SQDTool iteratively calculates the codebook values that minimize the mean squared error until the stopping criteria for the design process is satisfied. The block uses the resulting quantizer codebook values and boundary points to implement your scalar quantizer encoder and/or decoder.

For the **Training Set** parameter, enter a set of observations, or samples, of the signal you want to quantize. This data can be any variable defined in the MATLAB workspace including a variable created using a MATLAB function, such as the default value `randn(10000,1)`.

You have two choices for the **Source of initial codebook** parameter. Select **Auto-generate** to have the block choose the values of the initial codebook vector. In this case, the minimum training set value becomes the first codeword, and the maximum training set value becomes the last codeword. Then, the remaining initial codewords are equally spaced between these two values to form a codebook vector of length  $N$ , where  $N$  is the **Number of levels** parameter. When you select **User defined**, enter the initial codebook values in the **Initial codebook** field. Then, set the **Source of initial boundary points** parameter. You can select **Mid-points** to locate the boundary points at the midpoint between the codewords. To calculate the mid-points, the block internally arranges the initial codebook values in ascending order. You can also choose **User defined** and enter your own boundary points in the **Initial boundary points (unbounded)** field. Only one boundary point can be located between two codewords. When you select **User defined** for the **Source of initial boundary points** parameter, the values you enter in the **Initial codebook** and **Initial boundary points (unbounded)** fields must be arranged in ascending order.

---

**Note** This block assumes that you are designing an unbounded quantizer. Therefore, the first and last boundary points are always `-inf` and `inf` regardless of any other boundary point values you might enter.

---

After you have specified the quantization parameters, the block performs an iterative process to design the optimal scalar quantizer. Each step of the design process involves using the Lloyd

algorithm to calculate codebook values and quantizer boundary points. Then, the block calculates the squared quantization error and checks whether the stopping criteria has been satisfied.

The two possible options for the **Stopping criteria** parameter are `Relative threshold` and `Maximum iteration`. When you want the design process to stop when the fractional drop in the squared quantization error is below a certain value, select `Relative threshold`. Then, for **Relative threshold**, type the maximum acceptable fractional drop. When you want the design process to stop after a certain number of iterations, choose `Maximum iteration`. Then, enter the maximum number of iterations you want the block to perform in the **Maximum iteration** field. For **Stopping criteria**, you can also choose `Whichever comes first` and enter a **Relative threshold** and **Maximum iteration** value. The block stops iterating as soon as one of these conditions is satisfied.

With each iteration, the block quantizes the training set values based on the newly calculated codebook values and boundary points. When the training point lies on a boundary point, the algorithm uses the **Tie-breaking rules** parameter to determine which region the value is associated with. When you want the training point to be assigned to the lower indexed region, select `Lower indexed codeword`. To assign the training point with the higher indexed region, select `Higher indexed codeword`.

The **Searching methods** parameter determines how the block compares the training points to the boundary points. Select `Linear search` and SQDTool compares each training point to each quantization region sequentially. This process continues until all the training points are associated with the appropriate regions.

Select `Binary search` for the **Searching methods** parameter and the block compares the training point to the middle value of the boundary points vector. When the training point is larger than this boundary point, the block discards the lower boundary points. The block then compares the training point to the middle boundary point of the new range, defined by the remaining boundary points. This process continues until the training point is associated with the appropriate region.

Click **Design and Plot** to design the quantizer with the parameter values specified on the left side of the GUI. The performance curve and the staircase character of the quantizer are updated and displayed in the figures on the right side of the GUI.

---

**Note** You must click **Design and Plot** to apply any changes you make to the parameter values in the SQDTool dialog box.

---

SQDTool can export parameter values that correspond to the figures displayed in the GUI. Click the **Export Outputs** button, or press **Ctrl+E**, to export the **Final Codebook**, **Final Boundary Points**, and **Error** values to the workspace, a text file, or a MAT-file. The **Error** values represent the mean squared error for each iteration.

In the **Model** section of the GUI, specify the destination of the block that will contain the parameters of your quantizer. For **Destination**, select `Current model` to create a block with your parameters in the model you most recently selected. Type `gcs` in the MATLAB Command Window to display the name of your current model. Select `New model` to create a block in a new model file.

From the **Block type** list, select `Encoder` to design a Scalar Quantizer Encoder block. Select `Decoder` to design a Scalar Quantizer Decoder block. Select `Both` to design a Scalar Quantizer Encoder block and a Scalar Quantizer Decoder block.

In the **Encoder block name** field, enter a name for the Scalar Quantizer Encoder block. In the **Decoder block name** field, enter a name for the Scalar Quantizer Decoder block. When you have a Scalar Quantizer Encoder and/or Decoder block in your destination model with the same name, select the **Overwrite target block(s)** check box to replace the block's parameters with the current parameters. When you do not select this check box, a new Scalar Quantizer Encoder and/or Decoder block is created in your destination model.

Click **Generate Model**. SQDTool uses the parameters that correspond to the current plots to set the parameters of the Scalar Quantizer Encoder and/or Decoder blocks.

## Parameters

### Training Set

Enter the samples of the signal you would like to quantize. This data set can be a MATLAB function or a variable defined in the MATLAB workspace. The typical length of this data vector is  $1e6$ .

### Source of initial codebook

Select **Auto-generate** to have the block choose the initial codebook values. Select **User defined** to enter your own initial codebook values.

### Number of levels

Enter the length of the codebook vector. For a b-bit quantizer, the length should be  $N = 2^b$ .

### Initial codebook

Enter your initial codebook values. From the **Source of initial codebook** list, select **User defined** in order to activate this parameter.

### Source of initial boundary points

Select **Mid-points** to locate the boundary points at the midpoint between the codebook values. Choose **User defined** to enter your own boundary points. From the **Source of initial codebook** list, select **User defined** in order to activate this parameter.

### Initial boundary points (unbounded)

Enter your initial boundary points. This block assumes that you are designing an unbounded quantizer. Therefore, the first and last boundary point are  $-\infty$  and  $\infty$ , regardless of any other boundary point values you might enter. From the **Source of initial boundary points** list, select **User defined** in order to activate this parameter.

### Stopping criteria

Choose **Relative threshold** to enter the maximum acceptable fractional drop in the squared quantization error. Choose **Maximum iteration** to specify the number of iterations at which to stop. Choose **Whichever comes first** and the block stops the iteration process as soon as the relative threshold or maximum iteration value is attained.

### Relative threshold

Type the value that is the maximum acceptable fractional drop in the squared quantization error.

### Maximum iteration

Enter the maximum number of iterations you want the block to perform. From the **Stopping criteria** list, select **Maximum iteration** in order to activate this parameter.

**Searching methods**

Choose `Linear search` to use a linear search method when comparing the training points to the boundary points. Choose `Binary search` to use a binary search method when comparing the training points to the boundary points.

**Tie-breaking rules**

When a training point lies on a boundary point, choose `Lower indexed codeword` to assign the training point to the lower indexed quantization region. Choose `Higher indexed codeword` to assign the training point to the higher indexed region.

**Design and Plot**

Click this button to display the performance curve and the staircase character of the quantizer in the figures on the right side of the GUI. These plots are based on the current parameter settings.

You must click **Design and Plot** to apply any changes you make to the parameter values in the SQDTool GUI.

**Export Outputs**

Click this button, or press **Ctrl+E**, to export the **Final Codebook**, **Final Boundary Points**, and **Error** values to the workspace, a text file, or a MAT-file.

**Destination**

Choose `Current model` to create a Scalar Quantizer block in the model you most recently selected. Type `gcs` in the MATLAB Command Window to display the name of your current model. Choose `New model` to create a block in a new model file.

**Block type**

Select `Encoder` to design a Scalar Quantizer Encoder block. Select `Decoder` to design a Scalar Quantizer Decoder block. Select `Both` to design a Scalar Quantizer Encoder block and a Scalar Quantizer Decoder block.

**Encoder block name**

Enter a name for the Scalar Quantizer Encoder block.

**Decoder block name**

Enter a name for the Scalar Quantizer Decoder block.

**Overwrite target block(s)**

When you do not select this check box and a Scalar Quantizer Encoder and/or Decoder block with the same block name exists in the destination model, a new Scalar Quantizer Encoder and/or Decoder block is created in the destination model. When you select this check box and a Scalar Quantizer Encoder and/or Decoder block with the same block name exists in the destination model, the parameters of these blocks are overwritten by new parameters.

**Generate Model**

Click this button and SQDTool uses the parameters that correspond to the current plots to set the parameters of the Scalar Quantizer Encoder and/or Decoder blocks.

**References**

Gersho, A. and R. Gray. *Vector Quantization and Signal Compression*. Boston: Kluwer Academic Publishers, 1992.

## Supported Data Types

- Double-precision floating point

## See Also

Quantizer

Scalar Quantizer Decoder

Scalar Quantizer Encoder

Uniform Encoder

Uniform Decoder

Simulink

DSP System Toolbox

DSP System Toolbox

DSP System Toolbox

DSP System Toolbox

## Version History

Introduced before R2006a

**R2021b: Scalar Quantizer Design block has been removed**

The Scalar Quantizer Design block and the associated SQDTool have been removed.

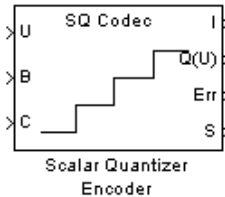
## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

## Scalar Quantizer Encoder

Encode each input value by associating it with index value of quantization region



## Library

Quantizers

dspquant2

## Description

The Scalar Quantizer Encoder block maps each input value to a quantization region by comparing the input value to the quantizer boundary points defined in the **Boundary points** parameter. The block outputs the zero-based index of the associated region.

You can select how you want to enter the **Boundary points** using the **Source of quantizer parameters**. When you select **Specify via dialog**, type the boundary points into the block parameters dialog box. When you select **Input port**, port B appears on the block. The block uses the input to port B as the **Boundary points** parameter.

Use the **Boundary points** parameter to specify the boundary points for your quantizer. These values are used to break up the set of input numbers into regions. Each region is specified by an index number.

Let  $N$  be the number of quantization regions. When the codebook is defined as  $[c_1 \ c_2 \ c_3 \ \dots \ c_N]$ , and the **Boundary points** parameter is defined as  $[p_0 \ p_1 \ p_2 \ p_3 \ \dots \ p_N]$ , then  $p_0 < c_1 < p_1 < c_2 \ \dots \ p_{(N-1)} < c_N < p_N$  for a regular quantizer. When your quantizer is bounded, from the **Partitioning** list, select **Bounded**. You need to specify  $N+1$  boundary points, or  $[p_0 \ p_1 \ p_2 \ p_3 \ \dots \ p_N]$ . When your quantizer is unbounded, from the **Partitioning** list, select **Unbounded**. You need to specify  $N-1$  boundary points, or  $[p_1 \ p_2 \ p_3 \ \dots \ p_{(N-1)}]$ ; the block sets  $p_0$  equal to  $-\text{inf}$  and  $p_N$  equal to  $\text{inf}$ .

The block uses the **Partitioning** parameter to interpret the boundary points you enter. For instance, to create a bounded quantizer, from the **Partitioning** list, select **Bounded** and enter the following boundary points:

```
[0 0.5 3.7 5.8 6.0 11]
```

The block assigns any input values between 0 and 0.5 to index 0, input values between 0.5 and 3.7 to index 1, and so on. The block assigns any values that are less than 0 to index 0, the lowest index value. The block assigns any values that are greater than 11 to index 4, the highest index value.

To create an unbounded quantizer, from the **Partitioning** list, select Unbounded and enter the following boundary points:

```
[0 0.5 3.7 5.8 6.0 11]
```

The block assigns any input values between 0 and 0.5 to index 1, input values between 0.5 and 3.7 to index 2, and so on. The block assigns any input values less than 0 to index 0 and any values greater than 11 to index 6.

The **Searching method** parameter determines how the appropriate quantizer index is found. When you select **Linear**, the Scalar Quantizer Encoder block compares the input value to the first region defined by the first two boundary points. When the input value does not fall within this region, the block then compares the input value to the next region. This process continues until the input value is determined to be within a region and is associated with the appropriate index value. The computational cost of this process is of the order  $P$ , where  $P$  is the number of boundary points.

When you select **Binary** for the **Searching method**, the block compares the input value to the middle value of the boundary points vector. When the input value is larger than this boundary point, the block discards the boundary points that are lower than this middle value. The block then compares the input value to the middle boundary point of the new range, defined by the remaining boundary points. This process continues until the input value is associated with the appropriate index value. The computational cost of this process is of the order  $\log_2 P$ , where  $P$  is the number of boundary points. In most cases, the Binary option is faster than the Linear option.

When an input value is the same as a boundary point, the **Tie-breaking rule** parameter determines the region to which the value is assigned. When you want the input value to be assigned to the lower indexed region, select **Choose the lower index**. To assign the input value with the higher indexed region, select **Choose the higher index**.

Select the **Output codeword** check box to output the codeword values that correspond to each index value at port Q(U).

Select the **Output the quantization error** check box to output the quantization error for each input value from the Err port on this block. The quantization error is the difference between the input value and the quantized output value.

When you select either the **Output codeword** check box or the **Output quantization error** check box, you must also enter your codebook values. If, from the **Source of quantizer parameters** list, you choose **Specify via dialog**, use the **Codebook** parameter to enter a vector of quantized output values that correspond to each region. If, from the **Source of quantizer parameters** list, you choose **Input port**, use input port C to specify your codebook values.

If, for the **Partitioning** parameter, you select **Bounded**, the **Output clipping status** check box and the **Action for out of range input** parameter appear. When you select the **Output clipping status** check box, port S appears on the block. Any time an input value is outside the range defined by the **Boundary points** parameter, the block outputs a 1 at the S port. When the value is inside the range, the blocks outputs a 0.

You can use the **Action for out of range input** parameter to determine the block's behavior when an input value is outside the range defined by the **Boundary points** parameter. Suppose the boundary points for a bounded quantizer are defined as  $[p_0 \ p_1 \ p_2 \ p_3 \ \dots \ p_N]$  and the possible index values are defined as  $[i_0 \ i_1 \ i_2 \ \dots \ i_{(N-1)}]$ , where  $i_0=0$  and  $i_0 < i_1 < i_2 < \dots < i_{(N-1)}$ . When you want any input value less than  $p_0$  to be assigned to index value  $i_0$  and any input values greater than  $p_N$  to be assigned to index value  $i_{(N-1)}$ , select **Clip**. When you want to be warned

when clipping occurs, select `Clip` and `warn`. When you want the simulation to stop and the block to display an error when the index values are out of range, select `Error`.

The Scalar Quantizer Encoder block accepts real floating-point and fixed-point inputs. For more information on the data types accepted by each port, see “Data Type Support” on page 2-1264 or “Supported Data Types” on page 2-1266.

### Data Type Support

The input data values, boundary points, and codebook values can be input to the block at ports U, B, and C, respectively. The data type of the inputs can be `double`, `single`, or `Fixed-point`.

The outputs of the block can be the index values, the quantized output values, the quantization error, and the clipping status. Use the **Index output data type** parameter to specify the data type of the index output from the block at port I. You can choose `int8`, `uint8`, `int16`, `uint16`, `int32`, or `uint32`. The data type of the quantized output and the quantization error can be `double`, `single`, or `Fixed-point`. The clipping status values output at port S are Boolean values.

---

**Note** The input data, boundary points, codebook values, quantized output values, and the quantization error must have the same data type whenever they are present.

---

## Dialog Box

### Main Tab

#### Source of quantizer parameters

Choose `Specify via dialog` to enter the boundary points and codebook values using the block parameters dialog box. Select `Input port` to specify the parameters using the block's input ports. Input the boundary points and codebook values using ports B and C, respectively.

#### Partitioning

When your quantizer is bounded, select `Bounded`. When your quantizer is unbounded, select `Unbounded`.

#### Boundary points

Enter a vector of values that represent the boundary points of the quantizer regions. This parameter is visible when you select `Specify via dialog` from the **Source of quantizer parameters** list. Tunable (Simulink).

#### Searching method

When you select `Linear`, the block finds the region in which the input value is located using a linear search. When you select `Binary`, the block finds the region in which the input value is located using a binary search.

#### Tie-breaking rule

Set this parameter to determine the behavior of the block when the input value is the same as the boundary point. When you select `Choose the lower index`, the input value is assigned to lower indexed region. When you select `Choose the higher index`, the value is assigned to the higher indexed region.

#### Output codeword

Select this check box to output the codeword values that correspond to each index value at port Q(U).



**Output quantization error**

Select this check box to output the quantization error for each input value at port Err.

**Codebook**

Enter a vector of quantized output values that correspond to each index value. If, for the **Partitioning** parameter, you select **Bounded** and your boundary points vector has length  $N$ , then you must specify a codebook of length  $N-1$ . If, for the **Partitioning** parameter, you select **Unbounded** and your boundary points vector has length  $N$ , then you must specify a codebook of length  $N+1$ .

This parameter is visible when you select **Specify** via dialog from the **Source of quantizer parameters** list and you select either the **Output codeword** or **Output quantization error** check box. Tunable (Simulink).

**Output clipping status**

When you select this check box, port S appears on the block. Any time an input value is outside the range defined by the **Boundary points** parameter, the block outputs a 1 at this port. When the value is inside the range, the block outputs a 0. This parameter is visible when you select **Bounded** from the **Partitioning** list.

**Action for out of range input**

Use this parameter to determine the behavior of the block when an input value is outside the range defined by the **Boundary points** parameter. Suppose the boundary points are defined as  $[p_0 p_1 p_2 p_3 \dots p_N]$  and the index values are defined as  $[i_0 i_1 i_2 \dots i_{(N-1)}]$ . When you want any input value less than  $p_0$  to be assigned to index value  $i_0$  and any input values greater than  $p_N$  to be assigned to index value  $i_{(N-1)}$ , select **Clip**. When you want to be warned when clipping occurs, select **Clip and warn**. When you want the simulation to stop and the block to display an error when the index values are out of range, select **Error**. This parameter is visible when you select **Bounded** from the **Partitioning** list.

**Index output data type**

Specify the data type of the index output from the block at port I. You can choose **int8**, **uint8**, **int16**, **uint16**, **int32**, **uint32**, or **Inherit** via back propagation.

**Data Types Tab****Rounding mode**

Specify the rounding mode for fixed-point operations as one of the following:

- Floor
- Ceiling
- Convergent
- Nearest
- Round
- Simplest
- Zero

For more details, see rounding mode.

**Saturate on integer overflow**

When you select this parameter, the block saturates the result of its fixed-point operation. When you clear this parameter, the block wraps the result of its fixed-point operation. For details on **saturate** and **wrap**, see overflow mode for fixed-point operations.

## References

Gersho, A. and R. Gray. *Vector Quantization and Signal Compression*. Boston: Kluwer Academic Publishers, 1992.

## Supported Data Types

| Port | Supported Data Types   |
|------|--|
| U    | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed only)</li> <li>• 8-, 16-, and 32-bit signed integers</li> </ul> |
| B    | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed only)</li> <li>• 8-, 16-, and 32-bit signed integers</li> </ul> |
| C    | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed only)</li> <li>• 8-, 16-, and 32-bit signed integers</li> </ul> |
| I    | <ul style="list-style-type: none"> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul>   |
| Q(U) | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed only)</li> <li>• 8-, 16-, and 32-bit signed integers</li> </ul> |
| Err  | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed only)</li> <li>• 8-, 16-, and 32-bit signed integers</li> </ul> |
| S    | <ul style="list-style-type: none"> <li>• Boolean</li> </ul>  |

For more information on what data types are supported for each quantizer mode, see “Data Type Support” on page 2-1264.

## See Also

Quantizer

Scalar Quantizer Decoder

Uniform Encoder

Uniform Decoder

Simulink

DSP System Toolbox

DSP System Toolbox

DSP System Toolbox

## **Version History**

**Introduced before R2006a**

### **Extended Capabilities**

#### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

#### **Fixed-Point Conversion**

Design and simulate fixed-point systems using Fixed-Point Designer™.

If the input is fixed point, it must be a signed integer or a signed fixed point value with a power-of-two slope and zero bias.

## Short-Time FFT

Nonparametric estimate of spectrum using short-time fast Fourier transform (STFT) method



### Libraries:

DSP System Toolbox / Transforms

## Description

The Short-Time FFT block computes a nonparametric estimate of the spectrum. The block buffers, applies a window, and zero pads the input signal. The block then takes the FFT of the signal, transforming it into the frequency domain.

## Ports

### Input

**x[n]** — Input signal  
vector | matrix

Specify the time-domain input signal as a vector or a matrix. If the input is a matrix, the block treats each column as an independent channel. The input can be a variable-size signal. That is, the frame size of the signal can change while the simulation is running. The number of channels must remain the same.

If the input is fixed point, it must be a signed integer or a signed fixed point value with a power-of-two slope and zero bias.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `fixed point`

**w[n]** — Analysis window vector  
vector

Connect your single-channel analysis window to the **w[n]** port. For the **Analysis window length** parameter, enter the length of the analysis window  $W$ . The block buffers the input signal such that it has a frame length of  $W$ .

If the window vector is fixed point, it must be a signed integer or signed fixed point with a power-of-two slope and zero bias.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `fixed point`

### Output

**X[nL,k]** — Short-time FFT output  
vector | matrix

The block outputs the complex-valued single-channel or multichannel short-time FFT at this port.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `fixed point`

## Parameters

**Analysis window length** — Length of analysis window

512 (default) | positive integer greater than 1

Specify the frame length of the analysis window as a positive integer greater than 1. The block buffers the input into overlapping window segments using the specified window length and overlap length, and then multiplies each overlapped segment by the window.

**Overlap between consecutive windows (in samples)** — Overlap length

384 (default) | nonnegative integer

Specify the number of samples by which consecutive windows overlap in each frame of the input signal as a nonnegative integer. The window overlap reduces artifacts at the data boundaries.

**FFT length** — FFT length

1024 (default) | positive integer greater than or equal to 2

Enter the length up to which the block pads the input signal as a positive integer greater than or equal to 2. After the block buffers and windows the signal, it zero-pads the signal before computing the FFT. This parameter determines the length of the short-time fast Fourier transform output (number of rows). The FFT length must be greater than or equal to the window length.

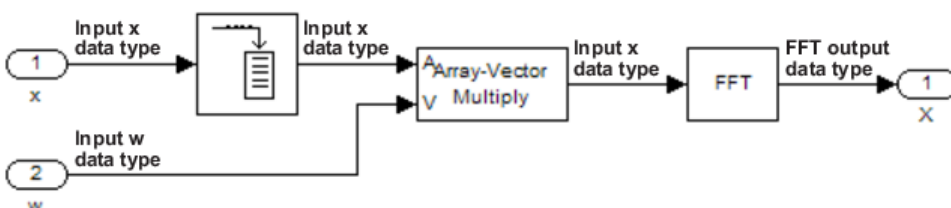
## Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | double   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no                                      |
| <b>Multidimensional Signals</b> | no                                      |
| <b>Variable-Size Signals</b>    | no                                      |
| <b>Zero-Crossing Detection</b>  | no                                      |

## More About

### Fixed-Point Data Types

The following diagram shows the data types used within the Short-Time FFT subsystem block for fixed-point signals.



The values of the fixed-point parameters of the Array-Vector Multiply block in this diagram are:

- **Rounding Mode** — Floor
- **Saturate on integer overflow** — Wrap
- **Product output** — Inherit via internal rule
- **Accumulator** — Inherit via internal rule
- **Output** — Same as first input

The values of the fixed-point parameters of the FFT block in this diagram are:

- **Rounding Mode** — Floor
- **Saturate on integer overflow** — Wrap
- **Sine table** — Same word length as input
- **Product output** — Inherit via internal rule
- **Accumulator** — Inherit via internal rule
- **Output** — Inherit via internal rule

See the FFT and Array-Vector Multiply block reference pages for more information.

## Version History

Introduced before R2006a

## References

- [1] Quatieri, Thomas E. *Discrete-Time Speech Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 2001.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Usage notes and limitations:

- Generated code relies on the `memcpy` or `memset` function (`string.h`) under certain conditions.
- When the FFT length is not a power of two, the executable generated from this block relies on prebuilt dynamic library files (`.dll` files) included with MATLAB. Use the `packNGo` function to package the code generated from this block and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see “How To Run a Generated Executable Outside MATLAB”.

When the FFT length is a power of two, you can generate standalone C and C++ code from this block.

### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

If the input is fixed point, it must be a signed integer or a signed fixed point value with a power-of-two slope and zero bias.

## See Also

### Objects

`dsp.SpectrumEstimator` | `dsp.STFT`

### Blocks

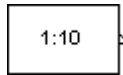
`Spectrum Estimator` | `Inverse Short-Time FFT` | `Magnitude FFT` | `Periodogram` | `Spectrum Analyzer` | `Window Function` | `Yule-Walker Method`

### Topics

“Spectral Analysis”

## Signal From Workspace

Import signal from MATLAB workspace



**Libraries:**  
DSP System Toolbox / Sources

### Description

The Signal From Workspace block imports a signal from the MATLAB workspace into the Simulink model. The **Signal** parameter specifies the name of a MATLAB workspace variable containing the signal to import, or any valid MATLAB expression defining a matrix or 3-D array.

Unlike the Simulink From Workspace block, the Signal From Workspace block holds the output value constant between successive output frames (that is, no linear interpolation takes place). Also, the initial signal values are always produced immediately at  $t=0$ .

### Ports

#### Output

**Port\_1** — Signal imported from workspace  
scalar | vector | matrix | 3-D array

Signal imported from workspace, as a scalar, vector, matrix, or 3-D array.

When the **Signal** parameter specifies an  $M$ -by- $N$  matrix ( $M \neq 1$ ), each of the  $N$  columns is treated as a distinct channel. You specify the frame size in the **Samples per frame** parameter,  $M_o$ . The output is an  $M_o$ -by- $N$  matrix containing  $M_o$  consecutive samples from each signal channel. You specify the output sample period in the **Sample time** parameter,  $T_s$ , and the output frame period is  $M_o T_s$ . For convenience, an imported row vector ( $M=1$ ) is treated as a single channel, so the output dimension is  $M_o$ -by-1.

When the **Signal** parameter specifies an  $M$ -by- $N$ -by- $P$  array, each of the  $P$  pages (an  $M$ -by- $N$  matrix) is output in sequence with period  $T_s$ . The **Samples per frame** parameter must be set to 1.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fixed point

### Parameters

**Signal** — Signal to import

1:10 (default) | MATLAB workspace variable | MATLAB expression

The name of the MATLAB workspace variable from which to import the signal, or a valid MATLAB expression specifying the signal.

**Sample time** — Output sample period



1 (default) | scalar | vector

The sample period,  $T_s$ , of the output, specified as a scalar or vector. The output frame period is  $M_o T_s$ .

**Samples per frame** — Samples per frame

1 (default) | positive integer

The number of samples,  $M_o$ , to buffer into each output frame, specified as a positive integer scalar. This value must be 1 when you specify a 3-D array in the **Signal** parameter.

**Form output after final data value by** — Values to output after final imported signal value

Setting to zero (default) | Holding final value | Cyclic repetition

Specifies the output after all of the specified signal samples have been generated.

- When you specify **Setting To Zero**, the block generates zero-valued outputs for the duration of the simulation after generating the last frame of the signal.
- When you specify **Holding Final Value**, the block repeats the final sample for the duration of the simulation after generating the last frame of the signal.
- When you specify **Cyclic Repetition**, the block repeats the signal from the beginning after it reaches the last sample in the signal. If the frame size you specify in the **Samples per frame** parameter does not evenly divide the input length, a buffer block is inserted into the Signal From Workspace subsystem, and the model becomes multirate. If you do not want your model to become multirate, make sure that the frame size evenly divides the input signal length.

The block does not extrapolate the imported signal beyond the last sample.

**Warn when frame size does not evenly divide input length** — Warn when input length is not an integer multiple of frame size

off (default) | on

Select the **Warn when frame size does not evenly divide input length** parameter to be alerted when the input length is not an integer multiple of the frame size. When the input length is not an integer multiple of the frame size, the model becomes multirate. Use the Model Explorer to turn these warnings on or off model-wide:

- In the **Modeling** tab, click **Model Explorer**.
- In the **Search** bar of the Model Explorer, search by **Property Name** for the `ignoreOrWarnInputAndFrameLengths` property. Each block with the **Warn when frame size does not evenly divide input length** check box appears in the list in the **Contents** pane.
- Select each of the blocks for which you want to toggle the warning parameter, and select or clear the check box in the `ignoreOrWarnInputAndFrameLengths` column.

### Dependencies

To enable this parameter set **Form output after final data value by** to **Cyclic Repetition**.

## Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | double   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no                                      |
| <b>Multidimensional Signals</b> | no                                      |
| <b>Variable-Size Signals</b>    | no                                      |
| <b>Zero-Crossing Detection</b>  | no                                      |

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

## See Also

### Blocks

[Signal To Workspace](#) | [From Workspace](#) | [To Workspace](#) | [Triggered Signal From Workspace](#)

### Objects

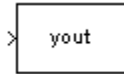
[dsp.SignalSource](#) | [dsp.SignalSink](#)

### Topics

[“Filter Frames of a Noisy Sine Wave Signal in Simulink”](#)  
[“Create Signals for Sample-Based Processing”](#)  
[“Create Signals for Frame-Based Processing”](#)  
[“Import and Export Signals for Sample-Based Processing”](#)  
[“Import and Export Signals for Frame-Based Processing”](#)

# Signal To Workspace

Write data to MATLAB workspace



## Compatibility

**Note** The Signal To Workspace block has been replaced by the To Workspace block in Simulink. Replace existing instances of the Signal To Workspace block with To Workspace block. For new models, use the To Workspace block.

## Library

Sinks

dspsnks4

## Description

The Signal To Workspace block writes data from your simulation into an array or structure in the main MATLAB workspace. You can specify a name for the workspace variable as well as whether the data is saved as an array, structure, or structure with time.

When the **Save format** is set to Array or Structure, the dimensions of the output depend on the input dimensions and the setting of the **Save 2-D signals as** parameter. The following table summarizes the output dimensions under various conditions. In the table,  $K$  represents the value of the **Limit data points to last** parameter.

| Input Signal Dimensions | Save 2-D Signals as ...                       | Signal To Workspace Output Dimension   |
|-------------------------|---|--|
| $M$ -by- $N$ matrix     | 2-D array (concatenate along first dimension) | <p><math>K</math>-by-<math>N</math> matrix.</p> <p>If you set the <b>Limit data points to last</b> parameter to <math>\text{inf}</math>, <math>K</math> represents the total number of samples acquired in each column by the end of simulation. This is equivalent to multiplying the input frame size (<math>M</math>) by the total number of <math>M</math>-by-<math>N</math> inputs acquired by the block.</p> |

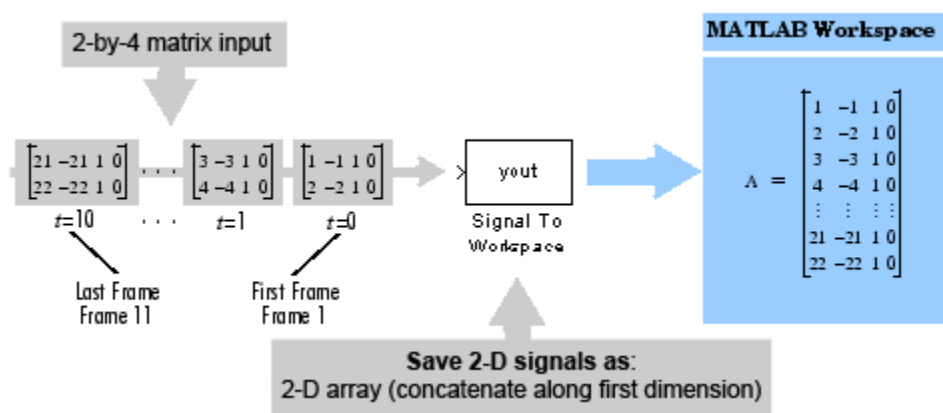
| Input Signal Dimensions              | Save 2-D Signals as ...                       | Signal To Workspace Output Dimension   |
|--------------------------------------|---|--|
| $M$ -by- $N$ matrix                  | 3-D array (concatenate along third dimension) | $M$ -by- $N$ -by- $K$ array.<br><br>If you set the <b>Limit data points to last</b> parameter to $\text{inf}$ , $K$ represents the total number of $M$ -by- $N$ inputs acquired by the end of the simulation.  |
| Length- $N$ unoriented vector        | Any setting                                   | $K$ -by- $N$ matrix  |
| $N$ -dimensional array where $N > 2$ | Any setting                                   | Array with $N+1$ dimensions, where the size of the last dimension is equal to $K$ . If you set the <b>Limit data points to last</b> parameter to $\text{inf}$ , $K$ represents the total number of $M$ -by- $N$ inputs acquired by the end of simulation |

## Examples

### Example 1: Save 2-D Signals as a 2-D Array

In the `ex_signaltoworkspace_ref2` model, the Signal To Workspace block receives a 2-by-4 matrix input and logs 11 frames (two samples per frame) by the end of the simulation. Because the **Save 2-D signals as** parameter is set to 2-D array (concatenate along first dimension), the block concatenates the input along the first dimension to create a 22-by-4 matrix,  $A$ , in the MATLAB workspace.

The following figure illustrates the behavior of the Signal to Workspace block in this example.

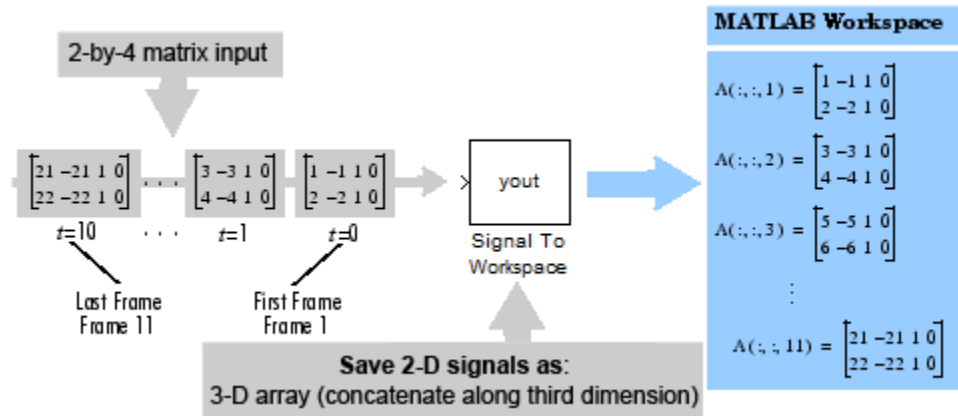


In the 2-D output mode, there is no indication of where one frame ends and another begins. To log input frames separately, set the **Save 2-D signals as** parameter to 3-D array (concatenate along third dimension), as shown in Example 2.

## Example 2: Save 2-D Signals as a 3-D Array

In the `ex_signaltoworkspace_ref1` model, the input to the Signal To Workspace block is a 2-by-4 matrix. The **Save 2-D signals as** parameter is set to 3-D array (concatenate along third dimension), so by the end of the simulation the Signal To Workspace block logs 11 frames of data as a 2-by-4-by-11 array, *A*, in the MATLAB workspace.

The following figure illustrates the behavior of the Signal to Workspace block in this example.



## Parameters

### Variable name

Specify the name of the array or structure into which the block logs the simulation data. The block creates this variable in the MATLAB workspace only after the simulation stops running. When you enter the name of an existing workspace variable, the block overwrites that variable with the simulation data.

### Limit data points to last

Specify the maximum number of samples or frames the block will save. When the simulation generates more than the specified maximum number of samples or frames, the simulation saves only the most recently generated data. To capture all data, set this parameter to `inf`. See the table in the Description on page 2-1275 section for more information on how this parameter affects the dimensions of the logged data.

### Decimation

Specify a positive integer *d* to determine how often the block writes data to the workspace array or structure. The block writes data to the array or structure every *d*th sample. With the default decimation value of 1, the block writes data at every time step.

### Save format

Specify the format in which to save simulation output to the workspace. You can select one of the following options:

- **Array** — Select this option to save the data as an N-dimensional array. If the input signal is an unoriented vector, the resulting workspace array is 2-D. Each input vector is saved in a row of the output matrix, vertically concatenated onto the previous vector. If the input signal is 2-dimensional, the dimensions of the resulting workspace array depend on the setting of the **Save 2-D signals as** parameter.

- **Structure** — Select this option to save the data as a structure consisting of three fields: `time`, `signals` and `blockName`. In this mode, the `time` field is empty, and the `blockName` field contains the name of the Signal To Workspace block. The `signals` field contains a structure with three additional fields: `values`, `dimensions`, and `label`. The `values` field contains the array of signal values, the `dimensions` field specifies the dimensions of the values array, and the `label` field contains the label of the input line.
- **Structure with time** — This option is the same as **Structure**, except that the `time` field contains a vector of simulation time steps. This is the only output format that can be read directly by a From Workspace block. When you select this option, the **Save 2-D signals as** parameter is not available. In this mode, the block always saves 2-D input arrays as a 3-D array.

The default setting of this parameter is **Array**.

### Save 2-D signals as

Specify whether the block outputs 2-D signals as a 2-D or 3-D array in the MATLAB workspace:

- **2-D array (concatenate along first dimension)** — When you select this option, the block saves an  $M$ -by- $N$  input signal as a  $(K*M)$ -by- $N$  matrix, where  $K*M$  is the total number of samples acquired by the end of the simulation. The block vertically concatenates each  $M$ -by- $N$  matrix input with the previous input to produce the 2-D output array. See “Example 1: Save 2-D Signals as a 2-D Array” on page 2-1276 for more information about this mode.
- **3-D array (concatenate along third dimension)** — When you select this option, the block saves an  $M$ -by- $N$  input signal as an  $M$ -by- $N$ -by- $K$  array, where  $K$  is the number of  $M$ -by- $N$  inputs logged by the end of the simulation.  $K$  has an upper bound equal to the value of the **Limit data points to last** parameter. See “Example 2: Save 2-D Signals as a 3-D Array” on page 2-1277 for more information about this mode.

This parameter is visible only when you set the **Save format** parameter to **Array** or **Structure**. When you set the **Save format** parameter to **Structure with time**, the block outputs the 2-D input signal as a 3-D array.

---

**Note** The **Inherit from input** (this choice will be removed - see release notes) option will be removed in a future release. See “Signal To Workspace Block Changes” in the *DSP System Toolbox Release Notes* for more information.

---

### Frames

The output dimensionality for frame-based inputs. **Frames** can be set to **Concatenate frames** (2-D array) or **Log frames separately** (3-D array). This parameter is ignored when inputs are not frame based.

### Log fixed-point data as a fi object

Select this parameter to log fixed-point data to the MATLAB workspace as a Fixed-Point Designer `fi` object. Otherwise, fixed-point data is logged to the workspace as `double`.

### Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point (signed and unsigned)

- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

## See Also

Triggered To Workspace  
To Workspace

DSP System Toolbox  
Simulink

## Version History

Introduced before R2006a

## Sine Wave

Generate continuous or discrete sine wave



### Libraries:

DSP System Toolbox / Sources

DSP System Toolbox HDL Support / Sources

### Description

The Sine Wave block generates a multichannel real or complex sinusoidal signal, with independent amplitude, frequency, and phase in each output channel. The block supports floating point and signed fixed-point data types.

The block generates a real sinusoidal signal when you set the **Output complexity** parameter to **Real**. The real sinusoidal output is defined by an expression of the type

$$y = A\sin(2\pi ft + \phi)$$

where you specify  $A$  in the **Amplitude** parameter,  $f$  in hertz in the **Frequency** parameter, and  $\phi$  in radians in the **Phase offset** parameter.

The block generates a complex exponential signal when you set the **Output complexity** parameter to **Complex**. This complex exponential signal is defined by an expression of the type

$$y = Ae^{j(2\pi ft + \phi)} = A\{\cos(2\pi ft + \phi) + j\sin(2\pi ft + \phi)\}$$

### Generating Multichannel Outputs

For both real and complex sinusoids, the **Amplitude**, **Frequency**, and **Phase offset** parameter values ( $A$ ,  $f$ , and  $\phi$ ) can be scalars or length- $N$  vectors, where  $N$  is the desired number of channels in the output. When you specify at least one of these parameters as a length- $N$  vector, scalar values specified for the other parameters are applied to every channel.

For example, to generate the three-channel output containing the following real sinusoids, set the block parameters as shown:

$$y = \begin{cases} \sin(2000\pi t) & \text{(channel 1)} \\ 2\sin(1000\pi t) & \text{(channel 2)} \\ 3\sin\left(500\pi t + \frac{\pi}{2}\right) & \text{(channel 3)} \end{cases}$$

- **Output complexity** = Real
- **Amplitude** = [1 2 3]
- **Frequency** = [1000 500 250]
- **Phase offset** = [0 0 pi/2]



## Ports

### Output

**Port 1** — Sinusoidal signal

scalar | vector | matrix

Output a sinusoidal signal as a scalar or vector. For more information about output complexity, see “Description” on page 2-1280. For information about multichannel support, see “Generating Multichannel Outputs” on page 2-1280.

---

**Tip** To output fixed-point data types, you must set **Sample mode** to Discrete and **Computation method** to Table lookup.

---

Data Types: single | double | fixed point  
Complex Number Support: Yes

## Parameters

### Main

**Amplitude** — Amplitude of sine waves

1 (default) | scalar | vector

A length- $N$  vector containing the amplitudes of the sine waves in each of  $N$  output channels, or a scalar to be applied to all  $N$  channels. The vector length must be the same as that specified for the **Frequency** and **Phase offset** parameters.

---

**Tip** This parameter is tunable (Simulink) only when the **Computation method** is Trigonometric fcn or Differential.

---

**Tunable:** Yes

**Frequency (Hz)** — Frequency of each sine wave

100 (default) | scalar | vector

A length- $N$  vector containing frequencies, in hertz, of the sine waves in each of  $N$  output channels, or a scalar to be applied to all  $N$  channels. The vector length must be the same as that specified for the **Amplitude** and **Phase offset** parameters. You can specify positive, zero, or negative frequencies.

---

**Tip** This parameter is tunable (Simulink) when you set either:

- **Sample mode** to Continuous.
  - **Sample mode** to Discrete and **Computation method** to Trigonometric fcn.
- 

**Tunable:** Yes

**Phase offset (rad)** — Phase offset

0 (default) | scalar | vector

A length- $N$  vector containing the phase offsets, in radians, of the sine waves in each of  $N$  output channels, or a scalar to be applied to all  $N$  channels. The vector length must be the same as that specified for the **Amplitude** and **Frequency** parameters.

---

**Tip** This parameter is tunable (Simulink) when you set either:

- **Sample mode** to Continuous.
  - **Sample mode** to Discrete and **Computation method** to Trigonometric fcn.
- 

**Tunable:** Yes

**Sample mode** — Continuous or discrete sampling mode

Discrete (default) | Continuous

Specify the sampling mode as Continuous or Discrete:

- Continuous

In continuous mode, the sinusoid in the  $i$ th channel,  $y_i$ , is computed as a continuous function,

$$y_i = A_i \sin(2\pi f_i t + \phi_i) \quad (\text{real})$$

or

$$y_i = A_i e^{j(2\pi f_i t + \phi_i)} \quad (\text{complex})$$

and the block's output is continuous. In this mode, the block operates the same as the Simulink Sine Wave block with **Sample time** set to 0. This mode offers high accuracy, but requires trigonometric function evaluations at each simulation step, which is computationally expensive. Also, because this method tracks absolute simulation time, a discontinuity will eventually occur when the time value reaches its maximum limit.

Note also that many DSP System Toolbox blocks do not accept continuous-time inputs.

- Discrete

In discrete mode, the block can generate discrete-time output by directly evaluating the trigonometric function, by table lookup, or by a differential method. For more information on these computation methods, see “Algorithms” on page 2-1286.

**Output complexity** — Real or complex waveform

Real (default) | Complex

The type of waveform to generate: Real specifies a real sine wave, Complex specifies a complex exponential.

**Computation method** — Method for computing discrete-time sinusoids

Trigonometric fcn (default) | Table lookup | Differential

The method by which discrete-time sinusoids are generated: `Trigonometric fcn`, `Table lookup`, or `Differential`. For more information on each of the available options, see “Algorithms” on page 2-1286.

**Dependencies**

This parameter is only visible when you set the **Sample mode** to `Discrete`.

---

**Note** To generate fixed-point sinusoids, you must set the **Computation method** to `Table lookup`.

---

**Optimize table for** — Optimize for speed or memory

Speed (default) | Memory

Optimizes the table of sine values for `Speed` or `Memory`. When optimized for speed, the table contains  $k$  elements, and when optimized for memory, the table contains  $k/4$  elements, where  $k$  is the number of input samples in one full period of the sine wave.

**Dependencies**

This parameter is only visible when you set the **Computation method** parameter to `Table lookup`.

**Sample time** — Sample period

1/1000 (default) | scalar

The period with which the sine wave is sampled,  $T_s$ , as a finite scalar, greater than zero. The output frame period of the block is  $MT_s$ , where you specify  $M$  in the **Samples per frame** parameter.

**Dependencies**

To enable this parameter, set **Sample mode** to `Discrete`.

**Samples per frame** — Samples per frame

1 (default) | positive integer

The number of consecutive samples from each sinusoid to buffer into the output frame,  $M$ , specified as a positive scalar integer. This parameter is not tunable.

The block output is an  $M$ -by- $N$  matrix with frame period  $MT_s$ , where you specify  $T_s$  in the **Sample time** parameter.

**Dependencies**

To enable this parameter, set **Sample mode** to `Discrete`.

**Resetting states when re-enabled** — State behavior inside enabled subsystems

Restart at time zero (default) | Catch up to simulation time

This parameter determines the behavior of the Sine Wave block when an enabled subsystem is reenabled. The block can either reset itself to its starting state (`Restart at time zero`), or resume generating the sinusoid based on the current simulation time (`Catch up to simulation time`).

### Dependencies

This parameter only applies when the Sine Wave block is located inside an enabled subsystem and the **States when enabling** parameter of the Enable block is set to `reset`.

### Data Types

**Output data type** — Output data type

`double (default) | single | fixdt(1,16) | fixdt(1,16,0) | <data type expression> | Inherit:Inherit via back propagation`

Select how you would like to specify the data type properties of the **Output data type**. You can choose:

- `Inherit` — Lets you specify a rule for inheriting a data type, for example, `Inherit: Inherit via back propagation`
- `Built in`— Lets you specify a built in data type, for example, `double`
- `Fixed point` — Lets you specify the fixed-point attributes of the data type.
- `Expression` — Lets you specify an expression that evaluates to a valid data type, for example, `fixdt(1,16)`

### Data Type Assistant

**Mode** — Data type mode

`Built in (default) | Inherit | Fixed point | Expression`

Select how you would like to specify the data type properties of the **Output data type**. You can choose:

- `Inherit` — Lets you specify a rule for inheriting a data type, for example, `Inherit: Inherit via back propagation`
- `Built in`— Lets you specify a built-in data type, for example, `double`
- `Fixed point` — Lets you specify the fixed-point attributes of the data type.
- `Expression` — Lets you specify an expression that evaluates to a valid data type, for example, `fixdt(1,16)`

For more information, see “Specify Data Types Using Data Type Assistant” (Simulink).

**Signedness** — Signedness of fixed-point data

`Signed (default)`

Specify the signedness of the fixed-point output. For more information, see “Specify Data Types Using Data Type Assistant” (Simulink).

**Limitations**

The Sine Wave only supports Signed data types.

**Scaling** — Method for scaling fixed-point data

Best precision (default) | Binary point

Specify the method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors. For more information, see “Specify Data Types Using Data Type Assistant” (Simulink).

**Dependencies**

To enable this parameter, set **Mode** to Fixed point

**Word length** — Bit size of the word that holds the quantized integer

16 (default) | integer from 2 to 128

Specify the bit size of the word that holds the quantized integer as a positive integer from 2 to 128. For more information, see “Specify Data Types Using Data Type Assistant” (Simulink).

**Dependencies**

To enable this parameter, set **Mode** to Fixed point

**Fraction length** — Fraction length for fixed-point data type

0 (default) | integer scalar

Specify the fraction length of the fixed-point data type as a positive or negative integer scalar.

**Dependencies**

To enable this parameter, set **Mode** to Fixed point and **Scaling** to Binary point.

**Data type override** — Data type override mode

Inherit (default) | Off

Select the data type override mode for this signal. You can select:

- **Inherit** — Inherits the data type override setting specified for the model.
- **Off** — Ignores the data type override setting specified for the model and uses the fixed-point data type you specify.

**Dependencies**

To enable this parameter, set **Mode** to Built in or Fixed point. For more information, see “Specify Data Types Using Data Type Assistant” (Simulink).

**Block Characteristics**

|                   |   |
|-------------------|---|
| <b>Data Types</b> | double   fixed point   integer   single |
|-------------------|---|

|                                 |    |
|---------------------------------|----|
| <b>Direct Feedthrough</b>       | no |
| <b>Multidimensional Signals</b> | no |
| <b>Variable-Size Signals</b>    | no |
| <b>Zero-Crossing Detection</b>  | no |

## Algorithms

When you select **Discrete** from the **Sample mode** parameter, the secondary **Computation method** parameter provides three options for generating the discrete sinusoid: **Trigonometric fcn**, **Table lookup**, and **Differential**.

### Trigonometric Fcn

The trigonometric function method computes the sinusoid in the  $i$ th channel,  $y_i$ , by sampling the continuous function

$$y_i = A_i \sin(2\pi f_i t + \phi_i) \quad (\text{real})$$

or

$$y_i = A_i e^{j(2\pi f_i t + \phi_i)} \quad (\text{complex})$$

with a period of  $T_s$ , where you specify  $T_s$  in the **Sample time** parameter. This mode of operation has the same benefits and liabilities as the **Continuous** sample mode.

At each sample time, the block evaluates the sine function at the appropriate time value *within the first cycle* of the sinusoid. By constraining trigonometric evaluations to the first cycle of each sinusoid, the block avoids the imprecision of computing the sine of very large numbers, and eliminates the possibility of discontinuity during extended operations (when an absolute time variable might overflow). This method therefore avoids the memory demands of the table lookup method at the expense of many more floating-point operations.

### Table Lookup

The table lookup method precomputes the *unique* samples of every output sinusoid at the start of the simulation, and recalls the samples from memory as needed. Because a table of finite length can only be constructed when all output sequences repeat, the method requires that the period of every sinusoid in the output be evenly divisible by the sample period. That is,  $1/(f_i T_s) = k_i$  must be an integer value for every channel  $i = 1, 2, \dots, N$ .

When the **Optimize table for** parameter is set to **Speed**, the table constructed for each channel contains  $k_i$  elements. When the **Optimize table for** parameter is set to **Memory**, the table constructed for each channel contains  $k_i/4$  elements.

For long output sequences, the table lookup method requires far fewer floating-point operations than any of the other methods, but can demand considerably more memory, especially for high sample rates (long tables). This method is recommended for models that are intended to emulate or generate code for DSP hardware, and that therefore must be optimized for execution speed.

---

**Note** The lookup table for this block is constructed from double-precision floating-point values. Thus, when you use the `Table` lookup computation mode, the maximum amount of precision you can achieve in your output is 53 bits. Setting the word length of the **Output** or **User-defined** data type to values greater than 53 bits does not improve the precision of your output.

---

**Tip** To generate fixed-point sinusoids, you must select `Table Lookup`.

---

## Differential

The differential method uses an incremental algorithm. This algorithm computes the output samples based on the output values computed at the previous sample time (and precomputed update terms) by using the following identities.

$$\sin(t + T_s) = \sin(t)\cos(T_s) + \cos(t)\sin(T_s)$$

$$\cos(t + T_s) = \cos(t)\cos(T_s) - \sin(t)\sin(T_s)$$

The update equations for the sinusoid in the  $i$ th channel,  $y_i$ , can therefore be written in matrix form as

$$\begin{bmatrix} \sin\{2\pi f_i(t + T_s) + \phi_i\} \\ \cos\{2\pi f_i(t + T_s) + \phi_i\} \end{bmatrix} = \begin{bmatrix} \cos(2\pi f_i T_s) & \sin(2\pi f_i T_s) \\ -\sin(2\pi f_i T_s) & \cos(2\pi f_i T_s) \end{bmatrix} \begin{bmatrix} \sin(2\pi f_i t + \phi_i) \\ \cos(2\pi f_i t + \phi_i) \end{bmatrix}$$

where you specify  $T_s$  in the **Sample time** parameter. Since  $T_s$  is constant, the right-hand matrix is a constant and can be computed once at the start of the simulation. The value of  $A_i \sin[2\pi f_i(t + T_s) + \phi_i]$  is then computed from the values of  $\sin(2\pi f_i t + \phi_i)$  and  $\cos(2\pi f_i t + \phi_i)$  by a simple matrix multiplication at each time step.

This mode offers reduced computational load, but is subject to drift over time due to cumulative quantization error. Because the method is not contingent on an absolute time value, there is no danger of discontinuity during extended operations (when an absolute time variable might overflow).

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

The Sine Wave block references absolute simulation time when configured in continuous sample mode.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

---

**Note** For a sine wave generator that is optimized for HDL code generation and has hardware-friendly control signals, see NCO (DSP HDL Toolbox), and “Generate Sine Wave” (DSP HDL Toolbox). The DSP HDL Toolbox blocks provide HDL-optimized algorithms with hardware-friendly control signals and simulate the latency of the HDL algorithm in Simulink.

---

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

**HDL Architecture**

This block has one default HDL architecture.

**Restrictions**

For HDL code generation, you must select the following Sine Wave block settings:

- **Computation method:** Table lookup
- **Sample mode:** Discrete

Output:

- The output port cannot have data types `single` or `double`.

**Complex Data Support**

This block supports code generation for complex signals.

**Fixed-Point Conversion**

Design and simulate fixed-point systems using Fixed-Point Designer™.

If the input is fixed point, it must be a signed integer or a signed fixed point value with a power-of-two slope and zero bias.

**See Also****Blocks**

Chirp | Signal From Workspace | Signal Generator | Sine Wave | Enabled Subsystem

**Functions**

`sin`

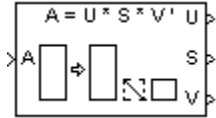
**Objects**

`dsp.SineWave`



# Singular Value Decomposition

Factor matrix using singular value decomposition



## Library

Math Functions / Matrices and Linear Algebra / Matrix Factorizations

dspfactors

## Description

The Singular Value Decomposition block factors the  $M$ -by- $N$  input matrix  $A$  such that

$$A = U \cdot \text{diag}(S) \cdot V^*$$

where

- $U$  is an  $M$ -by- $P$  matrix
- $V$  is an  $N$ -by- $P$  matrix
- $S$  is a length- $P$  vector
- $P$  is defined as  $\min(M,N)$

When

- $M = N$ ,  $U$  and  $V$  are both  $M$ -by- $M$  unitary matrices
- $M > N$ ,  $V$  is an  $N$ -by- $N$  unitary matrix, and  $U$  is an  $M$ -by- $N$  matrix whose columns are the first  $N$  columns of a unitary matrix
- $N > M$ ,  $U$  is an  $M$ -by- $M$  unitary matrix, and  $V$  is an  $N$ -by- $M$  matrix whose columns are the first  $M$  columns of a unitary matrix

In all cases,  $S$  is an unoriented vector of positive singular values having length  $P$ .

Length- $N$  row inputs are treated as length- $N$  columns.

Note that the first (maximum) element of output  $S$  is equal to the 2-norm of the matrix  $A$ .

## Parameters

### Show singular vector ports

Select to enable the  $U$  and  $V$  output ports.

### Show error status port

Select to enable the  $E$  output port, which reports a failure to converge. The possible values you can receive on the port are:

- 0 — The singular value decomposition calculation converges.
- 1 — The singular value decomposition calculation does not converge.

If the singular value decomposition calculation fails to converge, the output at ports U, S, and V are undefined matrices of the correct size.

### Simulate using

Type of simulation to run. You can set this parameter to:

- Interpreted execution (default)

Simulate model using the MATLAB interpreter. This option shortens startup time.

- Code generation

Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time.

## References

Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

## Supported Data Types

| Port | Supported Data Types   |
|------|--|
| A    | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> </ul> |
| U    | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> </ul> |
| S    | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> </ul> |
| V    | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> </ul> |
| E    | <ul style="list-style-type: none"> <li>• Boolean</li> </ul>  |

## See Also

Autocorrelation LPC

Cholesky Factorization

LDL Factorization

LU Inverse

Pseudoinverse

QR Factorization

SVD Solver

DSP System Toolbox

DSP System Toolbox

DSP System Toolbox

DSP System Toolbox

DSP System Toolbox

DSP System Toolbox

DSP System Toolbox

svd

MATLAB

See “Matrix Factorizations” for related information.

## **Version History**

**Introduced before R2006a**

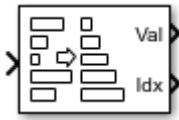
## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

## Sort

Sort input elements by value



**Libraries:**  
DSP System Toolbox / Statistics

### Description

The Sort block ranks the values of the input elements along each channel (column) in an **Ascending** or a **Descending** order, based on the **Sort order** you specify. Complex inputs are sorted by their magnitude, which is the sum of the squares of the real and imaginary components of the input. You can choose the **Sort algorithm** to be either **Quick sort** or **Insertion sort**. The quick sort algorithm uses a recursive sort method and is faster at sorting more than 32 elements. The insertion sort algorithm uses a nonrecursive method and is faster at sorting fewer than 32 elements. When you generate code, use the insertion sort algorithm to avoid recursive function calls.

The **Mode** parameter specifies the block's mode of operation, which you can set to **Value**, **Index**, or **Value and Index**.

### Ports

#### Input

**Port\_1** — Data input

vector | matrix

The block accepts real-valued or complex-valued multichannel inputs. The input data type must be double precision, single precision, integer, or fixed point, with power-of-two slope and zero bias.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fixed point

#### Output

**Val** — Sorted data

vector | matrix

The block sorts the data along each channel and outputs the sorted data through this port. The size, data type, and complexity of the sorted data matches that of the input data. The block sorts complex inputs according to their magnitude.

#### Dependencies

To enable this port, set the **Mode** parameter to **Value** and **index** or **Value**.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fixed point

**Idx** — Index of the sorted data  
vector | matrix

The output at this port contains the indices of the sorted data.

### Dependencies

To enable this port, set the **Mode** parameter to `Value` and `index` or `Index`.

Data Types: `uint32`

## Parameters

### Main Tab

**Mode** — Specify whether block returns values, indices, or both  
`Value` and `Index` (default) | `Value` | `Index`

When the **Mode** parameter is set to:

- `Value` — The block sorts the elements in each channel of the  $M$ -by- $N$  input matrix in an ascending or descending order, based on what you specify in the **Sort order** parameter. The output at each sample time, *Val*, is an  $M$ -by- $N$  matrix that contains the sorted columns of the input.

The block sorts complex inputs according to their magnitude.

- `Index` — The block sorts the elements in each channel of the  $M$ -by- $N$  input matrix, and outputs the index array, *I*. Each element in *I* is an integer of type `uint32` that indexes the sorted value in the corresponding column of the input.
- `Value` and `index` — The block outputs the sorted values of the input data, *Val*, and the corresponding indices in the index array, *I*.

**Sort order** — Order of sorting  
`Ascending` (default) | `Descending`

Specify to sort the input data in either ascending or descending order.

**Sort algorithm** — Sort method  
`Quick sort` (default) | `Insertion sort`

The quick sort algorithm uses a recursive sort method and is faster at sorting more than 32 elements. The insertion sort algorithm uses a nonrecursive method and is faster at sorting fewer than 32 elements. When you generate code, to avoid recursive function calls, use the insertion sort algorithm.

### Data Types Tab

---

**Note** To use these parameters, the data input must be complex and fixedpoint. For all other inputs, the parameters on the **Data Types** tab are ignored.

---

**Rounding mode** — Method of rounding operation  
`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

Specify the rounding mode for fixed-point operations. For more details, see rounding mode.

**Saturate on integer overflow** — Method of overflow action

off (default) | on

When you select this parameter, the block saturates the result of its fixed-point operation. When you clear this parameter, the block wraps the result of its fixed-point operation. For details on `saturate` and `wrap`, see overflow mode for fixed-point operations.

**Product output** — Product output data type

Inherit: Same as input (default) | `fixdt([],16,0)`

The squares of the real and imaginary parts of the complex input are stored in the **Product output** data type.

You can set this parameter to:

- `Inherit: Same as input` — The product output data type is the same as the input data type.
- `fixdt([],16,0)` — The product output data type is an autosigned, binary-point, scaled, fixed-point data type with a word length of 16 bits and a fraction length of 0.

Alternatively, you can set the **Product output** data type by using the **Data Type Assistant**. To use

the assistant, click the **Show data type assistant** button .

For more information on the data type assistant, see “Specify Data Types Using Data Type Assistant” (Simulink) in *Simulink User's Guide* (Simulink).

**Accumulator** — Accumulator data type

Inherit: Same as product output (default) | Inherit: Same as input | `fixdt([],16,0)`

The result of the sum of the squares of the real and imaginary parts of the complex input are stored in the **Accumulator** data type.

You can set this parameter to:

- `Inherit: Same as product output` — The accumulator data type is the same as the product output data type.
- `Inherit: Same as input` — The accumulator data type is the same as the input data type.
- `fixdt([],16,0)` — The accumulator data type is an autosigned, binary-point, scaled, fixed-point data type with a word length of 16 bits and a fraction length of 0.

Alternatively, you can set the **Accumulator** data type by using the **Data Type Assistant**. To use the

assistant, click the **Show data type assistant** button .

For more information on the data type assistant, see “Specify Data Types Using Data Type Assistant” (Simulink) in *Simulink User's Guide* (Simulink).

**Lock data type settings against changes by the fixed-point tools** — Prevent fixed-point tools from overriding data types  
off (default) | on

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block.

## Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | double   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no                                      |
| <b>Multidimensional Signals</b> | no                                      |
| <b>Variable-Size Signals</b>    | no                                      |
| <b>Zero-Crossing Detection</b>  | no                                      |

## Algorithms

### Sort

The block produces results identical to the MATLAB `sort` function.

The output of the block is equivalent to the following MATLAB code, when **Sort order** is set to:

- Ascending — `[Val,I] = sort(u,'ascend')`
- Descending — `[Val,I] = sort(u,'descend')`

where:

- `u` is the data input.
- `Val` is the sorted output.
- `I` is the index of the sorted output.

When the input is complex, the block sorts the data according to the magnitude. The block computes the magnitude by taking the sum of the squares of the real and imaginary components of the complex input. This is identical to calling the `sort` function as `[Val,I] = sort(u,...,'ComparisonMethod','abs')`.

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Generated code relies on the `memcpy` or `memset` function (`string.h`) under certain conditions.

**Fixed-Point Conversion**

Design and simulate fixed-point systems using Fixed-Point Designer™.

The parameters on the **Data Types** tab are used only for complex fixed-point inputs. Complex inputs are sorted by their magnitude, which is the sum of the squares of the real and imaginary components of the input. The results of the squares of the real and imaginary parts are stored in the **Product output** data type. The result of the sum of the squares is stored in the **Accumulator** data type. The parameters on the **Data Types** tab are ignored for all other inputs.

**See Also****Functions**

sort

**Blocks**

Median Filter | Histogram | Median



# Spectrum Analyzer

Display frequency spectrum



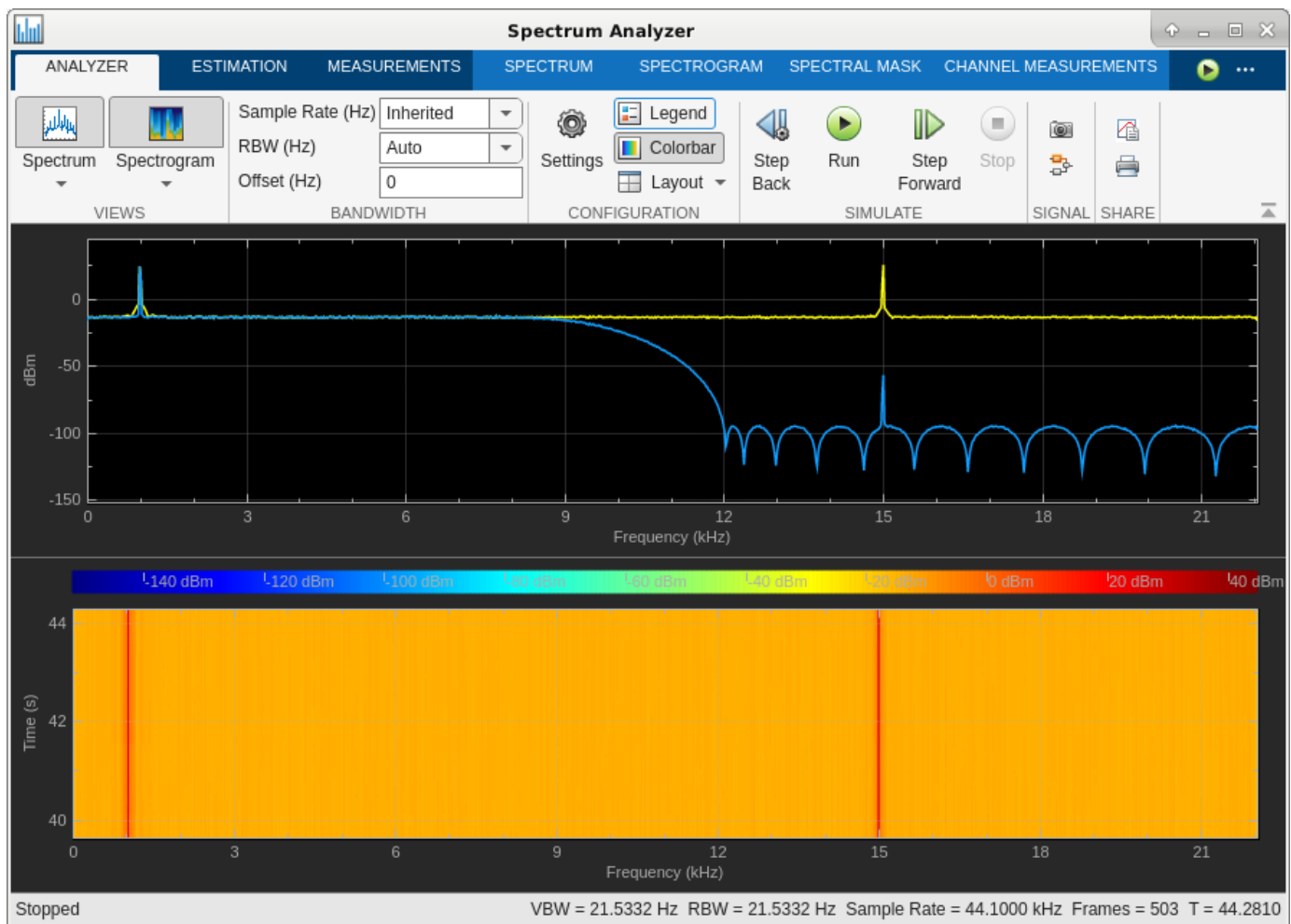
## Libraries:

DSP System Toolbox / Sinks

DSP System Toolbox HDL Support / Sinks

## Description

The Spectrum Analyzer block, referred to here as the scope, displays frequency-domain signals and the frequency spectrum of time-domain signals. The scope shows the spectrum view and the spectrogram view. The block algorithm performs spectral estimation using the filter bank method and Welch's method of averaged modified periodograms. You can customize the spectrum analyzer display to show the data and the measurement information that you need. For more details, see "Algorithms" on page 2-1335.



You can use the Spectrum Analyzer block in models running in `Normal` or `Accelerator` simulation modes. You can also use the Spectrum Analyzer block in models running in `Rapid Accelerator` or `External` simulation modes with some limitations.

You can use the Spectrum Analyzer block inside all subsystems and conditional subsystems. Conditional subsystems include enabled subsystems, triggered subsystems, enabled and triggered subsystems, and function-call subsystems. See “Conditionally Executed Subsystems Overview” (Simulink) for more information.

## Measurements

- “Cursor Measurements” on page 2-1330 — Measure signal values using vertical and horizontal cursors.
- “Peak Finder Measurements” on page 2-1330 — Find maxima and show the x-axis values at which they occur.
- “Channel Measurements” on page 2-1333 — Measure the occupied bandwidth or adjacent channel power ratio (ACPR).
- “Distortion Measurements” on page 2-1331 — Measure harmonic distortion and intermodulation distortion.
- “Spectral Mask” on page 2-1334 — Visualize spectrum limits and compare spectrum values to specification values.

## Programmatic Control

You can configure and display the Spectrum Analyzer settings from the command line with the `SpectrumAnalyzerConfiguration` object.

## Ports

### Input

**Signal** — Signals to visualize  
scalar | vector | matrix | array

Connect the signals you want to visualize. You can have up to 96 input ports. Input signals must have these characteristics:

- **Signal Domain** — Frequency or time signals.
- **Type** — Discrete signals.
- **Data type** — Any data type that Simulink supports. See “Data Types Supported by Simulink” (Simulink).
- **Dimension** — One dimensional (vector), two dimensional (matrix), or multidimensional (array) signals. Input signal must have a fixed number of channels. See “Signal Dimensions” (Simulink) and “Determine Signal Dimensions” (Simulink).

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fixed point

**Frequency** — Frequencies in Hz  
column vector

Specify the frequencies in Hz. The frequency vector must be a finite, monotonically increasing, column vector with two or more elements. The number of frequency vector points must be equal to the input frame size. You can also specify the frequencies using the **Frequency (Hz)** parameter on the **Estimation** tab.

#### Dependency

To enable this port, set the:

- **Input Domain** parameter on the **Estimation** tab to Frequency.
- **Frequency (Hz)** parameter on the **Estimation** tab to Input port.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | Boolean | fixed point

#### RBW — RBW Value

positive scalar

Specify the resolution bandwidth in Hz through this port. RBW defines the smallest positive frequency that can be resolved by the scope. You can also specify the RBW value using the **RBW (Hz)** parameter on the **Analyzer** tab.

#### Dependency

To enable this port, set the **RBW (Hz)** parameter on the **Analyzer** tab to Input port.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | Boolean | fixed point

#### VBW — VBW Value

positive scalar

Specify the video bandwidth in Hz through this port. Video bandwidth is the bandwidth of the lowpass filter that the scope uses to average or smooth the noise in the signal before displaying it. You can also specify the VBW value using the **VBW (Hz)** parameter on the **Estimation** tab.

#### Dependency

To enable this port, set the:

- **Input Domain** parameter on the **Estimation** tab to Time.
- **Averaging Method** parameter on the **Estimation** tab to VBW.
- **VBW (Hz)** parameter on the **Estimation** tab to Input port.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | Boolean | fixed point

## Parameters

### Analyzer Tab

#### Views

#### Spectrum — Type of spectrum to display

Power (default) | Power Density | RMS

Set the type of spectrum to display as one of these values:

- **Power** — Spectrum Analyzer shows the power spectrum.
- **Power Density** — Spectrum Analyzer shows the power spectral density. The power spectral density is the squared magnitude of the spectrum normalized to a bandwidth of 1 Hz.
- **RMS** — Spectrum Analyzer shows the root mean squared spectrum. Use this option to view the frequency of voltage or current signals.

**Tunable:** Yes

**Dependency**

To enable this parameter, set the **Input Domain** parameter on the **Estimation** tab to Time.

**Programmatic Use**

**Block Parameter:** SpectrumType

**Type:** character vector or string scalar

**Spectrogram** — Type of spectrogram to display

Power (default) | Power Density | RMS

Set the type of spectrogram to display as one of these values:

- **Power** — Spectrum Analyzer shows the power spectrogram.
- **Power Density** — Spectrum Analyzer shows the power density of the spectrogram. The power spectrogram density is the squared magnitude of the spectrogram normalized to a bandwidth of 1 Hz.
- **RMS** — Spectrum Analyzer shows the root mean square of the spectrogram. The root-mean-square shows the square root of the mean square. Use this option to view the frequency of voltage or current signals.

**Tunable:** Yes

**Dependency**

To enable this parameter, set the **Input Domain** parameter on the **Estimation** tab to Time.

**Programmatic Use**

**Block Parameter:** SpectrumType

**Type:** character vector or string scalar

**Bandwidth**

**Sample Rate (Hz)** — Sample rate the scope uses in Hz

Inherited (default) | positive scalar

Specify the sample rate the scope uses in Hz as one of the following:

- **Inherited** -- Use this option to specify the same sample rate as the input signal.
- **Positive scalar** -- The sample rate you specify must be at least twice the sample rate of the input signal. Otherwise, you might see unexpected behavior when visualizing your signal in the scope due to aliasing.

To display this property on the status bar, right-click the status bar at the bottom of the Spectrum Analyzer window and select **Sample Rate**.

**Tunable:** Yes

**Programmatic Use**

**Block Parameter:** SampleRate, SampleRateSource

**Type:** double

**RBW (Hz)** — Resolution bandwidth in Hz  
Auto (default) | Input port | positive scalar

Specify the resolution bandwidth in Hz. This parameter defines the smallest positive frequency that can be resolved by the scope. By default, this parameter is set to **Auto**. In this case, the Spectrum Analyzer determines the appropriate value to ensure that there are 1024 RBW intervals over the specified frequency span.

If you set this parameter to **Input port**, you can specify the RBW value through an input port on the block.

If you set this parameter to a numeric value, the value must allow at least two RBW intervals over the specified frequency span. In other words, the ratio of the overall frequency span to RBW must be greater than two:

$$\frac{\text{span}}{\text{RBW}} > 2$$

To display this property on the status bar, right-click the status bar at the bottom of the Spectrum Analyzer window and select **RBW**.

**Tunable:** Yes

**Programmatic Use**

**Block Parameter:** RBWSource, RBW

**Type:** character vector, string scalar, double

**Offset (Hz)** — Offset to apply to frequency axis  
0 (default) | scalar | vector

Specify the offset to apply to the frequency axis (x-axis) in units of Hz as one of the following:

- **Scalar** — Apply the same frequency offset to all channels.
- **Vector** — Apply a specific frequency offset for each channel. The vector length must be equal to the number of input channels.

The overall span must fall within the “Nyquist Frequency Interval” on page 2-1340. You can control the overall span in different ways based on how you set the **Span (Hz)** parameter.

**Tunable:** Yes

**Programmatic Use**

**Block Parameter:** FrequencyOffset

**Type:** double

**Configuration > Spectrum Analyzer Settings ( )**

**Num Inputs** — Number of input ports  
1 (default) | integer between 1 and 96

The number of input ports to the block, specified as an integer between 1 and 96. To change the number of input ports, drag a new input signal line to the block and the block automatically creates new ports.

**Programmatic Use**

**Block Parameter:** NumInputPorts  
**Type:** character vector or string scalar  
**Values:** scalar between 1 and 96

**Open at Simulation Start** — Automatically open scope when simulation starts  
on (default) | off

Select this parameter to automatically open the Spectrum Analyzer window when you run the simulation.

**Programmatic Use**

**Block Parameter:** OpenAtSimulationStart  
**Type:** logical

**Font Size** — Font size of labels  
Small (default) | Medium | Large | Extra Large

Specify the font size of labels in the display as Small, Medium, Large, and Extra Large.

**Tunable:** Yes

**Programmatic Use**

**Block Parameter:** -  
**Type:** character vector or string scalar

**Plot Type** — Plot type  
Line (default) | Stem

Specify whether to display a Line or Stem plot in the Spectrum display.

**Tunable:** Yes

**Dependency**

To enable this parameter:

- Select **Spectrum** in the **Analyzer** tab.
- Select the **Normal Trace** check box in the **Spectrum** tab.

**Programmatic Use**

**Block Parameter:** PlotType  
**Type:** character vector or string scalar

**Y-Label** — y-axis label

character vector | string scalar

Specify the y-axis label in the Spectrum display as a character vector or a string scalar. To display signal units, add (%<SignalUnits>) to the label. When simulation starts, Simulink replaces (%SignalUnits) with the units associated with the signals.

For example, for a velocity signal with units of m/s enter

Velocity (%<SignalUnits>)

**Tunable:** Yes

**Dependency**

To enable this parameter, select **Spectrum** in the **Analyzer** tab.

**Programmatic Use**

**Block Parameter:** YLabel

**Type:** character vector or string scalar

**Y-Axis Limits** — y-axis limits

[-80 20] | [ymin ymax]

Specify the y-axis limits in the Spectrum Analyzer display as a two-element numeric vector of the form [ymin ymax]. The units of the y-axis limits depend on the **Spectrum Unit** in the **Spectrum** tab.

**Tunable:** Yes

**Dependency**

To enable this parameter, select **Spectrum** in the **Analyzer** tab.

**Programmatic Use**

**Block Parameter:** YLimits

**Type:** double

**Title** — Display title

character vector | string

Specify the display title. Enter %<SignalLabel> to use the signal labels in the Simulink model as the axes titles.

**Tunable:** Yes

**Programmatic Use**

**Block Parameter:** Title

**Type:** character vector or string

**Show Grid** — Show internal grid lines

on (default) | off

Select this check box to show the grid in the Spectrum Analyzer display.

**Tunable:** Yes

**Programmatic Use****Block Parameter:** ShowGrid**Type:** logical**Color Map** — Spectrogram colormap

jet (default) | hot | bone | cool | copper | gray | parula | three-column matrix

Select a valid colormap name for the spectrogram, or enter a three-column matrix with values in the range [0,1] defining RGB triplets. For more information about colormaps, see `colormap`.

**Tunable:** Yes**Dependency**

To enable this parameter, select **Spectrogram** in the **Analyzer** tab.

**Programmatic Use****Block Parameter:** Colormap**Type:** character vector or string scalar**Color Limits** — Color limits of spectrogram

[-80, 20] (default) | [colorMin colorMax]

Specify the color limits of the spectrogram as a two-element numeric vector of the form [colorMin colorMax]. The units of the color limits directly depend upon the **Spectrum Unit** in the **Spectrogram** tab.

**Tunable:** Yes**Dependencies**

To enable this parameter, select **Spectrogram** in the **Analyzer** tab.

**Programmatic Use****Block Parameter:** ColorLimits**Type:** double**Preserve colors for copy to clipboard** — Preserve colors when copying to clipboard

off (default) | on

When you select this parameter, the scope preserves the colors when you copy the display to the clipboard using the **Copy Display** and the **Print** options in the **Analyzer** tab > **Share** section.

**Tunable:** Yes**Background** — Window background

black (default) | color picker

Specify the background color in the scope figure.

**Tunable:** Yes**Axes** — Axes background color

black (default) | color picker



Specify the background color of the axes.

**Tunable:** Yes

**Labels** — Color of labels  
gray (default) | color picker

Specify the color of the labels, grid, and the channel names in the legend.

**Tunable:** Yes

**Line** — Channel for line property settings  
channel names

Specify the channel for which you want to modify the visibility, line color, style, width, and marker properties.

**Tunable:** Yes

**Visible** — Channel visibility  
on (default) | off

Select this check box to display the channel you have selected. If you clear this check box, the selected channel is no longer visible. You can also click the signal name in the legend to control its visibility. For more details, see “Legend” on page 2-0 .

**Tunable:** Yes

**Style** — Line style  
- (default) | : | -. | -- | None

Specify the line style for the selected channel.

**Tunable:** Yes

**Width** — Line width  
1.5 (default) | 0.5 | 1 | 2 | ...

Specify the line width for the selected channel.

**Tunable:** Yes

**Marker** — Data point markers  
None (default) | + | x | ...

Specify a data point marker for the selected channel. This parameter is similar to the 'Marker' property for plots. You can choose any of the marker symbols from the drop-down list.

**Tunable:** Yes

**Color** — Line color  
yellow (default) | color picker

Specify the line color for the selected channel.


**Tunable:** Yes

## Configuration

**Legend** — Show or hide signal legend  
button

Click the **Legend** button to enable the Spectrum Analyzer to display the signal legend. The legend displays the signal names from the model. For signals with multiple channels, the scope appends a channel index after the signal name. Continuous signals have straight lines before their names and discrete signals have step-shaped lines.

You can control which signals are visible using the legend. To hide a signal, in the scope legend, click the signal name. To display the signal, click the signal name again. Alternatively, you can control

which signal is visible using the **Visible** parameter in the **Spectrum Analyzer Settings** () .

To display only one signal and hide all other signals, right-click the name of the signal you want the scope to display. To show all signals, press **Esc**.

---

**Note** The legend displays only the first 20 signals. You cannot view or control any additional signals from the legend.

---

**Tunable:** Yes

### Dependency

To enable the **Legend**, select **Spectrum** in the **Analyzer** tab.

### Programmatic Use

**Block Parameter:** ShowLegend  
**Type:** logical

**Colorbar** — Show or hide color bar  
button

When you select the **Colorbar** button, the Spectrum Analyzer shows the color bar.

**Tunable:** Yes

### Dependencies

To enable the **Colorbar** button, select **Spectrogram** in the **Analyzer** tab.

### Programmatic Use

**Block Parameter:** ShowColorbar  
**Type:** logical

**Layout** — Stack axes vertically or horizontally  
vertical layout (default) | horizontal layout

Stack axes vertically or horizontally by selecting the appropriate configuration in the layout grid.

**Tunable:** Yes

**Dependencies**

To enable the **Layout**, select **Spectrum** and **Spectrogram** in the **Analyzer** tab.

**Programmatic Use**

**Block Parameter:** AxesLayout

**Type:** character vector or string scalar

**Share**

**Copy Display** — Send display to clipboard  
button

Click this button to copy the scope display to the clipboard. You can preserve the color in the display by selecting the “Preserve colors for copy to clipboard” on page 2-0 parameter.

**Print** — Print display  
button

Click this button to save the scope display as an image or a PDF or to print the display.

**Estimation Tab****Domain**

**Input Domain** — Domain of the input signal

Time (default) | Frequency

The domain of the input signal you want to visualize. If you visualize time-domain signals, the scope transforms the signal to the frequency spectrum based on the algorithm you specify in the **Method** parameter.

**Programmatic Use**

**Block Parameter:** InputDomain

**Type:** character vector or string scalar

**Frequency (Hz)** — Frequency vector

Auto (default) | Input port | monotonically increasing vector

Set the frequency vector which determines the x-axis of the display to one of these values:

- Auto — The scope calculates the frequency vector from the length of the input. For more details, see “Frequency Vector” on page 2-1341.
- Input port — You specify the frequency vector at the **Frequency** input port on the block.
- Custom vector — You specify a custom vector as the frequency vector. The length of the custom vector must be equal to the frame size of the input signal.

**Tunable:** Yes

**Dependency**

To enable this parameter, set **Input Domain** to Frequency.

**Programmatic Use****Block Parameter:** FrequencyVectorSource, FrequencyVector**Type:** character vector, string scalar, double**Input Unit** — Units of frequency input

dBm (default) | dBV | dBW | Vrms | Watts | None

Select the units of the frequency-domain input. This parameter enables the Spectrum Analyzer to scale frequency data when you select a different display unit in the **Spectrum Unit** parameter in the **Estimation** tab.

**Tunable:** Yes**Dependency**

To enable this parameter, set **Input Domain** to Frequency.

**Programmatic Use****Block Parameter:** InputUnits**Type:** character vector or string scalar**Frequency Resolution****Method** — Spectrum estimation method

Filter Bank (default) | Welch

Select the spectrum estimation method as one of the following:

- **Filter Bank** -- Use an analysis filter bank to estimate the power spectrum. Compared to Welch's method, this method has a lower noise floor, better frequency resolution, lower spectral leakage, and requires fewer samples per update.
- **Welch** -- Use Welch's method of averaged modified periodograms.

For more details on these methods, see "Algorithms" on page 2-1335.

**Tunable:** Yes**Dependency**

To use this parameter, set **Input Domain** to Time.

**Programmatic Use****Block Parameter:** Method**Type:** character vector or string scalar**Sharpness** — Sharpness of lowpass filter

0.5 (default) | nonnegative scalar in the range [0,1]

Specify the sharpness of the prototype lowpass filter as a real nonnegative scalar in the range [0,1].

Increasing the filter sharpness decreases the spectral leakage and gives a more accurate power reading.

**Tunable:** Yes

**Dependencies**

To enable this property, set **Method** to `Filter bank`.

**Programmatic Use**

**Block Parameter:** `FilterSharpness`

**Type:** `double`

**Averaging**

**Averaging Method** — Smoothing method

`VBW (default)` | `Exponential`

Specify the smoothing method as one of the following:

- **VBW** — Video bandwidth method. The block uses a lowpass filter to smooth the trace and decrease the noise. Use the **VBW (Hz)** parameter to specify the video bandwidth (VBW) value.
- **Exponential** — Weighted average of samples. The block computes the average over samples weighted by an exponentially decaying forgetting factor. Use the **Forgetting Factor** parameter to specify the weighted forgetting factor.

For more information on the averaging methods, see “Averaging Method” on page 2-1343.

**Tunable:** Yes

**Dependency**

To enable this parameter, set **Input Domain** to `Time`.

**Programmatic Use**

**Block Parameter:** `AveragingMethod`

**Type:** `character vector` or `string scalar`

**VBW (Hz)** — Video bandwidth

`Auto (default)` | `Input port` | `positive scalar`

Specify the video bandwidth as one of the following:

- `Auto` -- The Spectrum Analyzer adjusts the VBW such that the equivalent forgetting factor is 0.9.
- `Input port` -- You specify the frequency vector at the **VBW** input port on the block.
- `Positive scalar` -- You specify a positive scalar. The Spectrum Analyzer adjusts the VBW using this value. The value you specify must be less than or equal to **Sample Rate (Hz)/2**.

For more details on the video bandwidth method, see “Averaging Method” on page 2-1343.

The Spectrum Analyzer shows the VBW value in the status bar at the bottom of the display. To display the VBW value, right-click the status bar and select **VBW**.

**Tunable:** Yes

**Dependency**

To enable this parameter, set:

- **Input Domain** to Time.
- Averaging Method to VBW.

#### Programmatic Use

**Block Parameter:** VBWSource, VBW

**Type:** double

**Forgetting Factor** — Forgetting factor of weighted average method

0.9 (default) | scalar in the range [0,1]

Specify the forgetting factor of the exponential weighted averaging method as a scalar in the range [0,1].

**Tunable:** Yes

#### Dependency

To enable this parameter, set:

- **Input Domain** to Time.
- **Averaging Method** to Exponential.

#### Programmatic Use

**Block Parameter:** ForgettingFactor

**Type:** double

#### Frequency Options

**Frequency Span** — Frequency span mode

Full (default) | Span and Center Frequency | Start and Stop Frequencies

Specify the frequency span mode as one of the following:

- **Full** -- The Spectrum Analyzer computes and plots the spectrum over the entire “Nyquist Frequency Interval” on page 2-1340.
- **Span and Center Frequency** -- The Spectrum Analyzer computes and plots the spectrum over the interval specified by the **Span (Hz)** and **Center Frequency (Hz)** parameters.
- **Start and Stop Frequencies** -- The Spectrum Analyzer computes and plots the spectrum over the interval specified by the **Start Frequency (Hz)** and **Stop Frequency (Hz)** parameters.

**Tunable:** Yes

#### Dependency

To enable this parameter, set **Input Domain** to Time.

#### Programmatic Use

**Block Parameter:** FrequencySpan

**Type:** character vector or string scalar

**Span (Hz)** — Frequency span to compute spectrum in Hz

10e3 (default) | real positive scalar

Specify the frequency span in Hz over which the Spectrum Analyzer computes and plots the spectrum. The overall span, defined by this parameter and the **Center Frequency (Hz)** parameter, must fall within the “Nyquist Frequency Interval” on page 2-1340. This parameter defines the range of the values shown on the Frequency axis in the Spectrum Analyzer window.

**Tunable:** Yes

**Dependency**

To enable this parameter, set:

- **Input Domain** to Time.
- **Frequency Span** to Span and Center Frequency.

**Programmatic Use**

**Block Parameter:** Span

**Type:** double

**Center Frequency (Hz)** — Center of frequency span in Hz  
0 (default) | real scalar

Specify the center of the frequency span in Hz over which the Spectrum Analyzer computes and plots the spectrum. Use this parameter with the **Span (Hz)** parameter to define the frequency span around a center frequency. This parameter defines the midpoint of the Frequency axis in the Spectrum Analyzer window.

**Tunable:** Yes

**Dependency**

To enable this parameter, set:

- **Input Domain** to Time.
- **Frequency Span** to Span and Center Frequency.

**Programmatic Use**

**Block Parameter:** CenterFrequency

**Type:** double

**Start Frequency (Hz)** — Start frequency in Hz  
-5e3 (default) | scalar

Specify the starting frequency in Hz of the frequency interval over which the Spectrum Analyzer computes and plots the spectrum. The overall span, which is defined by this parameter and the **Stop Frequency (Hz)** parameter, must fall within the “Nyquist Frequency Interval” on page 2-1340. This parameter defines the leftmost value on the Frequency axis in the Spectrum Analyzer window.

**Tunable:** Yes

**Dependency**

To enable this parameter, set:

- **Input Domain** to Time.

- **Frequency Span** to Start and Stop Frequencies.

**Programmatic Use****Block Parameter:** StartFrequency**Type:** double**Stop Frequency (Hz)** — Stop frequency in Hz  
5e3 (default) | scalar

Specify the stop frequency in Hz of the frequency interval over which the Spectrum Analyzer computes and plots the spectrum. The overall span, which is defined by this parameter and the **Start Frequency (Hz)** parameter, must fall within the “Nyquist Frequency Interval” on page 2-1340. This parameter defines the rightmost value on the Frequency axis in the Spectrum Analyzer window.

**Tunable:** Yes**Dependency**

To enable this parameter, set:

- **Input Domain** to Time.
- **Frequency Span** to Start and Stop Frequencies.

**Programmatic Use****Block Parameter:** StopFrequency**Type:** double**Window Options****Window** — Windowing method

Hann (default) | Blackman-Harris | Chebyshev | Flat Top | Hamming | Kaiser | Rectangular | custom window function name

Specify the windowing method to apply to the spectrum. Windowing is used to control the effect of sidelobes in spectral estimation. The window you specify affects the window length required to achieve a resolution bandwidth and the required number of samples per update. For more information about windowing, see “Windows”.

You can use your own spectral estimation window by directly specifying a custom window function name in the **Window** parameter.

**Tunable:** Yes**Dependency**

To enable this parameter, set:

- **Input Domain** to Time.
- **Method** to Welch.

**Programmatic Use****Block Parameter:** Window, CustomWindow**Type:** character vector or string scalar



**Attenuation (dB)** — Sidelobe attenuation in dB  
60 (default) | scalar greater than or equal to 45

Specify the sidelobe attenuation in dB as a scalar greater than or equal to 45.

**Tunable:** Yes

**Dependency**

To enable this parameter, set **Window** to Chebyshev or Kaiser.

**Programmatic Use**

**Block Parameter:** SidelobeAttenuation  
**Type:** double

**Overlap (%)** — Percentage of overlap  
0 (default) | scalar in the range [0 100)

Specify the percentage of overlap between the previous and the current buffered data segments as a scalar in the range [0 100). The overlap creates a window segment that the scope uses to compute a spectral estimate. The value must be greater than or equal to zero and less than 100.

**Tunable:** Yes

**Dependency**

To enable this parameter, set:

- **Input Domain** to Time.
- **Method** to Welch.

**Programmatic Use**

**Block Parameter:** OverlapPercent  
**Type:** double

**Measurements Tab**

**Channel**

**Channel** — Channel for which to obtain measurements  
1 (default) | positive integer

The channel for which you need to obtain measurements, specified as a positive integer in the range [1  $N$ ], where  $N$  is the number of input channels.

**Tunable:** Yes

**Dependency**

To enable this parameter, pass some data through the scope.

**Programmatic Use**

See MeasurementChannel.

**Cursors**

**Data Cursors** — Enable cursor measurements  
button

Click the **Data Cursors** button to enable data cursor measurements. Each cursor tracks a vertical line along the signal. The scope displays the difference between x- and y-values of the signal at the two cursors in the box between the cursors.

**Tunable:** Yes

**Programmatic Use**

See Enabled.

**Snap to data** — Snap cursors to data  
off (default) | on

Select this parameter to position the cursors on the signal data points.

**Tunable:** Yes

**Programmatic Use**

See SnapToData.

**Lock cursor spacing** — Lock cursor spacing  
off (default) | on

Select this parameter to lock the frequency difference between the two cursors.

**Tunable:** Yes

**Programmatic Use**

See LockSpacing.

**Peaks**

**Peak Finder** — Enable peak finder measurements  
button

Click the **Peak Finder** button to enable peak finder measurements. An arrow appears on the plot at each maxima and a Peaks panel appears at the bottom of the scope window.

**Tunable:** Yes

**Programmatic Use**

See Enabled.

**Num Peaks** — Maximum number of peaks to show  
3 (default) | positive integer less than 100

Specify the maximum number of peaks to show as a positive integer less than 100.

**Tunable:** Yes

**Programmatic Use**

See NumPeaks.

**Min Height** — Level above which scope detects peaks

-Inf (default) | real scalar value

Specify the level above which the scope detects peaks as a real scalar.

**Tunable:** Yes

**Programmatic Use**

See MinHeight.

**Min Distance** — Minimum number of samples between adjacent peaks

1 (default) | positive integer

Specify the minimum number of samples between adjacent peaks as a positive integer.

**Tunable:** Yes

**Programmatic Use**

See MinDistance.

**Threshold** — Minimum difference between height of peak and its neighboring samples

0 (default) | nonnegative scalar

Specify the minimum difference between the height of the peak and its neighboring samples as a nonnegative scalar.

**Tunable:** Yes

**Programmatic Use**

See Threshold.

**Label Peaks** — Label peaks

button

Click the **Label Peaks** button to label the peaks. The scope displays the labels (**P1**, **P2**, ...) above the arrows in the plot.

**Tunable:** Yes

**Programmatic Use**

See LabelPeaks.

**Distortion**

**Distortion** — Enable distortion measurements

button

Click the **Distortion** button to enable distortion measurements. A Distortion panel appears at the bottom of the scope window when you click this button.

**Tunable:** Yes

**Programmatic Use**

See Enabled.

**Distortion Type** — Type of measurement to display

Harmonic (default) | Intermodulation

Specify the type of measurement data to display as **Harmonic** or **Intermodulation**. For more details, see “Distortion Measurements” on page 2-1331.

**Tunable:** Yes

**Programmatic Use**

See Type.

**Num Harmonics** — Number of harmonics to measure

6 (default) | positive integer

Specify the number of harmonics to measure as a positive integer less than or equal to 99.

**Tunable:** Yes

**Dependency**

To enable this parameter, set **Distortion Type** to **Harmonic**.

**Programmatic Use**

See NumHarmonics.

**Label Harmonics** — Label harmonics

off (default) | on

When you select this parameter, the Spectrum Analyzer adds numerical labels to harmonics in the spectrum display.

**Tunable:** Yes

**Programmatic Use**

See LabelValues.

**Label Frequencies** — Label frequencies

off (default) | on

When you select this parameter, the Spectrum Analyzer adds numerical labels to the first-order intermodulation product and third-order frequencies in the Spectrum Analyzer display.

**Tunable:** Yes

**Programmatic Use**

See LabelValues.

## Spectrum Tab

---

**Note** This tab appears when you select **Spectrum** in the **Analyzer** tab.

---

### Trace Options

**Two-Sided Spectrum** — Enable two-sided spectrum view  
on (default) | off

Select this check box to enable a two-sided spectrum view. In this view, the Spectrum Analyzer shows both negative and positive frequencies. When the input signal is complex-valued, you must select this parameter. If you clear this check box, the Spectrum Analyzer shows a one-sided spectrum with positive frequencies only. In this case, the input signal data must be real valued.

When you clear this check box, the Spectrum Analyzer uses power folding. The y-axis values are twice the amplitude that they would be if you were to select this parameter, except at 0 and the Nyquist frequency. A one-sided power spectral density (PSD) contains the total power of the signal in the frequency interval from DC to half the Nyquist rate. For more information, see `pwelch`.

**Tunable:** Yes

### Programmatic Use

**Block Parameter:** `PlotAsTwoSidedSpectrum`  
**Type:** logical

**Normal Trace** — Normal trace view  
on (default) | off

When you select this check box, the Spectrum Analyzer calculates and plots the power spectral estimates. The Spectrum Analyzer performs a smoothing operation by averaging several spectral estimates and continues its spectral computations even when you clear this parameter.

**Tunable:** Yes

### Dependencies

To clear this check box, first select either the **Max-Hold Trace** or the **Min-Hold Trace** parameters.

To enable this parameter, select **Spectrum** in the **Analyzer** tab.

### Programmatic Use

**Block Parameter:** `PlotNormalTrace`  
**Type:** logical

**Max-Hold Trace** — Maximum hold trace view  
off (default) | on

Select this check box to enable the Spectrum Analyzer to plot the maximum spectral values of all the estimates. The Spectrum Analyzer computes the maximum-hold spectrum at each frequency bin by keeping the maximum value of all the power spectrum estimates. When you clear this check box, the Spectrum Analyzer resets its maximum-hold computations.

**Tunable:** Yes

**Dependency**

To enable this parameter, select **Spectrum** in the **Analyzer** tab.

**Programmatic Use**

**Block Parameter:** PlotMaxHoldTrace

**Type:** logical

**Min-Hold Trace** — Minimum hold trace view  
off (default) | on

Select this check box to enable the Spectrum Analyzer to plot the minimum spectral values of all the estimates. The Spectrum Analyzer computes the minimum-hold spectrum at each frequency bin by keeping the minimum value of all the power spectrum estimates. When you clear this check box, the Spectrum Analyzer resets its minimum-hold computations.

**Tunable:** Yes

**Dependency**

To enable this parameter, select **Spectrum** in the **Analyzer** tab.

**Programmatic Use**

**Block Parameter:** PlotMinHoldTrace

**Type:** logical

**Scale**

**Frequency Scale** — Scale of frequency axis  
Linear (default) | Log

Specify the scale to display frequencies as Linear or Log. When the frequency span contains negative frequency values, you cannot choose the logarithmic option.

**Tunable:** Yes

**Dependency**

To set the **Frequency Scale** to Log, clear the **Two-Sided Spectrum** check box in the **Trace Options** section in the **Spectrum** or the **Spectrogram** tab (if enabled). If you select the **Two-Sided Spectrum** check box, then the **Frequency Scale** parameter is set to Linear.

**Programmatic Use**

**Block Parameter:** FrequencyScale

**Type:** character vector or string scalar

**Reference load ( $\Omega$ )** — Reference load in  $\Omega$   
1 (default) | positive real scalar

Specify the reference load in ohms that the Spectrum Analyzer uses as a reference to compute the power values.

**Tunable:** Yes

**Dependency**

To enable this parameter, set:

- **Spectrum** type to Power or Power Density.
- **Spectrum Unit** to any option other than dBFS or dBFS/Hz.

**Programmatic Use**

**Block Parameter:** ReferenceLoad

**Type:** double

**Spectrum Unit** — Units of the spectrum

dBm (default) | dBFS | dBV | dBW | Vrms | Watts | dBm/Hz | dBW/Hz | dBFS/Hz | Watts/Hz | Auto

Specify the units in which the Spectrum Analyzer displays the power values as one of the following:

- dBm
- dBFS
- dBV
- dBW
- Vrms
- Watts
- dBm/Hz
- dBW/Hz
- dBFS/Hz
- Watts/Hz
- Auto

**Tunable:** Yes

**Dependency**

The units available depend on the value you choose for the **Spectrum** parameter in the **Analyzer** tab.

| Estimation tab > Input Domain parameter | Analyzer tab > Spectrum option | Available Units                   |
|---|--------------------------------|-----------------------------------|
| Time                                    | Power                          | dBm, dBW, dBFS, Watts             |
|   | Power Density                  | dBm/Hz, dBW/Hz, dBFS/Hz, Watts/Hz |
|   | RMS                            | dBV, Vrms                         |
| Frequency                               | —                              | Auto, dBm, dBV, dBW, Vrms, Watts  |

If you set the **Input Domain** parameter to Frequency and the **Spectrum Unit** parameter to Auto, the Spectrum Analyzer assumes the spectrum units to be equal to input units specified in the **Estimation tab > Input Unit** parameter. If you set the **Input Domain** parameter to Time and the **Spectrum Unit** parameter to any option other than Auto, the Spectrum Analyzer converts the units specified in the **Input Unit** parameter to the units specified in the **Spectrum Unit** parameter.

**Programmatic Use****Block Parameter:** SpectrumUnits**Type:** character vector or string scalar**Full Scale** — Full scale for dBFS units

Auto (default) | positive real scalar

The full scale used for the decibel full scale (dBFS) units. By default, the Spectrum Analyzer uses the entire spectrum scale. Specify a positive real scalar for the dBFS full scale.

**Tunable:** Yes**Dependencies**

To enable this parameter:

- In the **Analyzer** tab, set the spectrum type to Power or Power Density.
- In the **Estimation** tab, set **Input Domain** to Time.
- In the **Spectrum** tab, set **Spectrum Unit** to dBFS or dBFS/Hz (when spectrum type is set to Power Density).

**Programmatic Use****Block Parameter:** FullScale**Type:** double**Spectrogram Tab**

---

**Note** This tab appears when you select **Spectrogram** in the **Analyzer** tab.

---

**Channel****Channel** — Channel for which spectrogram is plotted

1 (default) | character vector of a positive integer | string scalar of a positive integer

Select the signal channel for which the spectrogram settings apply.

**Tunable:** Yes**Dependency**To enable this parameter, select **Spectrogram** in the **Analyzer** tab.**Programmatic Use****Block Parameter:** SpectrogramChannel**Type:** character vector, string scalar, double**Time Options****Time Resolution (s)** — Time resolution in seconds

Auto (default) | positive scalar



Time resolution is the amount of data, in seconds, used to compute a spectrogram line. The minimum attainable resolution is the amount of time it takes to compute a single spectral estimate. The tooltip displays the minimum attainable resolution given the current Spectrum Analyzer settings.

When you set **RBW (Hz)** and **Time Resolution (s)** to **Auto**, then the Spectrum Analyzer adjusts the RBW value such that there are 1024 RBW intervals in one frequency span and sets the time resolution is set to  $1/\text{RBW}$ .

When you set **RBW (Hz)** to **Auto** and **Time Resolution (s)** to a positive scalar, then time resolution becomes the main control and RBW is set to  $1/\text{Time Resolution (s)}$  Hz.

When you set **RBW (Hz)** to a positive scalar and **Time Resolution (s)** to **Auto**, then RBW becomes the main control and the time resolution is set  $1/\text{RBW (Hz)}$  s.

When you set **RBW (Hz)** and **Time Resolution (s)** to a positive value, then time resolution must be equal to or larger than the minimum attainable time resolution defined by  $1/\text{RBW (Hz)}$ . Several spectral estimates are combined into one spectrogram line to obtain the desired time resolution. Interpolation is used to obtain time resolution values that are not integer multiples of  $1/\text{RBW (Hz)}$ .

**Tunable:** Yes

**Dependency**

To enable this parameter, select **Spectrogram** in the **Analyzer** tab.

**Programmatic Use**

**Block Parameter:** TimeResolutionSource, TimeResolution

**Type:** character vector, string scalar, double

**Time Span (s)** — Time span in seconds

Auto (default) | positive scalar

The time span over which the Spectrum Analyzer displays the spectrogram specified as a positive scalar in seconds. The time span is the product of the desired number of spectral lines and the time resolution. When you set this parameter to **Auto**, the spectrogram displays 100 spectrogram lines at any given time. Otherwise, the spectrogram uses the time duration you specify in this parameter. The time span that you specify must be at least two times larger than the duration of the number of samples required for a spectral update.

**Tunable:** Yes

**Dependency**

To enable this parameter, select **Spectrogram** in the **Analyzer** tab.

**Programmatic Use**

**Block Parameter:** TimeSpanSource, TimeSpan

**Type:** character vector, string scalar, double

**Trace Options**

**Two-Sided Spectrum** — Enable two-sided spectrum view

on (default) | off

Select this check box to enable a two-sided spectrum view. In this view, the Spectrum Analyzer shows both negative and positive frequencies. When the input signal is complex-valued, you must select this

parameter. If you clear this check box, the Spectrum Analyzer shows a one-sided spectrum with positive frequencies only. In this case, the input signal data must be real valued.

When you clear this check box, the Spectrum Analyzer uses power folding. The y-axis values are twice the amplitude that they would be if you were to select this parameter, except at 0 and the Nyquist frequency. A one-sided power spectral density (PSD) contains the total power of the signal in the frequency interval from DC to half the Nyquist rate. For more information, see `pwelch`.

**Tunable:** Yes

**Programmatic Use**

**Block Parameter:** `PlotAsTwoSidedSpectrum`

**Type:** logical

**Spectral Mask Tab**

---

**Note** This tab appears when you select **Spectrum** in the **Analyzer** tab and set the **Spectrum** type to be Power or Power Density.

---

**Views**

**Upper Mask** — Enable upper spectral mask  
button

Select **Upper Mask** to display the upper mask in the spectrum plot. The **Spectral Mask** panel appears at the bottom of the Spectrum Analyzer window and displays mask details, such as number of times a mask succeeded, number of times a mask failed, channels causing the mask failure, and so on.

Use the **Upper Limits** parameter to specify the upper mask. If the entire spectrum plot is below the upper mask, the upper mask looks green. In all other cases, the upper mask looks red.

**Tunable:** Yes

**Programmatic Use**

See `EnabledMasks`.

**Lower Mask** — Enable lower spectral mask  
button

Select **Lower Mask** to display the lower mask in the spectrum plot. The **Spectral Mask** panel appears at the bottom of the Spectrum Analyzer window and displays mask details, such as number of times a mask succeeded, number of times a mask failed, channels causing the mask failure, and so on.

Use the **Lower Limits** parameter to specify the lower mask. If the entire spectrum plot is above the lower mask, the lower mask looks green. In all other cases, the lower mask looks red.

**Tunable:** Yes

**Programmatic Use**

See `EnabledMasks`.

## Configuration

**Upper Limits** — Limit for upper spectral mask

Inf (default) | scalar | two-column matrix

Specify the limit for the upper spectral mask as a scalar or a two-column matrix.

If **UpperMask** is a scalar, the upper limit mask uses the same power value for all frequencies specified in the Spectrum Analyzer.

If you specify a matrix, the first column contains the frequency values (Hz), which correspond to the x-axis values. The second column contains the power values, which correspond to the associated y-axis values.

To apply offsets to the power and frequency values, use the **Reference Level (dBr)** and the **Frequency Offset (Hz)** parameters.

**Tunable:** Yes

### Programmatic Use

See **UpperMask**.

**Lower Limits** — Limit for lower spectral mask

-Inf (default) | scalar | two-column matrix

Specify the limit for the lower spectral mask as a scalar or a two-column matrix.

If **LowerMask** is a scalar, the lower limit mask uses the same power value for all frequencies specified in the Spectrum Analyzer.

If you specify a matrix, the first column contains the frequency values (Hz), which correspond to the x-axis values. The second column contains the power values, which correspond to the associated y-axis values.

To apply offsets to the power and frequency values, use the **Reference Level (dBr)** and the **Frequency Offset (Hz)** parameters.

**Tunable:** Yes

### Programmatic Use

See **LowerMask**.

**Reference Level (dBr)** — Reference level for mask power values

0 (default) | real scalar | **Spectrum peak**

Specify the reference level for mask power values as a numeric scalar or set it to **Spectrum peak**.

When you set **Reference Level (dBr)** to a scalar value, the Spectrum Analyzer uses this value as the reference to the power values (in dBr) for the upper mask and the lower mask of the Spectrum Analyzer. The reference level should have the same units as the **Spectrum Unit** parameter in the **Spectrum** tab.

When you set **Reference Level (dBr)** to **Spectrum peak**, the Spectrum Analyzer uses the peak value of the current spectrum of the **Channel** in the **Spectral Mask** tab as the reference power value.

**Tunable:** Yes

**Programmatic Use**

See `ReferenceLevel` and `CustomReferenceLevel`.

**Channel** — Input channel with peak spectrum

1 (default) | integer

Select the input channel which the Spectrum Analyzer uses to determine the mask reference level. The peak value of the spectrum in this channel becomes the mask reference level when you set the **Reference Level (dBr)** parameter to `Spectrum peak`.

**Tunable:** Yes

**Dependency**

To enable this parameter, set **Reference Level (dBr)** to `Spectrum peak` and display some data on the scope.

**Programmatic Use**

See `SelectedChannel`.

**Frequency Offset (Hz)** — Frequency offset in Hz

0 (default) | finite numeric scalar

Specify the frequency offset in Hz as a finite numeric scalar. Using this value, the Spectrum Analyzer offsets the frequency values in the **Upper Mask** and the **Lower Mask** parameters.

**Tunable:** Yes

**Programmatic Use**

See `MaskFrequencyOffset`.

**Channel Measurements Tab**

---

**Note** This tab appears when you select **Spectrum** in the **Analyzer** tab.

---

**Channel**

**Channel** — Channel for computing occupied bandwidth and adjacent channel power ratio

1 (default) | positive integer

Specify the channel over which the Spectrum Analyzer computes and displays the occupied bandwidth and adjacent channel power ratio as a positive integer in the range [1 *N*], where *N* is the number of input channels.

**Tunable:** Yes

**Dependency**

To enable this parameter, pass data through the scope.

**Programmatic Use**

See MeasurementChannel.

**Channel Measurements**

**Channel Measurements** — Enable channel measurements  
button

Click **Channel Measurements** to enable channel measurements.

**Tunable:** Yes

**Programmatic Use**

See Enabled.

**Type** — Type of measurement data to display  
Occupied BW (default) | ACPR

Specify the type of measurement data to display as Occupied BW or ACPR.

**Tunable:** Yes

**Programmatic Use**

See Type.

**Occupied BW (%)** — Percentage of power to compute occupied bandwidth  
99 (default) | positive scalar

Specify the percentage of power over which the Spectrum Analyzer computes the occupied bandwidth as a positive scalar.

**Tunable:** Yes

**Dependency**

To enable this parameter, set **Type** to Occupied BW.

**Programmatic Use**

See PercentOccupiedBW.

**Frequency Options**

**Frequency Span** — Frequency span mode  
Span and Center Frequency (default) | Start and Stop Frequencies

Specify the frequency span mode as one of the following:

- Span and Center Frequency -- Measure over a frequency range specified in **Span (Hz)** and around the frequency value specified in **Center Frequency (Hz)**.
- Start and Stop Frequencies -- Measure over the frequency range [**Start Frequency (Hz)**, **Stop Frequency (Hz)**].

**Tunable:** Yes

**Programmatic Use**

See FrequencySpan.

**Span (Hz)** — Frequency span in Hz

2000 (default) | positive scalar

Specify the frequency span over which the Spectrum Analyzer computes the channel measurements as a positive scalar in Hz.

**Tunable:** Yes

**Dependency**

To enable this parameter, set **Frequency Span** to Span and Center Frequency.

**Programmatic Use**

See Span.

**Center Frequency (Hz)** — Center frequency of span in Hz

0 (default) | real scalar

Center frequency of the span over which the object computes the channel measurements, specified as a real scalar in Hz.

**Tunable:** Yes

**Dependency**

To enable this parameter, set **Frequency Span** to Span and Center Frequency.

**Programmatic Use**

See CenterFrequency.

**Start Frequency (Hz)** — Start frequency in Hz

-1000 (default) | real scalar

Specify the start frequency in Hz over which the Spectrum Analyzer computes the channel measurements.

**Tunable:** Yes

**Dependency**

To enable this parameter, set **Frequency Span** to Start and Stop Frequencies.

**Programmatic Use**

See StartFrequency.

**Stop Frequency (Hz)** — Stop frequency in Hz

1000 (default) | real scalar

Specify the stop frequency in Hz over which the Spectrum Analyzer computes the channel measurements.

**Tunable:** Yes

**Dependency**

To enable this parameter, set **Frequency Span** to Start and Stop Frequencies.

**Programmatic Use**

See StopFrequency.

**Adjacent Channels**

**Num Pairs** — Number of adjacent channel pairs

2 (default) | positive integer in range [1, 12]

Specify the number of adjacent channel pairs as a positive integer in the range [1, 12].

**Tunable:** Yes

**Dependency**

To enable this parameter, set **Type** to ACPR.

**Programmatic Use**

See NumOffsets.

**Offsets (Hz)** — ACPR offsets in Hz

[2000 3500] (default) | vector

Specify the frequency of the adjacent channel relative to the center frequency of the main channel as a real vector of length equal to the number of offset pairs specified in **Num Pairs**.

**Tunable:** Yes

**Dependency**

To enable this parameter, set **Type** to ACPR.

**Programmatic Use**

See ACPROffsets.

**Adjacent BW (Hz)** — Adjacent channel bandwidth in Hz

1000 (default) | positive scalar

Specify the adjacent channel bandwidth in Hz as a positive scalar.

**Tunable:** Yes

**Dependency**

To enable this parameter, set **Type** to ACPR.

**Programmatic Use**

See AdjacentBW.

**Filter Shape** — Filter shape

None (default) | RRC | Gaussian

Specify the filter shape for the main and adjacent channels as None, RRC, or Gaussian.

**Tunable:** Yes

**Dependency**

To enable this parameter, set **Type** to ACPR.

**Programmatic Use**

See `FilterShape`.

**Roll-off Factor** — Roll-off factor

0.5 (default) | real scalar in range [0, 1]

Specify the roll-off factor as a real scalar in the range [0, 1].

**Tunable:** Yes

**Dependency**

To enable this parameter, set **Type** to ACPR and **Filter Shape** to RRC.

**Programmatic Use**

See `FilterCoeff`.

**BT Product** — BT product

0.5 (default) | real scalar in range [0, 1]

Specify the BT product as a real scalar in the range [0, 1].

**Tunable:** Yes

**Dependency**

To enable this parameter, set **Type** to ACPR and **Filter Shape** to Gaussian.

**Programmatic Use**

See `FilterCoeff`.

**Property Inspector Only**

**Channel Names** — Input channel names

[] (default) | character vector | string | array of strings or character vectors.

Input channel names, specified as a character vector, string, or array. The names appear in the legend, **Settings**, and **Measurements** panels. If you do not specify the names, the scope labels the channels as Channel 1, Channel 2, etc.

Example: ["A","B"]

**Dependency**

To see channel names, select **Legend** in the **Analyzer** tab.

**Programmatic Use**

**Block Parameter:** ChannelNames

**Type:** cell array of character vectors or string array




**Maximize Axes** — Maximize size of plots

Auto (default) | Off | On

- Auto — If you have not specified **Title** and **Y-Label**, the scope maximizes all plots.
- On — The scope maximizes all plots and hides all values in **Title** and **Y-label**.
- Off — The scope does not maximize plots.



Hover over the Spectrum Analyzer to see the maximize axes button .

**Tunable:** Yes**Programmatic Use****Block Parameter:** MaximizeAxes**Type:** character vector or string scalar**Axes Scaling** — Y-axis scaling mode

OnceAtStop (default) | Manual | Auto | Updates

- OnceAtStop — Scale y-axis after simulation completes.
- Manual — Manually scale y-axis range with the **Scale Y-axis Limits** toolbar button.
- Auto — Scale y-axis range during and after simulation.
- Updates — Scale y-axis after the number of time steps specified in the “Number of Updates” on page 2-0 text box (100 by default). Scaling occurs only once during each run.

**Tunable:** Yes**Programmatic Use****Block Parameter:** AxesScaling**Type:** character vector or string scalar**Number of Updates** — Number of updates before scaling

100 (default) | integer

Set this property to delay auto scaling the y-axis.

**Tunable:** Yes**Dependency**

To enable this property, set “Axes Scaling” on page 2-0 to Updates.

**Programmatic Use****Block Parameter:** AxesScalingNumUpdates**Type:** character vector or string scalar**Values:** scalar**Block Characteristics**

|                   |  |
|-------------------|--|
| <b>Data Types</b> | Boolean   double   enumerated   fixed point   integer   single |
|-------------------|--|

|                                 |     |
|---------------------------------|-----|
| <b>Direct Feedthrough</b>       | no  |
| <b>Multidimensional Signals</b> | yes |
| <b>Variable-Size Signals</b>    | yes |
| <b>Zero-Crossing Detection</b>  | no  |

## More About

### Cursor Measurements

Measure signal values using vertical waveform cursors that track along the signal.

When you click the **Data Cursors** button in the **Measurements** tab of the Spectrum Analyzer, the spectrum display shows vertical cursors on each signal. Each cursor tracks a vertical line along the signal. The scope displays the difference between  $x$ - and  $y$ -values of the signal at the two cursors in the box between the cursors.

To enable cursor measurements, click the **Data Cursors** button in the **Measurements** tab. The cursors appear only when the Spectrum Analyzer has at least one signal in its display.

You can use the mouse to move the vertical cursors left and right.

In the **Measurements** tab, click the **Data Cursors** drop-down arrow to select one of these options:

- **Snap to Data** — To position the cursors on the signal data points.
- **Lock Cursor Spacing** — To lock the frequency difference between the two cursors.

For modifying the cursor measurements programmatically, see the `CursorMeasurementsConfiguration` object.

### Peak Finder Measurements

Compute and display peak values in the scope display.

When you click on the **Peak Finder** button in the **Measurements** tab of the Spectrum Analyzer, an arrow appears on the plot at each maxima and a **Peaks** panel appears at the bottom of the scope window. The Spectrum Analyzer computes peaks from the portion of the input signal that is currently on display in the scope, and the **Peaks** panel shows the peak values and the frequencies at which they occur.

The **Peaks** section in the **Measurements** tab allows you to specify the number of peaks you want the scope to display, the minimum height above which you want the scope to detect peaks, the minimum distance between peaks, and label the peaks.

The Spectrum Analyzer algorithm defines a peak as a local maximum with lower values present on either side of the peak. It does not consider end points as peaks. For more information on the algorithm, see the `findpeaks` function.

The peaks are valid for any units of the input signal. The letter after the value associated with each measurement indicates the abbreviation for the appropriate International System of Units (SI) prefix, such as *m* for *milli*-. For example, if the input signal is measured in volts, an *m* next to a measurement value indicates that this value is in units of millivolts.

For modifying the peak finder measurements programmatically, see the `PeakFinderConfiguration` object. For more information on these settings in the UI, see “Peaks” on page 2-0 .

## Distortion Measurements

Measure harmonic distortion and intermodulation distortion.

When you click the **Distortion** button in the **Distortion** section of the **Measurements** tab, a distortion panel opens at the bottom of the Spectrum Analyzer window. This panel shows the harmonic and distortion measurement values for the input signal currently on display in the scope. The **Distortion** section in the **Measurements** tab allows you to specify the distortion type, number of harmonics, and even label the harmonics.

---

**Note** For an accurate measurement, ensure that the fundamental signal (for harmonics) or primary tones (for intermodulation) is larger than any spurious or harmonic content. To do so, you may need to adjust the resolution bandwidth (RBW) of the Spectrum Analyzer. Make sure that the bandwidth is low enough to isolate the signal and harmonics from spurious noise content. In general, you should set the RBW value such that there is at least a 10 dB separation between the peaks of the sinusoids and the noise floor. You also might need to select a different spectral window to obtain a valid measurement.

---

You can set the **Distortion Type** parameter to one of these values:

- **Harmonic** -- Select **Harmonic** if your input is a single sinusoid.
- **Intermodulation** -- Select **Intermodulation** if your input is two equal-amplitude sinusoids. Intermodulation can help you determine distortion when the scope uses only a small portion of the available bandwidth.

See “Distortion Measurements” on page 2-1341 for information on how distortion measurements are calculated.

### Harmonic Distortion

When you set the **Distortion Type** to **Harmonic**, these fields appear in the **Harmonic Distortion** panel at the bottom of the Spectrum Analyzer window.

- **H1** — Fundamental frequency in Hz and its power in decibels of the measured power referenced to 1 milliwatt (dBm).
- **H2, H3, ...** — Harmonics frequencies in Hz and their power in decibels relative to the carrier (dBc). If the harmonics are at the same level or exceed the fundamental frequency, reduce the input power.
- **THD** — Total harmonic distortion. This value represents the ratio of the power in the harmonics  $D$  to the power in the fundamental frequency  $S$ . If the noise power is too high in relation to the harmonics, the THD value is not accurate. In this case, lower the resolution bandwidth or select a different spectral window.

$$THD = 10 \cdot \log_{10}(D/S)$$

- **SNR** — Signal-to-noise ratio (SNR). This value represents the ratio of the power in the fundamental frequency  $S$  to the power of all nonharmonic content  $N$ , including spurious signals, in decibels relative to the carrier (dBc).

$$SNR = 10 \cdot \log_{10}(S/N)$$

If you see — as the reported SNR, the total nonharmonic content of your signal is less than 30% of the total signal.

- **SINAD** — Signal-to-noise-and-distortion ratio. This value represents the ratio of the power in the fundamental frequency  $S$  to all other content (including noise  $N$  and harmonic distortion  $D$ ) in decibels relative to the carrier (dBc).

$$SINAD = 10 \cdot \log_{10}\left(\frac{S}{N + D}\right)$$

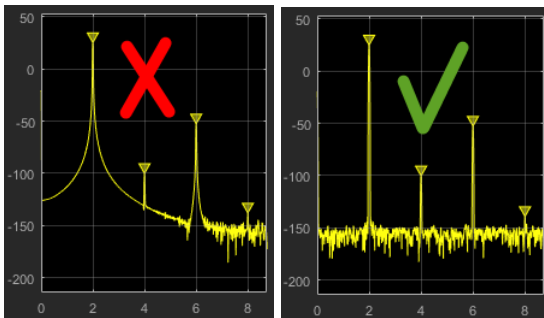
- **SFDR** — Spurious-free dynamic range (SFDR). This value represents the ratio of the power in the fundamental frequency  $S$  to power of the largest spurious signal  $R$  regardless of where it falls in the frequency spectrum. The worst spurious signal might or might not be a harmonic of the original signal. SFDR represents the smallest value of a signal that can be distinguished from a large interfering signal. SFDR includes harmonics.

$$SNR = 10 \cdot \log_{10}(S/R)$$

The harmonic distortion measurement automatically locates the largest sinusoidal component (fundamental signal frequency). It then computes the harmonic frequencies and power in each harmonic in your signal and ignores any DC component. The measurement does not include any harmonics that are outside the Spectrum Analyzer frequency span. Adjust your frequency span so that it includes all the desired harmonics.

**Note** To view the best harmonics, make sure that your fundamental frequency is set high enough to resolve the harmonics. However, this frequency should not be so high that aliasing occurs. For the best display of harmonic distortion, your plot should not show skirts, which indicate frequency leakage. The noise floor should be visible.

For a better display, try a Kaiser window with a large sidelobe attenuation (e.g. between 100–300 db).



### Intermodulation Distortion

When you set the **Distortion Type** to **Intermodulation**, the following fields appear in the **Intermodulation Distortion** panel at the bottom of the Spectrum Analyzer window.

- **F1** — Lower fundamental first-order frequency.
- **F2** — Upper fundamental first-order frequency.
- **2F1 - F2** — Lower intermodulation product from third-order harmonics.

- **2F2 - F1** — Upper intermodulation product from third-order harmonics.
- **TOI** — Third-order intercept point. If the noise power is too high in relation to the harmonics, the TOI value will not be accurate. In this case, you should lower the resolution bandwidth or select a different spectral window. If the TOI has the same amplitude as the input two-tone signal, reduce the power of that input signal.

The intermodulation distortion measurement automatically locates the fundamental and the first-order frequencies ( $F1$  and  $F2$ ). It then computes the frequencies of the third-order intermodulation products ( $2F1 - F2$  and  $2F2 - F1$ ).

For modifying the distortion measurements programmatically, see the `DistortionMeasurementsConfiguration` object. For more information on these settings in the UI, see “Distortion” on page 2-0 .

## Channel Measurements

Measure the occupied bandwidth or adjacent channel power ratio (ACPR).

When you click the **Channel Measurements** button in the **Channel Measurements** tab, a channel measurements panel opens at the bottom of the Spectrum Analyzer window. This panel displays the occupied bandwidth or the adjacent channel power ratio measurements. In the **Channel Measurements** tab, you can specify the occupied bandwidth or the ACPR settings, frequency span, center frequency, and start and stop frequencies.

You can select the channel measurements **Type** to:

- **Occupied BW** -- Occupied bandwidth
- **ACPR** -- Ratio of the power of the main channel to the power of the adjacent channel

For more details on how the Spectrum Analyzer calculates the occupied bandwidth, see “Occupied BW” on page 2-1341.

## Occupied Bandwidth

When you set the **Type** of channel measurement to compute and display to **Occupied BW**, these fields appear in the measurements panel at the bottom of the scope window.

- **Channel Power** — Total power in the channel
- **Occupied BW** — Bandwidth containing the specified **Occupied BW (%)** of the total power of the spectrum.
- **Frequency Error** — Difference between the center of the occupied band and the center frequency (**Center Frequency (Hz)**) of the channel

## ACPR

When you set the **Type** of channel measurement to compute and display to ACPR, these fields appear in the measurements panel at the bottom of the scope window.

- **Lower (Rel Power (dBc))** — Ratio of the power of the lower sideband to the power of the main channel
- **Upper (Rel Power (dBc))** — Ratio of the power of the upper sideband to the power of the main channel

To modify the channel measurements programmatically, see the `ChannelMeasurementsConfiguration` object. For more information on these settings in the UI, see “Channel Measurements” on page 2-0 .

## Spectral Mask

Visualize spectrum limits and compare spectrum values to specification values.

Add upper and lower masks to the Spectrum Analyzer to visualize spectrum limits and compare spectrum values to specification values. To enable the **Spectral Mask** tab, select **Spectrum** in the **Analyzer** tab. When you click the **Upper Mask** and **Lower Mask** buttons in the **Spectral Mask** tab, a **Spectral Mask** panel opens at the bottom of the Spectrum Analyzer window. This panel provides information on pass-fail statistics of masks, names of masks currently failing or passing, and names of channels causing the failure.

You can modify the mask settings in the **Spectral Mask** tab. For more information on these settings in the UI, see Spectral Mask on page 2-1322. For modifying the channel measurements programmatically, see the `SpectralMaskConfiguration` object.


### Check Spectral Masks

You can check the status of the spectral mask using the `getSpectralMaskStatus` function. This function gives details on the number of times a mask succeeded or failed, names of channels causing mask failure, and so on.

You can even use the `MaskTestFailed` event to perform an action every time the mask fails. To trigger a function when the mask fails, create a listener to the `MaskTestFailed` event and define a callback function to trigger it. For more details about using events, see “Events”.

### Customize Visualization

Set configuration and style settings in the Spectrum Analyzer.

To control the settings of the display and labels, color and styling, click on **Settings** () in the **Analyzer** tab of the Spectrum Analyzer toolbar.


In the dialog box that opens, you can customize the font size, plot type, y-axis properties of the spectrum plot, and color map properties of the spectrogram plot. You can change the color of the spectrum plot, background, axes, and labels and also change the line properties.




When you view the spectrum or the spectrogram, you see only the relevant options. For more details about these options, see Configuration > Spectrum Settings on page 2-0 .

### Display Controls

Zoom and pan axes using display controls.

To scale the plot axes, use the mouse to pan around the axes and the scroll button on your mouse to zoom in and out of the plot. Additionally, you can use the buttons that appear when you hover over the plot window.

-  — Maximize the axes, hide all labels and inset the axes values.

-  — Zoom in on the plot.
-  — Pan the plot.
-  — Autoscale the axes to fit the shown data.

## Algorithms

### Spectrum Estimation — Filter Bank

When you choose the `Filter Bank` method, the Spectrum Analyzer uses an analysis filter bank to estimate the power spectrum.

The filter bank splits the broadband input signal  $x(n)$ , of sample rate  $fs$ , into multiple narrow band signals  $y_0(m)$ ,  $y_1(m)$ , ...,  $y_{M-1}(m)$ , of sample rate  $fs/M$ .

The variable  $M$  represents the number of frequency bands in the filter bank. In the Spectrum Analyzer,  $M$  is equal to the number of data points needed to achieve the specified RBW value or 1024, whichever is larger. For more information on the analysis filter bank and its implementation, see the “More About” on page 4-198 and the “Algorithm” on page 4-200 sections in the `dsp.Channelizer` object.

After the Spectrum Analyzer splits the broadband input signal into multiple narrow bands, it computes the power in each narrow frequency band using the following equation. Each  $Z_i$  value is the power estimate over that narrow frequency band.

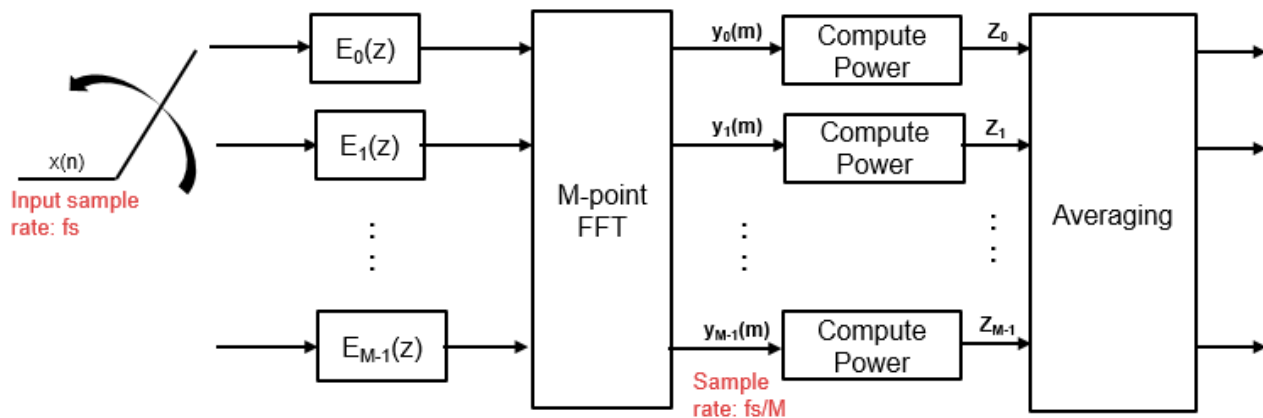
$$Z_i = \frac{1}{L} \sum_{m=0}^{L-1} |y_i[m]|^2$$

$L$  is length of the narrowband signal  $y_i(m)$  and  $i = 1, 2, \dots, M-1$ .

The power values in all the narrow frequency bands (denoted by  $Z_i$ ) form the  $Z$  vector.

$$Z = [Z_0, Z_1, Z_2, \dots, Z_{M-1}]$$

The Spectrum Analyzer averages the current  $Z$  vector with the previous  $Z$  vectors using one of the two moving average methods: video bandwidth or exponential weighting. The output of the averaging operation forms the spectral estimate vector. For details on the two averaging methods, see “Averaging Method” on page 2-1343.



The Spectrum Analyzer uses the value you specify in the **RBW (Hz)** parameter to determine the input frame length. To view the **RBW (Hz)** parameter in the scope, click the **Analyzer** tab on the Spectrum Analyzer toolstrip and navigate to the **Bandwidth** section.

Spectrum Analyzer requires a minimum number of samples to compute a spectral estimate. This value is directly related to the resolution bandwidth property **RBW (Hz)**.

When you set **RBW (Hz)** to:

- **Auto** -- The Spectrum Analyzer requires 1024 samples to update the display. The Spectrum Analyzer determines the appropriate resolution bandwidth to ensure that there are 1024 RBW intervals over the specified frequency span. When you set **RBW (Hz)** to Auto, the Spectrum Analyzer calculates RBW using this equation.

$$RBW_{auto} = \frac{span}{1024}$$

- **scalar value** -- The Spectrum Analyzer calculates the number of samples  $N_{samples}$  using this equation.

$$N_{samples} = \frac{F_s}{RBW}$$

$F_s$  is the sample rate of the input signal as specified in the **Sample Rate (Hz)** property. To view the **Sample Rate (Hz)** in the scope, click the **Analyzer** tab on the Spectrum Analyzer toolstrip and navigate to the **Bandwidth** section.

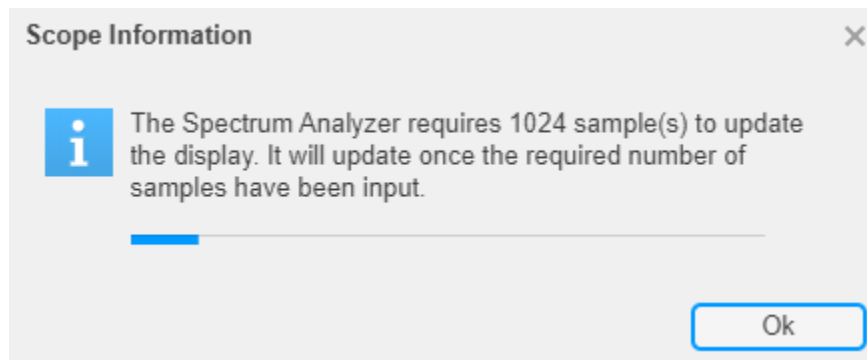
When you specify a resolution bandwidth using the **RBW (Hz)** parameter, you must specify a value such that there are at least two RBW intervals over the specified frequency span. The ratio of the overall span to RBW must be greater than two.

$$\frac{span}{RBW} > 2$$

$span$  is the frequency span over which the Spectrum Analyzer computes and plots the spectrum. To view the **Span (Hz)** in the scope, click the **Estimation** tab on the Spectrum Analyzer toolstrip and navigate to the **Frequency Options** section. To enable this property, set **Frequency Span** to Span and Center Frequency.



When the number of input samples is not sufficient to achieve the specified resolution bandwidth, the Spectrum Analyzer displays a message similar to this one.



The Spectrum Analyzer removes this message and displays a spectral estimate once you provide enough input samples.

### Spectrum Estimation – Welch's Method

When you select the Welch method, the power spectrum estimate is the averaged modified periodograms.

The algorithm in the Spectrum Analyzer consists of these steps:

- 1 The block buffers the input into  $N$ -point data segments. Each data segment is split into  $P$  overlapping data segments, each of length  $M$ , overlapping by  $D$  points. The data segments can be represented as:

$$x_i(n) = x(n + iD), \quad n = 0, 1, \dots, M - 1$$

$$i = 0, 1, \dots, P - 1$$

- If  $D = M/2$ , the overlap is 50%.
- If  $D = 0$ , the overlap is 0%.

- 2 Apply a window to each of the  $P$  overlapping data segments in the time domain.

The Spectrum Analyzer uses **RBW (Hz)** in the **Analyzer** tab to determine the data window length  $N_{window}$  internally. Then, it partitions the input signal into a number of windowed data segments.

Most window functions afford more influence to the data at the center of the set than to the data at the edges, which represents a loss of information. To mitigate that loss, the individual data sets are commonly overlapped in time. For each windowed segment, compute the periodogram by computing the discrete Fourier transform. Then compute the squared magnitude of the result and divide the result by  $M$ .

$$P_{xx}^i(f) = \frac{1}{MU} \left| \sum_{n=0}^{M-1} x_i(n)w(n)e^{-j2\pi fn} \right|^2, \quad i = 0, 1, \dots, P - 1$$

where  $U$  is the normalization factor for the power in the window function and is given by

$$U = \frac{1}{M} \sum_{n=0}^{M-1} w^2(n)$$

You can specify the window using the **Window** parameter in the **Estimation** tab of the Spectrum Analyzer toolstrip.

- 3 The Spectrum Analyzer calculates and plots the power spectrum, power spectrum density, and RMS using the modified Periodogram estimator. For more information about the Periodogram method, see [periodogram](#).

To determine the power spectrum estimate for Welch's method, the Spectrum Analyzer averages the result of the periodograms for the last  $P$  data segments. The averaging reduces the variance, compared to the original  $N$ -point data segment. For more details on the averaging, see "Averaging Method" on page 2-1343.

$$\text{PSD}(f) = \frac{1}{P} \sum_{i=0}^{P-1} P_{xx}^i(f)$$

- 4 The Spectrum Analyzer computes the power spectral density using:

$$\text{PSD}(f) = \frac{1}{P * F_s} \sum_{i=0}^{P-1} P_{xx}^i(f)$$

- 5 The power spectrum is the product of the power spectral density and the resolution bandwidth, as given by this equation.

$$P_{\text{spectrum}}(f) = \text{PSD}(f) \times \text{RBW} = \text{PSD}(f) \times \frac{F_s \times \text{NENBW}}{N_{\text{window}}}$$

- 6 The Spectrum Analyzer plots the power as a spectrogram in the **Spectrogram** mode. Each line of the spectrogram is one periodogram. The time resolution of each line is  $1/\text{RBW}$ , which is the minimum attainable resolution. Achieving the resolution you want might require combining several periodograms. You then use interpolation to calculate noninteger values of  $1/\text{RBW}$ . In the spectrogram display, time scrolls from top to bottom, so the most recent data appears at the top of the display. The offset shows the time value at which the center of the most current spectrogram line occurred.

The Spectrum Analyzer requires a minimum number of samples to compute a spectral estimate. This value is directly related to the resolution bandwidth (RBW).

$$N_{\text{samples}} = \frac{\left(1 - \frac{O_p}{100}\right) \times \text{NENBW} \times F_s}{\text{RBW}}$$

where  $O_p$  on page 2-1338 is the overlap percentage, NENBW on page 2-1339 is the normalized effective noise bandwidth,  $F_s$  on page 2-1339 is the input sample rate, and RBW on page 2-1339 is the resolution bandwidth.

The Spectrum Analyzer shows the number of samples per update in the Spectrum Analyzer status bar.

You can enable **Samples/Update** in the status bar only when you set **Input Domain** to Time and **Method** to Welch in the **Estimation** tab on the Spectrum Analyzer toolstrip.

### Overlap Percentage ( $O_p$ )

The overlap percentage  $O_p$  is the value you specify in the **Overlap %** property. To view the **Overlap %** in the scope, click the **Estimation** tab on the Spectrum Analyzer toolstrip and navigate to the **Window Options** section.

When you increase the overlap percentage, the Spectrum Analyzer needs fewer new input samples to compute a new spectral update.

| $O_p$ | $N_{samples}$ |
|-------|---------------|
| 0%    | 100           |
| 50%   | 50            |
| 80%   | 20            |

### Normalized Effective Noise Bandwidth (NENBW)

The normalized effective noise bandwidth  $NENBW$  is a window parameter that measures the noise performance of the window.  $NENBW$  is determined using the window length and the window coefficients, and is given by the following equation:

$$NENBW = N_{window} \times \frac{\sum_{n=1}^{N_{window}} w^2(n)}{\left[ \sum_{n=1}^{N_{window}} w(n) \right]^2}$$

$w(n)$  denotes the vector of window coefficients (calculated internally).  $N_{window}$  is the window length the Spectrum Analyzer needs to compute one spectral update, and is directly related to the resolution bandwidth and normalized effective noise bandwidth.

$$N_{window} = \frac{NENBW \times F_s}{RBW}$$

The rectangular window has the smallest  $NENBW$ , with a value of 1. All other windows have a larger  $NENBW$  value. For example, the Hann window has an  $NENBW$  value of approximately 1.5.

The Spectrum Analyzer shows the value of  $NENBW$  in the Spectrum Analyzer status bar.

You can enable  $NENBW$  only when you set **Input Domain** to Time and **Method** to Welch in the **Estimation** tab on the Spectrum Analyzer toolstrip.

### Input Sample Rate ( $F_s$ )

$F_s$  is the sample rate of the input signal. To view the **Sample Rate (Hz)** in the scope, click the **Analyzer** tab on the Spectrum Analyzer toolstrip and navigate to the **Bandwidth** section. You can enable this property in the status bar at the bottom of the Spectrum Analyzer window. Right-click the status bar and select **Sample Rate**.

### Resolution Bandwidth (RBW)

Resolution bandwidth controls the spectral resolution of the displayed signal. The RBW value determines the spacing between frequencies that the scope can resolve. A smaller value gives a higher spectral resolution and lowers the noise floor, that is, the Spectrum Analyzer can resolve frequencies that are closer to each other. However, this comes at the cost of a longer sweep time.

You can set the resolution bandwidth through the **RBW (Hz)** property. To view **RBW (Hz)** in the scope, click the **Analyzer** tab on the Spectrum Analyzer toolstrip and navigate to the **Bandwidth** section.

When you set **RBW (Hz)** to:

- **Auto** -- The Spectrum Analyzer requires 1024 samples to update the display. The Spectrum Analyzer determines the appropriate resolution bandwidth to ensure that there are 1024 RBW intervals over the specified frequency span. When you set **RBW (Hz)** to **Auto**, the Spectrum Analyzer calculates using this equation.

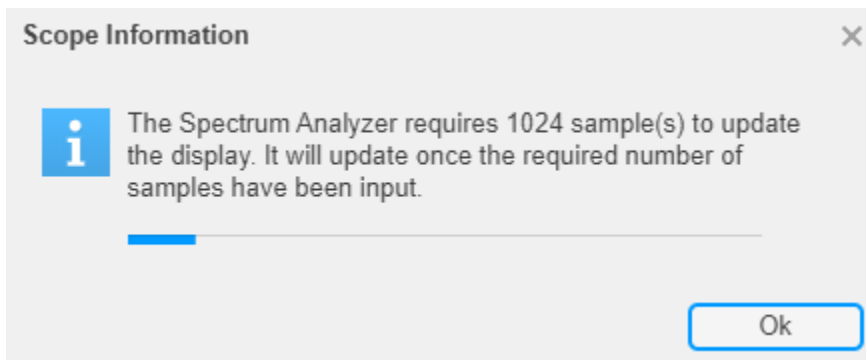
$$RBW_{auto} = \frac{span}{1024}$$

- **scalar value** -- Specify a value such that there are at least two RBW intervals over the specified frequency span. The ratio of the overall span to RBW must be greater than two:

$$\frac{span}{RBW} > 2$$

*span* is the frequency span over which the Spectrum Analyzer computes and plots the spectrum. Spectrum Analyzer shows the span through the **Span (Hz)** property. To view the **Span (Hz)** in the scope, click the **Estimation** tab on the Spectrum Analyzer toolstrip, navigate to the **Frequency Options** section, and set **Frequency Span** to **Span** and **Center Frequency**.

When the number of input samples is not sufficient to achieve the specified resolution bandwidth, the Spectrum Analyzer displays a message similar to this one.



The Spectrum Analyzer removes this message and displays a spectral estimate once you provide enough input samples.

You can enable this property in the status bar at the bottom of the Spectrum Analyzer window. Right-click the status bar and select **RBW**.

### Nyquist Frequency Interval

When you plot the two-sided spectrum by selecting **Two-Sided Spectrum** in the **Spectrum** or **Spectrogram** tab, the Nyquist frequency interval is

$$\left[ -\frac{SampleRate}{2}, \frac{SampleRate}{2} \right] + FrequencyOffset \text{ Hz.}$$

When you clear the **Two-Sided Spectrum**, the Nyquist frequency interval is

$$\left[ 0, \frac{SampleRate}{2} \right] + FrequencyOffset \text{ Hz.}$$

## Frequency Vector

When you set **Frequency (Hz)** to Auto, the software calculates the frequency vector for the frequency-domain input.

When you plot the two-sided spectrum by selecting **Two-Sided Spectrum** in the **Spectrum** or **Spectrogram** tab, the frequency vector is:

$$\left[ -\frac{\text{SampleRate}}{2}, \frac{\text{SampleRate}}{2} \right]$$

When you clear the **Two-Sided Spectrum**, the frequency vector is:

$$\left[ 0, \frac{\text{SampleRate}}{2} \right]$$

## Occupied BW

The Spectrum Analyzer calculates *Occupied BW* using these steps.

- 1 Calculate the total power in the measured frequency range.
- 2 Determine the lower frequency value. Starting at the lowest frequency in the range and moving upward, sum the power distributed in each frequency until the result is

$$\frac{100 - \text{OccupiedBW}\%}{2}$$

of the total power.

- 3 Determine the upper frequency value. Starting at the highest frequency in the range and moving downward, sum the power distributed in each frequency until the result reaches

$$\frac{100 - \text{OccupiedBW}\%}{2}$$

of the total power.

- 4 The bandwidth between the lower and upper power frequency values is the occupied bandwidth.
- 5 The frequency halfway between the lower and upper frequency values is the center frequency.

## Distortion Measurements

The Spectrum Analyzer calculates Distortion Measurements using these steps.

- 1 Estimate spectral content by finding peaks in the spectrum. When the algorithm detects a peak, it records the width of the peak and clears all monotonically decreasing values by treating all these values as if they belong to the peak. Using this method, the algorithm removes all spectral content centered at DC (0 Hz) from the spectrum and records the amount of bandwidth cleared ( $W_0$ ).
- 2 Determine the fundamental power ( $P_1$ ) from the remaining maximum value of the displayed spectrum. Create a local estimate ( $Fe_1$ ) of the fundamental frequency by computing the central moment of the power near the peak. Record the bandwidth of the fundamental power content ( $W_1$ ). Then remove the power from the fundamental as in step 1.
- 3 Determine the power and width of the higher-order harmonics ( $P_2, W_2, P_3, W_3$ , etc.) in succession by examining the frequencies closest to the appropriate multiple of the local estimate ( $Fe_1$ ).

Remove any spectral content that decreases monotonically about the harmonic frequency from the spectrum before proceeding to the next harmonic.

- 4 After removing the DC, fundamental, and harmonic content from the spectrum, examine the power of the remaining spectrum for its sum ( $P_{remaining}$ ), peak value ( $P_{maxspur}$ ), and median value ( $P_{estnoise}$ ).
- 5 Compute the sum of all the removed bandwidth as  $W_{sum} = W_0 + W_1 + W_2 + \dots + W_n$ .

Compute the sum of powers of the second and higher-order harmonics as  $P_{harmonic} = P_2 + P_3 + P_4 + \dots + P_n$ .

- 6 Estimate the sum of the noise power as:

$$P_{noise} = (P_{remaining} \cdot dF + P_{est.noise} \cdot W_{sum})/RBW$$

Where  $dF$  is the absolute difference between frequency bins, and  $RBW$  is the resolution bandwidth of the window.

- 7 Then compute the metrics for SNR, THD, SINAD, and SFDR from the estimates.

$$THD = 10 \cdot \log_{10} \left( \frac{P_{harmonic}}{P_1} \right)$$

$$SINAD = 10 \cdot \log_{10} \left( \frac{P_1}{P_{harmonic} + P_{noise}} \right)$$

$$SNR = 10 \cdot \log_{10} \left( \frac{P_1}{P_{noise}} \right)$$

$$SFDR = 10 \cdot \log_{10} \left( \frac{P_1}{\max(P_{maxspur}, \max(P_2, P_3, \dots, P_n))} \right)$$

## Harmonic Measurements

- 1 The harmonic distortion measurements use the spectrum trace shown in the display as the input to the measurements. The default Hann window setting of the Spectrum Analyzer might exhibit leakage that can completely mask the noise floor of the measured signal.

The harmonic measurements attempt to correct for leakage by ignoring all frequency content that decreases monotonically away from the maximum of harmonic peaks. If the window leakage covers more than 70% of the frequency bandwidth in your spectrum, you may see a blank reading (-) reported for **SNR** and **SINAD**. If your application can tolerate the increased equivalent noise bandwidth (ENBW), consider using a Kaiser window with a high attenuation (up to 330 dB) to minimize spectral leakage.

- 2 Ignore the DC component.
- 3 After windowing, the width of each harmonic component masks the noise power in the neighborhood of the fundamental frequency and harmonics. To estimate the noise power in each region, the Spectrum Analyzer computes the median noise level in the nonharmonic areas of the spectrum. It then extrapolates that value into each region.
- 4  $N^{\text{th}}$  order intermodulation products occur at  $A \cdot F1 + B \cdot F2$ ,

where  $F1$  and  $F2$  are the sinusoid input frequencies and  $|A| + |B| = N$ .  $A$  and  $B$  are integer values.

- 5 For intermodulation measurements, compute the third-order intercept (TOI) point as follows.

- $TOI_{lower} = P_{F1} + (P_{F2} - P_{(2F1-F2)})/2$
- $TOI_{upper} = P_{F2} + (P_{F1} - P_{(2F2-F1)})/2$
- $TOI = (TOI_{lower} + TOI_{upper})/2$

Where  $P$  is power in decibels of the measured power referenced to 1 milliwatt (dBm).

### Averaging Method

The Spectrum Analyzer can calculate the moving average using two methods:

- Video bandwidth — The Spectrum Analyzer uses a time-domain lowpass filter to smooth the noise in the signal. The video bandwidth (VBW) filter smooths the trace and decreases noise, and the Spectrum Analyzer applies the filter to the data before displaying it.

Video bandwidth is the bandwidth of the lowpass filter that Spectrum Analyzer uses to average or smooth the noise in the signal before displaying it in the scope. The Spectrum Analyzer computes the video bandwidth using this equation:

$$VBW = \frac{(1 - \lambda)RBW}{2\pi\lambda NENBW}$$

where,

- $\lambda$  is the forgetting factor.
- RBW on page 2-1339 is the resolution bandwidth.
- NENBW on page 2-1339 is the normalized effective noise bandwidth.

Video bandwidth does not affect the level of the noise (noise floor), but only increases the signal-to-noise ratio and smooths the trace of the noise. When you decrease the value of VBW, the signal-to-noise ratio improves.

The cutoff frequency of the video bandwidth filter is given by:

$$\omega_c = \frac{2\pi VBW}{F_s/NFFT}$$

where  $F_s$  is the input sample rate and NFFT is the number of FFT points.

The Spectrum Analyzer shows the values of sample rate, VBW, and NFFT in the status bar at the bottom of the display. To enable, right-click the status bar and select **Sample Rate**, **VBW**, and **NFFT**.

- Exponential — The moving average algorithm uses the exponential weighting method to update the weights and compute the moving average recursively for each  $Z$  vector that comes in by using the following recursive equations:

$$w_N = \lambda w_{N-1} + 1$$

$$\bar{z}_N = \left(1 - \frac{1}{w_N}\right)\bar{z}_{N-1} + \left(\frac{1}{w_N}\right)z_N$$

- $\lambda$  — Forgetting factor
- $w_N$  — Weighting factor applied to the current  $Z$  vector
- $z_N$  — Current  $Z$  vector

- $\bar{z}_{N-1}$  — Moving average until the previous  $Z$  vector
- $\left(1 - \frac{1}{w_N}\right)\bar{z}_{N-1}$  — Effect of the previous  $Z$  vectors on the average
- $\bar{z}_N$  — Moving average including the current  $Z$  vector

## Version History

Introduced in R2014b

### R2023a: Spectrum Analyzer with improved responsiveness and toolstrip interface in Simulink

In R2023a, the Spectrum Analyzer block is more responsive and its toolstrip interface is improved to provide you easy access to spectral analysis, estimation, and measurements. You can configure and display Spectrum Analyzer settings from the command line with the `SpectrumAnalyzerConfiguration` object.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block can be used for simulation visibility in systems that generate code, but is not included in the generated code.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

This block can be used for simulation visibility in subsystems that generate HDL code, but is not included in the hardware implementation.

### PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

This block can be used for simulation visibility in systems that generate PLC code, but is not included in the generated code.

### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

This block accepts fixed-point input, but converts it to `double` for display.

## See Also

### Objects

`SpectrumAnalyzerConfiguration` | `spectrumAnalyzer`

### Functions

`getSpectralMaskStatus` | `getSpectrumData` | `getMeasurementsData`



**Blocks**

Time Scope | Array Plot | Filter Visualizer

**Topics**

“Configure Spectrum Analyzer”

“Spectral Analysis”

“Estimate the Power Spectrum in Simulink”

“View the Spectrogram Using Spectrum Analyzer”

“Display Frequency-Domain Data in Spectrum Analyzer”

# Spectrum Estimator

Estimate power spectrum or power-density spectrum



## Library

Estimation / Power Spectrum Estimation

dspspect3

## Description

The Spectrum Estimator block outputs the power spectrum or power-density spectrum of a real or complex input signal, using the Welch method of averaged modified periodograms and the filter bank approach.

When you choose the filter bank approach, the block uses an analysis filter bank to estimate the power spectrum. The filter bank approach produces a spectral estimate with a higher resolution, a more accurate noise floor, and more precise peaks than the Welch method, with low or no spectral leakage. They come at the expense of increased computation and slower tracking.

When you choose the Welch method, the block computes the averaged modified periodograms to compute the spectral estimate. The block buffers the input data into overlapping segments. Use the block parameters to set the length of the data segments, the amount of data overlap between consecutive segments, and other features of the power spectrum.

For more information on the Welch method and the filter bank method, see “Algorithms” on page 2-1351.

Each column of the input signal is treated as a separate channel. If the input is a two-dimensional signal, the first dimension represents the channel length (or frame size) and the second dimension represents the number of channels. If the input is a one-dimensional signal, then it is interpreted as a single channel.

## Parameters

### Main Tab

#### Method

Specify the spectral estimation method.

- **Filter bank** (default) — An analysis filter bank splits the broadband input signal into multiple narrow subbands. The block computes the power in each narrow frequency band, and the computed value is the spectral estimate over the respective frequency band.

- **Welch** — The block uses the Welch averaged modified periodograms method to compute the power spectrum over the narrow subbands.

### Number of taps per band

Specify the number of filter coefficients, or taps, for each frequency band. This value corresponds to the number of filter coefficients per polyphase branch. The total number of filter coefficients is equal to **Number of taps per band** times the FFT length.

This parameter applies when you set **Method** to `Filter bank`. The default is 12.

### Spectrum type

Type of spectrum to compute. You can set this parameter to:

- **Power** (default) — Compute the power spectrum.
- **Power density** — Compute the power spectral density.

This parameter is nontunable.

### Frequency resolution method

Frequency resolution method. You can set this parameter to:

- **Auto** (default) — The Spectrum Estimator block computes the resolution bandwidth (RBW) so that the frequency span fits 1024 RBW intervals.
  - **Welch method** — The window length, *winLen*, is calculated using  $winLen = NENBW \times Fs/RBW$ . *NENBW* is the equivalent noise bandwidth of the window and *Fs* is the sample rate.
  - **Filter bank method** — The FFT length is the ceiling of the ratio of **Sample rate (Hz)** to the computed resolution bandwidth.
- **RBW** — Specify the resolution bandwidth, which is used to determine the window length (Welch method) or the FFT length (filter bank method). When the block uses the Welch method, the behavior is equivalent to that of the Spectrum Analyzer block. The window length is calculated using  $winLen = NENBW \times Fs/RBW$ . *NENBW* is the equivalent noise bandwidth of the window and *Fs* is the sample rate. The FFT length is equal to the ceiling of the ratio of **Sample rate (Hz)** to **RBW (Hz)**.
- **Window length** — Specify the window or segment length to use in the Welch algorithm. This option appears when you set **Method** to `Welch`.
- **Number of frequency bands** — Specify the number of polyphase branches of the analysis filter bank. This value corresponds to the FFT length that the filter bank uses. This option appears when you set **Method** to `Filter bank`.

This parameter is nontunable.

### RBW (Hz)

Resolution bandwidth, specified as a positive scalar in Hz. The default is 5. This parameter applies when you set **Frequency resolution method** to `RBW`. The ceiling of the ratio of the frequency span to RBW must be greater than 2.

This parameter is nontunable.

### Number of bands source

Source of the number of frequency bands. This parameter applies when you set **Method** to `Filter bank` and **Frequency resolution method** to `Number of frequency bands`. You can set this parameter to:

- Same as input frame length (default) — The FFT length is set to the frame size of the input.
- Specify on dialog — The FFT length is the value you specify in **Number of bands**.

This parameter is nontunable.

### Number of bands

Number of frequency bands, or the FFT length the filter bank uses to compute the power spectral estimate, specified as a positive scalar. The default is 1024. This parameter applies when you set **Method** to Filter bank, **Frequency resolution method** to Number of frequency bands, and **Number of bands source** to Specify on dialog. This parameter is nontunable.

### Window length source

Source of the window length value. This parameter applies when you set **Method** to Welch and **Frequency resolution method** to Window length. You can set this parameter to:

- Same as input frame length (default) — Window length is set to the frame size of the input. Specify this option to obtain behavior equivalent to that of the Periodogram block.
- Specify on dialog — Window length is the value you specify in the **Window length** parameter.

This parameter is nontunable.

### Window length

Length of the window used to compute the spectrum estimate, specified as a positive integer scalar greater than 2. The default is 1024. This parameter applies when you set **Method** to Welch, **Frequency resolution method** to Window length, and **Window length source** to Specify on dialog. This parameter is nontunable.

### FFT length source

Source of the FFT length value. This parameter applies when you set **Method** to Welch and **Frequency resolution method** to Window length. You can set this parameter to:

- Auto (default) — The block sets the FFT length to the frame size of the input.
- Property — The block sets the FFT length to the value you specify in **FFT length**.

This parameter is nontunable.

### FFT length

Length of the FFT used to compute the spectrum estimates, specified as a positive integer scalar. This parameter applies when you set **Method** to Welch, **Frequency resolution method** to Window length, and **FFT length source** to Property. The default is 1024. This parameter is nontunable.

### Inherit sample rate from input

When you select this check box, the block sample rate is computed as  $N/T_s$ , where  $N$  is the frame size of the input signal and  $T_s$  is the sample time of the input signal.

This check box applies when you do one of the following:

- Set **Method** to Welch and **Frequency resolution method** to Window length.
- Set **Method** to Filter bank and **Frequency resolution method** to Number of frequency bands.

When you clear this check box, the block sample rate is the value you specify in **Sample rate (Hz)**. By default, this check box is selected. This parameter is nontunable.

### Sample rate (Hz)

Sample rate of the input signal, specified as a positive scalar. The default is 44100. This parameter applies when you do one of the following:

- Set **Frequency resolution method** to Auto or RBW.
- Set **Method** to Welch, **Frequency resolution method** to Window length, and clear the **Inherit sample rate from input** check box.
- Set **Method** to Filter bank, **Frequency resolution method** to Number of frequency bands, and clear the **Inherit sample rate from input** check box.

This parameter is nontunable.

### Window function

Window function the Welch algorithm uses, specified as one of Chebyshev | Flat Top | Hamming | Hann | Kaiser | Rectangular. This parameter appears when you set **Method** to Welch. The default is Hann. This parameter is nontunable.

### Sidelobe attenuation of window (dB)

Sidelobe attenuation of the window, specified as a real positive scalar greater than or equal to 45, in dB. The default is 60. This parameter appears when you set **Method** to Welch and **Window function** to Chebyshev or Kaiser. This parameter is nontunable.

### Averaging method

Specify the averaging method as Exponential (default) or Running. In the exponential method, the block computes the average over samples weighted by an exponentially decaying forgetting factor. In the running averaging method, the block computes an equally weighted average of specified number of spectrum estimates defined by **Number of spectral averages** parameter.

### Number of spectral averages

Number of spectral averages, specified as a positive integer scalar. The default is 1. The spectrum estimator computes the current power spectrum estimate by averaging the last  $N$  power spectrum estimates, where  $N$  is the number of spectral averages defined in **Number of spectral averages**. This parameter is nontunable.

This parameter applies when **Averaging method** is set to Running.

### Specify forgetting factor from input port

Select this check box to specify the forgetting factor from an input port. When you do not select this check box, the forgetting factor is specified through the **Forgetting factor** parameter.

This parameter applies when **Averaging method** is set to Exponential.

### Forgetting factor

Specify the exponential weighting forgetting factor as a scalar value greater than zero and smaller than or equal to one. The default is 0.9.

This parameter applies when you set **Averaging method** to Exponential and clear the **Specify forgetting factor from input port** parameter.

**Advanced Tab****Window overlap (%)**

Percentage of overlap between successive data windows, specified as a scalar from 0 and 100. The default value is 0. To enable this parameter, on the **Main Tab**, set **Method** to **Welch**. This parameter is nontunable.

**Reference load (ohms)**

Load used as a reference to compute the power values, specified as a real positive scalar expressed in ohms. The default value is 1. This parameter is nontunable.

**Frequency range**

Frequency range of the spectrum estimator. You can set this parameter to:

- **One-sided** — The spectrum estimator computes the one-sided spectrum of a real input signal. When the FFT length, `NFFT`, is even, the spectrum estimate has length  $(NFFT/2) + 1$  and is computed over the frequency range  $[0 \text{ SampleRate}/2]$ . `SampleRate` is the sample rate of the input signal. When `NFFT` is odd, the spectrum estimate has length  $(NFFT + 1)/2$  and is computed over the frequency range  $[0 \text{ SampleRate}/2]$ .
- **Two-sided** — The spectrum estimator computes the two-sided spectrum of a complex or real input signal. The length of the spectrum estimate is equal to the FFT length. The spectrum estimate is computed over the frequency range  $[0 \text{ SampleRate})$ , where `SampleRate` is the sample rate of the input signal.
- **Centered (default)** — The spectrum estimator computes the centered two-sided spectrum of a complex or real input signal. The length of the spectrum estimate is equal to the FFT length. The spectrum estimate is computed over the frequency range  $(-\text{SampleRate}/2 \text{ SampleRate}/2]$  when the FFT length is even and  $(-\text{SampleRate}/2 \text{ SampleRate}/2)$  when the FFT length is odd.

This parameter is nontunable.

**Power units**

Units used to measure power. You can set this parameter to:

- **'Watts'** (default) — The spectrum estimator measures power in watts.
- **'dBw'** — The spectrum estimator measures power in decibel-watts.
- **'dBm'** — The spectrum estimator measures power in decibel-milliwatts.

This parameter is nontunable.

**Output max-hold spectrum**

When you select this check box, the block computes the max-hold spectrum of the input signal by keeping, at each frequency bin, the maximum value of all the power spectrum estimates. By default, this check box is not selected. This parameter is nontunable.

**Output min-hold spectrum**

When you select this check box, the block computes the min-hold spectrum of the input signal by keeping, at each frequency bin, the minimum value of all the power spectrum estimates. By default, this check box is not selected. This parameter is nontunable.

**Output frequency vector**

When you select this check box, the block outputs the frequency vector. By default, this check box is not selected. This parameter is nontunable.

**Output effective RBW**

When you select this check box, the block computes the effective resolution bandwidth. By default, this check box is not selected. This parameter is nontunable.

**Simulate using**

Type of simulation to run. You can set this parameter to:

- **Code generation** (default) — Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than **Interpreted execution**.
- **Interpreted execution** — Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than **Code generation**.

**Supported Data Types**

| Port   | Supported Data Types   |
|--------|--|
| Input  | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> </ul> |
| Output | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> </ul> |

**Algorithms****Welch's Method of Averaged Modified Periodograms**

When you choose the Welch method, the power spectrum estimate is averaged modified periodograms.

Given the signal input,  $x$ :

- 1 Multiply  $x$  by the window and scale the result by the window power.
- 2 Compute the FFT of the signal,  $Y$ , and take the square magnitude using  $Z = Y \cdot \text{conj}(Y)$ .
- 3 Compute the current power spectrum estimate by taking the moving average of the last  $N$  number of  $Z$ 's, and scaling the answer by the sample rate. For details on the moving average methods, see "Averaging Method" on page 2-1352.

**Filter Bank**

The filter-bank-based spectrum estimator uses an analysis filter bank to estimate the power spectrum. The filter bank splits a broadband input signal,  $x(n)$ , of sample rate  $fs$  into multiple narrow band signals,  $y_0(m), y_1(m), \dots, y_{M-1}(m)$ , of sample rate  $fs/M$ .

The variable  $M$  represents the number of frequency bands in the filter bank. When you specify FFT length,  $M$  equals the FFT length. When you do not specify FFT length,  $M$  is equal to the number of rows in the input signal. The number of taps per frequency band sets the number of filter coefficients for each frequency band of the filter bank. The total number of filter coefficients is equal to number of taps per band times the number of frequency bands  $M$ . For more information on the analysis filter bank and how it is implemented, see the "More About" on page 4-198 and the "Algorithm" on page 4-200 sections in `dsp.Channelizer`.

After the broadband input signal is split into multiple narrow bands, the spectrum estimator computes the power in each narrow band using the following equation. Each  $Z_i$  value becomes the estimate of the power over that narrow frequency band.

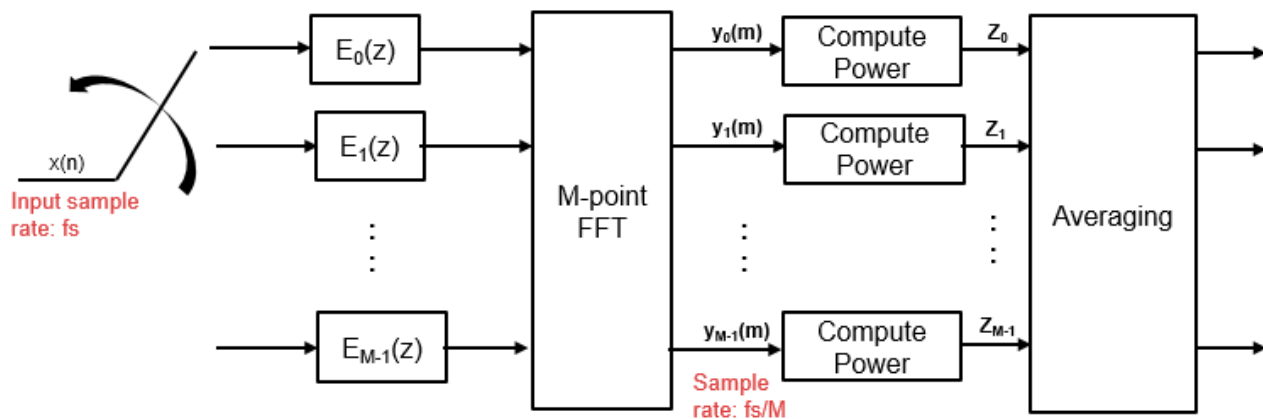
$$Z_i = \frac{1}{L} \sum_{m=0}^{L-1} |y_i[m]|^2$$

$L$  is length of the narrow band signal  $y_i(m)$ , where  $i = 1, 2, \dots, M-1$ .

The power values in all the narrow bands (denoted by  $Z_i$ ) form the  $Z$  vector.

$$Z = [Z_0, Z_1, Z_2, \dots, Z_{M-1}]$$

The filter bank estimator algorithm averages the current  $Z$  vector with the previous  $Z$  vectors using one of the two moving average methods: running or exponential weighting. The output of the averaging operation forms the spectral estimate vector. For details on the two averaging methods, see "Averaging Method" on page 2-1352.

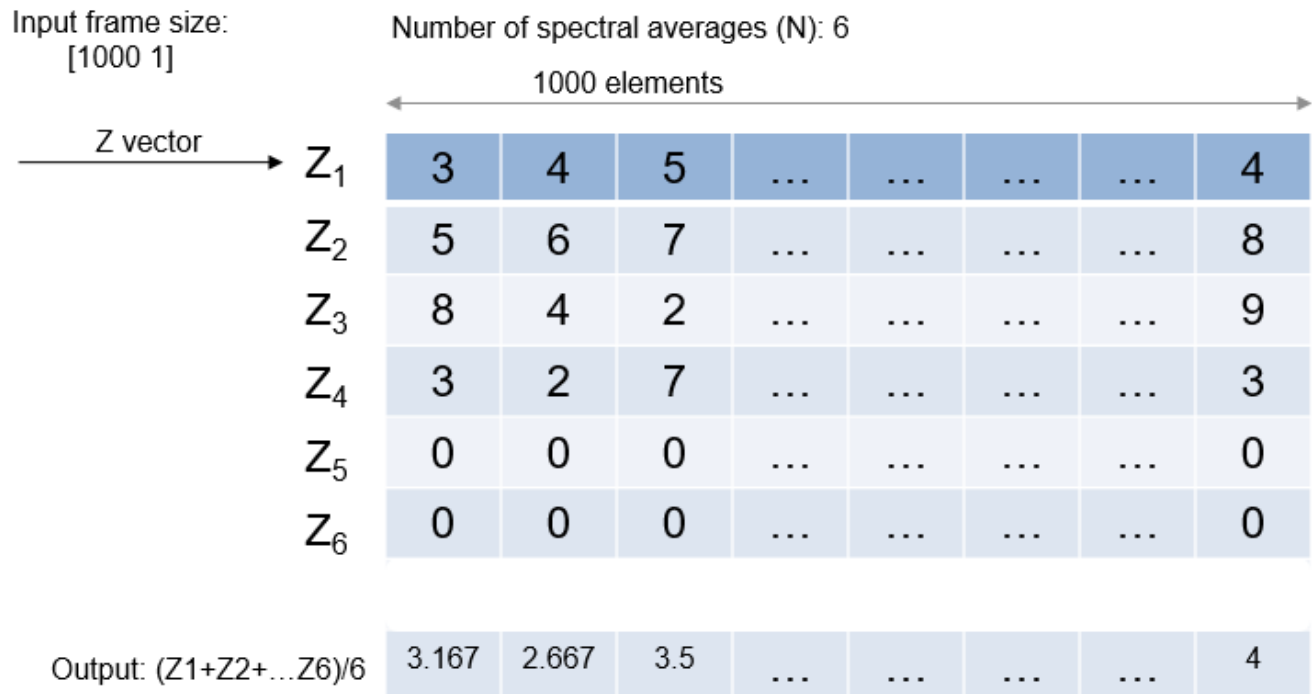


### Averaging Method

The moving average is calculated using one of the two methods:

- **Running** — For each frame of input, average the last  $N$ -scaled  $Z$  vectors, which are computed by the algorithm. The variable  $N$  is the value you specify for the number of spectral averages. If the algorithm does not have enough  $Z$  vectors, the algorithm uses zeros to fill the empty elements.





- **Exponential** — The moving average algorithm using the exponential weighting method updates the weights and computes the moving average recursively for each Z vector that comes in by using the following recursive equations:

$$w_N = \lambda w_{N-1} + 1$$

$$\bar{z}_N = \left(1 - \frac{1}{w_N}\right)\bar{z}_{N-1} + \left(\frac{1}{w_N}\right)z_N$$

- $\lambda$  — Forgetting factor
- $w_N$  — Weighting factor applied to the current Z vector
- $z_N$  — Current Z vector
- $\bar{z}_{N-1}$  — Moving average until the previous Z vector
- $\left(1 - \frac{1}{w_N}\right)\bar{z}_{N-1}$  — Effect of the previous Z vectors on the average
- $\bar{z}_N$  — Moving average including the current Z vector

## Version History

Introduced in R2015b

## References

- [1] Hayes, Monson H. *Statistical Digital Signal Processing and Modeling*. Hoboken, NJ: John Wiley & Sons, 1996.

- [2] Kay, Steven M. *Modern Spectral Estimation: Theory and Application*. Englewood Cliffs, NJ: Prentice Hall, 1999.
- [3] Stoica, Petre, and Randolph L. Moses. *Spectral Analysis of Signals*. Englewood Cliffs, NJ: Prentice Hall, 2005.
- [4] Welch, P. D. "The Use of Fast Fourier Transform for the Estimation of Power Spectra: A Method Based on Time Averaging Over Short, Modified Periodograms." *IEEE Transactions on Audio and Electroacoustics*. Vol. 15, No. 2, June 1967, pp. 70–73.
- [5] *Bluetooth Specification Version 4.2*. Bluetooth SIG. December 2014, p. 217. Specification of the Bluetooth System

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

## See Also

### Objects

`dsp.SpectrumEstimator`

### Blocks

Cross-Spectrum Estimator | Discrete Transfer Function Estimator | Periodogram | Spectrum Analyzer

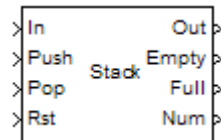
### Topics

"Streaming Power Spectrum Estimation Using Welch's Method"

"Estimate Power Spectral Density of Chirp"

# Stack

Store inputs into LIFO register



## Library

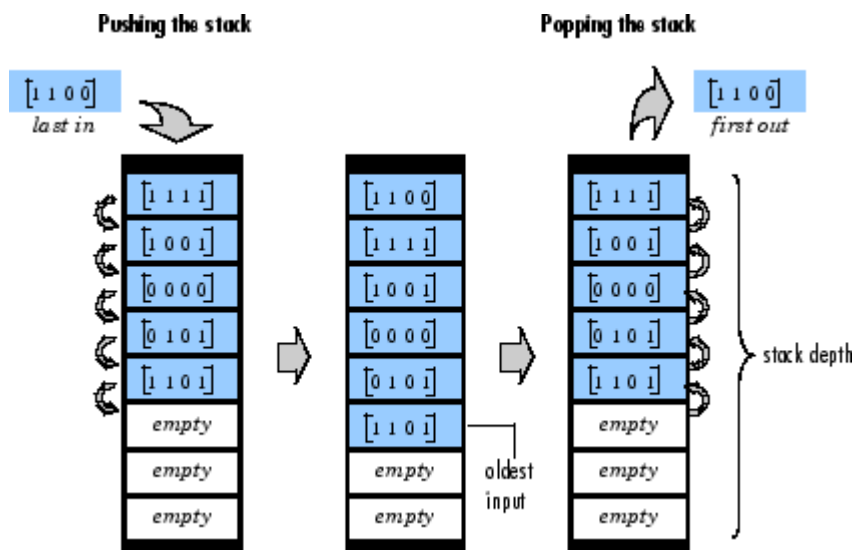
Signal Management / Buffers

dspbuff3

## Description

The Stack block stores a sequence of input samples in a last in, first out (LIFO) register. The register capacity is set by the **Stack depth** parameter, and inputs can be scalars, vectors, or matrices.

The block *pushes* the input at the In port onto the top of the stack when a trigger event is received at the Push port. When a trigger event is received at the Pop port, the block *pops* the top element off the stack and holds the Out port at that value. The last input to be pushed onto the stack is always the first to be popped off.



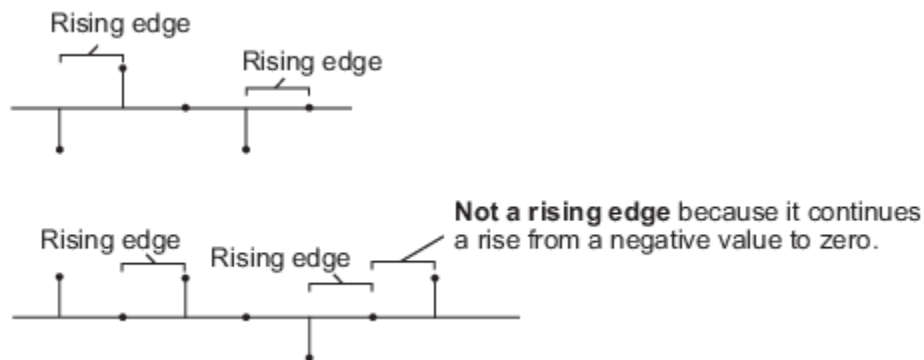
A trigger event at the optional Rst port empties the stack contents. When you select **Clear output port on reset**, then a trigger event at the Rst port empties the stack *and* sets the value at the Out port to zero. This setting also applies when a disabled subsystem containing the Stack block is reenabled; the Out port value is only reset to zero in this case when you select **Clear output port on reset**.

When two or more of the control input ports are triggered at the same time step, the operations are executed in the following order:

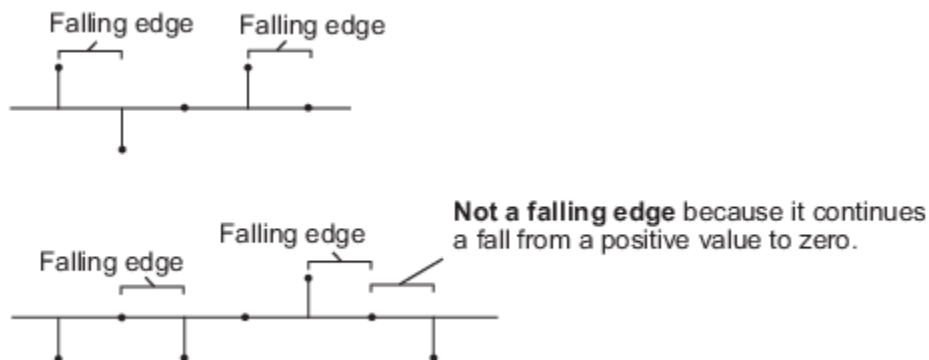
- 1 Rst
- 2 Push
- 3 Pop

The rate of the trigger signal must be the same as the rate of the data signal input. You specify the triggering event for the Push, Pop, and Rst ports in the **Trigger type** pop-up menu:

- **Rising edge** — Triggers execution of the block when the trigger input does one of the following:
  - Rises from a negative value to a positive value or zero
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure)



- **Falling edge** — Triggers execution of the block when the trigger input does one of the following:
  - Falls from a positive value to a negative value or zero
  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero (see the following figure)



- **Either edge** — Triggers execution of the block when the trigger input is a **Rising edge** or **Falling edge** (as described above).
- **Non-zero sample** — Triggers execution of the block at each sample time that the trigger input is not zero.

---

**Note** If your model contains any referenced models that use a Stack block with the **Push full stack** parameter set to `Dynamic reallocation`, you cannot simulate your top-level model in Simulink Accelerator mode.

---

The **Push full stack** parameter specifies the block's behavior when a trigger is received at the Push port but the register is full. The **Pop empty stack** parameter specifies the block's behavior when a trigger is received at the Pop port but the register is empty. The following options are available for both cases:

- **Ignore** — Ignore the trigger event, and continue the simulation.
- **Warning** — Ignore the trigger event, but display a warning message in the MATLAB command window.
- **Error** — Display an error dialog box and terminate the simulation.

---

**Note** The **Push full stack** and **Pop empty stack** parameters are diagnostic parameters. Like all diagnostic parameters on the Configuration Parameters dialog box, they are set to `Ignore` in the code generated for this block by Simulink Coder code generation software.

---

The **Push full stack** parameter additionally offers the **Dynamic reallocation** option, which dynamically resizes the register to accept as many additional inputs as memory permits. To find out how many elements are on the stack at a given time, enable the Num output port by selecting the **Show number of stack entries port** parameter.

---

**Note** When `Dynamic reallocation` is selected, the **System target file** parameter on the **Code Generation** pane of the Model Configuration Parameters dialog box must be set to `grt_malloc.tlc` – Generic Real-Time Target with dynamic memory allocation.








---

## Examples

### Example 1

The table below illustrates the Stack block's operation for a **Stack depth** of 4, **Trigger type** of `Either edge`, and **Clear output port on reset** enabled. Because the block triggers on both rising and falling edges in this example, each transition from 1 to 0 or 0 to 1 in the Push, Pop, and Rst columns below represents a distinct trigger event. A 1 in the Empty column indicates an empty buffer, while a 1 in the Full column indicates a full buffer.

| In | Push | Pop | Rst | Stack                                      | Out | Empty | Full | Num |
|----|------|-----|-----|--|-----|-------|------|-----|
| 1  | 0    | 0   | 0   | <i>top</i> █         █ <i>bottom</i>       | 0   | 1     | 0    | 0   |
| 2  | 1    | 0   | 0   | <i>top</i> █ 2         █ <i>bottom</i>     | 0   | 0     | 0    | 1   |
| 3  | 0    | 0   | 0   | <i>top</i> █ 3   2       █ <i>bottom</i>   | 0   | 0     | 0    | 2   |
| 4  | 1    | 0   | 0   | <i>top</i> █ 4   3   2     █ <i>bottom</i> | 0   | 0     | 0    | 3   |
| 5  | 0    | 0   | 0   | <i>top</i> █ 5   4   3   2 █ <i>bottom</i> | 0   | 0     | 1    | 4   |

| In | Push | Pop | Rst | Stack  | Out | Empty | Full | Num |
|----|------|-----|-----|--|-----|-------|------|-----|
| 6  | 0    | 1   | 0   | top  bottom | 5   | 0     | 0    | 3   |
| 7  | 0    | 0   | 0   | top  bottom | 4   | 0     | 0    | 2   |
| 8  | 0    | 1   | 0   | top  bottom | 3   | 0     | 0    | 1   |
| 9  | 0    | 0   | 0   | top  bottom | 2   | 1     | 0    | 0   |
| 10 | 1    | 0   | 0   | top  bottom | 2   | 0     | 0    | 1   |
| 11 | 0    | 0   | 0   | top  bottom | 2   | 0     | 0    | 2   |
| 12 | 1    | 0   | 1   | top  bottom | 0   | 0     | 0    | 1   |

Note that at the last step shown, the Push and Rst ports are triggered simultaneously. The Rst trigger takes precedence, and the stack is first cleared and then pushed.

## Parameters

### Stack depth

The number of entries that the LIFO register can hold.

### Trigger type

The type of event that triggers the block's execution. The rate of the trigger signal must be the same as the rate of the data signal input.

### Push full stack

Response to a trigger received at the Push port when the register is full. Inputs to this port must have the same built-in data type as inputs to the Pop and Rst input ports.

When Dynamic reallocation is selected, the **System target file** parameter on the **Code Generation** pane of the Model Configuration Parameters dialog box must be set to `grt_malloc.tlc` – Generic Real-Time Target with dynamic memory allocation.

### Pop empty stack

Response to a trigger received at the Pop port when the register is empty. Inputs to this port must have the same built-in data type as inputs to the Push and Rst input ports.

### Show empty stack indicator port

Enable the Empty output port, which is high (1) when the stack is empty, and low (0) otherwise.

### Show full stack indicator port

Enable the Full output port, which is high (1) when the stack is full, and low (0) otherwise. The Full port remains low when you select **Dynamic reallocation** from the **Push full stack** parameter.

### Show number of stack entries port

Enable the Num output port, which tracks the number of entries currently on the stack. When inputs to the In port are double-precision values, the outputs from the Num port are double-precision values. Otherwise, the outputs from the Num port are 32-bit unsigned integer values.

**Show reset port to clear internal stack buffer**

Enable the Rst input port, which empties the stack when the trigger specified by the **Trigger type** is received. Inputs to this port must have the same built-in data type as inputs to the Push and Pop input ports.

**Clear output port on reset**

Reset the Out port to zero (in addition to clearing the stack) when a trigger is received at the Rst input port.

**Supported Data Types**

| Port | Supported Data Types  |
|------|---|
| In   | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed and unsigned)</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul>  |
| Push | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul> <p>Inputs to this port must have the same built-in data type as inputs to the Pop and Rst input ports</p>   |
| Pop  | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul> <p>Inputs to this port must have the same built-in data type as inputs to the Push and Rst input ports.</p> |
| Rst  | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul> <p>Inputs to this port must have the same built-in data type as inputs to the Push and Pop input ports.</p> |

| Port  | Supported Data Types   |
|-------|--|
| Out   | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed and unsigned)</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul>   |
| Empty | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Boolean</li> </ul>   |
| Full  | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Boolean</li> </ul>   |
| Num   | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> </ul> <p>The block outputs a double-precision floating-point value at this port when the data type of the In port is double-precision floating-point.</p> <ul style="list-style-type: none"> <li>• 32-bit unsigned integers</li> </ul> <p>The block outputs a 32-bit unsigned integer value at this port when the data type of the In port is anything other than double-precision floating-point.</p> |

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

The following limitations apply:

- Generated code relies on the `memcpy` or `memset` function (`string.h`) under certain conditions.
- When `Dynamic reallocation` is selected, the **System target file** parameter on the **Code Generation** pane of the Configuration Parameters dialog box must be set to `grt_malloc.tlc` - Generic Real-Time Target with dynamic memory allocation.

### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

## See Also

### Blocks

Buffer | Delay Line | Queue

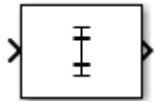
### Topics

“Queues”



# Standard Deviation

Standard deviation of input or sequence of inputs



**Libraries:**  
DSP System Toolbox / Statistics

## Description

The Standard Deviation block computes the standard deviation of each row or column of the input, or along vectors of a specified dimension of the input. It can also compute the standard deviation of the entire input. You can specify the dimension using the **Find the standard deviation value over** parameter. The Standard Deviation block can also track the standard deviation in a sequence of inputs over a period of time. To track the standard deviation in a sequence of inputs, select the **Running standard deviation** parameter.

---

**Note** The **Running** mode in the Standard Deviation block will be removed in a future release. To compute the running standard deviation in Simulink, use the Moving Standard Deviation block instead.

---

## Ports

### Input

**In** — Data input  
vector | matrix | *N*-D array

The block accepts real-valued or complex-valued multichannel and multidimensional inputs.

This port is unnamed until you select the **Running standard deviation** parameter and set the **Reset port** parameter to any option other than None.

Data Types: single | double

**Rst** — Reset port  
scalar

Specify the reset event that causes the block to reset the running standard deviation. The sample time of the **Rst** input must be a positive integer multiple of the input sample time.

### Dependencies

To enable this port, select the **Running standard deviation** parameter and set the **Reset port** parameter to any option other than None.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean

## Output

**Port 1** — Standard deviation along the specified dimension

scalar | vector | matrix |  $N$ -D array

The data type of the output matches the data type of the input.

When you do not select the **Running standard deviation** parameter, the block computes the standard deviation in each row or column of the input, or along vectors of a specified dimension of the input. It can also compute the standard deviation of the entire input at each individual sample time. Each element in the output array  $y$  is the standard deviation of the corresponding column, row, or entire input. The output array  $y$  depends on the setting of the **Find the standard deviation value over** parameter. Consider a three-dimensional input signal of size  $M$ -by- $N$ -by- $P$ . When you set **Find the standard deviation value over** to:

- **Entire input** — The output at each sample time is a scalar that contains the standard deviation of the  $M$ -by- $N$ -by- $P$  input matrix.
- **Each row** — The output at each sample time consists of an  $M$ -by-1-by- $P$  array, where each element contains the standard deviation of each vector over the second dimension of the input. For an  $M$ -by- $N$  matrix input, the output at each sample time is an  $M$ -by-1 column vector.
- **Each column** — The output at each sample time consists of a 1-by- $N$ -by- $P$  array, where each element contains the standard deviation of each vector over the first dimension of the input. For an  $M$ -by- $N$  matrix input, the output at each sample time is a 1-by- $N$  row vector.

In this mode, the block treats length- $M$  unoriented vector inputs as  $M$ -by-1 column vectors.

- **Specified dimension** — The output at each sample time depends on the value of the **Dimension** parameter. If you set the **Dimension** to 1, the output is the same as when you select **Each column**. If you set the **Dimension** to 2, the output is the same as when you select **Each row**. If you set the **Dimension** to 3, the output at each sample time is an  $M$ -by- $N$  matrix containing the standard deviation of each vector over the third dimension of the input.

When you select **Running standard deviation**, the block tracks the standard deviation of each channel in a time sequence of inputs. In this mode, you must also specify a value for the **Input processing** parameter.

- **Elements as channels (sample based)** — The block treats each element of the input as a separate channel. For a three-dimensional input signal of size  $M$ -by- $N$ -by- $P$ , the block outputs an  $M$ -by- $N$ -by- $P$  array. Each element  $y_{ijk}$  of the output contains the standard deviation of the element  $u_{ijk}$  for all inputs since the last reset.

When a reset event occurs, the running standard deviation  $y_{ijk}$  in the current frame is reset to the element  $u_{ijk}$ .

- **Columns as channels (frame based)** — The block treats each column of the input as a separate channel. This option does not support input signals with more than two dimensions. For a two-dimensional input signal of size  $M$ -by- $N$ , the block outputs an  $M$ -by- $N$  matrix. Each element  $y_{ij}$  of the output contains the standard deviation of the elements in the  $j$ th column of all inputs since the last reset, up to and including the element  $u_{ij}$  of the current input.

When a reset event occurs, the running standard deviation for each channel becomes the standard deviation of all the samples in the current input frame, up to and including the current input sample.

Data Types: single | double

## Parameters

### Main Tab

**Running standard deviation** — Option to select running standard deviation  
off (default) | on

When you select the **Running standard deviation** parameter, the block tracks the standard deviation value of each channel in a time sequence of inputs.

**Find the standard deviation value over** — Dimension over which the block computes the standard deviation  
Each column (default) | Entire input | Each row | Specified dimension

- **Each column** — The block outputs the standard deviation over each column.
- **Each row** — The block outputs the standard deviation over each row.
- **Entire input** — The block outputs the standard deviation over the entire input.
- **Specified dimension** — The block outputs the standard deviation over the dimension, specified in the **Dimension** parameter.

### Dependencies

To enable this parameter, clear the **Running standard deviation** parameter.

**Dimension** — Custom dimension  
1 (default) | scalar

Specify the dimension (one-based value) of the input signal over which the standard deviation is computed. The value of this parameter must be greater than 0 and less than the number of dimensions in the input signal.

### Dependencies

To enable this parameter, set **Find the standard deviation value over** to **Specified dimension**.

**Input processing** — Method to process the input in running mode  
Columns as channels (frame based) (default) | Elements as channels (sample based)

- **Columns as channels (frame based)** — The block treats each column of the input as a separate channel. This option does not support input signals with more than two dimensions. For a two-dimensional input signal of size  $M$ -by- $N$ , the block outputs an  $M$ -by- $N$  matrix. Each element  $y_{ij}$  of the output contains the standard deviation of the elements in the  $j$ th column of all inputs since the last reset, up to and including the element  $u_{ij}$  of the current input.

When a reset event occurs, the running standard deviation for each channel becomes the standard deviation of all the samples in the current input frame, up to and including the current input sample.

- **Elements as channels (sample based)** — The block treats each element of the input as a separate channel. For a three-dimensional input signal of size  $M$ -by- $N$ -by- $P$ , the block outputs an

$M$ -by- $N$ -by- $P$  array. Each element  $y_{ijk}$  of the output contains the standard deviation of the element  $u_{ijk}$  for all inputs since the last reset.

When a reset event occurs, the running standard deviation  $y_{ijk}$  in the current frame is reset to the element  $u_{ijk}$ .

### Variable-Size Inputs

When your inputs are of variable size, and you select the **Running standard deviation** parameter, then:

- If you set the **Input processing** parameter to `Elements as channels (sample based)`, the state is reset.
- If you set the **Input processing** parameter to `Columns as channels (frame based)`, then:
  - When the input size difference is in the number of channels (number of columns), the state is reset.
  - When the input size difference is in the length of channels (number of rows), no reset occurs and the running operation is carried out as usual.

### Dependencies

To enable this parameter, select the **Running standard deviation** parameter.

#### Reset port — Reset event

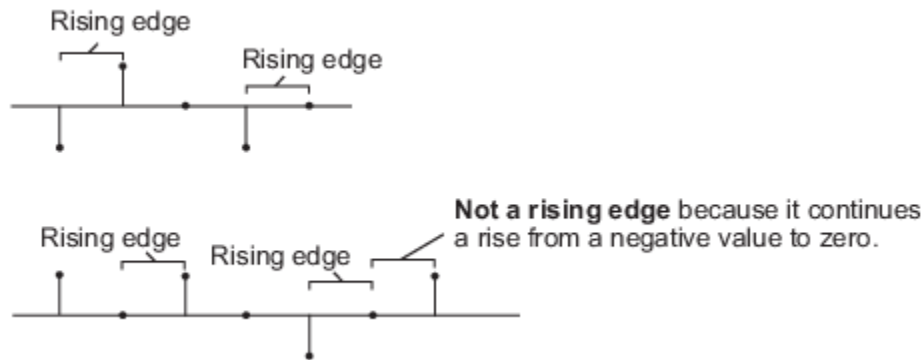
None (default) | Rising edge | Falling edge | Either edge | Non-zero sample

The block resets the running standard deviation whenever a reset event is detected at the optional **Rst** port. The reset sample time must be a positive integer multiple of the input sample time.

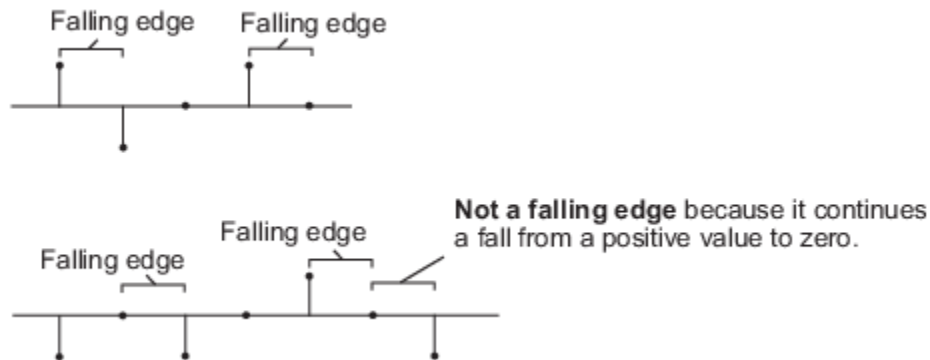
When a reset event occurs while the **Input processing** parameter is set to `Elements as channels (sample based)`, the running standard deviation for each channel is initialized to the value in the corresponding channel of the current input. Similarly, when the **Input processing** parameter is set to `Columns as channels (frame based)`, the running standard deviation for each channel becomes the standard deviation of all the samples in the current input frame, up to and including the current input sample.

Use this parameter to specify the reset event.

- `None` — Disables the **Rst** port.
- `Rising edge` — Triggers a reset operation when the **Rst** input does one of the following:
  - Rises from a negative value to either a positive value or zero.
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero.



- **Falling edge** — Triggers a reset operation when the **Rst** input does one of the following:
  - Falls from a positive value to a negative value or zero.
  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero.



- **Either edge** — Triggers a reset operation when the **Rst** input is a Rising edge or Falling edge.
- **Non-zero sample** — Triggers a reset operation at each sample time, when the **Rst** input is not zero.

---

**Note** When running simulations in the Simulink multitasking mode, reset signals have a one-sample latency. Therefore, when the block detects a reset event, there is a one-sample delay at the reset port rate before the block applies the reset. For more information on latency and the Simulink tasking modes, see “Excess Algorithmic Delay (Tasking Latency)” and “Time-Based Scheduling and Code Generation” (Simulink Coder).

---

### Dependencies

To enable this parameter, select the **Running standard deviation** parameter.

### Block Characteristics

|                           |                 |
|---------------------------|-----------------|
| <b>Data Types</b>         | double   single |
| <b>Direct Feedthrough</b> | no              |

|                                 |     |
|---------------------------------|-----|
| <b>Multidimensional Signals</b> | no  |
| <b>Variable-Size Signals</b>    | yes |
| <b>Zero-Crossing Detection</b>  | no  |

## More About

### Standard Deviation

The standard deviation of a discrete-time signal is the square root of the variance of the signal.

Standard deviation gives a measure of deviation of the signal from its mean value.

For purely real or imaginary input,  $u$ , of size  $M$ -by- $N$ , the standard deviation is given by the following equation:

$$y = \sigma = \sqrt{\frac{\sum_{i=1}^M \sum_{j=1}^N |u_{ij}|^2 - \frac{\left| \sum_{i=1}^M \sum_{j=1}^N u_{ij} \right|^2}{M*N}}{M*N - 1}}$$

- $u_{ij}$  is the input data element at indices  $i, j$ .
- $M$  is the length of the  $j$ th column.
- $N$  is the number of columns.

For complex inputs, the standard deviation is given by the following equation:

$$\sigma = \sqrt{\sigma_{Re}^2 + \sigma_{Im}^2}$$

- $\sigma_{Re}^2$  is the variance of the real part of the complex input.
- $\sigma_{Im}^2$  is the variance of the imaginary part of the complex input.

## Algorithms

### Standard Deviation

When you clear the **Running standard deviation** parameter in the block and specify a dimension, the block produces results identical to the MATLAB `std` function, when it is called as `y = std(u, D)`.

- $u$  is the data input.
- $D$  is the dimension.
- $y$  is the standard deviation along the specified dimension.

The standard deviation along the entire input is identical to calling the `std` function as `y = std(u(:))`.

For a complex input signal, the standard deviation is the square root of the sum of the variances of the real and imaginary parts.

$$\sigma = \sqrt{\sigma_{\text{Re}}^2 + \sigma_{\text{Im}}^2}$$

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

## See Also

### Functions

`std`

### Objects

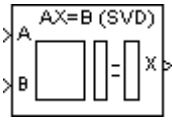
`dsp.MovingStandardDeviation`

### Blocks

Moving Standard Deviation

## SVD Solver

Solve  $AX = B$  using singular value decomposition



### Libraries:

DSP System Toolbox / Math Functions / Matrices and Linear Algebra / Linear System Solvers

## Description

The SVD Solver block solves the linear system  $AX = B$ . The linear system can be overdetermined, underdetermined, or exactly determined. The block solves the system by applying singular value decomposition (SVD) factorization to the input matrix at the **A** port.

## Ports

### Input

**A** — Input *A* matrix  
*M*-by-*N* matrix

Specify the input matrix *A* as an *M*-by-*N* matrix through this port. Inputs *A* and *B* must have the same number of rows.

Data Types: `single` | `double`

**B** — *B* matrix  
*M*-by-*L* matrix | *M*-by-1 vector

Specify the matrix *B* in the  $AX = B$  equation as an *M*-by-*L* matrix or an *M*-by-1 vector.

When you specify an *M*-by-1 vector, the block treats the length-*M* unoriented vector input at port **B** as an *M*-by-1 matrix. Inputs *A* and *B* must have the same number of rows.

Data Types: `single` | `double`

### Output

**X** — Solution to system of equations  
*N*-by-*L* matrix

The output at the **X** port is the *N*-by-*L* matrix *X*. The block computes *X* to minimize the sum of the squares of the elements of  $B - AX$  (the residual).

When *B* is a vector, this solution minimizes the vector 2-norm of the residual. When *B* is a matrix, this solution minimizes the matrix Frobenius norm of the residual. In this case, the columns of *X* are the solutions to the *L* corresponding systems  $AX_k = B_k$ , where  $B_k$  is the *k*th column of *B*, and  $X_k$  is the *k*th column of *X*.

*X* is known as the minimum-norm-residual solution to  $AX = B$ . The minimum-norm-residual solution is unique for overdetermined and exactly determined linear systems, but the solution is not unique for



underdetermined linear systems. Thus when you apply the SVD Solver block to an underdetermined system, the block computes  $X$  to minimize the number of nonzero entries in  $X$ .

Data Types: `single` | `double`

**E** — Show error status report  
logical scalar

The block returns a logical scalar value of 0 if the singular value decomposition calculation converges, and a logical scalar value of 1 if the singular value decomposition calculation does not converge.

Data Types: `Boolean`

## Parameters

**Show error status port (E)** — Select to report failure to converge

`off` (default) | `on`

Select to enable the **E** output port, which reports the failure to converge as one of the following:

- 0 — The singular value decomposition calculation converges.
- 1 — The singular value decomposition calculation does not converge.

If the singular value decomposition calculation fails to converge, the output at port **X** is an undefined matrix of the expected size.

## Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | <code>double</code>   <code>single</code> |
| <b>Direct Feedthrough</b>       | <code>no</code>                           |
| <b>Multidimensional Signals</b> | <code>no</code>                           |
| <b>Variable-Size Signals</b>    | <code>no</code>                           |
| <b>Zero-Crossing Detection</b>  | <code>no</code>                           |

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Generated code relies on the `memcpy` or `memset` function (`string.h`) under certain conditions.

## **See Also**

### **Blocks**

Autocorrelation LPC | Cholesky Solver | LDL Solver | Levinson-Durbin | LU Inverse | Pseudoinverse | QR Solver | Singular Value Decomposition

# Time Scope

Display and analyze signals generated during simulation and log signal data to MATLAB

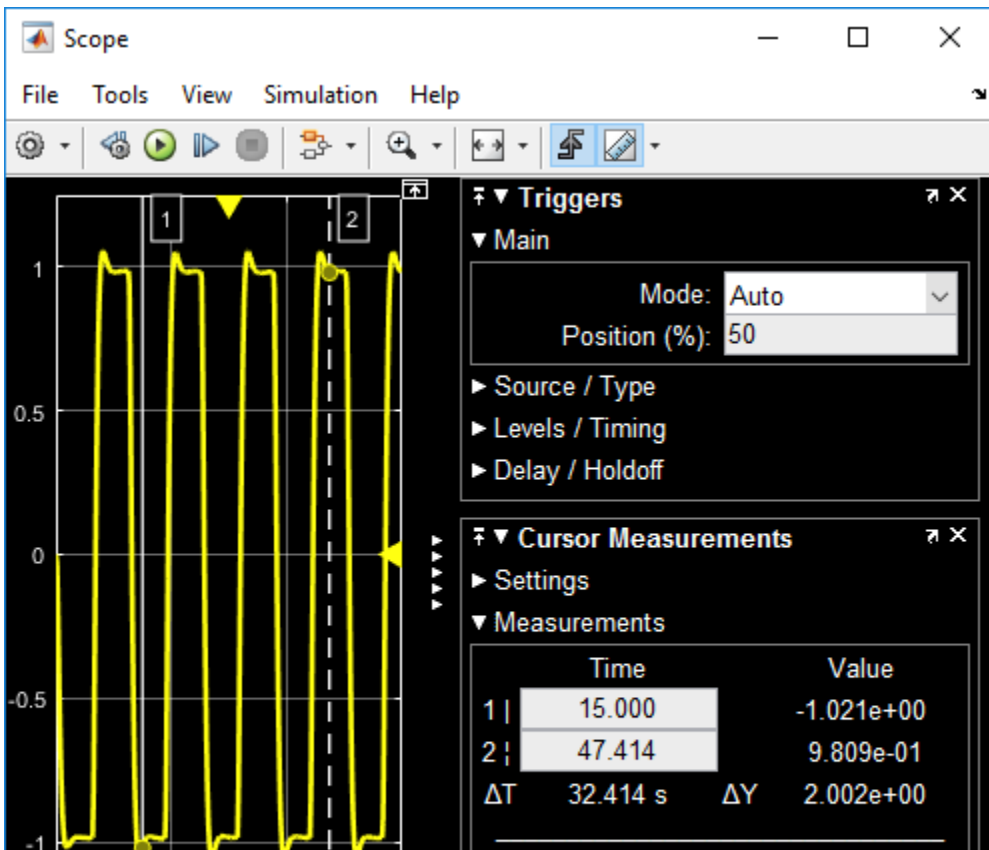
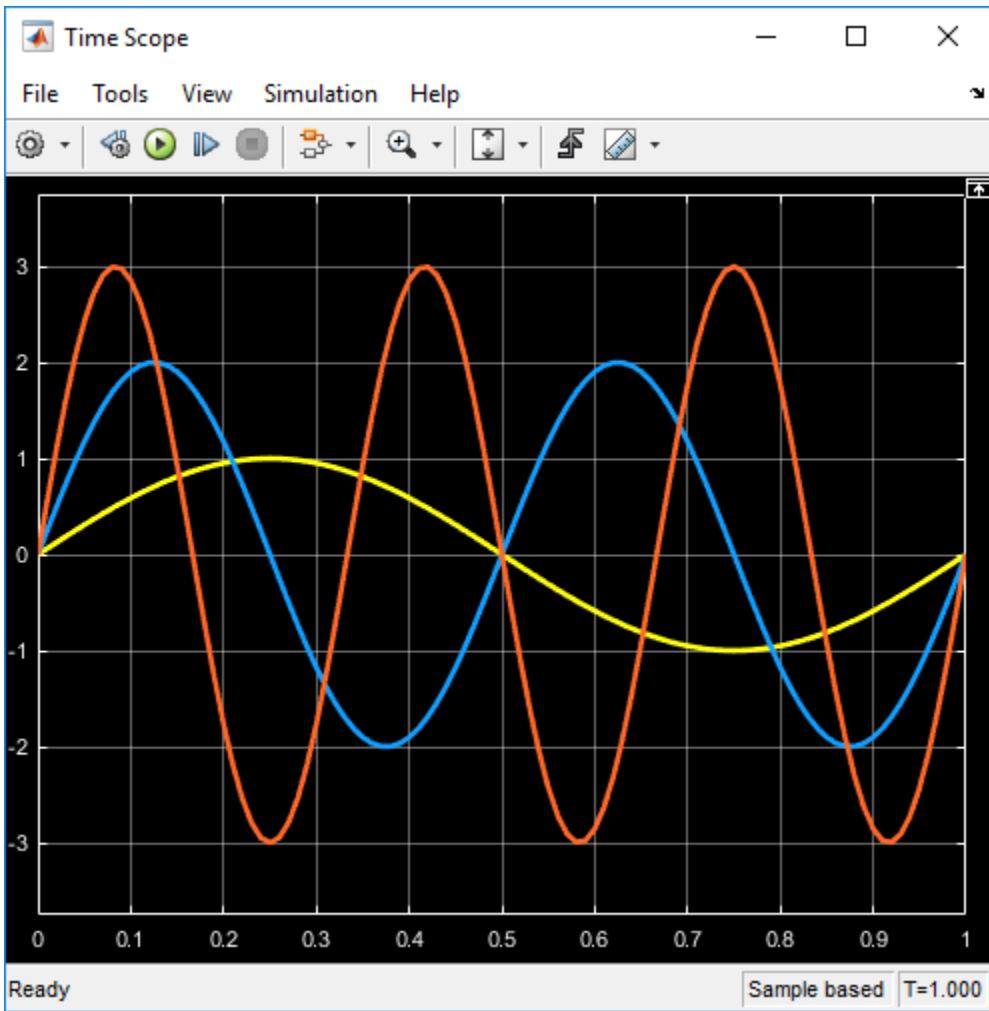
**Libraries:**

DSP System Toolbox / Sinks

DSP System Toolbox HDL Support / Sinks

## Description

The Simulink Scope block and DSP System Toolbox Time Scope block display time domain signals.



The two blocks have identical functionality, but different default settings. The Time Scope is optimized for discrete time processing. The Scope is optimized for general time-domain simulation. For a side-by-side comparison, see “Simulink Scope Versus DSP System Toolbox Time Scope” (Simulink).

Oscilloscope features:

- Triggers — Set triggers to sync repeating signals and pause the display when events occur.
- Cursor Measurements — Measure signal values using vertical and horizontal cursors.
- Signal Statistics — Display the maximum, minimum, peak-to-peak difference, mean, median, and RMS values of a selected signal.
- Peak Finder — Find maxima, showing the x-axis values at which they occur.
- Bilevel Measurements — Measure transitions, overshoots, undershoots, and cycles.

You must have a Simscape™ or DSP System Toolbox license to use the Peak Finder, Bilevel Measurements, and Signal Statistics.

Scope display features:

- Simulation control — Debug models from a Scope window using **Run**, **Step Forward**, and **Step Backward** toolbar buttons.
- Multiple signals — Plot multiple signals on the same y-axis (display) using multiple input ports.
- Multiple y-axes (displays) — Display multiple y-axes. All the y-axes have a common time range on the x-axis.
- Modify parameters — Modify scope parameter values before and during a simulation.
- Axis autoscaling — Autoscale axes during or at the end of a simulation. Margins are drawn at the top and bottom of the axes.
- Display data after simulation — Scope data is saved during a simulation. If a scope is closed at the start of a simulation, when you open the scope after a simulation, the scope displays simulation results for attached input signals.

---

**Note** If you have a high sample rate or long simulation time, you may run into issues with memory or system performance because the scope saves data internally. To limit the amount of data saved for scope visualization, use the Limit data points to last property.

---

For information on controlling a scope programmatically, see “Control Scope Blocks Programmatically” (Simulink).

## Limitations

- Do not use scope blocks in a Library. If you place a scope block inside a library block with a locked link or in a locked library, Simulink displays an error when trying to open the scope window. To display internal data from a library block, add an output port to the library block, and then connect the port to a Scope block in your model.
- If you step through a model, the scope only updates when the scope block runs. This means that the time shown in the status bar may not match the model time.
- When connected to a constant signal, a scope block may plot a single point.
- The scope shows gaps in the display when the signal value is NaN.

- When you visualize multiple frame-based signals in the scope, some samples of signals with a frame size of 1 might not be displayed. To visualize these signals, move the signals with frame size of 1 to a separate scope.
- Scope displays have limitations in Rapid Accelerator mode. See “Behavior of Scopes and Viewers with Rapid Accelerator Mode” (Simulink)
- When the Scope is in a ForEach subsystem, the scope only displays the last index.

## Ports

### Input

**Port\_1** — Signal or signals to visualize

scalar | vector | matrix | array | bus | nonvirtual bus

Connect the signals you want to visualize. You can have up to 96 input ports. Input signals can have these characteristics:

- **Type** — Continuous (sample-based) or discrete (sample-based and frame-based).
- **Data type** — Any data type that Simulink supports. See “Data Types Supported by Simulink” (Simulink).
- **Dimension** — Scalar, one dimensional (vector), two dimensional (matrix), or multidimensional (array). Display multiple channels within one signal depending on the dimension. See “Signal Dimensions” (Simulink) and “Determine Signal Dimensions” (Simulink).

### Input Limitations

- When the input is a constant signal, the scope plots a single point.
- The scope shows gaps in the display when the signal value is NaN.
- When you visualize multiple frame-based signals in the scope, some samples of signals with a frame size of 1 might not be displayed. To visualize these signals, move the signals with frame size of 1 to a separate scope.

### Bus Support

You can connect nonvirtual bus and arrays of bus signals to a scope. To display the bus signals, use normal or accelerator simulation mode. The scope displays each bus element signal in the order the elements appear in the bus, from the top to the bottom. Nested bus elements are flattened.

To log nonvirtual bus signals with a scope, set the **Save format** parameter to **Dataset**. You can use any **Save format** to log virtual bus signals.

Data Types: single | double | half | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | Boolean | fixed point | enumerated | bus

## Properties

### Configuration Properties

The Configuration Properties dialog box controls various properties about the scope displays. From the scope menu, select **View > Configuration Properties**.

**Main**

**Open at simulation start** — Specify when scope window opens

off (default for Scope) | on (default for Time Scope)

Select this check box to open the scope window when simulation starts.

**Programmatic Use**

See `OpenAtSimulationStart`.

**Display the full path** — Display block path on scope title bar

off (default) | on

Select this check box to display the block path in addition to the block name.

**Number of input ports** — Number of input ports on scope block

1 (default) | integer

Specify number of input ports on a Scope block, specified as an integer. The maximum number of input ports is 96.

**Programmatic Use**

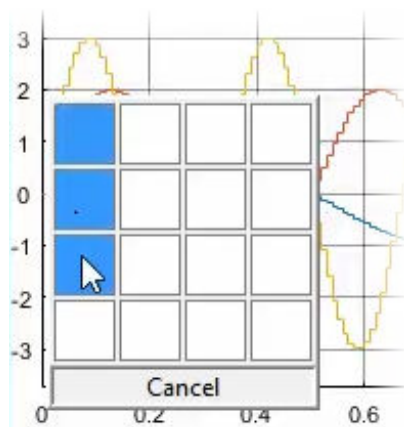
See `NumInputPorts`.

**Layout** — Number and arrangement of displays

1-by-1 display (default) | an arrangement of m-by-n axes

Specify number and arrangement of displays. The maximum layout is 16 rows by 16 columns.

To expand the layout grid beyond 4 by 4, click within the dialog box and drag. Maximum of 16 rows by 16 columns.



If the number of displays is equal to the number of ports, signals from each port appear on separate displays. If the number of displays is less than the number of ports, signals from additional ports appear on the last display. For layouts with multiple columns and rows, ports are mapped down then across.

**Programmatic Use**

See `LayoutDimensions`.

**Sample time** — Simulation interval between scope updates

-1 (for inherited) (default) | positive real number

Specify the time interval between updates of the scope display. This property does not apply to floating scopes and scope viewers. For a more detailed explanation of sample time with the scope, see “Sample Time with Scope Blocks” (Simulink)

**Programmatic Use**

See `SampleTime`.

**Input processing** — Channel or element signal processing

`Elements as channels (sample based)` (default for Scope) | `Columns as channels (frame based)` (default for Time Scope)

- `Elements as channels (sample based)` - Process each element as a unique sample.
- `Columns as channels (frame based)` - Process signal values in a channel as a group of values from multiple time intervals. Frame-based processing is available only with discrete input signals.

**Programmatic Use**

See `FrameBasedProcessing`.

**Maximize axes** — Maximize size of plots

`Off` (default for Scope) | `Auto` (default for Time Scope) | `On`

- `Auto` - If “Title” on page 2-0 and “Y-label” on page 2-0 properties are not specified, maximize all plots.
- `On` - Maximize all plots. Values in **Title** and **Y-label** are hidden.
- `Off` - Do not maximize plots.

**Programmatic Use**

See `MaximizeAxes`.

**Time****Time span** — Length of x-axis to display

`Auto` (default) | `User defined` | `One frame period`

- `Auto` — Difference between the simulation start and stop times.

The block calculates the beginning and end times of the time range using the “Time display offset” on page 2-0 and “Time span” on page 2-0 properties. For example, if you set the **Time display offset** to 10 and the **Time span** to 20, the scope sets the time range from 10 to 30.



- **User defined** — Enter any value less than the total simulation time.
- **One frame period** — Use the frame period of the input signal to the Time Scope block. This option is only available when the **Input processing** parameter is set to **Columns as channels (frame based)**.

#### **Programmatic Use**

See `TimeSpan`.

**Time span overrun action** — Display data beyond visible x-axis

Wrap (default) | Scroll

Specify how to display data beyond the visible x-axis range.

You can see the effects of this option only when plotting is slow with large models or small step sizes.

- **Wrap** — Draw a full screen of data from left to right, clear the screen, and then restart drawing the data from the left.
- **Scroll** — Move data to the left as new data is drawn on the right. This mode is graphically intensive and can affect run-time performance.

#### **Programmatic Use**

See `TimeSpanOverrunAction`.

**Time units** — x-axis units

None (default for Scope) | Metric (default for Time Scope) | Seconds

- **Metric** — Display time units based on the length of “Time span” on page 2-0 .
- **Seconds** — Display time in seconds.
- **None** — Do not display time units.

#### **Programmatic Use**

See `TimeUnits`.

**Time display offset** — x-axis offset

0 (default) | scalar | vector

Offset the x-axis by a specified time value, specified as a real number or vector of real numbers.

For input signals with multiple channels, you can enter a scalar or vector:

- **Scalar** — Offset all channels of an input signal by the same time value.
- **Vector** — Independently offset the channels.

#### **Programmatic Use**

See `TimeDisplayOffset`.

**Time-axis labels** — Display of x-axis labels

Bottom Displays Only (default for Scope) | All (default for Time Scope) | None

Specify how x-axis (time) labels display:

- All — Display x-axis labels on all y-axes.
- None — Do not display labels. Selecting None also clears the **Show time-axis label** check box.
- Bottom displays only — Display x-axis label on the bottom y-axis.

#### **Dependencies**

To enable this property, set:

- “Show time-axis label” on page 2-0 to on.
- “Maximize axes” on page 2-0 to off.

The “Active display” on page 2-0 property determines which display is affected.

#### **Programmatic Use**

See `TimeAxisLabels`.

**Show time-axis label** — Display or hide x-axis labels

off (default for Scope) | on (default for Time Scope)

Select this check box to show the x-axis label for the active display

#### **Dependencies**

To enable this property, set “Time-axis labels” on page 2-0 to All or Bottom Displays Only.

The “Active display” on page 2-0 property determines which display is affected.

#### **Programmatic Use**

See `ShowTimeAxisLabel`.

#### **Display**

**Active display** — Selected display

1 (default) | positive integer

Selected display. Use this property to control which display is changed when changing style properties and axes-specific properties.

Specify the desired display using a positive integer that corresponds to the column-wise placement index. For layouts with multiple columns and rows, display numbers are mapped down and then across.

#### **Programmatic Use**

See `ActiveDisplay`.

**Title** — Display name

%<SignalLabel> (default) | string

Title for a display. The default value %<SignalLabel> uses the input signal name for the title.

**Dependency**

The “Active display” on page 2-0 property determines which display is affected.

**Programmatic Use**

See Title.

**Show legend** — Display signal legend

off (default) | on

Toggle signal legend. The names listed in the legend are the signal names from the model. For signals with multiple channels, a channel index is appended after the signal name. Continuous signals have straight lines before their names, and discrete signals have step-shaped lines.

From the legend, you can control which signals are visible. This control is equivalent to changing the visibility in the **Style** properties. In the scope legend, click a signal name to hide the signal in the scope. To show the signal, click the signal name again. To show only one signal, right-click the signal name, which hides all other signals. To show all signals, press **Esc**.

---

**Note** The legend only shows the first 20 signals. Any additional signals cannot be controlled from the legend.

---

**Dependency**

The “Active display” on page 2-0 property determines which display is affected.

**Programmatic Use**

See ShowLegend.

**Show grid** — Show internal grid lines

on (default) | off

Select this check box to show grid lines.

**Dependency**

The “Active display” on page 2-0 property determines which display is affected.

**Programmatic Use**

See ShowGrid.

**Plot signals as magnitude and phase** — Split display into magnitude and phase plots

off (default) | on

- On — Display magnitude and phase plots. If the signal is real, plots the absolute value of the signal for the magnitude. The phase is 0 degrees for positive values and 180 degrees for negative values. This feature is useful for complex-valued input signals. If the input is a real-valued signal, selecting this check box returns the absolute value of the signal for the magnitude.

- Off — Display signal plot. If the signal is complex, plots the real and imaginary parts on the same y-axis.

**Dependency**

The “Active display” on page 2-0 property determines which display is affected.

**Programmatic Use**

See `PlotAsMagnitudePhase`.

**Y-limits (Minimum)** — Minimum y-axis value

-10 (default) | real scalar

Specify the minimum value of the y-axis as a real number.

**Tunable:** Yes

**Dependency**

If you select **Plot signals as magnitude and phase**, this property only applies to the magnitude plot. The y-axis limits of the phase plot are always [-180 180].

The “Active display” on page 2-0 property determines which display is affected.

**Programmatic Use**

See `YLimits`.

**Y-limits (Maximum)** — Maximum y-axis value

10 (default) | real scalar

Specify the maximum value of the y-axis as a real number.

**Tunable:** Yes

**Dependency**

If you select **Plot signals as magnitude and phase**, this property only applies to the magnitude plot. The y-axis limits of the phase plot are always [-180 180].

The “Active display” on page 2-0 property determines which display is affected.

**Programmatic Use**

See `YLimits`.

**Y-label** — Y-axis label

none (default for Scope) | Amplitude (default for Time Scope) | string

Specify the text to display on the y-axis. To display signal units, add (`%<SignalUnits>`) to the label. At the beginning of a simulation, Simulink replaces (`%SignalUnits`) with the units associated with the signals.

Example: For a velocity signal with units of m/s, enter `Velocity (%<SignalUnits>)`.

**Dependency**

If you select **Plot signals as magnitude and phase**, this property does not apply. The y-axes are labeled Magnitude and Phase.

The “Active display” on page 2-0 property determines which display is affected.

**Programmatic Use**

See YLabel.

**Logging****Limit data points to last** — Limit buffered data values

off and 5000 (default) | on | positive integer

Limit data saved by the scope internally. By default all data points are saved so that you can view the scope visualization after the simulation finishes. For simulations with **Stop time** set to `inf`, consider selecting **Limit data points to last**.

---

**Note** If you do not select **Limit data points to last** and you have a high sample rate or long simulation time, you may run into issues with memory or system performance.

---

When you select this property, the scope saves the latest  $n$  data points, where  $n$  the specified number of data points.

- Off — Save and plot all data values.
- On — Save specified number of data values for each signal. If the signal is frame-based, the number of buffered data values is the specified number of data values multiplied by the frame size.

In some cases, selecting this property can have the effect of plotting signals for less than the entire time range of a simulation (for example if your sample time is small). If the scope plots a portion of your signals, consider increasing the number of data points to save.

This property limits the data values plotted in the scope and the data values saved to a MATLAB variable specified in “Variable name” on page 2-0 .

**Programmatic Use**

See `DataLoggingLimitDataPoints` and `DataLoggingMaxPoints`.

**Decimation** — Reduce amount of scope data to display and save

off, 2 (default) | on | positive integer

- On — Plot and log (save) scope data every  $N$ th data point, where  $N$  is the decimation factor entered in the text box. A value of 1 buffers all data values.
- Off — Save all scope data values.

**Dependency**

To enable this property, select “Log data to workspace” (Simulink).

This property limits the data values plotted in the scope and the data values saved to a MATLAB variable specified in “Variable name” on page 2-0 .

**Programmatic Use**

See `DataLoggingDecimateData` and `DataLoggingDecimation`.

**Log data to workspace** — Save data to MATLAB workspace

off (default) | on

Select this check box to enable logging and enable the **Variable name**, **Save format**, and **Decimation** properties. This property does not apply to floating scopes and scope viewers.

For an example of saving signals to the MATLAB Workspace using a Scope block, see “Save Simulation Data Using Scope Block” (Simulink).

**Programmatic Use**

See `DataLogging`.

**Variable name** — Name of saved data variable

`ScopeData` (default) | string

Specify a variable name for saving scope data in the MATLAB workspace. This property does not apply to floating scopes and scope viewers.

**Dependency**

To enable this property, select “Log data to workspace” (Simulink).

**Programmatic Use**

See `DataLoggingVariableName`.

**Save format** — MATLAB variable format

`Dataset` (default) | `Structure With Time` | `Structure` | `Array`

Select variable format for saving data to the MATLAB workspace. This property does not apply to floating scopes and scope viewers.

- `Dataset` — Save data as a `Dataset` object, by default a `timeseries` object.
- `Structure With Time` — Save data as a structure with associated time information.
- `Structure` — Save data as a structure.
- `Array` — Save data as an array with associated time information. This format does not support variable-size data.

**Dependency**

To enable this property, select “Log data to workspace” (Simulink).

**Programmatic Use**

See `DataLoggingSaveFormat`.

## Axes Scaling Properties

The **Axes Scaling** dialog controls the axes limits of the scope. To open the Axes Scaling properties, in the scope menu, select **Tools > Axes Scaling > Axes Scaling Properties**.

**Axes scaling** — Y-axis scaling mode

Manual (default) | Auto | After N Updates

- **Manual** — Manually scale the y-axis range with the **Scale Y-axis Limits** toolbar button.
- **Auto** — Scale the y-axis range during and after simulation. Selecting this option displays the **Do not allow Y-axis limits to shrink** check box. If you want the y-axis range to increase and decrease with the maximum value of a signal, set **Axes scaling** to Auto and clear the **Do not allow Y-axis limits to shrink** check box.
- **After N Updates** — Scale y-axis after the number of time steps specified in the **Number of updates** text box (10 by default). Scaling occurs only once during each run.

### Programmatic Use

See AxesScaling.

**Do not allow Y-axis limits to shrink** — When y-axis limits can change

on (default) | off

Allow y-axis range limits to increase but not decrease during a simulation.

### Dependency

To use this property, set **Axes scaling** to Auto.

**Number of updates** — Number of updates before scaling

10 (default) | integer

Set this property to delay auto scaling the y-axis.

### Dependency

To use this property, set **Axes scaling** to After N Updates.

### Programmatic Use

See AxesScalingNumUpdates.

**Scale axes limits at stop** — When y-axis limits can change

on (default) | off

- **On** — Scale axes when simulation stops.
- **Off** — Scale axes continually.

### Dependency

To use this property, set **Axes scaling** to Auto.

**Y-axis Data range (%)** — Percent of y-axis to use for plotting

80 (default) | integer between [1, 100]

Specify the percentage of the y-axis range used for plotting data. If you set this property to 100, the plotted data uses the entire y-axis range.

**Y-axis Align** — Alignment along y-axis

Center (default) | Top | Bottom

Specify where to align plotted data along the y-axis data range when **Y-axis Data range** is set to less than 100 percent.

- Top — Align signals with the maximum values of the y-axis range.
- Center — Center signals between the minimum and maximum values.
- Bottom — Align signals with the minimum values of the y-axis range.

**Autoscale X-axis limits** — Scale x-axis range limits

off (default) | on

Scale x-axis range to fit all signal values. If **Axes scaling** is set to Auto, the data currently within the axes is scaled, not the entire signal in the data buffer.

**X-axis Data range (%)** — Percent of x-axis to use for plotting

100 (default) | integer in the range [1, 100]

Specify the percentage of the x-axis range to plot data on. For example, if you set this property to 100, plotted data uses the entire x-axis range.

**X-axis Align** — Alignment along x-axis

Center (default) | Left | Right

Specify where to align plotted data along the x-axis data range when **X-axis Data range** is set to less than 100 percent.

- Right — Align signals with the maximum values of the x-axis range.
- Center — Center signals between the minimum and maximum values.
- Left — Align signals with the minimum values of the x-axis range.

**Style Properties**

To open the Style dialog box, from the scope menu, select **View > Style**.

**Figure color** — Background color for window

black (default) | color

Background color for the scope.

**Plot type** — How to plot signal



Auto (default for Scope) | Line (default for Time Scope) | Stairs | Stem

When you select **Auto**, the plot type is a line graph for continuous signals, a stair-step graph for discrete signals, and a stem graph for Simulink message signals.

**Axes colors** — Background and axes color for individual displays

black (default) | color

Select the background color for axes (displays) with the first color palette. Select the grid and label color with the second color palette.

**Preserve colors for copy to clipboard** — Copy scope without changing colors

off (default) | on

Specify whether to use the displayed color of the scope when copying.

When you select **File > Copy to Clipboard**, the software changes the color of the scope to be printer friendly (white background, visible lines). If you want to copy and paste the scope with the colors displayed, select this check box.

**Properties for line** — Line to change

Channel 1 (default)

Select active line for setting line style properties.

**Visible** — Line visibility

on (default) | off

Show or hide a signal on the plot.

#### Dependency

The values of “Active display” on page 2-0 and “Properties for line” on page 2-0 determine which line is affected.

**Line** — Line style

solid line (default style) | 0.75 (default width) | yellow (default color)

Select line style, width, and color.

#### Dependency

The values of “Active display” on page 2-0 and “Properties for line” on page 2-0 determine which line is affected.

**Marker** — Data point marker style

None (default) | marker shape

Select marker shape.

#### Dependency

The values of “Active display” on page 2-0 and “Properties for line” on page 2-0 determine which line is affected.

## Block Characteristics

|                                 |  |
|---------------------------------|--|
| <b>Data Types</b>               | Boolean   bus <sup>a</sup>   double   enumerated   fixed point   half   integer   single |
| <b>Direct Feedthrough</b>       | no   |
| <b>Multidimensional Signals</b> | yes  |
| <b>Variable-Size Signals</b>    | yes  |
| <b>Zero-Crossing Detection</b>  | no   |

<sup>a</sup> Virtual bus not supported. Nonvirtual bus supported only in normal and accelerator mode simulation. Data logging for nonvirtual bus supported only in the dataset format.

## Tips

If you run your simulation for a long time, you may run into out-of-memory issues because the scope saves data. To limit the amount of data saved for scope visualization, use the Limit data points to last property.

## Version History

Introduced in R2015b

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block can be used for simulation visibility in systems that generate code, but is not included in the generated code.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

This block can be used for simulation visibility in subsystems that generate HDL code, but is not included in the hardware implementation.

### PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

This block can be used for simulation visibility in systems that generate code, but is not included in the generated code.

### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

This block accepts fixed-point input, but converts it to `double` for display.

---

## See Also

### Blocks

Floating Scope | Scope Viewer

### Objects

timescope

### Topics

[“Simulate a Model Interactively” \(Simulink\)](#)

[“Step Through Simulation” \(Simulink\)](#)

[“Common Scope Block Tasks” \(Simulink\)](#)

[“Floating Scope and Scope Viewer Tasks” \(Simulink\)](#)

[“Scope Triggers Panel” \(Simulink\)](#)

[“Cursor Measurements Panel” \(Simulink\)](#)

[“Scope Signal Statistics Panel” \(Simulink\)](#)

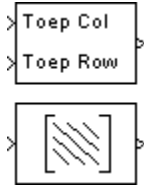
[“Scope Bilevel Measurements Panel” \(Simulink\)](#)

[“Peak Finder Measurements Panel” \(Simulink\)](#)

[“Control Scope Blocks Programmatically” \(Simulink\)](#)

## Toeplitz

Generate matrix with Toeplitz symmetry



## Library

Math Functions / Matrices and Linear Algebra / Matrix Operations

dspmtx3

## Description

The Toeplitz block generates a Toeplitz matrix from inputs defining the first column and first row. The top input (*Col*) is a vector containing the values to be placed in the first *column* of the matrix, and the bottom input (*Row*) is a vector containing the values to be placed in the first *row* of the matrix.

```
y = toeplitz(Col,Row) % Equivalent MATLAB code
```

The other elements of the matrix obey the relationship

$$y(i,j) = y(i-1,j-1)$$

and the output has dimension  $[\text{length}(\text{Col}) \ \text{length}(\text{Row})]$ . The  $y(1,1)$  element is inherited from the *Col* input. For example, the following inputs

```
Col = [1 2 3 4 5]
Row = [7 7 3 3 2 1 3]
```

produce the Toeplitz matrix

$$\begin{bmatrix} 1 & 7 & 3 & 3 & 2 & 1 & 3 \\ 2 & 1 & 7 & 3 & 3 & 2 & 1 \\ 3 & 2 & 1 & 7 & 3 & 3 & 2 \\ 4 & 3 & 2 & 1 & 7 & 3 & 3 \\ 5 & 4 & 3 & 2 & 1 & 7 & 3 \end{bmatrix}$$

When you select the **Symmetric** check box, the block generates a symmetric (Hermitian) Toeplitz matrix from a single input, *u*, defining both the first row and first column of the matrix.

```
y = toeplitz(u) % Equivalent MATLAB code
```

The output has dimension  $[\text{length}(u) \ \text{length}(u)]$ . For example, the Toeplitz matrix generated from the input vector  $[1 \ 2 \ 3 \ 4]$  is

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 1 & 2 & 3 \\ 3 & 2 & 1 & 2 \\ 4 & 3 & 2 & 1 \end{bmatrix}$$

The Toeplitz block supports real and complex floating-point and fixed-point inputs.

## Parameters

### Symmetric

When selected, enables the single-input configuration for symmetric Toeplitz matrix output.

### Saturate on integer overflow

When you generate a symmetric Toeplitz matrix with this block, if the input vector is complex, the output is a symmetric Hermitian matrix whose elements satisfy the relationship

$$y(i, j) = \text{conj}(y(j, i))$$

For fixed-point signals the conjugate operation could result in an overflow. When you select this parameter, overflows saturate. This parameter is only visible with the **Symmetric** parameter is selected. This parameter is ignored for floating-point signals.

## Supported Data Types

| Port     | Supported Data Types   |
|----------|--|
| Input    | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed and unsigned)</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers (real signals only)</li> </ul> |
| Toep Col | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed and unsigned)</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul>                     |
| Toep Row | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed and unsigned)</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul>                     |

| <b>Port</b> | <b>Supported Data Types</b>   |
|-------------|---|
| Output      | <ul style="list-style-type: none"><li>• Double-precision floating point</li><li>• Single-precision floating point</li><li>• Fixed point (signed and unsigned)</li><li>• Boolean</li><li>• 8-, 16-, and 32-bit signed integers</li><li>• 8-, 16-, and 32-bit unsigned integers</li></ul> |

## See Also

toeplitz

MATLAB

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

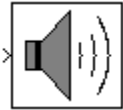
Generate C and C++ code using Simulink® Coder™.

### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

## To Audio Device

Write audio data to computer's audio device



## Compatibility

---

**Note** The To Audio Device block will be removed in a future release. Existing instances of the block continue to run. For new models, use the Audio Device Writer block instead.

---

## Library

Sinks

dspsnks4

## Description

The To Audio Device block sends audio data to your computer's audio device. This block is not supported for use with the Simulink Model block.

Use the **Device** parameter to specify the device to which you want to send the audio data. This parameter is automatically populated based on the audio devices installed on your system.

Select the **Inherit sample rate from input** check box if you want the block to inherit the sample rate of the audio signal from the input to the block. If you clear this check box, the **Sample rate (Hz)** parameter appears on the block. Use this parameter to specify the number of samples per second in the signal.

The range of supported audio device sample rates and data type formats, depend on both the sound card and the API which is chosen for the sound card.

Use the **Device data type** to specify the data type of the audio data that is sent to the device. You can choose:

- 8-bit integer
- 16-bit integer
- 24-bit integer
- 32-bit float
- Determine from input data type

If you choose Determine from input data type, the following table summarizes the block's behavior.

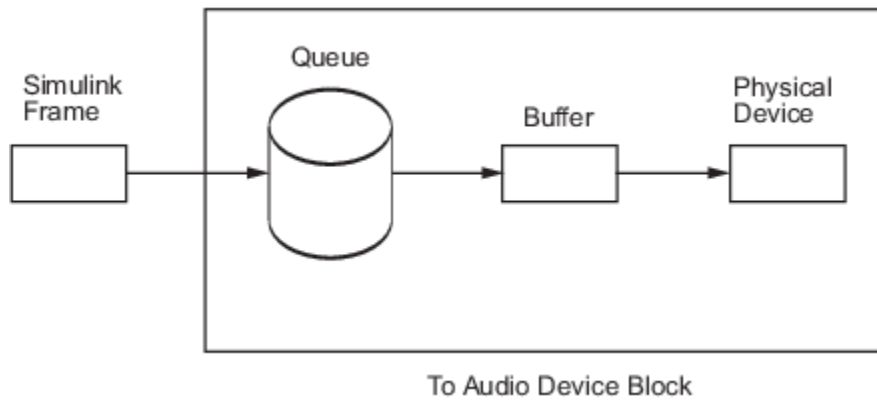
| Input Data Type  | Device Data Type      |
|--|-----------------------|
| Double-precision floating point or single-precision floating point | 32-bit floating point |
| 32-bit integer   | 24-bit integer        |
| 16-bit integer   | 16-bit integer        |
| 8-bit integer  | 8-bit integer         |

If you choose **Determine** from **input data type** and the device does not support the input data type, the block uses the next lowest-precision data type supported by the device.

The generated code for this block relies on prebuilt .dll files. You can run this code outside the MATLAB environment, or redeploy it, but be sure to account for these extra .dll files when doing so. The packNGo function creates a single zip file containing all of the pieces required to run or rebuild this code. See packNGo for more information.

### Buffering

The To Audio Device block buffers the data from a Simulink signal using the process illustrated by the following figure.



- 1 At the start of the simulation, the queue is filled with silence. Specify the size of this queue using the **Queue duration (seconds)** parameter. As Simulink runs, the block appends Simulink frames to the bottom of the queue.
- 2 At each time step, the blocks sends a buffer of samples from the top of the queue to the audio device. Select the **Automatically determine buffer size** check box to allow the block to use a conservative buffer size. See the From Audio Device block reference page for the equation the block uses to calculate this buffer size. If you clear this check box, the **Buffer size (samples)** parameter appears on the block. Use this parameter to specify the size of the buffer in samples.
- 3 The block writes the buffer of audio data to the device. If the queue did not contain enough data to completely fill the buffer, the block fills the remaining portion of the buffer with zeros. This data has a the data type specified by the **Device data type** parameter.

When the simulation throughput rate is lower than the hardware throughput rate, the queue, which is initially full, becomes empty. If the queue is empty, the block sends zeros (silence) to the audio device. You can monitor inserted zeroes using the optional **Under run** output port. When the simulation throughput rate is higher than the hardware throughput rate, the To Audio Device block waits to write data to the queue.



To minimize the chance of dropouts, the block checks to make sure the queue duration is at least as large as the maximum of the buffer size and the frame size. If it is not, the queue duration is automatically set to this maximum value.

### Channel Mapping

The term Channel Mapping refers to a 1-to-1 mapping that associates channels on the selected audio device to channels of the data. When you play audio, channel mapping allows you to specify which channel of the audio device directs input to a specific channel of audio data. You can specify channel mapping as a vector of output channel indices corresponding to each output channel of data being written. The default value in the **Device Output Channels** parameter is 1:MAXOUTPUTCHANNELS. If you do not select the default mapping, you must specify the **Device Output Channels** parameter in the dialog box.

Example: The selected output audio device contains 8 channels. The data being output has dimensions  $N \times 3$  (3-channel data). You want the output to be redirected as follows:

- First data channel to Audio Device channel 3
- Second data channel to Audio Device channel 1
- Third data channel to Audio Device channel 8

Thus, you would specify the **Device Output Channels** as [3 1 8].

### Troubleshooting

#### Not Keeping Up in Real Time

When Simulink cannot keep up with an audio device that is operating in real time, the queue becomes empty and gaps occur in the audio data that the block sends to the device. Select the **Output number of samples by which the queue was underrun** check box to add an output port indicating when the queue was empty. Here are several ways to deal with this situation:

- *Increase the queue duration.*

The **Queue duration (seconds)** parameter specifies the duration of the signal, in seconds, that can be buffered during the simulation. This is the maximum length of time that the block's data supply can lag the hardware's data demand.

- *Increase the buffer size.*

The size of the buffer processed in each interrupt from the audio device affects the performance of your model. If the buffer is too small, a large portion of hardware resources are used to write data to the device. If the buffer is too big, Simulink must wait for the device to empty the buffer before it can write the data to the queue, which introduces latency.

- *Increase the simulation throughput rate.*

Two useful methods for improving simulation throughput rates are increasing the signal frame size and compiling the simulation into native code:

- Increase frame sizes and use frame-based processing throughout the model to reduce the amount of block-to-block communication overhead. This can increase throughput rates in many cases. However, larger frame sizes generally result in greater model latency due to initial buffering operations.
- Generate executable code with Simulink Coder code generation software. Native code runs much faster than Simulink and should provide rates adequate for real-time audio processing.

Other ways to improve throughput rates include simplifying the model and running the simulation on a faster PC processor. For other ideas on improving simulation performance, see “Delay and Latency” and “Optimize Performance” (Simulink).

### Running an Executable Outside MATLAB

To run your generated standalone executable application in Shell, you need to set your environment to the following:

| Platform | Command   |
|----------|---|
| Mac      | <pre>setenv DYLD_LIBRARY_PATH \$LD_LIBRARY_PATH: \$MATLABROOT/bin/ maci64 (csh/tcsh)  export DYLD_LIBRARY_PATH \$LD_LIBRARY_PATH: \$MATLABROOT/bin/ maci64 (Bash)</pre> |
| Linux    | <pre>setenv LD_LIBRARY_PATH \$LD_LIBRARY_PATH: \$MATLABROOT/bin/ glnxa64 (csh/tcsh)  export LD_LIBRARY_PATH \$LD_LIBRARY_PATH: \$MATLABROOT/bin/ glnxa64 (Bash)</pre>   |
| Windows  | <pre>set PATH = \$MATLABROOT\bin\win64;%PATH %</pre>  |

### Channel-to-Speaker Mapping on Windows Operating Systems

The To Audio Device and From Audio Device blocks can support multiple channels. On Windows operating systems, the channel-to-speaker mapping is defined as listed below. This mapping only applies when your sound card is properly configured and capable of receiving the audio data you send. If the number of channels on the card does not match the number of channels on the block, or if you specify a data type for the **Device data type** parameter that is not supported by your device, the Windows mixer intervenes to translate from one format to another. If the Windows mixer does intervene, the channel-to-speaker mapping might differ from what is specified here.

- Single channel input — Front center speaker

On systems with two speakers, the front center channel is split between the right and left speakers.

- Multichannel input — Channels are assigned to speakers as follows:
  - One channel — Front center
  - Two channels — Front left, front right
  - Four channels — Front left, front right, rear left, rear right
  - Six channels — Front left, front right, front center, low frequency, rear left, rear right
  - Eight channels — Front left, front right, front center, low frequency, rear left, rear right, front left center, front right center
  - For all other channel combinations, the channel assignment is dictated by the audio card.

## Audio Hardware API

The To Audio Device and From Audio Device blocks use the open-source PortAudio library in order to communicate with the audio hardware on a given computer. The PortAudio library supports a range of API's designed to communicate with the audio hardware on a given platform. The following API choices were made when building the PortAudio library for the DSP System Toolbox product:

- Windows: DirectSound, WDM-KS, ASIO
- Linux: ALSA, OSS
- Mac: CoreAudio

For Windows, the default is DirectSound, for Linux, the default is ALSA, and for Mac there is only one choice.

To determine the audio hardware API currently selected, type the following command in the MATLAB command prompt.

```
getpref('dsp','portaudioHostApi')
```

The output is a scalar indicating the choice of the API.

- 1 — DirectSound
- 3 — ASIO
- 7 — OSS
- 8 — ALSA
- 11 — WDM-KS

To select a particular API, type the following command in the MATLAB command prompt.

```
setpref('dsp','portaudioHostApi',N)
```

where  $N$  is a scalar. Choose  $N$  based on the API choice above.

## Parameters

### Device

Specify which device to send the audio data to.

### Inherit sample rate from input

Select this check box if you want the block to inherit the sample rate of the audio signal from the input to the block.

### Sample rate (Hz)

Specify the number of samples per second in the signal. This parameter is visible when the **Inherit sample rate from input** check box is cleared.

### Device data type

Specify the data type of the audio data sent to the device.

### Automatically determine buffer size

Select this check box to allow the block to calculate a conservative buffer size.

**Buffer size (samples)**

Specify the size of the buffer. This parameter is visible when the **Automatically determine buffer size** check box is cleared.

**Queue duration (seconds)**

Specify the size of the queue in seconds.

**Use default mapping between Data and Device Output Channels**

Select this check box to have the default mapping, where the data from the first channel of audio device is sent to the first channel of the input data, data from second channel of audio device is sent to second channel of data and so on. The maximum number of channels in the input data is determined by the **Number of channels** property.

**Device Output Channels**

Specify the channel mapping. This parameter is visible when the **Use default mapping between Device Input Channels and Data** check box is disabled.

**Output number of samples by which the queue was underrun**

Select this check box to output the number of zero samples inserted into the audio stream due to queue underrun since the last transfer of a frame to the audio device. You can use this value to debug throughput problems and adjust the queues and buffers in your model. To learn how to improve throughput, see “Troubleshooting” on page 2-1393.

**Supported Data Types**

| Port     | Supported Data Types  |
|----------|---|
| Input    | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• 32-bit signed integers</li> <li>• 16-bit signed integers</li> <li>• 8-bit unsigned integers</li> </ul> |
| Underrun | 32-bit unsigned integer   |

**See Also**

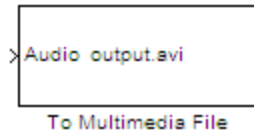
|                    |                    |
|--------------------|--------------------|
| From Audio Device  | DSP System Toolbox |
| To Multimedia File | DSP System Toolbox |
| audioplayer        | MATLAB             |
| sound              | MATLAB             |

**Version History**

**Introduced in R2007b**

## To Multimedia File

Stream video frames and audio samples to multimedia file



## Library

Sinks

dspsnks4

## Description

The To Multimedia File block writes video frames, audio samples, or both to a multimedia (.avi, .wav, .wma, .mp4, .ogg, .flac, or .wmv) file.

You can compress the video frames or audio samples by selecting a compression algorithm. You can connect as many of the input ports as you want. Therefore, you can control the type of video and/or audio the multimedia file receives.

---

**Note** This block supports code generation for platforms that have file I/O available. You cannot use this block with Simulink Desktop Real-Time software, because that product does not support file I/O.

This block performs best on platforms with Version 11 or later of Windows Media Player software. This block supports only uncompressed RGB24 AVI files on Linux and Mac platforms.

Windows 7 UAC (User Account Control), may require administrative privileges to encode WMV and WMA files.

---

The generated code for this block relies on prebuilt library files. You can run this code outside the MATLAB environment, or redeploy it, but be sure to account for these extra library files when doing so. The packNGo function creates a single zip file containing all of the pieces required to run or rebuild this code. See packNGo for more information.

To run an executable file that was generated from a model containing this block, you may need to add precompiled shared library files to your system path. See “Understanding C Code Generation in DSP System Toolbox” for details.

### Cross-Platform Supported File Formats for Audio Files

Audio files can be of the following formats on all platforms:

- WAV
- FLAC

- OGG
- OPUS
- MPEG4 (only on Windows 7 and macOS)

The default format is WAV. This block supports MPEG-4 AAC audio files on Windows 7, and macOS. You can use both M4A and MP4 extensions.

The following platform specific restrictions apply when writing these files:

### Windows 7

- Only sample rates of 44100 and 48000 Hz are supported.
- Only mono or stereo outputs are allowed for MPEG-4 AAC file format. For all other formats, more than two audio output channels are allowed.
- The output data is padded on both the front and back of the signal, with extra samples of silence.

Windows AAC encoder places sharp fade-in and fade-out on audio signal, causing signal to be slightly longer in samples when written to disk.

- A minimum of 1025 samples per channel must be written to the MPEG-4 AAC file.

### macOS

- Only mono or stereo outputs are allowed for MPEG-4 AAC file format. For all other formats, more than two audio output channels are allowed.
- Not all sampling rates are supported, although the Mac Audio Toolbox API do not explicitly specify a restriction.

### Linux

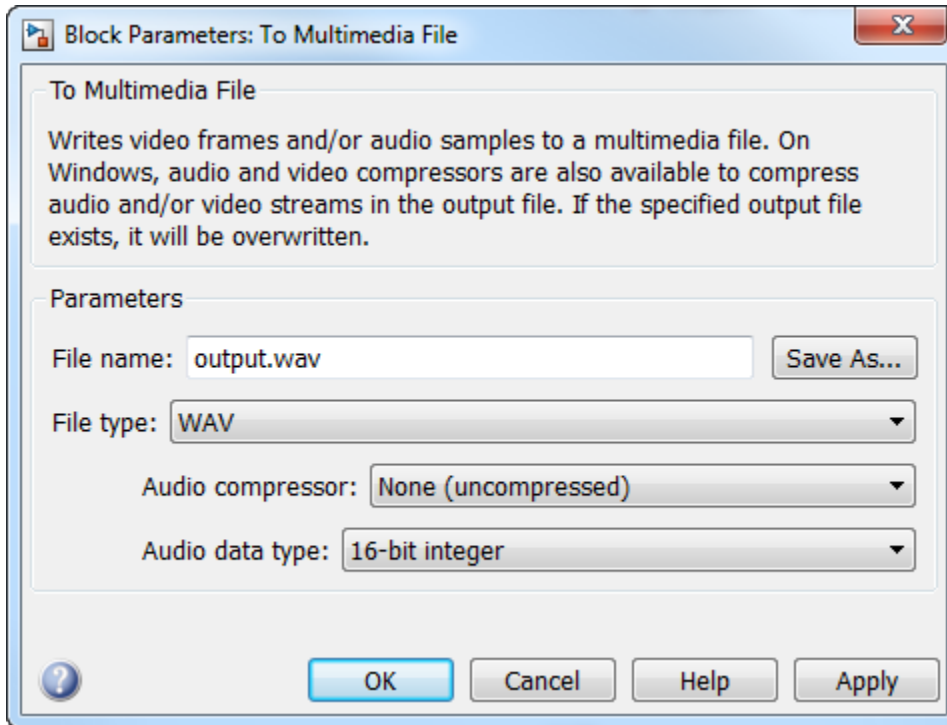
- To support OPUS file format on a Linux machine, you must have `libsndfile 1.0.29` or a later version installed. On Windows and Mac, this file is shipped with MATLAB and is available under `$MATLABROOT/bin/<platform>`.

### Ports

| Port             | Description  |
|------------------|--|
| <b>Image</b>     | $M$ -by- $N$ -by-3 matrix RGB, Intensity, or YCbCr 4:2:2 signal.   |
| <b>R, G, B</b>   | Matrix that represents one plane of the RGB video stream. Inputs to the R, G, or B port must have the same dimensions and data type.   |
| <b>Audio</b>     | $M$ -by- $N$ matrix. $M$ is the number of samples in each channel, and $N$ is the number of channels.  |
| <b>Y, Cb, Cr</b> | Matrix that represents one frame of the YCbCr video stream. The Y, Cb, and Cr ports use the following dimensions:<br>$Y: M \times N$<br>$Cb: M \times \frac{N}{2}$<br>$Cr: M \times \frac{N}{2}$ |

## Dialog Box

The **Main** pane of the To Multimedia File block dialog appears as follows.



### File name

Specify the name of the multimedia file. The block saves the file in your current folder. To specify a different file or location, click the **Save As...** button.

### File type

Specify the file type of the multimedia file. You can select AVI, WAV, MJ2000, WMA, WMV, MPEG4, FLAC, OGG, or OPUS. By default, the **File type** is set to WAV.

### Write

Specify whether the block writes video frames, audio samples, or both to the multimedia file. You can select **Video and audio**, **Video only**, or **Audio only**. This parameter is visible only when you set **File type** to AVI, MPEG4, OGG or OPUS.

### Audio compressor

Select the type of compression algorithm to use to compress the audio data. This compression reduces the size of the multimedia file. Choose **None (uncompressed)** to save uncompressed audio data to the multimedia file.

---

**Note** The other items available in this parameter list are the audio compression algorithms installed on your system. For information about a specific audio compressor, see the documentation for that compressor.

---

**Audio data type**

Select the audio data type. You can use the **Audio data type** parameter only for uncompressed wave files.

**Video compressor**

Select the type of compression algorithm to use to compress the video data. This compression reduces the size of the multimedia file. Choose **None (uncompressed)** to save uncompressed video data to the multimedia file.

---

**Note** The other items available in this parameter list are the video compression algorithms installed on your system. For information about a specific video compressor, see the documentation for that compressor.

---

**Compression Factor (>1)**

Specify the compression factor as an integer scalar greater than 1. This parameter is applicable only when the **File type** is set to MJ2000 and **Video compressor** is set to Lossy. By default, this parameter is set to 10.

**File color format**

Select the color format of the data stored in the file. You can select either RGB or YCbCr 4:2:2.

**Image signal**

Specify how the block accepts a color video signal. If you select **One multidimensional signal**, the block accepts an *M*-by-*N*-by-*P* color video signal, where *P* is the number of color planes, at one port. If you select **Separate color signals**, additional ports appear on the block. Each port accepts one *M*-by-*N* plane of an RGB video stream.

**Video Quality (0-100)**

Quality of the video specified as an integer scalar in the range [0 100]. This parameter is applicable only when **File name** is set to MPEG4 and **Write** is set to **Video only**. By default, this parameter is set to 75.

**Troubleshooting****Running an Executable Outside MATLAB**

To run your generated standalone executable application in Shell, you need to set your environment to the following:

| Platform | Command   |
|----------|---|
| Mac      | <pre>setenv DYLD_LIBRARY_PATH "\$ {DYLD_LIBRARY_PATH}:\$MATLABROOT/bin/ maci64" (csh/tcsh)  export DYLD_LIBRARY_PATH= \$DYLD_LIBRARY_PATH:\$MATLABROOT/bin/ maci64 (Bash)</pre> <p>For more information, see <a href="#">Append library path to "DYLD_LIBRARY_PATH" in MAC</a>.</p> |



| Platform | Command   |
|----------|---|
| Linux    | <pre>setenv LD_LIBRARY_PATH \$ {LD_LIBRARY_PATH}:%MATLABROOT/bin/ glnxa64 (csh/tcsh)  export LD_LIBRARY_PATH= \$LD_LIBRARY_PATH:%MATLABROOT/bin/ glnxa64 (Bash)</pre> |
| Windows  | <pre>set PATH=%PATH%;%MATLABROOT%\bin\win64</pre>   |

## Supported Data Types

For the block to display video data properly, double- and single-precision floating-point pixel values must be between 0 and 1. Any other data type requires the pixel values between the minimum and maximum values supported by their data type.

Check the specific codecs you are using for supported audio rates.

| Port      | Supported Data Types  | Supports Complex Values? |
|-----------|---|--------------------------|
| Image     | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Boolean</li> <li>• 8-, 16- 32-bit signed integers</li> <li>• 8-, 16- 32-bit unsigned integers</li> </ul> | No                       |
| R, G, B   | Same as Image port  | No                       |
| Audio     | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• 16-bit signed integers</li> <li>• 32-bit signed integers</li> <li>• 8-bit unsigned integers</li> </ul>   | No                       |
| Y, Cb, Cr | Same as Image port  | No                       |

## Version History

Introduced before R2006a

### R2022b: Support for OPUS audio file format

The To Multimedia File block supports OPUS file format (.opus).

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Usage notes and limitations:

- Host computer only. Excludes Simulink Desktop Real-Time code generation.
- The executable generated from this block relies on prebuilt dynamic library files (.dll files) included with MATLAB. Use the `packNGo` function to package the code generated from this block and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see “How To Run a Generated Executable Outside MATLAB”.

## **See Also**

### **Blocks**

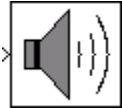
From Multimedia File

### **Topics**

“How To Run a Generated Executable Outside MATLAB”

## To Wave Device (Obsolete)

Send audio data to standard Windows audio device in real time



### Library

dspwin32

### Description

---

**Note** The To Wave Device block is still supported but is likely to be obsoleted in a future release. We strongly recommend replacing this block with the To Audio Device block.

---

The To Wave Device block sends audio data to a standard Windows audio device in real time. It is compatible with most popular Windows hardware, including Sound Blaster cards. The data is sent to the hardware in uncompressed pulse code modulation (PCM) format, and should typically be sampled at one of the standard Windows audio device rates: 8000, 11025, 22050, or 44100 Hz. Some hardware might support other rates in addition to these.

---

**Note** Models that contain both the To Wave Device block and the From Wave Device block require a duplex-capable sound card.

---

The **Use default audio device** check box allows the To Wave Device block to detect and use the system's default audio hardware. You should select this option for systems that have a single sound device installed, or when the default sound device on a multiple-device system is your desired target. When the default sound device is *not* your desired output device, clear **Use default audio device**, and set the desired hardware in the **Audio device** parameter. This parameter lists the names of the installed audio devices.

The block input can contain audio data from a mono or stereo signal. A mono signal is represented as either a sample-based scalar or a frame-based length- $M$  vector, where  $M$  is frame size. A stereo signal is represented as a sample-based length-2 vector or a frame-based  $M$ -by-2 matrix.

When the input data type is `uint8`, the block conveys the signal samples to the audio device using 8 bits. When the input data type is `double`, `single`, `int16`, or fixed point with a word length of 16 and a fraction length of 15, the block conveys the signal samples to the audio device using 16 bits by default. For inputs of data type `double` and `single`, you can also set the block to convey the signal samples using 24 bits by selecting the **Enable 24-bit output for double- and single-precision input signals** check box. The 24-bit sample width requires more memory but in general yields better fidelity.

The amplitude of the input must be in a valid range that depends on the input data type, as shown in the following table. Amplitudes outside the valid range are clipped to the nearest allowable value.

| Input Data Type  | Valid Input Amplitude Range                   |
|--|---|
| double   | $-1 \leq \textit{amplitude} < 1$              |
| single   | $-1 \leq \textit{amplitude} < 1$              |
| int16  | $-32768 \leq \textit{amplitude} \leq 32767$   |
| uint8  | $0 \leq \textit{amplitude} \leq 255$          |
| Fixed point with a word length of 16 and a fraction length of 15 | $-1 \leq \textit{amplitude} \leq 1 - 2^{-15}$ |

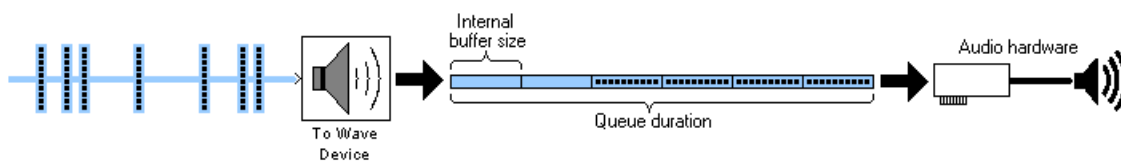
### Buffering

Because audio devices generate real-time audio output, the Simulink environment must maintain a continuous flow of data to a device throughout simulation. Delays in passing data to the audio hardware can result in hardware errors or distortion of the output. This means that the To Wave Device block must in principle supply data to the audio hardware as quickly as the hardware reads the data. However, the To Wave Device block often *cannot* match the throughput rate of the audio hardware, especially when the simulation is running within Simulink rather than as generated code. Simulink execution speed can vary during the simulation as the host operating system services other processes. The block must therefore rely on a buffering strategy to ensure that signal data is available to the hardware on demand.

**Note** This block requires real-time execution of the parent model for best performance.

The following block parameters control the memory management for this block:

- **Queue duration**
- **Automatically determine internal buffer size** or **User-defined internal buffer size**
- **Initial output delay**



The **Queue duration** parameter defines the overall size of the block's buffer. The block reads in chunks of data in the size of the input dimensions and stores them in the buffer. The internal buffer size defines the dimensions of the block output to the hardware. You can define the internal buffer size yourself in the **User-defined internal buffer size parameter**. If you select **Automatically determine internal buffer size** instead, the internal buffer size is calculated for you according to the following rules:

- If the input to the block has a frame size of 32 samples or larger, the internal buffer size be the same as the input frame size.
- If the input to the block has a frame size smaller than 32 samples, the internal buffer size is based on the input sample rate according to the following table, where

$$F_s = \textit{samplingfrequency} = \frac{1}{\textit{samptime}}$$

| $F_s$ (Hz)                 | Internal Buffer Size (samples) |
|----------------------------|--------------------------------|
| $F_s < 8000$               | $\min(64, 2 * F_s)$            |
| $8000 \leq F_s < 22,050$   | 128                            |
| $22,050 \leq F_s < 44,100$ | 256                            |
| $44,100 \leq F_s < 96,000$ | 512                            |
| $F_s \geq 96,000$          | 1024                           |

To minimize the chance of dropouts, the block checks to make sure that the queue duration is at least as big as twice the internal buffer size. If it is not, the queue duration is automatically set to twice the internal buffer size.

The **Initial output delay** parameter enables you to preload the buffer before the block starts to output data to the audio device, which can be helpful for models that do not run in real time. However, for real-time applications, it is best to set the initial output delay to zero (one frame of delay), or as close to zero as possible.

### Troubleshooting

If you are getting undesirable audio output using the To Wave Device block, first determine whether your model can run in real time. Replace the To Wave Device block with a To Wave File block, run the model, and compare the model's simulation stop time to the elapsed time on your watch. If the model simulation stop time is less than the elapsed time on your watch, your model can probably run in real time. Then,

- If your model can run in real time,
  - 1 Select **Automatically determine internal buffer size**. This alone might solve the problem. If not,
  - 2 Try increasing the **Queue duration** parameter to a relatively large value, such as 0.5 s.

If one or both of these options restores desirable audio output, you can try reducing the internal buffer size and/or queue duration until the quality of the audio output again degrades.

- If your model is not running in real time, try to make it run in real time by
  - 1 Optimizing the model (using a more efficient implementation), or
  - 2 Using a Simulink "Acceleration" (Simulink) mode, or
  - 3 Generating stand-alone code

If none of these are possible, but the model only runs for a short period of time, set the **Queue duration** parameter to a size equal to a significant fraction of the model stop time and use a similarly large initial delay. This is not an optimal solution, but might work in some cases.

### Parameters

#### Queue duration (seconds)

Specify the overall buffer size. To minimize the chance of dropouts, the block checks to make sure that the queue duration is at least as large as twice the internal buffer size. If it is not, the queue duration is automatically set to twice the internal buffer size.

**Automatically determine internal buffer size**

Select to have the block automatically select the internal buffer size for you. For details, see “Buffering” on page 2-1404.

**User-defined internal buffer size (samples)**

Define the internal buffer size, or the size of the chunks of data sent by the block to the audio hardware device.

This parameter is only visible when **Automatically determine internal buffer size** is not selected.

**Initial output delay (seconds)**

Specify the amount of time by which to delay the initial output to the audio device. During this time data accumulates in the block's buffer. Any value less than or equal to the queue duration specifies the smallest possible initial delay, which is a single frame.

**Use default audio device**

Select to direct audio output to the system's default audio device.

**Audio device**

This parameter lists the names of the installed audio devices. Specify the name of the audio device to receive the audio output. Select **Use default audio device** when the system has only a single audio card installed.

This parameter is only enabled when the **Use default audio device** check box is not selected.

**Enable 24-bit output for double and single precision input signals**

Select to output 24-bit data when inputs are double- or single-precision. Otherwise, the block outputs 16-bit data for double- and single-precision inputs.

**Supported Data Types**

| Port  | Supported Data Types   |
|-------|--|
| Input | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Signed fixed point with a word length of 16 and a fraction length of 15</li> <li>• 16-bit signed integers</li> <li>• 8-bit unsigned integers</li> </ul> |

**See Also**

|                             |                    |
|-----------------------------|--------------------|
| From Wave Device (Obsolete) | DSP System Toolbox |
| To Wave File (Obsolete)     | DSP System Toolbox |
| audioplayer                 | MATLAB             |
| sound                       | MATLAB             |

**Version History**

**Introduced in R2008b**

## To Wave File (Obsolete)

Write audio data to file in Microsoft Wave (.wav) format



## Library

dspwin32

## Description

---

**Note** The To Wave File block is still supported but is likely to be obsoleted in a future release. We strongly recommend replacing this block with the To Multimedia File block.

---

The To Wave File block streams audio data to a Microsoft Wave (.wav) file in the uncompressed pulse code modulation (PCM) format. For compatibility reasons, the sample rate of the discrete-time input signal should typically be one of the standard Windows audio device rates (8000, 11025, 22050, or 44100 Hz), although the block supports arbitrary rates.

The input to the block,  $u$ , can contain audio data with one or more channels. A signal with  $C$  channels is represented as a sample-based length- $C$  vector or a frame-based  $M$ -by- $C$  matrix. The amplitude of the input should be in the range  $\pm 1$ . Values outside this range are clipped to the nearest allowable value.

```
wavwrite(u,Fs,bits,'filename')    % Equivalent MATLAB code
```

---

**Note** AVI files are the only supported file type for non-Windows platforms.

---

## Parameters

### File name

Specify the path and name of the file to write. Paths can be relative or absolute. You do not need to specify the .wav extension.

### Sample width (bits)

Specify the number of bits used to represent the signal samples in the file. The higher sample width settings require more memory but yield better fidelity for double- and single-precision inputs:

- 8 — Allocates 8 bits to each sample, allowing a resolution of 256 levels
- 16 — Allocates 16 bits to each sample, allowing a resolution of 65536 levels
- 24 — Allocates 24 bits to each sample, allowing a resolution of 16777216 levels

- 32 — Allocates 32 bits to each sample, allowing a resolution of  $2^{32}$  levels ranging from -1 to 1

The 8-, 16-, and 24-bit modes output integer data, while the 32-bit mode outputs single-precision floating-point data.

#### Minimum number of samples for each write to file

Specify the number of consecutive samples,  $L$ , to write with each file access. To reduce the required number of file accesses, the block writes  $L$  consecutive samples to the file during each access for  $L \geq M$ . For  $L < M$ , the block instead writes  $M$  consecutive samples during each access. Larger values of  $L$  result in fewer file accesses, which reduces run-time overhead.

## Supported Data Types

| Port  | Supported Data Types   |
|-------|--|
| Input | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Signed fixed point with a word length of 16 and a fraction length of 15</li> <li>• 16-bit signed integers</li> <li>• 8-bit unsigned integers</li> </ul> |

## See Also

From Multimedia File

DSP System Toolbox

To Audio Device

DSP System Toolbox

To Workspace

Simulink

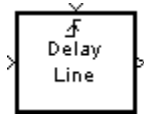
## Version History

Introduced in R2010a



## Triggered Delay Line (Obsolete)

Buffer sequence of inputs into frame-based output



### Library

dspobslib

### Description

---

**Note** The Triggered Delay Line block is still supported but is likely to be obsoleted in a future release. We strongly recommend replacing this block with the Delay Line block.

---

The Triggered Delay Line block acquires a collection of  $M_o$  input samples into a frame, where you specify  $M_o$  in the **Delay line size** parameter. The block buffers a single sample from input 1 whenever it is triggered by the control signal at input 2 (⚡). When the next triggering event occurs, the newly acquired input sample is appended to the output frame so that the new output overlaps the previous output by  $M_o-1$  samples. Between triggering events the block ignores input 1 and holds the output at its last value.

You specify the triggering event at input 2 in the **Trigger type** pop-up menu:

- **Rising edge** triggers execution of the block when the trigger input rises from a negative value to zero or a positive value, or from zero to a positive value.
- **Falling edge** triggers execution of the block when the trigger input falls from a positive value to zero or a negative value, or from zero to a negative value.
- **Either edge** triggers execution of the block when either a rising or falling edge (as described above) occurs.

The Triggered Delay Line block has zero latency, so the new input appears at the output in the same simulation time step. The output frame period is the same as the input sample period,  $T_{fo}=T_{si}$ .

#### Sample-Based Operation

In sample-based operation, the Triggered Delay Line block buffers a sequence of sample-based length- $N$  vector inputs (1-D, row, or column) into a sequence of overlapping sample-based  $M_o$ -by- $N$  matrix outputs, where you specify  $M_o$  in the **Delay line size** parameter ( $M_o>1$ ). That is, each input vector becomes a *row* in the sample-based output matrix. When  $M_o=1$ , the input is simply passed through to the output, and retains the same dimension. Sample-based full-dimension matrix inputs are not accepted.

#### Frame-Based Operation

In frame-based operation, the Triggered Delay Line block rebuffers a sequence of frame-based  $M_f$ -by- $N$  matrix inputs into an sequence of overlapping frame-based  $M_o$ -by- $N$  matrix outputs, where  $M_o$  is

the output frame size specified by the **Delay line size** parameter (that is, the number of consecutive samples from the input frame to rebuffer into the output frame).  $M_o$  can be greater or less than the input frame size,  $M_i$ . Each of the  $N$  input channels is rebuffered independently.

### Initial Conditions

The Triggered Delay Line block's buffer is initialized to the value specified by the **Initial condition** parameter. The block always outputs this buffer at the first simulation step ( $t=0$ ). When the block's output is a vector, the **Initial condition** can be a vector of the same size or a scalar value to be repeated across all elements of the initial output. When the block's output is a matrix, the **Initial condition** can be a matrix of the same size or a scalar to be repeated across all elements of the initial output.

### Parameters

#### Trigger type

The type of event that triggers the block's execution.

#### Delay line size

The length of the output frame (number of rows in output matrix),  $M_o$ .

#### Initial condition

The value of the block's initial output, a scalar, vector, or matrix.

### Supported Data Types

| Port    | Supported Data Types   |
|---------|--|
| Input   | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed and unsigned)</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul> |
| Trigger | <ul style="list-style-type: none"> <li>• Any data type supported by the Trigger block</li> </ul>   |
| Output  | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed and unsigned)</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul> |

### See Also

|            |                    |
|------------|--------------------|
| Buffer     | DSP System Toolbox |
| Delay Line | DSP System Toolbox |

Unbuffer

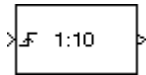
DSP System Toolbox

## **Version History**

**Introduced in R2008b**

## Triggered Signal From Workspace

Import signal samples from MATLAB workspace when triggered



**Libraries:**  
DSP System Toolbox / Signal Operations

### Description

The Triggered Signal From Workspace block imports signal samples from the MATLAB workspace into the Simulink model when triggered by the control signal at the input port (⚡). The **Signal** parameter specifies the name of a MATLAB workspace variable containing the signal to import, or any valid MATLAB expression defining a matrix or 3-D array.

### Ports

#### Input

**trigger** — Trigger signal  
scalar | vector | matrix

Triggering input signal, specified as a scalar, vector, or matrix. This port is labelled with ⚡.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | Boolean | fixed point

#### Output

**Port\_1** — Signal imported from workspace  
scalar | vector | matrix | 3-D array

The block outputs the signal imported from the workspace when triggered by the control signal at the input port.

When the **Signal** parameter specifies an  $M$ -by- $N$  matrix ( $M \neq 1$ ), each of the  $N$  columns is treated as a distinct channel. You specify the frame size in the **Samples per frame** parameter,  $M_0$ . When triggered, the block outputs an  $M_0$ -by- $N$  matrix containing  $M_0$  consecutive samples from each signal channel. For convenience, an imported row vector ( $M=1$ ) is treated as a single channel, so the output dimension is  $M_0$ -by-1.

When the **Signal** parameter specifies an  $M$ -by- $N$ -by- $P$  array, the block generates a single page of the array (an  $M$ -by- $N$  matrix) at each trigger time. The **Samples per frame** parameter must be set to 1.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | Boolean | fixed point

### Parameters

**Signal** — Signal to import

1:10 (default) | MATLAB workspace variable | MATLAB expression

The name of the MATLAB workspace variable from which to import the signal, or a valid MATLAB expression specifying the signal.

When the **Signal** parameter specifies an  $M$ -by- $N$  matrix ( $M \neq 1$ ), each of the  $N$  columns is treated as a distinct channel. You specify the frame size in the **Samples per frame** parameter,  $M_o$ . When triggered, the block outputs an  $M_o$ -by- $N$  matrix containing  $M_o$  consecutive samples from each signal channel. For convenience, an imported row vector ( $M=1$ ) is treated as a single channel, so the output dimension is  $M_o$ -by-1.

When the **Signal** parameter specifies an  $M$ -by- $N$ -by- $P$  array, the block generates a single page of the array (an  $M$ -by- $N$  matrix) at each trigger time. The **Samples per frame** parameter must be set to 1.

**Trigger type** — Type of triggering event

Rising edge (default) | Falling edge | Either edge

The type of event that triggers the block to execute.

- **Rising edge** triggers execution of the block when the trigger input rises from a negative value to zero or a positive value, or from zero to a positive value.
- **Falling edge** triggers execution of the block when the trigger input falls from a positive value to zero or a negative value, or from zero to a negative value.
- **Either edge** triggers execution of the block when either a rising or falling edge occurs.

**Initial output** — Initial output value

0 (default) | scalar | vector | matrix

The value to output until the first trigger event is received. Between trigger events, the block holds the output value constant at its most recent value (that is, no linear interpolation takes place). For single-channel signals, the **Initial output** can be a vector of length  $M_o$  or a scalar to repeat across the  $M_o$  elements of the initial output frames. For matrix outputs ( $M_o$ -by- $N$  or  $M$ -by- $N$ ), the **Initial output** parameter value can be a matrix of the same size or a scalar to be repeated across all elements of the initial output.

**Samples per frame** — Samples per frame

1 (default) | positive integer

The number of samples,  $M_o$ , to buffer into each output frame specified as a positive integer scalar. This value must be 1 when you specify a 3-D array in the **Signal** parameter.

**Form output after final data value by** — Values to output after final imported signal value

Setting to zero (default) | Holding final value | Cyclic repetition

Specifies the output after all of the specified signal samples have been generated.

- When you specify **Setting To Zero**, the block generates zero-valued outputs for the duration of the simulation after generating the last frame of the signal.
- When you specify **Holding Final Value**, the block repeats the final sample for the duration of the simulation after generating the last frame of the signal.

- When you specify **Cyclic Repetition**, the block repeats the signal from the beginning after generating the last frame. When there are not enough samples at the end of the signal to fill the final frame, the block zero-pads the final frame as necessary to ensure that the output for each cycle is identical. For example, the *i*th frame of one cycle contains the same samples as the *i*th frame of any other cycle.

The block does not extrapolate the imported signal beyond the last sample.

## Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | double   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no                                      |
| <b>Multidimensional Signals</b> | no                                      |
| <b>Variable-Size Signals</b>    | no                                      |
| <b>Zero-Crossing Detection</b>  | no                                      |

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

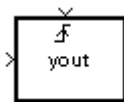
## See Also

### Blocks

Signal From Workspace | To Workspace | Triggered To Workspace

# Triggered To Workspace

Write input sample to MATLAB workspace when triggered



**Libraries:**  
DSP System Toolbox / Sinks

## Description

The Triggered To Workspace block creates a matrix or array variable in the MATLAB workspace, where it stores the acquired inputs at the end of a simulation. The block overwrites an existing variable with the same name.

To save a record of the sample time corresponding to each sample value, open the Configuration Parameters dialog box. To open the Configuration Parameters, in the **Modeling** tab of the Simulink model toolstrip, click **Model Settings**. In the left pane of the window that opens, click **Data Import/Export** and select the **Time** check box.

The nontriggered version of this block is the To Workspace block.

## Ports

### Input

**Input** — Data to log to workspace

scalar | vector | matrix | bus

Specify the data to log to the workspace when the block receives a trigger at the **Trigger** port. The Triggered To Workspace block supports logging real and complex data of any built-in data type or user-defined data types such as buses, enumerations, and fixed-point data.

### Tips

Logging fixed-point data requires a Fixed-Point Designer license.

Data Types: single | double | half | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | Boolean | string | fixed point | enumerated | bus | image

**Trigger** — Trigger signal

scalar | vector | matrix

Specify the trigger signal that controls the execution of the block as a scalar, vector, or a matrix.

Data Types: single | double | half | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | Boolean | fixed point

## Parameters

**Trigger type** — Trigger type

Rising edge (default) | Falling edge | Either edge

The type of event that triggers the execution of the block.

The block acquires and buffers a single frame from input 1 whenever it is triggered by the control signal at input 2 (✚). At all other times, the block ignores input 1. You specify the triggering event at input 2 using the **Trigger type** parameter:

- **Rising edge** triggers execution of the block when the trigger input rises from a negative value to zero or a positive value, or from zero to a positive value.
- **Falling edge** triggers execution of the block when the trigger input falls from a positive value to zero or a negative value, or from zero to a negative value.
- **Either edge** triggers execution of the block for a **Rising edge** or a **Falling edge**.

**Variable name** — Workspace variable name

yout (default) | character vector | string

The name of the workspace variable in which to store the data. By default, the Triggered To Workspace block saves logged data in a variable called `yout`. When a model contains multiple Triggered To Workspace blocks, you must specify a unique value in the **Variable name** parameter of each block.

**Decimation** — Decimation factor

1 (default) | scalar

Specify a decimation factor to reduce the effective sample rate for the logged data. For a decimation factor  $D$  the Triggered To Workspace block logs the value of every  $D^{\text{th}}$  sample. For example, when you specify the **Decimation** value as 2, the Triggered To Workspace block logs every other data point.

For more information about controlling which samples the block logs during simulation, see “Specify Signal Values to Log” (Simulink).

**Maximum number of rows** — Maximum number of rows to be saved

1000 (default) | positive integer

Specify the maximum number of rows (one row per time step) that the block saves ( $P$ ) as a positive integer.

**Save 2-D signals as** — Arrangement of 2-D samples values in logged data

2-D array (concatenate along first dimension) (default) | 3-D array (concatenate along third dimension)

Specify whether the block saves 2-D signals as a 2-D or 3-D array in the MATLAB workspace.

When you select:

- **2-D array (concatenate along first dimension)** — The block vertically concatenates each  $M$ -by- $N$  matrix input with the previous input to produce a 2-D output array. The block saves the data as a  $P$ -by- $N$  matrix, where  $P$  is the value of the **Maximum number of rows** parameter.



When the simulation progresses long enough for the block to acquire more than  $P$  samples, the block stores only the most recent  $P$  samples. The **Decimation factor**  $D$  allows you to store only every  $D^{\text{th}}$  input matrix.

- **3-D array (concatenate along third dimension)** — The block saves an  $M$ -by- $N$  input as a 3-D array in which each  $M$ -by- $N$  page represents a single sample from each of the  $M*N$  channels (the most recent input matrix occupies the last page). The maximum size of this 3-D array is limited to  $M$ -by- $N$ -by- $P$ , where  $P$  is the value of the value of the **Maximum number of rows** parameter.

When the simulation progresses long enough for the block to acquire more than  $P$  inputs, the block stores only the last  $P$  inputs. The **Decimation factor**  $D$  allows you to store only every  $D^{\text{th}}$  input matrix.

**Log fixed-point data as a fi object** — Log fixed-point data as a fi object

off (default) | on

Specify how to log fixed-point input data. By default, the Triggered To Workspace block logs fixed-point input data as double. When you select this parameter, the Triggered To Workspace block logs fixed-point input as a Fixed-Point Designer fi object.

Using the Triggered To Workspace block to log fixed-point data requires a license for Fixed-Point Designer.

## Block Characteristics

|                                 |  |
|---------------------------------|--|
| <b>Data Types</b>               | Boolean   double   enumerated   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no   |
| <b>Multidimensional Signals</b> | no   |
| <b>Variable-Size Signals</b>    | no   |
| <b>Zero-Crossing Detection</b>  | no   |

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

This block can be used for simulation visibility in subsystems that generate HDL code, but is not included in the hardware implementation.

**Fixed-Point Conversion**

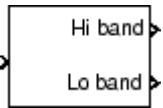
Design and simulate fixed-point systems using Fixed-Point Designer™.

**See Also**

[Signal From Workspace](#) | [To Workspace](#) | [Triggered Signal From Workspace](#) | [Trigger](#)

# Two-Channel Analysis Subband Filter

Decompose signal into high-frequency and low-frequency subbands



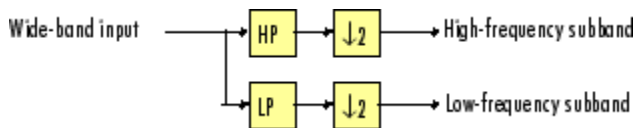
## Libraries:

DSP System Toolbox / Filtering / Multirate Filters

## Description

The Two-Channel Analysis Subband Filter block decomposes the input into high-frequency and low-frequency subbands, each with half the bandwidth and half the sample rate of the input.

The block filters the input with a pair of highpass and lowpass FIR filters, and then downsamples the results by 2, as shown in this figure.



The block implements the FIR filtering and downsampling steps together using a polyphase filter structure, which is more efficient than the filter-then-decimate algorithm shown in the preceding figure. Each subband is the first phase of the respective polyphase filter. You can implement a multilevel dyadic analysis filter bank by connecting multiple copies of this block or by using the Dyadic Analysis Filter Bank block. See “Creating Multilevel Dyadic Analysis Filter Banks” on page 2-1428 for more information.

You must provide a vector of filter coefficients for the lowpass and highpass FIR filters. Each filter should be a half-band filter that passes the frequency band that the other filter stops.

## Ports

### Input

#### Input 1 — Data input

column vector | matrix

Specify the data input as a column vector or a matrix of size  $M$ -by- $N$ . The columns in the input signal represent  $N$  independent channels.

The block supports variable-size input signals (frame length changes during simulation) when you set **Input processing** to Columns as channels (frame based) and **Rate options** to Enforce single-rate processing. When you specify variable-size input signals, they can be of arbitrary frame length, that is, the input frame length does not have to be even. When you specify fixed-size signals, the frame length can be arbitrary under certain conditions. For more details, see “Frame-Based Processing” on page 2-1425 and “Sample-Based Processing” on page 2-1427.

When the input is fixed point, it must be signed only.

Data Types: single | double | int8 | int16 | int32 | fixed point

## Output

**Hi band** — High-frequency subband  
column vector | matrix

The block outputs the high-frequency subband as a column vector or a matrix.

When you set **Rate options** to:

- **Enforce single-rate processing** -- The block maintains the input sample rate and decimates the signal by decreasing the output frame size by a factor of 2.

The output has an upper bound size of  $\text{ceil}(M/2)$ -by- $N$  for an input of size  $M$ -by- $N$ .

- **Allow multirate processing** -- The block decimates the signal such that the output sample rate is half the input sample rate.

The output frame size is the same as the input frame size.

For more details, see “Frame-Based Processing” on page 2-1425 and “Sample-Based Processing” on page 2-1427.

Data Types: single | double | int8 | int16 | int32 | fixed point

**Lo band** — Low-frequency subband  
column vector | matrix

The block outputs the low-frequency subband as a column vector or a matrix.

When you set **Rate options** to:

- **Enforce single-rate processing** -- The block maintains the input sample rate and decimates the signal by decreasing the output frame size by a factor of 2.

The output has an upper bound size of  $\text{ceil}(M/2)$ -by- $N$  for an input of size  $M$ -by- $N$ .

- **Allow multirate processing** -- The block decimates the signal such that the output sample rate is half the input sample rate.

The output frame size is the same as the input frame size.

For more details, see “Frame-Based Processing” on page 2-1425 and “Sample-Based Processing” on page 2-1427.

Data Types: single | double | int8 | int16 | int32 | fixed point

## Parameters

### Main Tab

**Lowpass FIR filter coefficients** — Lowpass FIR filter coefficients

[0.0352 -0.0854 -0.1350 0.4599 0.8069 0.3327] (default) | row vector

Specify a vector of lowpass FIR filter coefficients, in descending powers of  $z$ . The lowpass filter should be a half-band filter that passes the frequency band stopped by the filter specified in the **Highpass FIR filter coefficients** parameter. The default values of this parameter specify a filter

based on a third-order Daubechies wavelet. When you use the Two-Channel Synthesis Subband Filter block to reconstruct the input to this block, you need to design perfect reconstruction filters to use in the synthesis subband filter. For more information, see “Specify FIR Filters” on page 2-1425.

#### **Highpass FIR filter coefficients** — Highpass FIR filter coefficients

`[-0.3327 0.8069 -0.4599 -0.1350 0.0854 0.0352]` (default) | row vector

Specify a vector of highpass FIR filter coefficients, in descending powers of  $z$ . The highpass filter should be a half-band filter that passes the frequency band stopped by the filter specified in the **Lowpass FIR filter coefficients** parameter. The default values of this parameter specify a filter based on a third-order Daubechies wavelet. When you use the Two-Channel Synthesis Subband Filter block to reconstruct the input to this block, you need to design perfect reconstruction filters to use in the synthesis subband filter. For more information, see “Specify FIR Filters” on page 2-1425.

#### **Input processing** — Input processing

`Columns as channels (frame based)` (default) | `Elements as channels (sample based)`

Specify how the block should process the input. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` (default) — When you select this option, the block treats each column of the input as a separate channel.
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

For more information, see “Frame-Based Processing” on page 2-1425 and “Sample-Based Processing” on page 2-1427.

#### **Rate options** — Rate options

`Enforce single-rate processing` (default) | `Allow multirate processing`

Specify the rate processing rule for the block. You can set this parameter to one of the following options:

- `Enforce single-rate processing` — When you select this option, the block treats each column of the input as an independent channel and decomposes each channel over time. The output has the same sample rate as the input, but the output frame size is half that of the input frame size. To select this option, you must set the **Input processing** parameter to `Columns as channels (frame based)`.
- `Allow multirate processing` — When you select this option, the input and output of the block are the same size, but the sample rate of the output is half that of the input.

Some settings of this parameter cause the block to have nonzero latency. See “Latency” on page 2-1427 for more information.

#### **Allow arbitrary frame length for fixed-size input signals** — Allow arbitrary frame length for fixed-size input signals

`off` (default) | `on`

Specify whether fixed-size input signals (whose size does not change during simulation) can have an arbitrary frame length, where the frame length does not have to be even. The block uses this parameter only for fixed-size input signals and ignores it if the input has a variable-size.

When the input signal is a variable-size signal, the signal can have an arbitrary frame length, that is, the frame length does not have to be an even number.

For fixed-size input signals, if you:

- Select the **Allow arbitrary frame length for fixed-size input signals** parameter, the frame length of the signal does not have to be even. The output is a variable-size signal.
- Clear the **Allow arbitrary frame length for fixed-size input signals** parameter, the input frame length must be even.

### Dependency

To enable this parameter, set **Input processing** to Columns as channels (frame based) and **Rate options** to Enforce single-rate processing.

### Data Types Tab

**Rounding mode** — Rounding mode for fixed-point operations

Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Select the rounding mode for fixed-point operations. The filter coefficients do not obey this parameter; they are always rounded to Nearest.

---

**Note** The **Rounding mode** and **Saturate on integer overflow** settings have no effect on numerical results when all the following conditions exist:

- **Product output** is Inherit: Inherit via internal rule
- **Accumulator** is Inherit: Inherit via internal rule
- **Output** is Inherit: Same as accumulator

With these data type settings, the block effectively operates in full-precision mode.

---

**Saturate on integer overflow** — Saturate on integer overflow

off (default) | on

When you select this parameter, the block saturates the result of its fixed-point operation. When you clear this parameter, the block wraps the result of its fixed-point operation. For details on saturate and wrap, see overflow mode for fixed-point operations.

---

**Note** The **Rounding mode** and **Saturate on integer overflow** parameters have no effect on numeric results when all these conditions are met:

- **Product output** data type is Inherit: Inherit via internal rule.
- **Accumulator** data type is Inherit: Inherit via internal rule.

With these data type settings, the block operates in full-precision mode.


---

**Coefficients** — Data type of the coefficients

Inherit: Same word length as input (default) | `fixd(1,16)` | `fixd(1,16,0)`

Specify the coefficients data type. See “Fixed-Point Data Types” on page 2-1429 and “Multiplication Data Types” for illustrations depicting the use of the coefficients data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same word length as input`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Coefficients** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in *Simulink User's Guide* (Simulink) for more information.

**Coefficients Minimum** — Minimum value of filter coefficients

`[]` (default) | scalar

Specify the minimum value of the filter coefficients. The default value is `[]` (unspecified). Simulink software uses this value to perform automatic scaling of fixed-point data types.

**Coefficients Maximum** — Maximum value of filter coefficients

`[]` (default) | scalar

Specify the maximum value of the filter coefficients. The default value is `[]` (unspecified). Simulink software uses this value to perform automatic scaling of fixed-point data types.

**Product output** — Product output data type

Inherit: `Inherit via internal rule` (default) | `Inherit: Same as input` | `fixd(1,16,0)`

Specify the product output data type. See “Fixed-Point Data Types” on page 2-1429 and “Multiplication Data Types” for illustrations depicting the use of the product output data type in this block. You can set it to:


- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.

---

**Note** The actual product output word length can be equal to or greater than the calculated ideal product output word length, depending on the settings on the **Hardware Implementation** pane of the Configuration Parameters dialog box.

---

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Product output** parameter.


See “Specify Data Types Using Data Type Assistant” (Simulink) in *Simulink User's Guide* (Simulink) for more information.

**Accumulator** — Data type of accumulator

Inherit: `Inherit via internal rule (default)` | `Inherit: Same as input` | `Inherit: Same as product output` | `fixd(1,16,0)`

Specify the accumulator data type. See “Fixed-Point Data Types” on page 2-1429 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Accumulator** parameter.


See “Specify Data Types Using Data Type Assistant” (Simulink) in *Simulink User's Guide* (Simulink) for more information.

**Output** — Data type of output

Inherit: `Same as accumulator (default)` | `Inherit: Same as input` | `Inherit: Same as product output` | `fixd(1,16,0)`

Specify the output data type. See “Fixed-Point Data Types” on page 2-1429 for illustrations depicting the use of the output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as accumulator`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Output** parameter.

See “Control Data Types of Signals” (Simulink) in *Simulink User's Guide* (Simulink) for more information.

**Output Minimum** — Output minimum

`[]` (default) | scalar

Specify the minimum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Specify Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

**Output Maximum** — Output maximum

`[]` (default) | scalar

Specify the maximum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Specify Signal Ranges” (Simulink))



- Automatic scaling of fixed-point data types

**Lock data type settings against changes by the fixed-point tools** — Prevent fixed-point tools from overriding data types

off (default) | on

Select this parameter to prevent the fixed-point tools from overriding the data types you specify in the block dialog box.

## Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | double   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no                                      |
| <b>Multidimensional Signals</b> | no                                      |
| <b>Variable-Size Signals</b>    | yes                                     |
| <b>Zero-Crossing Detection</b>  | no                                      |

## More About

### Specify FIR Filters

You must provide the vector of numerator coefficients for the lowpass and highpass filters in the **Lowpass FIR filter coefficients** and **Highpass FIR filter coefficients** parameters.

For example, to specify a filter with the following transfer function, enter the vector `[b(1) b(2) ... b(m)]`.

$$H(z) = B(z) = b_1 + b_2z^{-1} + \dots + b_mz^{-(m-1)}$$

Each filter should be a half-band filter that passes the frequency band that the other filter stops. You can use the Two-Channel Synthesis Subband Filter block to reconstruct the input to this block. To do so, you must design perfect reconstruction filters to use in the synthesis subband filter.

The best way to design perfect reconstruction filters is to use the Wavelet Toolbox `wfilters` function to design both the filters both in this block and in the Two-Channel Synthesis Subband Filter block. You can also use other DSP System Toolbox and Signal Processing Toolbox functions.

The Two-Channel Analysis Subband Filter block initializes all filter states to zero.

### Frame-Based Processing

When you set the **Input processing** parameter to `Columns as channels (frame based)`, the block accepts an  $M$ -by- $N$  matrix. The block treats each column of the input as the high- or low-frequency subbands of the corresponding output channel. You can use the **Rate options** parameter to specify how the block resamples the input:

- When you set the **Rate options** parameter to `Enforce single-rate processing`, the input and the two outputs of the block have the same sample rate. The block treats each column of the

input as an independent channel and decomposes each channel over time. The block outputs two matrices, where each column of the output is the high- or low-frequency subband of the corresponding input column. The frame length of each output has an upper bound size of  $\text{ceil}(M/2)$ .

In this mode, if you input a fixed-size signal (frame length does not change during simulation) and select the **Allow arbitrary frame length for fixed-size input signals** parameter, the input frame length can be arbitrary and does not have to be an even number. If you clear the **Allow arbitrary frame length for fixed-size input signals** parameter, the input frame length must be an even number.

In this mode, if you input a variable-size signal (frame length changes during simulation), the **Allow arbitrary frame length for fixed-size input signals** parameter appears on the block dialog box but does not have any impact on the input frame length. You can input a variable-size signal of any frame length even if you do not select the **Allow arbitrary frame length for fixed-size input signals** parameter.

This table summarizes the support for arbitrary input frame length when you set **Input processing** to Columns as channels (frame based) and **Rate options** to Enforce single-rate processing.

| Input Signal               | Block Support for this Signal | Support for Arbitrary Input Frame Length   | Input Size   | Output Size                                     |
|----------------------------|-------------------------------|--|--------------|---|
| Fixed-size input signal    | Yes                           | When you select <b>Allow arbitrary frame length for fixed-size input signals</b> | $M$ -by- $N$ | Upper bound size of $\text{ceil}(M/2)$ -by- $N$ |
| Variable-size input signal | Yes                           | Always   | $M$ -by- $N$ | Upper bound size of $\text{ceil}(M/2)$ -by- $N$ |

- When you set the **Rate options** parameter to Allow multirate processing, the input and the two outputs of the Two-Channel Analysis Subband Filter block are of the same size, but the sample rate of the output is half that of the input. In this mode, the block treats a  $M$ -by- $N$  matrix input as  $N$  independent channels and decomposes each channel over time. The block outputs two  $M$ -by- $N$  matrices, where each column of the output is the high- or low-frequency subband of the corresponding input column.

In this mode, the block accepts only fixed-size signals and these signals can have arbitrary frame length. That is, the frame length does not have to be even.

This table summarizes the support for arbitrary input frame length when you set **Input processing** to Columns as channels (frame based) and **Rate options** to Allow multirate processing.

| Input Signal            | Block Support for this Signal | Support for Arbitrary Input Frame Length | Input Size   | Output Size  |
|-------------------------|-------------------------------|--|--------------|--------------|
| Fixed-size input signal | Yes                           | Always                                   | $M$ -by- $N$ | $M$ -by- $N$ |

| Input Signal               | Block Support for this Signal | Support for Arbitrary Input Frame Length | Input Size     | Output Size    |
|----------------------------|-------------------------------|--|----------------|----------------|
| Variable-size input signal | No                            | Not applicable                           | Not applicable | Not applicable |

In this mode, the block has one frame of latency, as described in the “Latency” on page 2-1427 section.

### Sample-Based Processing

When you set the **Input processing** parameter to `Elements as channels (sample based)`, the block treats an  $M$ -by- $N$  matrix input as  $M \cdot N$  independent channels. The block decomposes each channel over time and outputs two  $M$ -by- $N$  matrices whose sample rates are half the input sample rate. Each element in the output matrix is the high- or low-frequency subband output of the corresponding element of the input matrix.

In this mode, the block accepts only fixed-size signals and these signals can have an arbitrary frame length.

This table summarizes the support for arbitrary input frame length when you set **Input processing** to `Elements as channels (sample based)`. **Rate options** is automatically set to `Allow multirate processing`.

| Input Signal               | Block Support for this Signal | Support for Arbitrary Input Frame Length | Input Size     | Output Size    |
|----------------------------|-------------------------------|--|----------------|----------------|
| Fixed-size input signal    | Yes                           | Always                                   | $M$ -by- $N$   | $M$ -by- $N$   |
| Variable-size input signal | No                            | Not applicable                           | Not applicable | Not applicable |

Depending on the setting of your Simulink configuration parameters, the output may have one sample of latency, as described in the “Latency” on page 2-1427 section.

### Latency

When you set the **Input processing** parameter to `Columns as channels (frame based)` and the **Rate options** parameter to `Enforce single-rate processing`, the Two-Channel Analysis Subband Filter block always has zero-tasking latency. Zero-tasking latency means that the block propagates the first input sample (received at time  $t=0$ ) as the first output sample.

When you set the **Rate options** parameter to `Allow multirate processing`, the Two-Channel Analysis Subband Filter block may exhibit latency. The amount of latency depends on the setting of the **Input processing** parameter of this block, and the setting of the Simulink **Treat each discrete rate as a separate task** configuration parameter. The following table summarizes the conditions that produce latency when the block is performing multirate processing.

| Input processing                    | Treat each discrete rate as a separate task | Latency  |
|-------------------------------------|---|--|
| Elements as channels (sample based) | Off   | None.  |
|                                     | On  | One sample. The first output sample in each channel always has a value of 0. |
| Columns as channels (frame based)   | On or Off                                   | One frame. All samples in the first output frame have a value of 0.          |

---

**Note** For more information on latency and the Simulink tasking modes, see “Excess Algorithmic Delay (Tasking Latency)” and “Time-Based Scheduling and Code Generation” (Simulink Coder).

---

### Creating Multilevel Dyadic Analysis Filter Banks

The Two-Channel Analysis Subband Filter block is the basic unit of a dyadic analysis filter bank. You can connect several of these blocks to implement an  $n$ -level filter bank, as illustrated in the following figure. For a review of dyadic analysis filter banks, see the Dyadic Analysis Filter Bank block reference page.

When you create a filter bank by connecting multiple copies of this block, the output values of the filter bank differ depending on whether there is latency. Though the output values differ, both sets of values are valid; the difference arises from changes in latency. See the “Latency” on page 2-1427 section for more information about when latency can occur in the Two-Channel Analysis Subband Filter block.

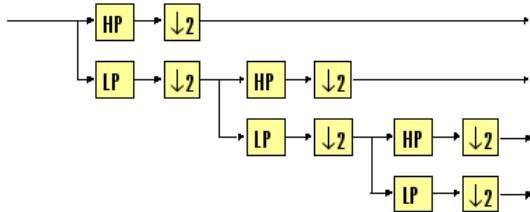
In some cases, rather than connecting several Two-Channel Analysis Subband Filter blocks, you can use the Dyadic Analysis Filter Bank block, which is faster and requires lesser memory. In particular, the Dyadic Analysis Filter Bank block is more efficient under the following conditions:

- The frame size of the signal you are decomposing is a multiple of  $2^n$ .
- You are decomposing the signal into  $n+1$  or  $2^n$  subbands.

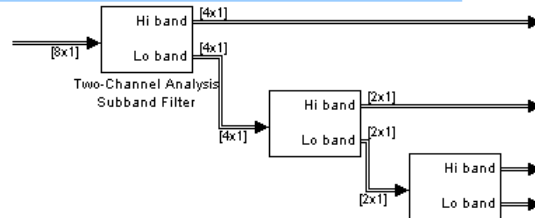
In all other cases, use Two-Channel Analysis Subband Filter blocks to implement your filter banks.

### 3-Level Dyadic Analysis Filter Banks

#### Conceptual illustration



#### Two-Channel Analysis Subband Filter block implementation

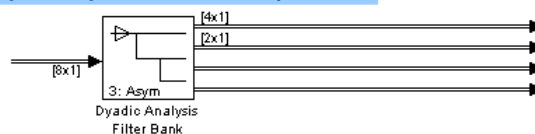


Both implementations of the dyadic analysis filter bank decompose a frame-based signal with frame size a multiple of  $2^n$  into  $n+1$  subbands, where  $n = 3$ .

In this case, the Dyadic Analysis Filter Bank block's implementation is more efficient.

Use the Two-Channel Analysis Subband Filter block implementation for other cases, such as to handle sample-based inputs, or to handle frame-based inputs whose frame size is not a multiple of  $2^n$ .

#### Dyadic Analysis Filter Bank block implementation

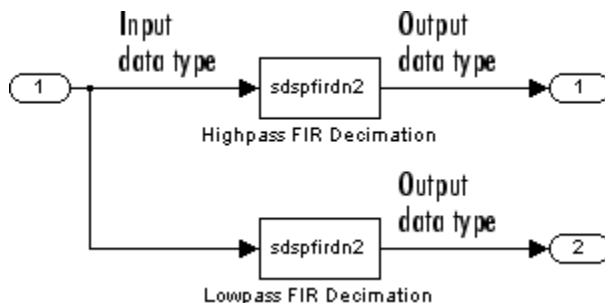


The Dyadic Analysis Filter Bank block allows you to specify the filter bank filters by providing vectors of filter coefficients, just as this block does. The Dyadic Analysis Filter Bank block provides an additional option of using wavelet-based filters that the block designs by using a wavelet you specify.

### Fixed-Point Data Types

Fixed-point data types in the Two-Channel Analysis Subband Filter Bank block.

The Two-Channel Analysis Subband Filter Bank block is composed of two FIR Decimation blocks as shown in the following diagram.



For fixed-point signals, you can set the coefficient, product output, accumulator, and output data types of the FIR Decimation blocks as discussed in "Parameters" on page 2-1420. For a diagram showing the usage of these data types, see the FIR Decimation block reference page.

## Version History

Introduced before R2006a

**R2023a: Support for arbitrary input frame length**

This block supports input signals with arbitrary frame lengths when the:

- Input signal is a fixed-size signal (frame length does not change during simulation) and the block allows for multirate processing.
- Input signal is a fixed-size signal, the block enforces single-rate processing, and you select the **Allow arbitrary frame length for fixed-size input signals** parameter.
- Input signal is a variable-size signal (frame length changes during simulation).

When this block supports an input signal with an arbitrary frame length, the input frame length does not have to be even.

For more details, see “Frame-Based Processing” on page 2-1425 and “Sample-Based Processing” on page 2-1427.

## References

- [1] Fliege, N. J. *Multirate Digital Signal Processing: Multirate Systems, Filter Banks, Wavelets* . West Sussex, England: John Wiley & Sons, 1994.
- [2] Strang, G. and T. Nguyen. *Wavelets and Filter Banks* . Wellesley, MA: Wellesley-Cambridge Press, 1996.
- [3] Vaidyanathan, P. P. *Multirate Systems and Filter Banks* . Englewood Cliffs, NJ: Prentice Hall, 1993.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Generated code relies on the `memcpy` or `memset` function (`string.h`) under certain conditions.

### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

If the input is fixed point, it must be a signed integer or a signed fixed point value with a power-of-two slope and zero bias.

## See Also

### Functions

`fir1` | `fir2` | `firls`

### Blocks

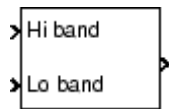
DWT | Dyadic Analysis Filter Bank | FIR Decimation | IDWT | Two-Channel Synthesis Subband Filter

### Topics

“Multirate and Multistage Filters”

# Two-Channel Synthesis Subband Filter

Reconstruct signal from high-frequency and low-frequency subbands



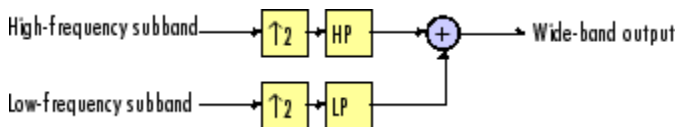
## Libraries:

DSP System Toolbox / Filtering / Multirate Filters

## Description

The Two-Channel Synthesis Subband Filter block reconstructs a signal from its high-frequency and low-frequency subbands, each with half the bandwidth and half the sample rate of the original signal. Use this block to reconstruct signals decomposed by the Two-Channel Analysis Subband Filter block.

The block upsamples the high- and low-frequency subbands by 2, and then filters the results with a pair of highpass and lowpass FIR filters, as shown in this figure.



The block implements the FIR filtering and downsampling steps together using a polyphase filter structure, which is more efficient than the interpolate-then-filter algorithm shown in the preceding figure. You can implement a multilevel dyadic synthesis filter bank by connecting multiple copies of this block or by using the Dyadic Synthesis Filter Bank block. For more information, see “Creating Multilevel Dyadic Synthesis Filter Banks” on page 2-1438.

You must provide a vector of filter coefficients for the lowpass and highpass FIR filters. Each filter should be a half-band filter that passes the frequency band that the other filter stops. You can use this block to reconstruct the output of a Two-Channel Analysis Subband Filter block. To do so, you must design the filters in this block such that they perfectly reconstruct the outputs of the analysis filters.

## Ports

### Input

**Hi band** — High-frequency subband  
vector | matrix

Specify the high-frequency subband as a vector or a matrix. Both the high-frequency subband and the low-frequency subbands must have the same size and data type.

When the input is fixed-point, it must be signed only.

Data Types: single | double | int8 | int16 | int32 | fixed point

**Lo band** — Low-frequency subband  
vector | matrix

Specify the low-frequency subband as a vector or a matrix. Both the high-frequency subband and the low-frequency subbands must have the same size and data type.

When the input is fixed-point, it must be signed only.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `fixed point`

## Output

**Output 1** — Synthesized output

`vector` | `matrix`

The block returns the synthesized output as a vector or a matrix. The number of rows in the output is the sum of the number of rows of the input signals.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `fixed point`

## Parameters

### Main Tab

**Lowpass FIR filter coefficients** — Lowpass FIR filter coefficients

[0.3327 0.8069 0.4599 -0.1350 -0.0854 0.0352] (default)

A vector of lowpass FIR filter coefficients, in descending powers of  $z$ . The lowpass filter should be a half-band filter that passes the frequency band stopped by the filter specified in the **Highpass FIR filter coefficients** parameter. To use this block to reconstruct the output of a Two-Channel Analysis Subband Filter block, you must design the filters in this block to perfectly reconstruct the outputs of the analysis filters. For more information, see “Specify FIR Filters” on page 2-1436.

**Highpass FIR filter coefficients** — Highpass FIR filter coefficients

[0.0352 0.0854 -0.1350 -0.4599 0.8069 -0.3327] (default)

A vector of highpass FIR filter coefficients, in descending powers of  $z$ . The highpass filter should be a half-band filter that passes the frequency band stopped by the filter specified in the **Lowpass FIR filter coefficients** parameter. To use this block to reconstruct the output of a Two-Channel Analysis Subband Filter block, you must design the filters in this block to perfectly reconstruct the outputs of the analysis filters. For more information, see “Specify FIR Filters” on page 2-1436.

**Input processing** — Input processing

`Columns as channels (frame based)` (default) | `Elements as channels (sample based)`

Specify how the block should process the input. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` (default) — When you select this option, the block treats each column of the input as a separate channel.
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

For more information, see “Frame-Based Processing” on page 2-1436 and “Sample-Based Processing” on page 2-1437.



**Rate options** — Rate options

Enforce single-rate processing (default) | Allow multirate processing

Specify the rate processing rule for the block. You can set this parameter to one of the following options:

- **Enforce single-rate processing** — When you select this option, the block treats each column of the input as an independent channel and reconstructs each channel over time. The output has the same sample rate as the input, but the output frame size is twice that of the input frame size. To select this option, you must set the **Input processing** parameter to `Columns as channels (frame based)`.
- **Allow multirate processing** — When you select this option, the input and output of the block are the same size, but the sample rate of the output is twice that of the input.

Some settings of this parameter cause the block to have nonzero latency. See “Latency” on page 2-1437 for more information.

**Data Types Tab****Rounding mode** — Rounding mode

Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Select the rounding mode for fixed-point operations. The filter coefficients do not obey this parameter; they always round to **Nearest**.

---

**Note** The **Rounding mode** and **Saturate on integer overflow** settings have no effect on numerical results when all the following conditions exist:

- **Product output** is `Inherit`: `Inherit via internal rule`
- **Accumulator** is `Inherit`: `Inherit via internal rule`
- **Output** is `Inherit`: `Same as accumulator`

With these data type settings, the block effectively operates in full-precision mode.

---

**Saturate on integer overflow** — Saturate on integer overflow

off (default) | on

When you select this parameter, the block saturates the result of its fixed-point operation. When you clear this parameter, the block wraps the result of its fixed-point operation. For details on saturate and wrap, see overflow mode for fixed-point operations.

---

**Note** The **Rounding mode** and **Saturate on integer overflow** parameters have no effect on numeric results when all these conditions are met:

- **Product output** data type is `Inherit`: `Inherit via internal rule`.
- **Accumulator** data type is `Inherit`: `Inherit via internal rule`.

With these data type settings, the block operates in full-precision mode.


---

**Coefficients** — Data type of the coefficients

Inherit: Same word length as input (default) | `fixd(1,16)` | `fixd(1,16,0)`

Specify the coefficients data type. See “Fixed-Point Data Types” on page 2-1438 and “Multiplication Data Types” for illustrations depicting the use of the coefficients data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same word length as input`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Coefficients** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in *Simulink User's Guide* (Simulink) for more information.

**Coefficients Minimum** — Minimum value of filter coefficients

`[]` (default) | scalar

Specify the minimum value of the filter coefficients. The default value is `[]` (unspecified). Simulink software uses this value to perform automatic scaling of fixed-point data types.

**Coefficients Maximum** — Maximum value of filter coefficients

`[]` (default) | scalar

Specify the maximum value of the filter coefficients. The default value is `[]` (unspecified). Simulink software uses this value to perform automatic scaling of fixed-point data types.

**Product output** — Product output data type

Inherit: `Inherit via internal rule` (default) | `Inherit: Same as input` | `fixd(1,16,0)`

Specify the product output data type. See “Fixed-Point Data Types” on page 2-1438 and “Multiplication Data Types” for illustrations depicting the use of the product output data type in this block. You can set it to:


- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.

---

**Note** The actual product output word length can be equal to or greater than the calculated ideal product output word length, depending on the settings on the **Hardware Implementation** pane of the Configuration Parameters dialog box.

---

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Product output** parameter.


See “Specify Data Types Using Data Type Assistant” (Simulink) in *Simulink User's Guide* (Simulink) for more information.

**Accumulator** — Data type of accumulator

Inherit: `Inherit via internal rule (default)` | Inherit: `Same as input` | Inherit: `Same as product output` | `fixd(1,16,0)`

Specify the accumulator data type. See “Fixed-Point Data Types” on page 2-1438 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Accumulator** parameter.


See “Specify Data Types Using Data Type Assistant” (Simulink) in *Simulink User's Guide* (Simulink) for more information.

**Output** — Data type of output

Inherit: `Same as accumulator (default)` | Inherit: `Same as input` | Inherit: `Same as product output` | `fixd(1,16,0)`

Specify the output data type. See “Fixed-Point Data Types” on page 2-1438 for illustrations depicting the use of the output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as accumulator`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Output** parameter.

See “Control Data Types of Signals” (Simulink) in *Simulink User's Guide* (Simulink) for more information.

**Output Minimum** — Output Minimum

`[]` (default) | scalar

Specify the minimum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Specify Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

**Output Maximum** — Output Maximum

`[]` (default) | scalar

Specify the maximum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Specify Signal Ranges” (Simulink))

- Automatic scaling of fixed-point data types

**Lock data type settings against changes by the fixed-point tools** — Prevent fixed-point tools from overriding data types

off (default) | on

Select this parameter to prevent the fixed-point tools from overriding the data types you specify in the block dialog box.

## Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | double   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no                                      |
| <b>Multidimensional Signals</b> | no                                      |
| <b>Variable-Size Signals</b>    | yes                                     |
| <b>Zero-Crossing Detection</b>  | no                                      |

## More About

### Specify FIR Filters

You must provide the vector of numerator coefficients for the lowpass and highpass filters in the **Lowpass FIR filter coefficients** and **Highpass FIR filter coefficients** parameters.

For example, to specify a filter with the following transfer function, enter the vector [b(1) b(2) ... b(m)].

$$H(z) = B(z) = b_1 + b_2z^{-1} + \dots + b_mz^{-(m-1)}$$

Each filter should be a half-band filter that passes the frequency band that the other filter stops. You can use this block to reconstruct the output of a Two-Channel Analysis Subband Filter block. To do so, you must design the filters in this block such that they perfectly reconstruct the outputs of the analysis filters.

The best way to design perfect reconstruction filters is to use the Wavelet Toolbox `wfilters` function for the filters in both this block *and* in the corresponding Two-Channel Analysis Subband Filter block. You can also use DSP System Toolbox and Signal Processing Toolbox functions.

The Two-Channel Synthesis Subband Filter block initializes all filter states to zero.

### Frame-Based Processing

When you set the **Input processing** parameter to Columns as channels (frame based), the block accepts any two  $M$ -by- $N$  matrices with the same frame rates. The block treats each column of the input as the high- or low-frequency subbands of the corresponding output channel. You can use the **Rate options** parameter to specify how the block resamples the input:

- When you set the **Rate options** parameter to Enforce single-rate processing, the input to the block can be any two  $M$ -by- $N$  matrices with the same frame rate. The block treats each input

column as the high- or low-frequency subbands of the corresponding output channel. The input to the topmost input port should contain the high-frequency subbands. The block outputs one matrix, where each column is reconstructed from the corresponding columns of each input matrix. The input and output frame *rates* are the same, but the frame *size* of the output is twice that of the input.

- When you set the **Rate options** parameter to `Allow multirate processing`, the block treats each column of the input as the high- or low-frequency subbands of the corresponding output channel. The input to the topmost input port should contain the high-frequency subbands. The block outputs one matrix, where each column is reconstructed from the corresponding columns of the input matrices. The input and output frame *sizes* are the same, but the frame *rate* of the output is twice that of the input. Thus, the overall sample rate of the output is twice that of the input sample rate.

In this mode, the block has one frame of latency, as described in the “Latency” on page 2-1437 section.

### Sample-Based Processing

When you set the **Input processing** parameter to `Elements as channels (sample based)`, the block accepts any two  $M$ -by- $N$  matrices with the same sample rates. The block treats each  $M$ -by- $N$  matrix as  $M \cdot N$  independent subbands. Each element of the input matrices is the high- or low-frequency subband of the corresponding channel in the output matrix. The input to the topmost input port should contain the high-frequency subbands. The block outputs one matrix with the same dimensions as the input matrices, but a sample rate that is twice that of the input. The block reconstructs each element of the output from the corresponding elements in the input matrices.

Depending on the setting of your Simulink configuration parameters, the output may have one sample of latency, as described in the “Latency” on page 2-1437 section.

### Latency

When you set the **Input processing** parameter to `Columns as channels (frame based)` and the **Rate options** parameter to `Enforce single-rate processing`, the Two-Channel Synthesis Subband Filter block always has zero-tasking latency. Zero-tasking latency means that the block propagates the first input sample (received at time  $t=0$ ) as the first output sample.

When you set the **Rate options** parameter to `Allow multirate processing`, the Two-Channel Synthesis Subband Filter block may exhibit latency. The amount of latency depends on the setting of the **Input processing** parameter of this block and the setting of the Simulink **Treat each discrete rate as a separate task** configuration parameter. The following table summarizes the conditions that produce latency when the block is performing multirate processing.

| Input processing                    | Treat each discrete rate as a separate task | Latency  |
|-------------------------------------|---|--|
| Elements as channels (sample based) | Off   | None.  |
|                                     | On  | One sample. The first output sample in each channel always has a value of 0. |
| Columns as channels (frame based)   | Off or On                                   | One frame. All samples in the first output frame have a value of 0.          |

**Note** For more information on latency and the Simulink tasking modes, see “Excess Algorithmic Delay (Tasking Latency)” and “Time-Based Scheduling and Code Generation” (Simulink Coder).

## Creating Multilevel Dyadic Synthesis Filter Banks

The Two-Channel Synthesis Subband Filter block is the basic unit of a dyadic synthesis filter bank. You can connect several of these blocks to implement an  $n$ -level filter bank, as illustrated in the following figure. For a review of dyadic synthesis filter banks, see the Dyadic Synthesis Filter Bank block reference page.

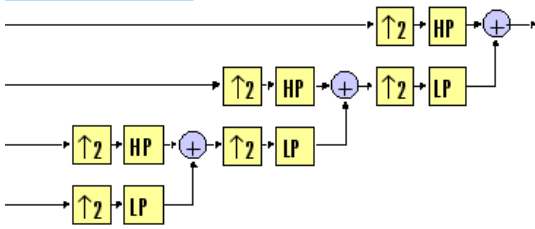
When you create a filter bank by connecting multiple copies of this block, the output values of the filter bank differ depending on whether there is latency. Though the output values differ, both sets of values are valid; the difference arises from changes in latency. See the “Latency” on page 2-1437 section for more information about when latency can occur in the Two-Channel Analysis Subband Filter block.

In some cases, rather than connecting several Two-Channel Analysis Subband Filter blocks, you can use the Dyadic Analysis Filter Bank block, which is faster and requires lesser memory. In particular, the Dyadic Analysis Filter Bank block is more efficient under the following conditions:

- You are reconstructing a signal from  $2^n$  or  $n+1$  subbands.
- The frame size of the signal you are reconstructing is a multiple of  $2^n$ .
- The properties of the subbands you are working with match those of the outputs of the Dyadic Analysis Filter Bank block. These properties are described in the Dyadic Analysis Filter Bank reference page.

### 3-Level Dyadic Synthesis Filter Banks

#### Conceptual illustration

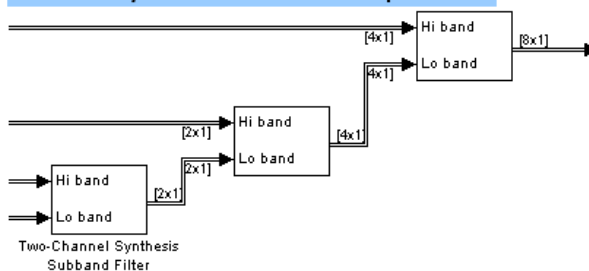


Both implementations of the dyadic analysis filter bank reconstruct a frame-based signal from  $n+1$  subbands, where  $n = 3$ .

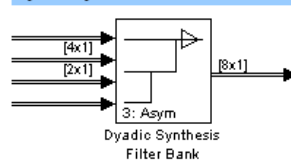
In this case, the Dyadic Synthesis Filter Bank block's implementation is more efficient, since the input subbands have the properties of the outputs of a Dyadic Analysis Filter Bank block.

Use the Two-Channel Synthesis Subband Filter block implementation for other cases, such as to handle separate sample-based vectors or matrices of subbands (rather than a single sample-based vector or matrix of concatenated subbands), or to output sample-based signals.

#### Two-Channel Synthesis Subband Filter block implementation



#### Dyadic Synthesis Filter Bank block implementation

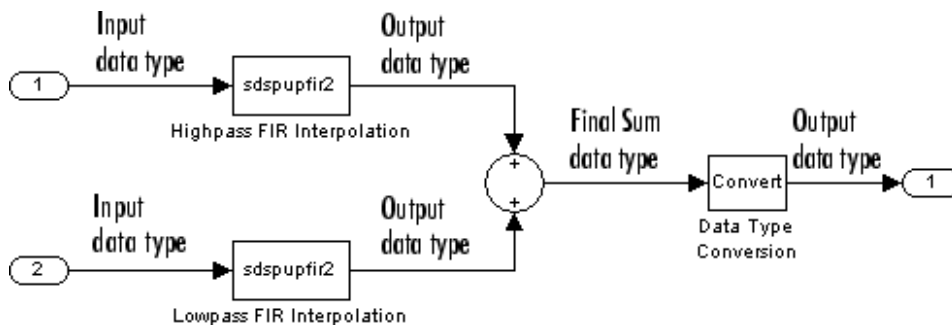


The Dyadic Synthesis Filter Bank block allows you to specify the filter bank filters by providing vectors of filter coefficients, just as this block does. The Dyadic Synthesis Filter Bank block provides an additional option of using wavelet-based filters that the block designs by using a wavelet you specify.

### Fixed-Point Data Types

Fixed-point data types in the Two-Channel Synthesis Subband Filter block.

The Two-Channel Synthesis Subband Filter block is composed of two FIR Interpolation blocks as shown in the following diagram.



For fixed-point signals, you can set the coefficient, product output, accumulator, and output data types used in the FIR Interpolation blocks as discussed in “Parameters” on page 2-1432. For a diagram showing the usage of these data types within the FIR blocks, see the FIR Interpolation block reference page.

In addition, the inputs to the Sum block shown in the previous diagram are accumulated using the accumulator data type. The output of the Sum block is then cast from the accumulator data type to the output data type. Therefore the output of the Two-Channel Synthesis Subband Filter block is in the output data type. You also set these data types in the block dialog box as discussed in the “Parameters” on page 2-1432 section.

If the input is fixed point, it must be a signed integer or a signed fixed point value with a power-of-two slope and zero bias.

## Version History

Introduced before R2006a

## References

- [1] Fliege, N. J. *Multirate Digital Signal Processing: Multirate Systems, Filter Banks, Wavelets*. West Sussex, England: John Wiley & Sons, 1994.
- [2] Strang, G. and T. Nguyen. *Wavelets and Filter Banks*. Wellesley, MA: Wellesley-Cambridge Press, 1996.
- [3] Vaidyanathan, P. P. *Multirate Systems and Filter Banks*. Englewood Cliffs, NJ: Prentice Hall, 1993.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Generated code relies on the memcpy or memset function (string.h) under certain conditions.

### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

If the input is fixed point, it must be a signed integer or a signed fixed point value with a power-of-two slope and zero bias.

## **See Also**

### **Functions**

`fir1` | `fir2` | `firls`

### **Blocks**

DWT | Dyadic Synthesis Filter Bank | FIR Interpolation | IDWT | Two-Channel Analysis Subband Filter

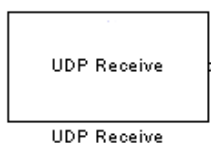
### **Topics**

“Multirate and Multistage Filters”



# UDP Receive

Receive UDP packets on a given IP port



**Libraries:**  
DSP System Toolbox / Sources

## Description

The UDP Receive block receives UDP packets from an IP network port and saves them to its buffer. Due to the nature of the UDP transmission protocol, the receiver is not guaranteed to receive all the data packets that you send using the UDP Send block. With each sample, the block outputs the contents of a single UDP packet as a data vector.

The local IP port number on which the block receives the UDP packets is tunable in the generated code. The generated code for this block relies on prebuilt `.dll` files. You can run this code outside the MATLAB environment or redeploy it, but you must account for the extra `.dll` files. The `packNGo` function creates a ZIP file that contains the pieces required to run or rebuild this code. For more details, see “How To Run a Generated Executable Outside MATLAB”.

## Ports

### Output

**Message** — Data received  
vector

Data received by the UDP Receive block, returned as a one-dimensional vector of the data type specified in **Data type for Message**. If you select **Message is complex**, the block treats the received message as complex data.

If the packet gets dropped during transmission and the receiver does not receive the packet, the data received output is empty.

This port is unnamed until you clear the **Output variable-size signal** parameter.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean`

**Length** — Length of data packet  
scalar

Length of the data packet received, returned as a scalar. This value is determined by the value of the **Maximum length for Message** parameter.

### Dependencies

This port appears only when you clear the **Output variable-size signal** parameter.

Data Types: `uint16`

## Parameters

**Local IP port** — Number of IP port

25000 (default) | scalar in the range [1, 65535]

Specify the IP port number on which to receive UDP packets. This parameter is tunable in the generated code but is not tunable during simulation.

On Linux, to set the IP port number below 1024, run MATLAB with root privileges. For example, at the Linux command line, enter `sudo matlab`.

**Remote IP address ('0.0.0.0' to accept all)** — IP address from which to accept packets

'0.0.0.0' (default) | IP address

Specify the IP address from which to accept UDP packets. Specify a specific IP address to block UDP packets from other addresses. To accept packets from any IP address, specify '0.0.0.0'.

**Receive buffer size (bytes)** — Size of buffer that receives UDP packets

8192 (default) | scalar in the range [1, 67108864]

Specify the size of the buffer, in bytes, that receives the UDP packets. Make the buffer large enough to avoid data loss caused by buffer overflows.

**Maximum length for Message** — Maximum length of output data

255 (default) | scalar

Specify the maximum length, in vector elements, of the data output vector. Set this parameter to a value equal to or greater than the data size of a UDP packet. The system truncates data that exceeds this length.

If you disable **Output variable-size signal**, the block output is the length specified by this parameter.

**Data type for Message** — Data type of message

uint8 (default) | single | double | int8 | int16 | int32 | uint16 | uint32 | boolean

Specify the data type of the vector elements in the message output. Match the data type to the data input used to create the UDP packets.

**Message is complex** — Message data complexity

off (default) | on

Specify whether the block receives a message as complex data. Select this parameter to receive a message as complex data. Clear this parameter if a received message is real data.

**Output variable-size signal** — Message output that varies in length

on (default) | off

Specify whether your model supports signals of varying length. If your model supports signals of varying length, select this parameter. In that case:

- The output vector varies in length, depending on the amount of data in the UDP packet.
- The block emits the data vector from a single unlabeled output.

If your model does not support signals of varying length, clear this parameter. In that case:

- The block emits a fixed-length output that is the same length as specified by **Maximum length for Message**.
- If the UDP packet contains less data than the fixed-length output, the difference contains invalid data.
- The block emits the data vector from the **Message** output.
- The block emits the length of the valid data from the **Length** output.

If you disable this parameter, the block emits output that is the length specified by **Maximum length for Message**.

**Sample time (seconds)** — Frequency of calls to block

1 (default) | scalar

Specify the frequency at which the scheduler calls the UDP Receive block, in seconds. Enter a value greater than zero. In real-time operation, setting this parameter to a smaller value reduces the likelihood of dropped UDP messages.

## Block Characteristics

|                                 |                                     |
|---------------------------------|-------------------------------------|
| <b>Data Types</b>               | Boolean   double   integer   single |
| <b>Direct Feedthrough</b>       | no                                  |
| <b>Multidimensional Signals</b> | no                                  |
| <b>Variable-Size Signals</b>    | yes                                 |
| <b>Zero-Crossing Detection</b>  | no                                  |

## Version History

Introduced in R2010a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Usage notes and limitations:

- The executable generated from this block relies on prebuilt dynamic library files (.dll files) included with MATLAB. Use the `packNGo` function to package the code generated from this block and all the relevant files in a compressed ZIP file. Using this ZIP file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see “How To Run a Generated Executable Outside MATLAB”.
- The **Local IP port** parameter is tunable in the generated code, but not tunable during simulation. You can control the parameter tunability in the generated code in several ways. One of the ways is to configure the parameter as a tunable field of a global structure in the generated code. Other ways include applying a built-in storage class or custom storage class to a `Simulink.Parameter` object and using this object to set the value of the block parameter. For details, see “Create Tunable Calibration Parameter in the Generated Code” (Simulink Coder).

## **See Also**

### **Objects**

`dsp.UDPReceiver` | `dsp.UDPSender`

### **Blocks**

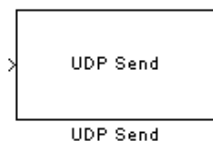
UDP Send

### **Topics**

“How To Run a Generated Executable Outside MATLAB”

# UDP Send

Send UDP message



## Libraries:

DSP System Toolbox / Sinks

## Description

The UDP Send block sends data packets over the network using the User Datagram Protocol (UDP). UDP is a simple message-based connectionless protocol and does not have an end-to-end connection. The protocol sends data packets in one direction from source to destination without verifying the readiness of the receiver. The protocol has no handshaking mechanism. The data packets can get dropped for several reasons. There is no acknowledgement, retransmission, or timeout in UDP. However, UDP is a very simple transmission protocol and is suitable for time-sensitive applications where dropping packets is preferable to waiting for packets delayed due to retransmission.

The UDP Send block transmits an input data vector as a UDP packet to a remote IP network port. The remote IP port number to which the block sends the UDP packets is tunable in the generated code.

Some Simulink blocks and .exe files built from models that contain those blocks require shared libraries, such as .dll files on Windows. The UDP Send block requires the `networkdevice.dll` library file. To meet this requirement, follow the example on the `packNGo` function page to package the code files for your model. The resulting compressed folder contains the .dll files that the model requires, including `networkdevice.dll`. To run this type of .exe file outside of a MATLAB environment, place the required .dll files in the same folder as the .exe file or place them in a folder on the Windows system path. For more details, see “How To Run a Generated Executable Outside MATLAB”.

## Ports

### Input

**Input** — Input data vector  
vector

Specify the input data vector that the block transmits as a UDP packet.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean`

## Parameters

**Remote IP address ('255.255.255.255' for broadcast)** — IP address from which to accept UDP packets

'127.0.0.1' (default) | IP address | string

Specify the IP address or host name to which to send UDP packets. If you specify a host name, specify it as a string. To broadcast a UDP packet, specify '255.255.255.255'.

**Remote IP port** — Number of remote IP port  
25000 (default) | scalar in the range [1, 65535]

Specify the IP port number to which to send UDP packets. This parameter is tunable in the generated code but is not tunable during simulation.

On Linux, to set the IP port number below 1024, run MATLAB with root privileges. For example, at the Linux command line, enter `sudo matlab`.

**Local IP port source** — Source of local IP port  
Automatically determine (default) | Specify via dialog

Specify whether the block uses a local port number that the system assigns or that you specify by using the **Local IP port** parameter. For the system to assign the port number, select **Automatically determine**. If the receiving address expects UDP packets from a specific port number, select **Specify via dialog** and specify the port number by using the **Local IP port** parameter.

**Local IP port** — Number of local IP port  
-1 (default) | scalar in the range [1, 65535]

Specify the IP port number from which the block sends UDP packets. Use this parameter when the receiving address expects messages from a specific port number.

#### Parameter Dependencies

To enable this parameter, set **Local IP port source** to **Specify via dialog**.

**Send buffer size (bytes)** — Size of buffer that sends UDP packets  
8192 (default) | scalar in the range [1, 67108864]

Specify the size of the buffer, in bytes, that sends the UDP packets. Make the buffer large enough to avoid data loss caused by buffer overflows.

## Block Characteristics

|                                 |                                     |
|---------------------------------|-------------------------------------|
| <b>Data Types</b>               | Boolean   double   integer   single |
| <b>Direct Feedthrough</b>       | no                                  |
| <b>Multidimensional Signals</b> | no                                  |
| <b>Variable-Size Signals</b>    | no                                  |
| <b>Zero-Crossing Detection</b>  | no                                  |

## Version History

Introduced in R2010a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Usage notes and limitations:

- The executable generated from this block relies on prebuilt dynamic library files (.dll files) included with MATLAB. Use the `packNGo` function to package the code generated from this block and all the relevant files in a compressed ZIP file. Using this ZIP file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see “How To Run a Generated Executable Outside MATLAB”.
- The **Remote IP port** parameter is tunable in the generated code, but not tunable during simulation. You can control the parameter tunability in the generated code through several ways. One of the ways is to configure the parameter as a tunable field of a global structure in the generated code. Other ways include applying a built-in storage class or custom storage class to a `Simulink.Parameter` object and using this object to set the value of the block parameter. For details, see “Create Tunable Calibration Parameter in the Generated Code” (Simulink Coder).

## See Also

### Objects

`dsp.UDPReceiver` | `dsp.UDPSender`

### Blocks

UDP Receive

### Topics

“How To Run a Generated Executable Outside MATLAB”

## Unbuffer

Unbuffer input frame into sequence of scalar outputs



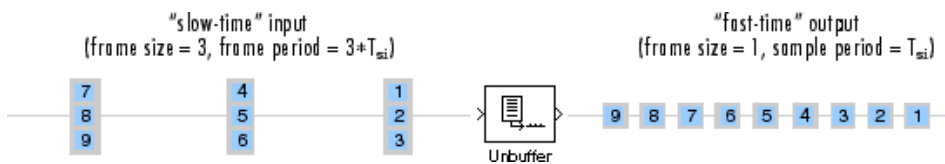
## Library

Signal Management / Buffers

dspbuff3

## Description

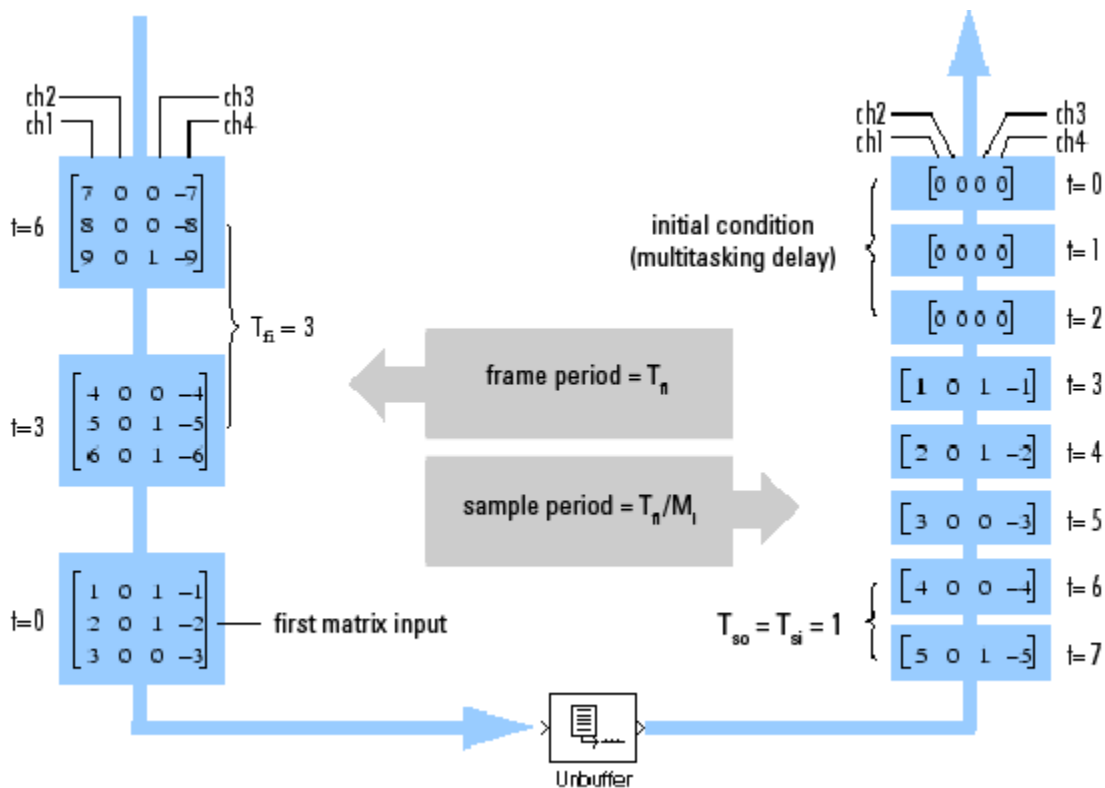
The Unbuffer block unbuffers an  $M_i$ -by- $N$  input into a 1-by- $N$  output. That is, inputs are unbuffered row-wise so that each matrix row becomes an independent time-sample in the output. The rate at which the block receives inputs is generally less than the rate at which the block produces outputs.



The block adjusts the output rate so that the *sample period* is the same at both the input and output,  $T_{so}=T_{si}$ . Therefore, the output sample period for an input of frame size  $M_i$  and frame period  $T_{fi}$  is  $T_{fi}/M_i$ , which represents a *rate*  $M_i$  times higher than the input frame rate. In the example above, the block receives inputs only once every three sample periods, but produces an output once every sample period. To rebuffer inputs to a larger or smaller frame size, use the Buffer block.

In the `ex_unbuffer_ref1` on page 5-1369 example model, the block unbuffers a four-channel input with a frame size of three. The **Initial conditions** parameter is set to zero and the tasking mode is set to multitasking, so the first three outputs are zero vectors.





## Zero Latency

The Unbuffer block has *zero-tasking latency* in Simulink single-tasking mode. Zero-tasking latency means that the first input sample (received at  $t=0$ ) appears as the first output sample.

## Nonzero Latency

For *multitasking* operation, the Unbuffer block's buffer is initialized with the value specified by the **Initial conditions** parameter, and the block begins unbuffering this frame at the start of the simulation. Inputs to the block are therefore delayed by one buffer length, or  $M_i$  samples.

The **Initial conditions** parameter can be one of the following:

- A scalar to be repeated for the first  $M_i$  output samples of every channel
- A length- $M_i$  vector containing the values of the first  $M_i$  output samples for every channel
- An  $M_i$ -by- $N$  matrix containing the values of the first  $M_i$  output samples in each of  $N$  channels

---

**Note** For more information on latency and the Simulink tasking modes, see “Excess Algorithmic Delay (Tasking Latency)” and “Time-Based Scheduling and Code Generation” (Simulink Coder).

---

## Parameters

### Initial conditions

The value of the block's initial output for cases of nonzero latency. You can specify a scalar, vector, or matrix.

## Supported Data Types

| Port   | Supported Data Types   |
|--------|--|
| Input  | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed and unsigned)</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul> |
| Output | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed and unsigned)</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul> |

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

The Unbuffer block supports code generation for Simulink Real-Time. The code generated for this target is executed concurrently. To generate this code, set the system target file of the model to `slrealtime.tlc`.

To set the system target file:

- 1 In the Simulink model window, on the **Modeling** tab, click **Model Settings**.
- 2 In the Model Settings > **Code Generation** pane, set the **System target file** to `slrealtime.tlc`.

### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

## See Also

### Functions

`rebuffer_delay`

### Blocks

Delay Line | Buffer

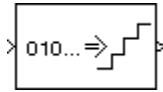
### Topics

“Buffering and Frame-Based Processing”

“Convert Sample and Frame Rates in Simulink Using Frame Rebuffering Blocks”

## Uniform Decoder

Decode integer input into floating-point output



## Library

Quantizers

dspquant2

## Description

The Uniform Decoder block performs the inverse operation of the Uniform Encoder block, and reconstructs quantized floating-point values from encoded integer input. The block adheres to the definition for uniform decoding specified in ITU-T Recommendation G.701.

Inputs can be real or complex values of the following six integer data types: `uint8`, `uint16`, `uint32`, `int8`, `int16`, or `int32`.

The block first casts the integer input values to floating-point values, and then uniquely maps (decodes) them to one of  $2^B$  uniformly spaced floating-point values in the range  $[-V, (1-2^{1-B})V]$ , where you specify  $B$  in the **Bits** parameter (as an integer between 2 and 32) and  $V$  is a floating-point value specified by the **Peak** parameter. The smallest input value representable by  $B$  bits (0 for an unsigned input data type;  $-2^{B-1}$  for a signed input data type) is mapped to the value  $-V$ . The largest input value representable by  $B$  bits ( $2^B-1$  for an unsigned input data type;  $2^{B-1}-1$  for a signed input data type) is mapped to the value  $(1-2^{1-B})V$ . Intermediate input values are linearly mapped to the intermediate values in the range  $[-V, (1-2^{1-B})V]$ .

To correctly decode values encoded by the Uniform Encoder block, the **Bits** and **Peak** parameters of the Uniform Decoder block should be set to the same values as the **Bits** and **Peak** parameters of the Uniform Encoder block. The **Overflow mode** parameter specifies the Uniform Decoder block's behavior when the integer input is outside the range representable by  $B$  bits. When you select **Saturate**, *unsigned* input values greater than  $2^B-1$  saturate at  $2^B-1$ ; *signed* input values greater than  $2^{B-1}-1$  or less than  $-2^{B-1}$  saturate at those limits. The real and imaginary components of complex inputs saturate independently.

When you select **Wrap**, *unsigned* input values,  $u$ , greater than  $2^B-1$  are wrapped back into the range  $[0, 2^B-1]$  using  $\text{mod-}2^B$  arithmetic.

$$u = \text{mod}(u, 2^B)$$

*Signed* input values,  $u$ , greater than  $2^{B-1}-1$  or less than  $-2^{B-1}$  are wrapped back into that range using  $\text{mod-}2^B$  arithmetic.

$$u = (\text{mod}(u+2^B/2, 2^B) - (2^B/2))$$

The real and imaginary components of complex inputs wrap independently.

The **Output type** parameter specifies whether the decoded floating-point output is single or double precision. Either level of output precision can be used with any of the six integer input data types.

## Parameters

### Peak

Specify the largest amplitude represented in the encoded input. To correctly decode values encoded with the Uniform Encoder block, set the **Peak** parameters in both blocks to the same value.

### Bits

Specify the number of input bits, B, used to encode the data. (This can be less than the total number of bits supplied by the input data type.) To correctly decode values encoded with the Uniform Encoder block, set the **Bits** parameters in both blocks to the same value.

### Overflow mode

Specify the block's behavior when the integer input is outside the range representable by B bits. Out-of-range inputs can either saturate at the extreme value, or wrap back into range.

### Output type

Specify the precision of the floating-point output, `single` or `double`.

## References

*General Aspects of Digital Transmission Systems: Vocabulary of Digital Transmission and Multiplexing, and Pulse Code Modulation (PCM) Terms*, International Telecommunication Union, ITU-T Recommendation G.701, March, 1993

## Supported Data Types

| Port   | Supported Data Types   |
|--------|--|
| Input  | <ul style="list-style-type: none"> <li>8-, 16-, and 32-bit integers</li> </ul>   |
| Output | <ul style="list-style-type: none"> <li>Double-precision floating point</li> <li>Single-precision floating point</li> </ul> |

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

## See Also

### Functions

`udecode` | `uencode`

**Blocks**

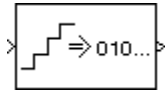
Data Type Conversion | Quantizer | Scalar Quantizer Decoder | Uniform Encoder

**Topics**

“Decode Integer Data Using Uniform Decoder Block”

# Uniform Encoder

Quantize and encode floating-point input into integer output



## Library

Quantizers

dspquant2

## Description

The Uniform Encoder block performs the following two operations on each floating-point sample in the input vector or matrix:

- 1 Quantizes the value using the same precision
- 2 Encodes the quantized floating-point value to an integer value

In the first step, the block quantizes an input value to one of  $2^B$  uniformly spaced levels in the range  $[-V, (1-2^{1-B})V]$ , where you specify  $B$  in the **Bits** parameter and you specify  $V$  in the **Peak** parameter. The quantization process rounds both positive and negative inputs *downward* to the nearest quantization level, with the exception of those that fall exactly on a quantization boundary. The real and imaginary components of complex inputs are quantized independently.

The number of bits,  $B$ , can be any integer value between 2 and 32, inclusive. Inputs greater than  $(1-2^{1-B})V$  or less than  $-V$  saturate at those respective values. The real and imaginary components of complex inputs saturate independently.

In the second step, the quantized floating-point value is uniquely mapped (encoded) to one of  $2^B$  integer values. When the **Output type** is set to **Unsigned integer**, the smallest quantized floating-point value,  $-V$ , is mapped to the integer 0, and the largest quantized floating-point value,  $(1-2^{1-B})V$ , is mapped to the integer  $2^B-1$ . Intermediate quantized floating-point values are linearly (uniformly) mapped to the intermediate integers in the range  $[0, 2^B-1]$ . For efficiency, the block automatically selects an *unsigned* output data type (`uint8`, `uint16`, or `uint32`) with the minimum number of bits equal to or greater than  $B$ .

When the **Output type** is set to **Signed integer**, the smallest quantized floating-point value,  $-V$ , is mapped to the integer  $-2^{B-1}$ , and the largest quantized floating-point value,  $(1-2^{1-B})V$ , is mapped to the integer  $2^{B-1}-1$ . Intermediate quantized floating-point values are linearly mapped to the intermediate integers in the range  $[-2^{B-1}, 2^{B-1}-1]$ . The block automatically selects a *signed* output data type (`int8`, `int16`, or `int32`) with the minimum number of bits equal to or greater than  $B$ .

Inputs can be real or complex, double or single precision. The output data types that the block uses are shown in the table below. Note that most of the DSP System Toolbox blocks accept only double-precision inputs. Use the Simulink Data Type Conversion block to convert integer data types to double precision. See “About Data Types in Simulink” (Simulink) for a complete discussion of data types, as well as a list of Simulink blocks capable of reduced-precision operations.

| Bits     | Unsigned Integer | Signed Integer |
|----------|------------------|----------------|
| 2 to 8   | uint8            | int8           |
| 9 to 16  | uint16           | int16          |
| 17 to 32 | uint32           | int32          |

The Uniform Encoder block operations adhere to the definition for uniform encoding specified in ITU-T Recommendation G.701.

## Parameters

### Peak

The largest input amplitude to be encoded,  $V$ . Real or imaginary input values greater than  $(1-2^{1-B})V$  or less than  $-V$  saturate (independently for complex inputs) at those limits.

### Bits

Specify the number of bits,  $B$ , needed to represent the integer output. The number of levels at which the block quantizes the floating-point input is  $2^B$ .

### Output type

The data type of the block's output, `Unsigned integer` or `Signed integer`. Unsigned outputs are `uint8`, `uint16`, or `uint32`, while signed outputs are `int8`, `int16`, or `int32`.

## References

*General Aspects of Digital Transmission Systems: Vocabulary of Digital Transmission and Multiplexing, and Pulse Code Modulation (PCM) Terms*, International Telecommunication Union, ITU-T Recommendation G.701, March, 1993

## Supported Data Types

| Port   | Supported Data Types   |
|--------|--|
| Input  | <ul style="list-style-type: none"> <li>Double-precision floating point</li> <li>Single-precision floating point</li> </ul> |
| Output | <ul style="list-style-type: none"> <li>8-, 16-, and 32-bit integers</li> </ul>   |

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

## See Also

### Functions

`udecode` | `uencode`



**Blocks**

Data Type Conversion | Quantizer | Scalar Quantizer Decoder | Uniform Decoder

**Topics**

“Encode Data Using Uniform Encoder”

# Unwrap

Unwrap signal phase



**Libraries:**  
DSP System Toolbox / Signal Operations

## Description

The Unwrap block unwraps each channel of the input by adding or subtracting appropriate multiples of  $2\pi$  to each channel element. The block recognizes phase discontinuities larger than the value of the **Tolerance** parameter. For more information on phase unwrapping, see “Definition of Phase Unwrap” on page 2-1460.

## Ports

### Input

**Port\_1** — Signal input  
scalar | vector | matrix

Signal input to unwrap, specified as a scalar, vector, or matrix. The input must have radian phase entries.

Data Types: single | double

### Output

**Port\_1** — Signal output  
scalar | vector | matrix

Unwrapped signal output, returned as a scalar, vector, or matrix.

The block preserves the input size and dimension, and the output port rate equals the input port rate.

Data Types: single | double

## Parameters

**Tolerance (radians)** — Jump size  
pi (default) | scalar value

The jump size that the block recognizes as a true phase discontinuity. The default is set to  $\pi$  (rather than a smaller value) to avoid altering legitimate signal features. To increase the block's sensitivity, set the **Tolerance** to a value slightly less than  $\pi$ .

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32

**Input processing** — Method to process the input  
Columns as channels (frame based) (default) | Elements as channels (sample based)

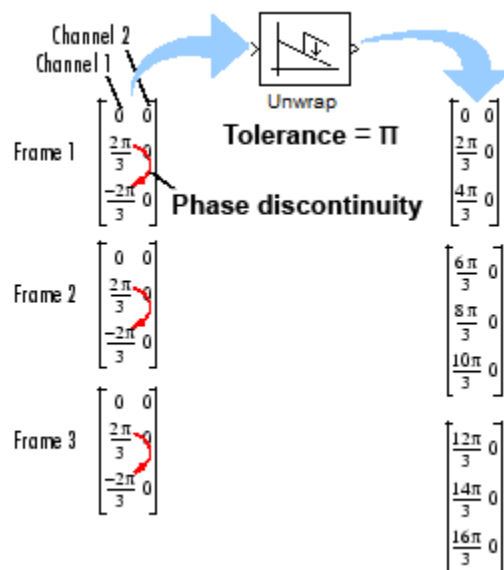
Specify how the block should process the input. You can set this parameter to one of these options:

- **Columns as channels (frame based) (default)** — When you select this option, the block treats each column of the input as a separate channel.
- **Elements as channels (sample based)** — When you select this option, the block treats each element of the input as a separate channel.

**Reset initial phase to zero for each input frame** — Checking successive frames  
off (default) | on

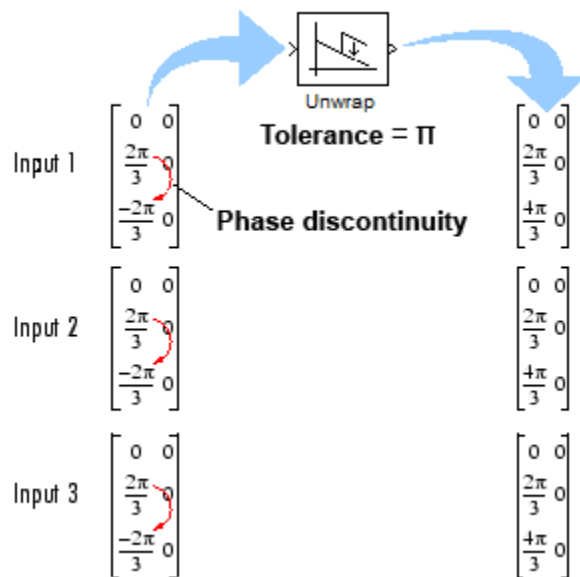
When you clear this parameter, the block ignores boundaries between input frames and does not reset the initial phase value to zero each time a new input is received. In this mode, the block continuously unwraps the data in each column of the input.

This figure illustrates how the block unwraps data when you clear this parameter.



When you select this parameter, the block treats each frame of input data independently and resets the initial phase value for each new input frame.

This figure illustrates how the block unwraps data when you select this parameter.



For more information, see “Frame-Based Processing” on page 2-1462.

### Dependencies

To enable this parameter, set the **Input processing** parameter to Columns as channels (frame based). In sample-based processing mode, the block does not reset the initial phase value to zero for each new input. See “Sample-Based Processing” on page 2-1463 for more information.

### Block Characteristics

|                                 |                 |
|---------------------------------|-----------------|
| <b>Data Types</b>               | double   single |
| <b>Direct Feedthrough</b>       | no              |
| <b>Multidimensional Signals</b> | no              |
| <b>Variable-Size Signals</b>    | no              |
| <b>Zero-Crossing Detection</b>  | no              |

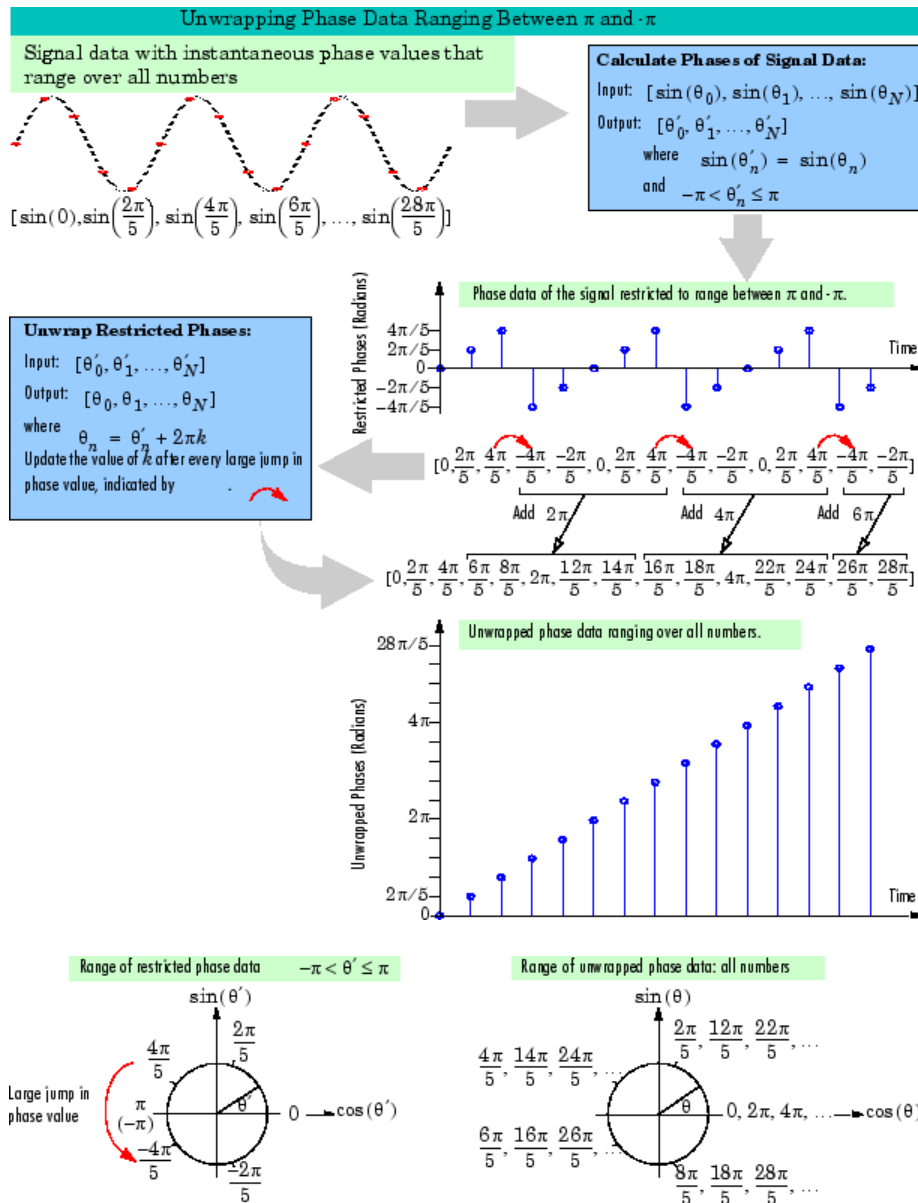
### More About

#### Definition of Phase Unwrap

Phase unwrap or unwrap is a process often used to reconstruct a signal's original phase. Unwrap algorithms add appropriate multiples of  $2\pi$  to each phase input to restore original phase values, as illustrated in the diagram. For more information on the unwrap algorithm used by this block, see “Unwrap Method” on page 2-1461.

Algorithms that compute the phase of a signal often only output phases between  $-\pi$  and  $\pi$ . For instance, such algorithms compute the phase of  $\sin(2\pi + 3)$  to be 3, since  $\sin(3) = \sin(2\pi + 3)$ , and

since the actual phase,  $2\pi + 3$ , is not between  $-\pi$  and  $\pi$ . Such algorithms compute the phases of  $\sin(-4\pi + 3)$  and  $\sin(16\pi + 3)$  to be 3 as well.



## Unwrap Method

The Unwrap block unwraps each channel of its input matrix or input vector by adding  $2\pi k$  to each successive channel element, and updating  $k$  at each *phase jump*. A phase jump occurs when the difference between two adjacent phase value entries exceeds the value of the **Tolerance** parameter.

This code shows how the block unwraps the data in a given input channel  $u$ .

```
k=0; % initialize k to 0
i=1; % initialize the counter to 1
alpha=pi; % set alpha to the desired Tolerance. In this case, pi

for i = 1:(size(u)-1)
    yout(i,:) = u(i) + (2*pi*k); % add 2*pi*k to ui
```

```

if((abs(u(i+1)-u(i))>(abs(alpha))) %if diff is greater than alpha, increment or decrement k
    if u(i+1)<u(i) % if the phase jump is negative, increment k
        k=k+1;
    else % if the phase jump is positive, decrement k
        k=k-1;
    end
end
end
end
yout((i+1,:)=u(i+1)+(2*pi*k); % add 2*pi*k to the last element of the input

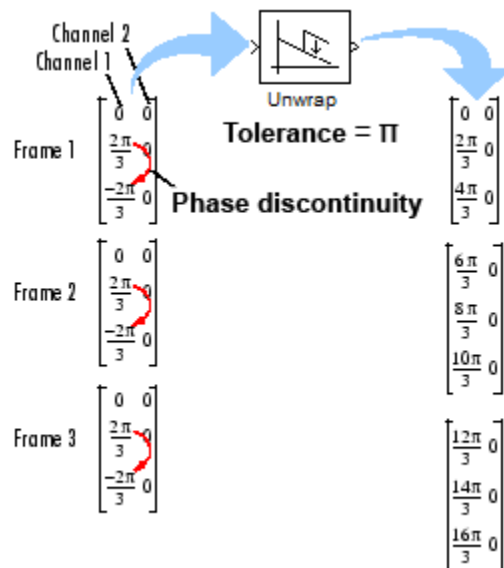
```

## Frame-Based Processing

When you configure the block to perform frame-based processing, by setting the **Input processing** parameter to **Columns as channels (frame based)**, the block supports two different unwrap modes. In both modes, the block adds  $2\pi k$  to each input channel's elements, and updates  $k$  at each phase discontinuity. The difference between the two modes is how often the block resets the initial phase value ( $k$ ) to zero. You can choose to unwrap data across frame boundaries (default), or to unwrap only within input frames, by resetting the initial phase value each time a new input frame is received.

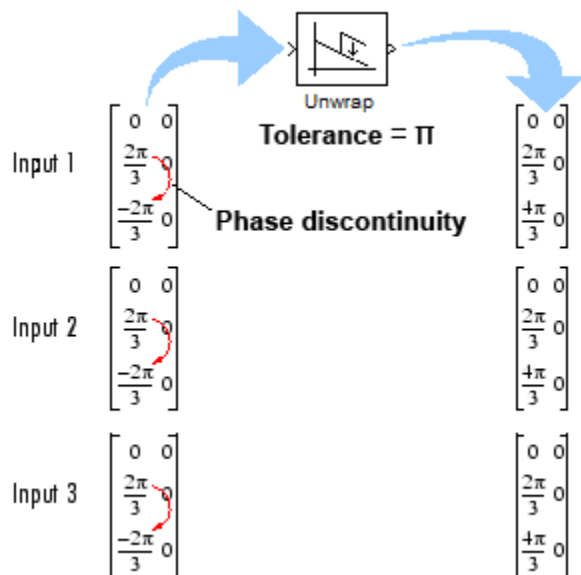
### Unwrapping Across Frame Boundaries

In the default mode, the block ignores boundaries between input frames, and continues to unwrap the data in each channel without resetting the initial phase value to zero. To specify this mode, clear the **Reset initial phase to zero for each input frame** parameter. This figure illustrates how the block unwraps data in this mode.



### Unwrapping Within Frames

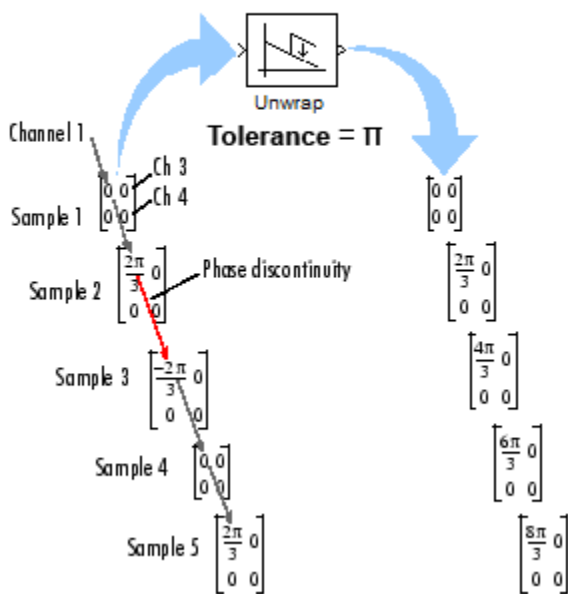
When you select the **Reset initial phase to zero for each input frame** parameter, the block treats each frame of input data independently. In this mode, the block resets the initial phase value to zero each time a new input frame is received. This figure illustrates how the block unwraps data in this mode.



### Sample-Based Processing

When you configure the block to perform sample-based processing, by setting the **Input processing** parameter to **Elements as channels (sample based)**, the block treats each element of the input as an individual channel. The block unwraps the data in each channel of the input, and does not reset the initial phase to zero each time a new input is received. This figure illustrates how the block unwraps data when performing sample-based processing.

### Sample-Based Unwrap Mode



## **Version History**

Introduced before R2006a

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

Generated code relies on the memcpy or memset function (string.h) under certain conditions.

## **See Also**

### **Functions**

unwrap



# Upsample

Resample input at higher rate by inserting zeros



## Libraries:

DSP System Toolbox / Signal Operations

DSP System Toolbox HDL Support / Signal Operations

## Description

The Upsample block resamples each channel of the  $M_1$ -by- $N$  input at a rate  $L$  times higher than the input sample rate by inserting  $L-1$  zeros between consecutive samples. Specify the integer  $L$  in the **Upsample factor**, **L** parameter or through the input port **L**.

You can use this block inside triggered subsystems when you set the **Rate options** parameter to Enforce single-rate processing.

## Ports

### Input

**in** — Input signal

scalar | vector | matrix

Specify the input signal as a real or a complex-valued scalar, vector, or a matrix signal.

When you set **Input processing** to Columns as channels (frame based) and **Rate options** to Enforce single-rate processing, the input can be a variable-size signal. When the input is a variable-size signal, the frame size (number of rows) and the number of channels (columns) of the signal can change during simulation.

This port is unnamed until you set **Upsample factor source** to Input port.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

**L** — Upsample factor

positive integer  $\leq L_{max}$

Specify the upsample factor  $L$  as a positive integer less than or equal to the value you specify in the **Maximum upsample factor**, **Lmax** parameter.

### Dependency

To enable this port, set the **Upsample factor source** parameter to Input port.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32

### Output

**Port\_1** — Upsampled signal

scalar | vector | matrix

The Upsample block outputs a signal that has the same data type and complexity as the input signal. The size of the output depends on the value of the upsample factor and the option you select in the **Rate options** parameter.

If you set **Upsample factor, L** to  $L$  and **Rate options** to:

- **Enforce single-rate processing** -- The block maintains the input sample rate by increasing the output frame size by a factor of  $L$ .
- **Allow multirate processing** -- The block resamples the signal such that the output sample rate is  $L$  times faster than the input sample rate. The output signal has the same frame size as the input signal.

When you input the upsample factor through the input port **L**, the **Rate options** parameter is automatically set to **Enforce single-rate processing**. In this case, the block maintains the input sample rate at the output by increasing the output frame size by a factor of  $L$ .

The output is a variable-size signal when one or both of these conditions are met:

- Input is a variable-size signal.
- Upsample factor is input through the port **L**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixed point`

## Parameters

**Upsample factor source** — Mode of specifying upsample factor

`Dialog parameter (default)` | `Input port`

Specify the upsample factor through one of these options:

- `Dialog parameter` -- The block enables the **Upsample factor, L** parameter on the block dialog box.
- `Input port` -- The block enables the input port **L** through which you can specify the upsample factor.

**Upsample factor, L** — Upsample factor

`3 (default)` | `positive integer`

Specify the upsample factor  $L$  by which to increase the input sample rate as a positive integer.

### Dependency

To enable this parameter, set **Upsample factor source** to `Dialog parameter`.

**Sample offset (0 to L-1)** — Sample offset

`0 (default)` | `integer in the range [0, L-1]`

Specify the sample offset  $D$  as an integer in the range  $[0, L-1]$ .

Use the **Sample offset (0 to L-1)** parameter to delay the output samples by a specific number of sample periods. Doing so enables you to select any of the  $L$  possible output phases. The value you specify in the **Sample offset (0 to L-1)** parameter must be an integer in the range  $0 \leq D < (L - 1)$ .

When you input the upsample factor through the input port **L**, the **Sample offset (0 to L-1)** parameter is not enabled in the block dialog box. In this case, the sample offset  $D$  is automatically set to 0.

#### Dependency

To enable this parameter, set **Upsample factor source** to `Dialog` parameter.

#### Maximum upsample factor, $L_{max}$ — Maximum upsample factor

16 (default) | positive integer  $\geq 1$

Specify the maximum upsample factor  $L_{max}$  as a positive integer greater than or equal to 1. The upsample factor you specify through the input port **L** must be less than or equal to  $L_{max}$ .

#### Dependency

To enable this parameter, set **Upsample factor source** to `Input` port.

#### Input processing — Method to process input signals

`Columns as channels (frame based)` (default) | `Elements as channels (sample based)`

Specify how the block should process the input. You can set this parameter to one of these options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel. In this mode, the block can perform single-rate or multirate processing.
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel. In this mode, the block always performs multirate processing.

#### Dependency

To enable this parameter, set **Upsample factor source** to `Dialog` parameter.

When you set **Upsample factor source** to `Input` port, the **Input processing** parameter is automatically set to `Columns as channels (frame based)`.

#### Rate options — Method to upsample input

`Enforce single-rate processing` (default) | `Allow multirate processing`

Specify the method that the block uses to upsample the input. You can select one of the following options:

- `Enforce single-rate processing` — When you select this option, the block maintains the input sample rate by increasing the output frame size by a factor of  $L$ . To select this option, you must set the **Input processing** parameter to `Columns as channels (frame based)`.
- `Allow multirate processing` — When you select this option, the block resamples the signal such that the output sample rate is  $L$  times faster than the input sample rate.

**Dependency**

To enable this parameter, set **Upsample factor source** to `Dialog` parameter.

When you set **Upsample factor source** to `Input port`, the **Rate options** parameter is automatically set to `Enforce single-rate processing`.

**Initial conditions** — Initial conditions

0 (default) | scalar | vector | matrix

Specify the value with which the block is initialized for cases of nonzero latency as a scalar, vector, or a matrix. The scalar value appears in the output as the sample  $D+1$ . If this parameter is set to a vector or a matrix, the size of this parameter should be the same as that of the input. This parameter appears only when you configure the block to perform multirate processing.

**Dependencies**

To enable this parameter, set **Rate options** to `Allow multirate processing`.

**Block Characteristics**

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | Boolean   double   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no  |
| <b>Multidimensional Signals</b> | no  |
| <b>Variable-Size Signals</b>    | yes   |
| <b>Zero-Crossing Detection</b>  | no  |

**More About****Frame-Based Processing**

When you set the **Input processing** parameter to `Columns as channels (frame based)`, the block upsamples each column of the input over time. In this mode, the block can perform either single-rate or multirate processing. You can use the **Rate options** parameter to specify how the block upsamples the input.

- When you set the **Rate options** parameter to `Enforce single-rate processing`, the input and output of the block have the same sample rate. In this mode, the block outputs a signal with a proportionally larger frame size than the input. For upsampling by a factor of  $L$ , the output frame size is  $L$  times larger than the input frame size ( $M_o = M_i * L$ ), but the input and output frame rates are equal.

For an example of single-rate upsampling, see “Upsample Signal In Single-Rate Frame-Based Processing Mode”.

- When you set the **Rate options** parameter to `Allow multirate processing`, the block treats an  $M_i$ -by- $N$  matrix input as  $N$  independent channels. The block upsamples each column of the input over time by keeping the frame size constant ( $M_i = M_o$ ), and making the output frame period ( $T_{fo}$ )  $L$  times shorter than the input frame period ( $T_{fi} = T_{fo} / L$ ).

See the “Upsample Signal In Multirate Frame-Based Processing Mode” example to see the Upsample block in this mode.

### Sample-Based Processing

When you set the **Input processing** parameter to `Elements as channels (sample based)`, the block treats an  $M$ -by- $N$  matrix input as  $M*N$  independent channels, and upsamples each channel over time. In this mode, the block always performs multirate processing. The output sample rate is  $L$  times higher than the input sample rate ( $T_{so} = T_{si}/L$ ), and the input and output sizes are identical.

### Zero Latency

The Upsample block has *zero-tasking latency* for all single-rate operations. The block is in a single-rate mode if you specify an upsample factor of 1 or if you set the **Input processing** parameter to `Columns as channels (frame based)` and the **Rate options** parameter to `Enforce single-rate processing`.

The Upsample block also has zero-tasking latency for multirate operations if you run your model in Simulink single-tasking mode.

Zero-tasking latency means that the block propagates the first input (received at  $t = 0$ ) immediately after the consecutive zeros  $D$ , which you specify in the **Sample offset (0 to L-1)** parameter. When you input the upsample factor through the input port **L**,  $D = 0$ . This output ( $D+1$ ) is followed in turn by the  $L-1$  inserted zeros and the next input sample.

### Nonzero Latency

The Upsample block has tasking latency for a multirate multitasking operation:

- In multirate sample-based processing mode, the initial condition for each channel appears as output sample  $D+1$ , and is followed by  $L-1$  inserted zeros. The channel first input appears as output sample  $D+L+1$ . The **Initial conditions** parameter can be an  $M_i$ -by- $N$  matrix containing one value for each channel, or a scalar value that the block applies to all signal channels.
- In multirate frame-based processing mode, the first row of the initial condition matrix appears as output sample  $D+1$ , and is followed by  $L-1$  inserted rows of zeros, the second row of the initial condition matrix, and so on. The first row of the first input matrix appears in the output as sample  $M_iL+D+1$ . The **Initial conditions** parameter can be an  $M_i$ -by- $N$  matrix, or a scalar value that the block repeats across all elements of the input matrix.

---

**Note** For more information on latency and the Simulink tasking modes, see “Excess Algorithmic Delay (Tasking Latency)” and “Time-Based Scheduling and Code Generation” (Simulink Coder).

---

## Version History

Introduced before R2006a

### R2022b: Support for variable-size input signal

Starting in R2022b, when you set **Input processing** to `Columns as channels (frame based)` and **Rate options** to `Enforce single-rate processing`, the input can be a variable-size signal. That is, the frame size (number of rows) and the number of channels (columns) of the signal can change during simulation.

When the input is a variable-size signal, the output is also a variable-size signal.

### R2023a: Tune upsample factor through input port

You can now specify the upsample factor through an input port when the Upsample block operates in the single-rate frame-based processing mode. When you specify the upsample factor through the input port, you can change the factor during simulation.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Generated code relies on the memcpy or memset function (`string.h`) under certain conditions.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

### Best Practices

Consider whether your model can use the Repeat block instead of the Upsample block. The Repeat block uses fewer hardware resources, so as a best practice use the Upsample block only when your algorithm requires zero-padding upsampling.

See also “Multirate Model Requirements for HDL Code Generation” (HDL Coder).

### HDL Architecture

This block has one default HDL architecture.

### HDL Block Properties

|                                  |  |
|----------------------------------|--|
| <b>ConstrainedOutputPipeline</b> | Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder). |
| <b>InputPipeline</b>             | Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).                 |
| <b>OutputPipeline</b>            | Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).               |

### Restrictions

- **Input processing** set to `Columns as channels (frame based)` is not supported.

- **Upsample factor source** set to `Input` port is not supported.

**Complex Data Support**

This block supports code generation for complex signals.

**Fixed-Point Conversion**

Design and simulate fixed-point systems using Fixed-Point Designer™.

**See Also****Blocks**

Downsample | FIR Interpolation | FIR Rate Conversion | Repeat

# Variable Bandwidth FIR Filter

Design tunable bandwidth FIR filter



## Libraries:

DSP System Toolbox / Filtering / Filter Designs

## Description

The Variable Bandwidth FIR Filter block filters each channel of the input signal over time using the specified FIR filter specifications. This block offers tunable filter design parameters, which enable you to tune the filter characteristics while the simulation is running.

The block designs the FIR filter according to the filter parameters specified in the block dialog box. The output port properties, such as datatype, complexity, and dimension, are identical to the input port properties.

This block also supports SIMD code generation. For details, see “Code Generation” on page 2-1480.

## Ports

### Input

**x** — Data input  
vector | matrix

Specify the data input as a vector or a matrix. The block treats each column of the input signal as a separate channel. If the input is a two-dimensional signal, the first dimension represents the channel length (or frame size) and the second dimension represents the number of channels. If the input is a one-dimensional signal, then the block interprets it as having a single channel.

The block accepts variable-size input signals, that is, you can change the size of each input channel during simulation but you cannot change the number of channels.

This port is unnamed until you select one of these parameters:

- **Specify cutoff frequency from input port**
- **Specify center frequency from input port**
- **Specify bandwidth from input port**

Data Types: `single` | `double`

**Fcut** — Filter cutoff frequency  
positive scalar

Specify the cutoff frequency of the FIR filter as a real positive scalar in Hz or in normalized frequency units.



**Dependencies**

To enable this port, select the **Specify cutoff frequency from input port** parameter.

Data Types: `single` | `double`

**Fc** — Filter center frequency

positive scalar

Specify the center frequency of the FIR filter as a real positive scalar in Hz or in normalized frequency units.

**Dependencies**

To enable this port, select the **Specify center frequency from input port** parameter.

Data Types: `single` | `double`

**BW** — Filter bandwidth

positive scalar

Specify the bandwidth of the FIR filter as a real positive scalar in Hz or in normalized frequency units.

**Dependencies**

To enable this port, select the **Specify bandwidth from input port** parameter.

Data Types: `single` | `double`

**Output****Port\_1** — Filtered output

vector | matrix

Filtered output, returned as a vector or a matrix. The size, data type, and complexity of the output signal matches that of the input signal.

Data Types: `single` | `double`

**Parameters****FIR filter order** — FIR filter order

30 (default) | positive integer

Specify the order of the FIR filter as a positive integer scalar.

**Filter type** — Filter type

Lowpass (default) | Highpass | Bandpass | Bandstop

Specify the type of FIR filter. You can set this parameter to:

- Lowpass
- Highpass

- Bandpass
- Bandstop

**Specify cutoff frequency from input port** — Specify cutoff frequency from input port

off (default) | on

When you select this check box, specify the cutoff frequency through the **Fcut** port. When you clear this check box, specify the cutoff frequency in the block dialog box through the **Filter cutoff frequency** parameter.

#### Dependency

To enable this parameter, set **Filter type** to Lowpass or Highpass.

**Filter cutoff frequency** — Filter cutoff frequency

512 (default) | positive scalar

Specify the cutoff frequency of the FIR filter as a real positive scalar in Hz or in normalized frequency units.

If you set the **Sample rate mode** parameter to:

- **Specify on dialog or Inherit from input port** -- The value of the filter cutoff frequency is in Hz and must be less than half the value of the input sample rate.
- **Use normalized frequency (0 to 1)** -- The value of the filter cutoff frequency is in normalized frequency units. The value must be a positive scalar less than 1.0.

**Tunable:** Yes

#### Dependencies

To enable this parameter:

- Set **Filter type** to Lowpass or Highpass.
- Clear the **Specify cutoff frequency from input port** parameter.

**Specify center frequency from input port** — Specify center frequency from input port

off (default) | on

When you select this check box, specify the center frequency through the **Fc** port. When you clear this check box, specify the center frequency in the block dialog box through the **Filter center frequency** parameter.

#### Dependencies

To enable this parameter, set **Filter type** to Bandpass or Bandstop.

**Filter center frequency** — Filter center frequency

44100/4 (default) | positive scalar

Specify the center frequency of the FIR filter as a real positive scalar in Hz or in normalized frequency units.

If you set the **Sample rate mode** parameter to:

- Specify on dialog or Inherit from input port -- The value of the filter center frequency is in Hz and must be less than half the value of the input sample rate.
- Use normalized frequency (0 to 1) -- The value of the filter center frequency is in normalized frequency units. The value must be a positive scalar less than 1.0.

**Tunable:** Yes

**Dependencies**

To enable this parameter:

- Set **Filter type** to Bandpass or Bandstop.
- Clear the **Specify center frequency from input port** parameter.

**Specify bandwidth from input port** — Specify bandwidth from input port

off (default) | on

When you select this check box, specify the filter bandwidth through the **BW** port. When you clear this check box, specify the filter bandwidth in the block dialog box through the **Filter bandwidth** parameter.

**Dependency**

To enable this parameter, set **Filter type** to Bandpass or Bandstop.

**Filter bandwidth** — Filter bandwidth

7680 (default) | positive scalar

Specify the bandwidth of the FIR filter as a real positive scalar in Hz or in normalized frequency units.

If you set the **Sample rate mode** parameter to:

- Specify on dialog or Inherit from input port -- The value of the filter bandwidth is in Hz and must be less than half the value of the input sample rate.
- Use normalized frequency (0 to 1) -- The value of the filter bandwidth is in normalized frequency units. The value must be a positive scalar less than 1.0.

**Tunable:** Yes

**Dependencies**

To enable this parameter:

- Set **Filter type** to Bandpass or Bandstop.
- Clear the **Specify bandwidth from input port** parameter.

**Window function** — Window function

Hann (default) | Hamming | Chebyshev | Kaiser

Specify the window function used to design the FIR filter. You can set this parameter to:

- Hann
- Hamming
- Chebyshev
- Kaiser

**Chebyshev window sidelobe attenuation (dB)** — Chebyshev window sidelobe attenuation

60 (default) | positive scalar

Specify the sidelobe attenuation of the Chebyshev window as a real positive scalar.

**Dependencies**

To enable this parameter, set **Window function** to Chebyshev.

**Kaiser window parameter** — Kaiser window parameter

0.5 (default) | real scalar

Specify the Kaiser window parameter as a real scalar.

**Dependencies**

To enable this parameter, set **Window function** to Kaiser.

**Sample rate mode** — Mode to specify the input sample rate

Specify on dialog (default) | Inherit from input port | Use normalized frequency (0 to 1)

Specify the input sample rate using one of these options:

- **Specify on dialog** -- Specify the input sample rate in the block dialog box using the **Input sample rate (Hz)** parameter.
- **Inherit from input port** -- The block inherits the sample rate from the input signal as  $N / T_s$ , where  $N$  is the frame size of the input signal and  $T_s$  is the sample time of the input signal.
- **Use normalized frequency (0 to 1)** -- Specify the filter cutoff frequency, center frequency, and the filter bandwidth in normalized frequency units (0 to 1).

**Input sample rate (Hz)** — Input sample rate

44100 (default) | positive scalar

Specify the sample rate of the input signal as a positive scalar in Hz.

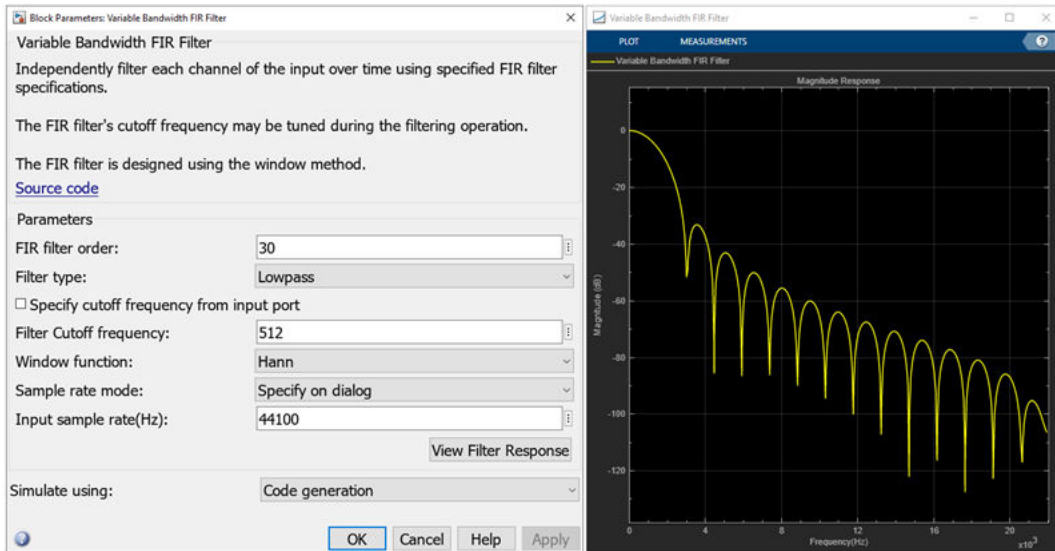
**Dependencies**

To enable this parameter, set the **Sample rate mode** parameter to **Specify on dialog**.

**View Filter Response** — View Filter Response  
button

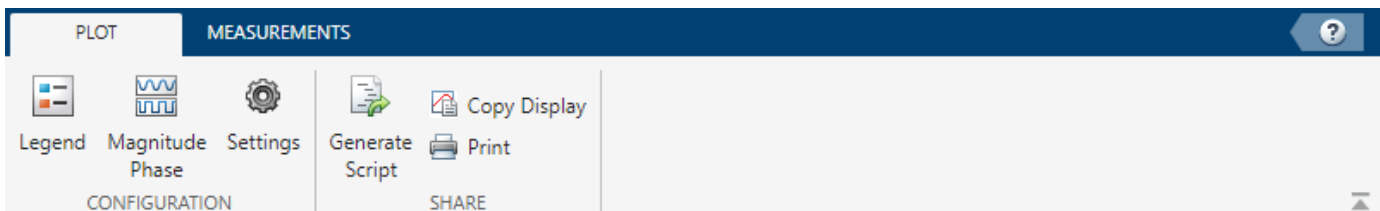
Open the dynamic filter visualizer and display the magnitude response of the variable bandwidth FIR filter. The response is based on the parameters you select in the block dialog box. To update the

magnitude response while the dynamic filter visualizer is running, modify the parameters in the dialog box and click **Apply**.

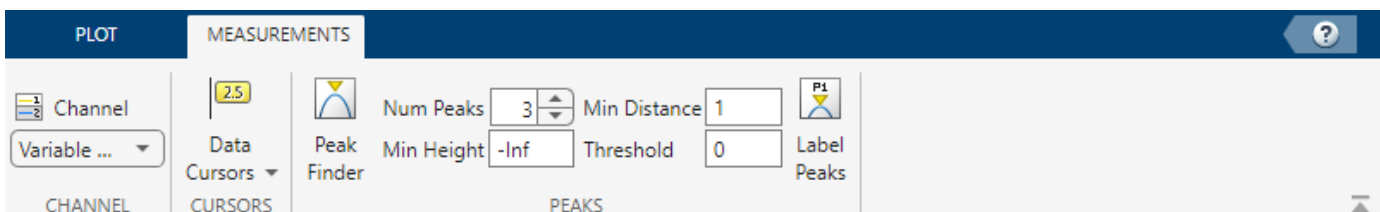


You can configure the plot settings and the signal measurements from the interface of the visualizer.

On the **Plot** tab, the **Configuration** section allows you to modify the plot settings.



On the **Measurements** tab, you can measure the signal statistics, place data cursors, and display the peak values of the selected signal.



For more details on the dynamic filter visualizer interface and its tools, see `dsp.DynamicFilterVisualizer`.

**Simulate using** — Simulate using

Code generation (default) | Interpreted execution

Specify the type of simulation to run. You can set this parameter to:

- Code generation (default)

Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than Interpreted execution.

- Interpreted execution

Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than Code generation.

## Block Characteristics

|                                 |                 |
|---------------------------------|-----------------|
| <b>Data Types</b>               | double   single |
| <b>Multidimensional Signals</b> | No              |
| <b>Variable-Size Signals</b>    | Yes             |

## Algorithms

### FIR Transformations

All transformations assume a lowpass filter of length  $2N+1$ .

#### Lowpass to Lowpass

Consider an ideal lowpass brickwall filter with normalized cutoff frequency  $\omega_{c1}$ . By taking the inverse discrete Fourier transform of the ideal frequency response, and clipping the resulting sequence to length  $2N+1$ , the impulse response is:

$$\begin{aligned} & \text{for } n = 0 \\ & h_{LP}(n) = \frac{\omega_c}{\pi} \cdot w(0) \\ & \text{for } 1 \leq |n| \leq N \\ & h_{LP}(n) = \frac{\sin(\omega_c n)}{\pi n} \cdot w(n) \end{aligned}$$

where  $w(n)$  is the window vector. Tune the lowpass filter coefficients to a new cutoff frequency  $\omega_{c2}$  as follows:

$$\begin{aligned} & \text{for } n = 0 \\ & h_{LP}(n) = \frac{\omega_{c2}}{\pi} \cdot \omega(0) \\ & \text{for } 1 \leq |n| \leq N \\ & h_{LP}(n) = \frac{\sin(\omega_{c2} n)}{\pi n} \cdot \omega(n) \end{aligned}$$

You do not need to recompute the window every time you tune the cutoff frequency.

### Lowpass to Highpass

Assuming a lowpass filter with normalized 6-dB cutoff frequency  $\omega_c$ , you can obtain a highpass filter with the same cutoff frequency by taking the complementary of the lowpass frequency response:

$$H_{HP}(e^{j\omega}) = 1 - H_{LP}(e^{j\omega})$$

Taking the inverse discrete Fourier transform of the above response, you have the following highpass filter coefficients:

$$\begin{aligned} & \text{for } n = 0 \\ & h_{hp}(n) = 1 - h_{LP}(n) \\ & \text{for } 1 \leq |n| \leq N \\ & h_{hp}(n) = -h_{LP}(n) \end{aligned}$$

### Lowpass to Bandpass

Obtain a bandpass filter centered at frequency  $\omega_0$  by shifting the lowpass response:

$$H_{BP}(e^{j\omega}) = H_{LP}(e^{j(\omega-\omega_0)}) + H_{LP}(e^{j(\omega+\omega_0)})$$

The bandwidth of the resulting bandpass filter is  $2\omega_c$ , as measured between the two cutoff frequencies of the bandpass filter. The equivalent bandpass filter coefficients are then:

$$\begin{aligned} h_{BP}(n) &= (e^{j\omega_0 n} + e^{-j\omega_0 n})h_{LP}(n) \\ & \text{which can be written as:} \\ h_{BP}(n) &= 2\cos(\omega_0 n)h_{LP}(n) \end{aligned}$$

### Lowpass to Bandstop

You can transform a lowpass filter to a bandstop filter by combining the highpass and bandpass transformations. First make the filter bandpass by shifting the lowpass response, and then invert it to get a bandstop response centered at  $\omega_0$ .

$$H_{BS}(e^{j\omega}) = 1 - (H_{LP}(e^{j(\omega-\omega_0)}) + H_{LP}(e^{j(\omega+\omega_0)}))$$

This yields the following coefficients:

$$\begin{aligned} & \text{for } n = 0 \\ & h_{BS}(n) = 1 - 2\cos(\omega_0 n)h_{LP}(n) \\ & \text{for } 1 \leq |n| \leq N \\ & h_{BS}(n) = -2\cos(\omega_0 n)h_{LP}(n) \end{aligned}$$

### Generalized Transformation

You can combine these transformations to transform a lowpass filter to a lowpass, highpass, bandpass, or bandstop filter with arbitrary cutoffs.

For example, to transform a lowpass filter with cutoff  $\omega_{c1}$  to a highpass with cutoff  $\omega_{c2}$ , you first apply the lowpass-to-lowpass transformation to get a lowpass filter with cutoff  $\omega_{c2}$ , and then apply the lowpass-to-highpass transformation to get the highpass with cutoff  $\omega_{c2}$ .

To get a bandpass filter with center frequency  $\omega_0$  and bandwidth  $\beta$ , we first apply the lowpass-to-lowpass transformation to go from a lowpass with cutoff  $\omega_c$  to a lowpass with cutoff  $\beta/2$ , and then

apply the lowpass-to-bandpass transformation to get the desired bandpass filter. You can use the same approach for a bandstop filter.

## Version History

Introduced in R2015a

### R2023a: Support for normalized frequencies

When you set the **Sample rate mode** parameter to Use normalized frequency (0 to 1), you can specify the filter cutoff frequency, center frequency, and filter bandwidth in normalized frequency units (0 to 1).

## References

[1] Jarske, P., Y. Neuvo, and S. K. Mitra. "A Simple Approach to the Design of Linear Phase FIR Digital Filters with Variable Characteristics." *Signal Processing* 14, no. 4 \*(1988): 313-326.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

The Variable Bandwidth FIR Filter block supports SIMD code generation using Intel AVX2 technology when the input signal has a data type of `single` or `double`.

The SIMD technology significantly improves the performance of the generated code.

## See Also

### Objects

`dsp.VariableBandwidthFIRFilter` | `dsp.VariableBandwidthIIRFilter`

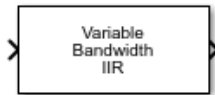
### Blocks

Biquad Filter | Variable Bandwidth IIR Filter



# Variable Bandwidth IIR Filter

Design tunable bandwidth IIR filter



## Libraries:

DSP System Toolbox / Filtering / Filter Designs

## Description

The Variable Bandwidth IIR Filter block filters each channel of the input signal over time using the IIR filter specifications. This block offers tunable filter design parameters, which enable you to tune the filter characteristics while the simulation is running.

The block designs the IIR filter according to the filter parameters set in the block dialog box. The output port properties, such as datatype, complexity, and dimension, are identical to the input port properties.

## Ports

### Input

**x** — Data input  
vector | matrix

Specify the data input as a vector or a matrix. The block treats each column of the input signal as a separate channel. If the input is a two-dimensional signal, the first dimension represents the channel length (or frame size) and the second dimension represents the number of channels. If the input is a one-dimensional signal, then the block interprets it as having a single channel.

The block accepts variable-size input signals, that is, you can change the size of each input channel during simulation but you cannot change the number of channels.

This port is unnamed until you select one of these parameters:

- **Specify passband frequency from input port**
- **Specify center frequency from input port**
- **Specify bandwidth from input port**

Data Types: single | double

**Fp** — Filter passband frequency  
positive scalar

Specify the passband frequency of the IIR filter as a real positive scalar in Hz or in normalized frequency units.

### Dependencies

To enable this port, select the **Specify passband frequency from input port** parameter.

Data Types: `single` | `double`

**Fc** — Filter center frequency  
positive scalar

Specify the center frequency of the IIR filter as a real positive scalar in Hz or in normalized frequency units.

#### **Dependencies**

To enable this port, select the **Specify center frequency from input port** parameter.

Data Types: `single` | `double`

**BW** — Filter bandwidth  
positive scalar

Specify the bandwidth of the IIR filter as a real positive scalar in Hz or in normalized frequency units.

#### **Dependencies**

To enable this port, select the **Specify bandwidth from input port** parameter.

Data Types: `single` | `double`

#### **Output**

**Port\_1** — Filtered output  
vector | matrix

Filtered output, returned as a vector or a matrix. The size, data type, and complexity of the output signal matches that of the input signal.

Data Types: `single` | `double`

### **Parameters**

**Filter type** — Filter type

Lowpass (default) | Highpass | Bandpass | Bandstop

Specify the type of IIR filter. You can set this parameter to:

- Lowpass
- Highpass
- Bandpass
- Bandstop

**IIR filter order** — IIR filter order

8 (default) | positive integer

Specify the order of the IIR filter as a positive integer.

**Specify passband frequency from input port** — Specify passband frequency from input port

off (default) | on

When you select this check box, specify the filter passband frequency through the **Fp** port. When you clear this check box, specify the passband frequency in the block dialog box through the **Filter passband frequency** parameter.

**Dependency**

To enable this parameter, set **Filter type** to Lowpass or Highpass.

**Filter passband frequency** — Filter passband frequency

512 (default) | positive scalar

Specify the filter passband frequency as a real, positive scalar in Hz or in normalized frequency units.

If you set the **Sample rate mode** parameter to:

- Specify on dialog or Inherit from input port -- The value of the filter passband frequency is in Hz and must be less than half the value of the input sample rate.
- Use normalized frequency (0 to 1) -- The value of the filter passband frequency is in normalized frequency units. The value must be a positive scalar less than 1.0.

**Tunable:** Yes

**Dependencies**

To enable this parameter:

- Set **Filter type** to Lowpass or Highpass.
- Clear the **Specify passband frequency from input port** parameter.

**Specify center frequency from input port** — Specify center frequency from input port

off (default) | on

When you select this check box, specify the center frequency of the IIR filter through the **Fc** port. When you clear this check box, specify the center frequency in the block dialog box through the **Filter center frequency** parameter.

**Dependencies**

To enable this parameter, set **Filter type** to Bandpass or Bandstop.

**Filter center frequency** — Filter center frequency

44100/4 (default) | positive scalar

Specify the filter center frequency as a real positive scalar in Hz or in normalized frequency units.

If you set the **Sample rate mode** parameter to:

- Specify on dialog or Inherit from input port -- The value of the filter center frequency is in Hz and must be less than half the value of the input sample rate.
- Use normalized frequency (0 to 1) -- The value of the filter center frequency is in normalized frequency units. The value must be a positive scalar less than 1.0.

**Tunable:** Yes

**Dependencies**

To enable this parameter:

- Set **Filter type** to Bandpass or Bandstop.
- Clear the **Specify center frequency from input port** parameter.

**Specify bandwidth from input port** — Specify bandwidth from input port

off (default) | on

When you select this check box, specify the bandwidth of the IIR filter through the **BW** port. When you clear this check box, specify the filter bandwidth in the block dialog box through the **Filter bandwidth** parameter.

**Dependencies**

To enable this parameter, set **Filter type** to Bandpass or Bandstop.

**Filter bandwidth** — Filter bandwidth

7680 (default) | positive scalar

Specify the filter bandwidth as a positive scalar in Hz or in normalized frequency units.

If you set the **Sample rate mode** parameter to:

- **Specify on dialog or Inherit from input port** -- The value of the filter bandwidth is in Hz and must be less than half the value of the input sample rate.
- **Use normalized frequency (0 to 1)** -- The value of the filter bandwidth is in normalized frequency units. The value must be a positive scalar less than 1.0.

**Tunable:** Yes

**Dependencies**

To enable this parameter:

- Set **Filter type** to Bandpass or Bandstop.
- Clear the **Specify bandwidth from input port** parameter.

**Filter passband ripple (dB)** — Filter passband ripple

1 (default) | positive scalar

Specify the filter passband ripple as a real, positive scalar.

**Filter Stopband attenuation (dB)** — Filter Stopband attenuation

60 (default) | positive scalar

Specify the filter stopband attenuation as a real, positive scalar.

**Sample rate mode** — Mode to specify the input sample rate

Specify on dialog (default) | Inherit from input port | Use normalized frequency (0 to 1)

Specify the input sample rate using one of these options:

- **Specify on dialog** -- Specify the input sample rate in the block dialog box using the **Input sample rate (Hz)** parameter.
- **Inherit from input port** -- The block inherits the sample rate from the input signal as  $N / T_s$ , where  $N$  is the frame size of the input signal and  $T_s$  is the sample time of the input signal.
- **Use normalized frequency (0 to 1)** -- Specify the filter passband frequency, center frequency, and the filter bandwidth in normalized frequency units (0 to 1).

**Input sample rate (Hz)** — Input sample rate

44100 (default) | positive scalar

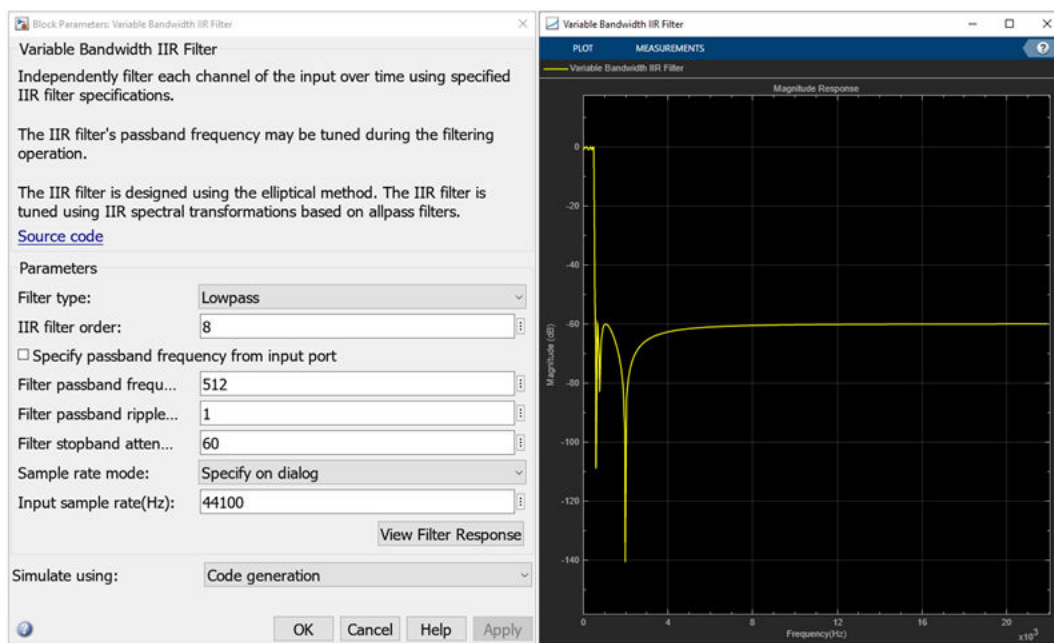
Specify the sample rate of the input signal as a positive scalar in Hz.

#### Dependencies

To enable this parameter, set the **Sample rate mode** parameter to **Specify on dialog**.

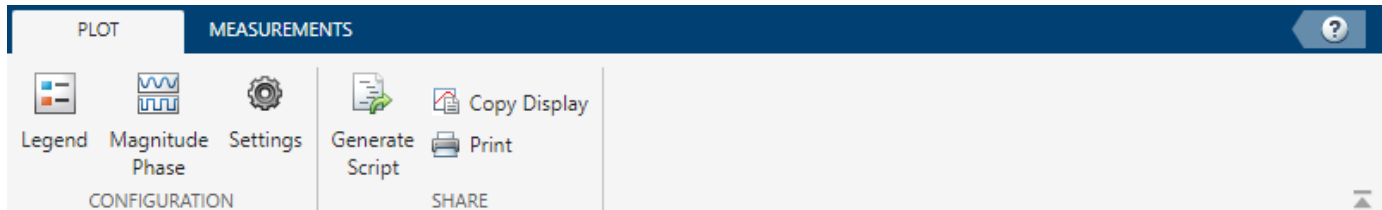
**View Filter Response** — View Filter Response button

Opens the dynamic filter visualizer and displays the magnitude response of the variable bandwidth IIR filter. The response is based on the parameters you select in the Block Parameters dialog box. To update the magnitude response while the dynamic filter visualizer is running, modify the parameters in the dialog box and click **Apply**.

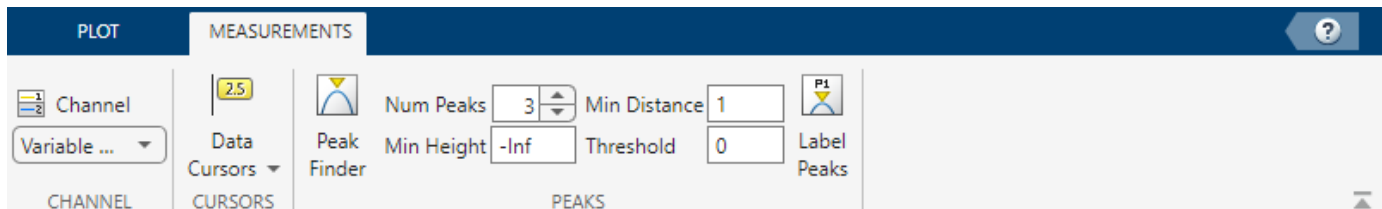


You can configure the plot settings and the signal measurements from the interface of the visualizer.

On the **Plot** tab, the **Configuration** section allows you to modify the plot settings.



On the **Measurements** tab, you can measure the signal statistics, place data cursors, and display the peak values of the selected signal.



For more details on the dynamic filter visualizer interface and its tools, see `dsp.DynamicFilterVisualizer`.

**Simulate using** — Simulate using

Code generation (default) | Interpreted execution

Specify the type of simulation to run. You can set this parameter to:

- Code generation (default)

Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than Interpreted execution.

- Interpreted execution

Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than Code generation.

## Block Characteristics

|                                 |                 |
|---------------------------------|-----------------|
| <b>Data Types</b>               | double   single |
| <b>Multidimensional Signals</b> | No              |
| <b>Variable-Size Signals</b>    | Yes             |

## Algorithms

This filter covers frequency transformations. The algorithm designs a lowpass IIR prototype using the elliptical method by specifying its order, passband frequency, passband ripple, and stopband

attenuation. The passband ripple and stopband attenuation are equal to the values of the `PassbandRipple` and `StopbandAttenuation` properties. The algorithm sets the prototype passband frequency to 0.5. If the `FilterType` property is 'Lowpass' or 'Highpass', the order of the prototype filter is equal to the value of `FilterOrder`. If the `FilterType` property is 'Bandpass' or 'Bandstop', the order of the prototype filter is equal to `FilterOrder/2`. The prototype is a Direct Form II Transposed cascade of second-order sections (Biquad filter). The prototype is transformed into the desired filter using the algorithms used in "Digital Frequency Transformations". Each prototype SOS section is transformed separately. When `FilterType` is 'Lowpass' or 'Highpass', the resulting filter remains a Direct Form II Transposed cascade of second order sections. If the `FilterType` is 'Bandpass' or 'Bandstop', the resulting filter is a cascade of Direct Form II Transposed cascade of fourth order sections.

## Version History

Introduced in R2015a

### R2023a: Support for normalized frequencies

When you set the **Sample rate mode** parameter to Use normalized frequency (0 to 1), you can specify the filter passband frequency, center frequency, and filter bandwidth in normalized frequency units (0 to 1).

## References

[1] A. G. Constantinides. "Spectral Transformations for Digital Filters." Proceedings of the Institution of Electrical Engineers 117, no. 8 (1970):1585-1590.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

## See Also

### Blocks

Biquad Filter | Variable Bandwidth FIR Filter

### Objects

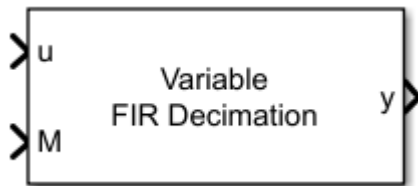
`dsp.VariableBandwidthFIRFilter` | `dsp.VariableBandwidthIIRFilter`

### Topics

"Tunable Lowpass Filtering of Noisy Input in Simulink"

## Variable FIR Decimation

Polyphase FIR decimation with tunable decimation factor



### Libraries:

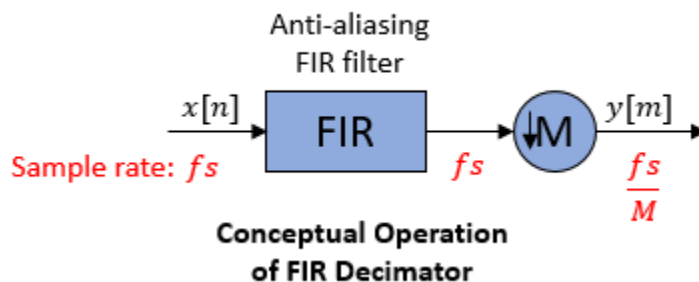
DSP System Toolbox / Filtering / Multirate Filters

### Description

The Variable FIR Decimation block performs an efficient polyphase FIR decimation with a tunable decimation factor. You can update the decimation factor and the filter coefficients in the block dialog box or through an input port while the simulation is running.

Conceptually, the FIR decimator (as shown in the schematic) consists of an anti-aliasing FIR filter followed by a downsampler. To design an FIR anti-aliasing filter, use the `designMultirateFIR` function.

The FIR filter filters the data in each channel of the input using a direct-form FIR filter. The downsampler that follows downsamples each channel of filtered data by taking every  $M$ -th sample and discarding the  $M - 1$  samples that follow.  $M$  is the value of the decimation factor that you specify. The resulting discrete-time signal has a sample rate that is  $1/M$  times the original sample rate.



Note that the actual block algorithm implements a direct-form FIR polyphase structure, an efficient equivalent of the combined system depicted in the diagram. For more details, see “Algorithms” on page 2-1500.

The block supports C and C++ code generation.

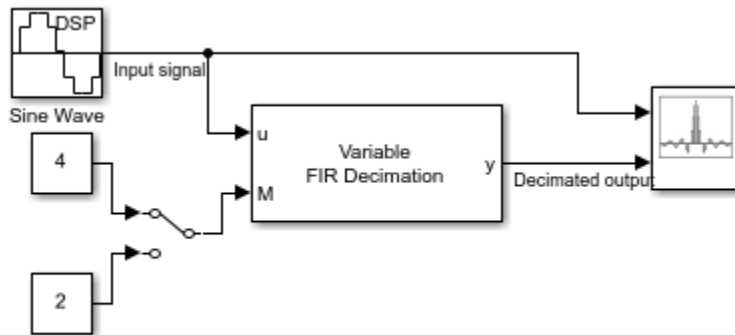
### Examples



## Decimate Sinusoidal Signal with Tunable Decimation Factor

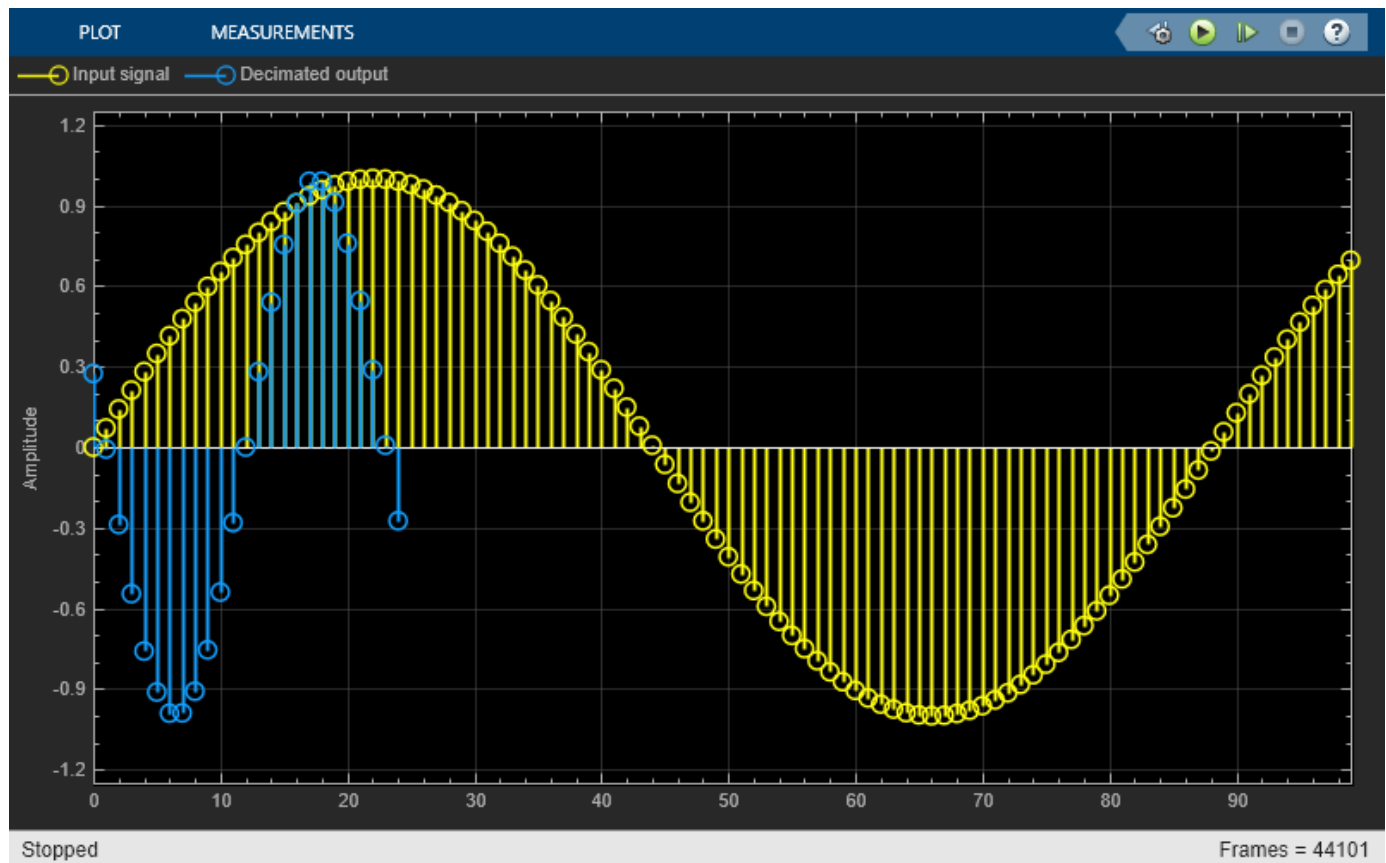
Decimate a sinusoidal signal by varying the decimation factor using the Variable FIR Decimation block. You can vary the decimation factor in the block dialog box or through an input port while the simulation is running.

Open the model. The input is a sinusoidal signal with a frequency of 500 Hz, sample time of  $1/44100$  s, and contains 100 samples per frame. Pass this signal through the Variable FIR Decimation block. The **Maximum decimation factor** parameter in the block is 24. The decimation factor that you input through the port is 4.

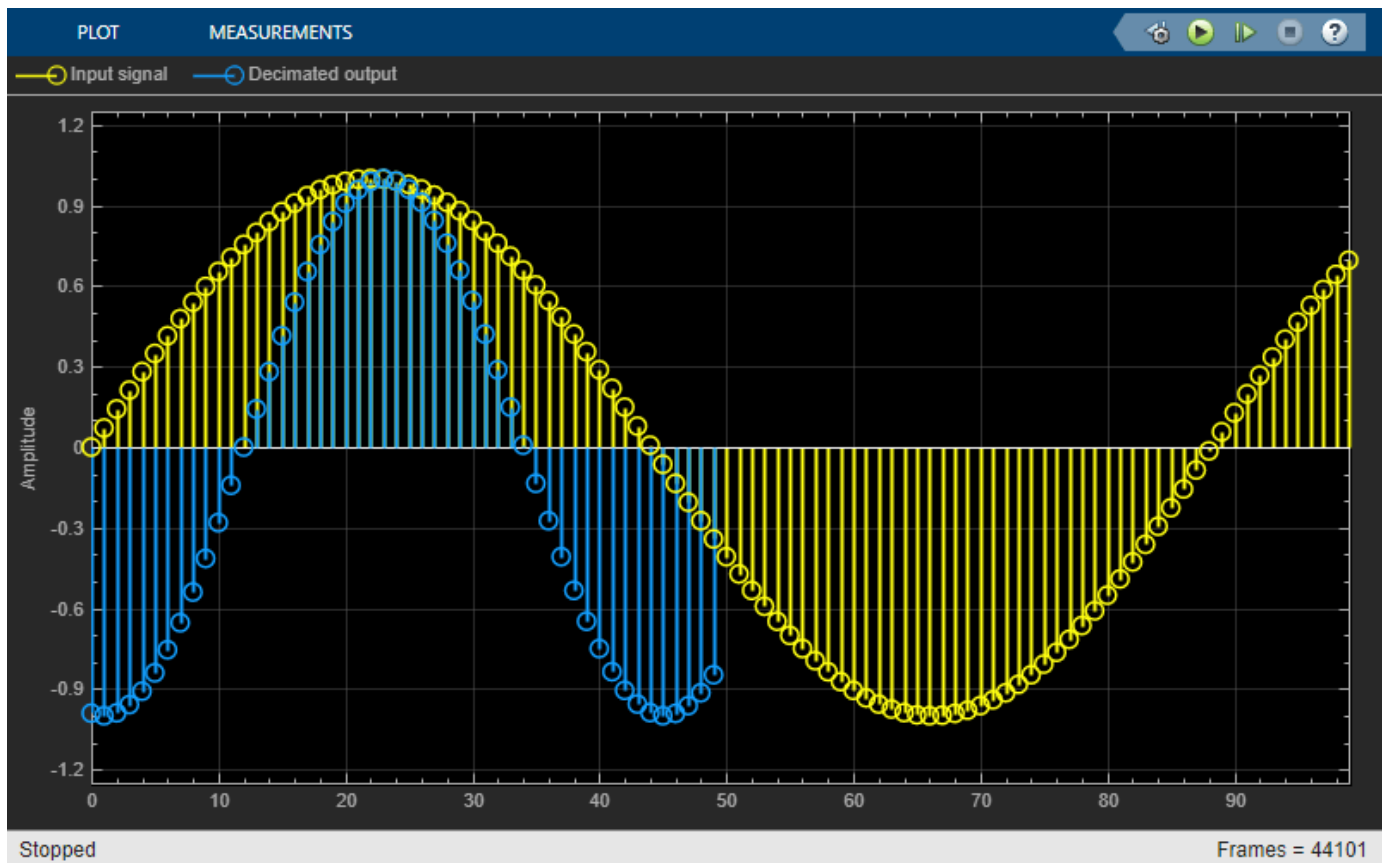


Copyright 2023 The MathWorks, Inc.

Run the model. The Array Plot block shows the input signal and the decimated output on the display.



While the simulation is running, change the decimation factor to 2 by clicking the Manual Switch. The span of the decimated output updates in the Array Plot display. You can change the decimation factor to any value that is an integer factor of the maximum decimation factor of 24.



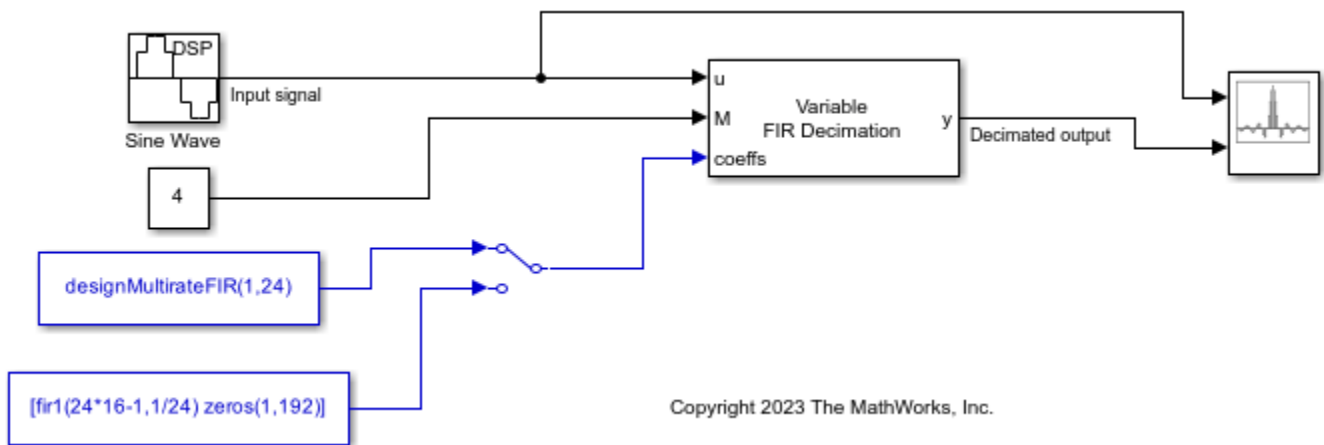
If you specify the decimation factor in the block dialog box, you can tune the **Decimation factor** parameter in the block dialog box while the simulation is running and the block updates the decimated output accordingly.

### Specify Filter Coefficients Through Input Port

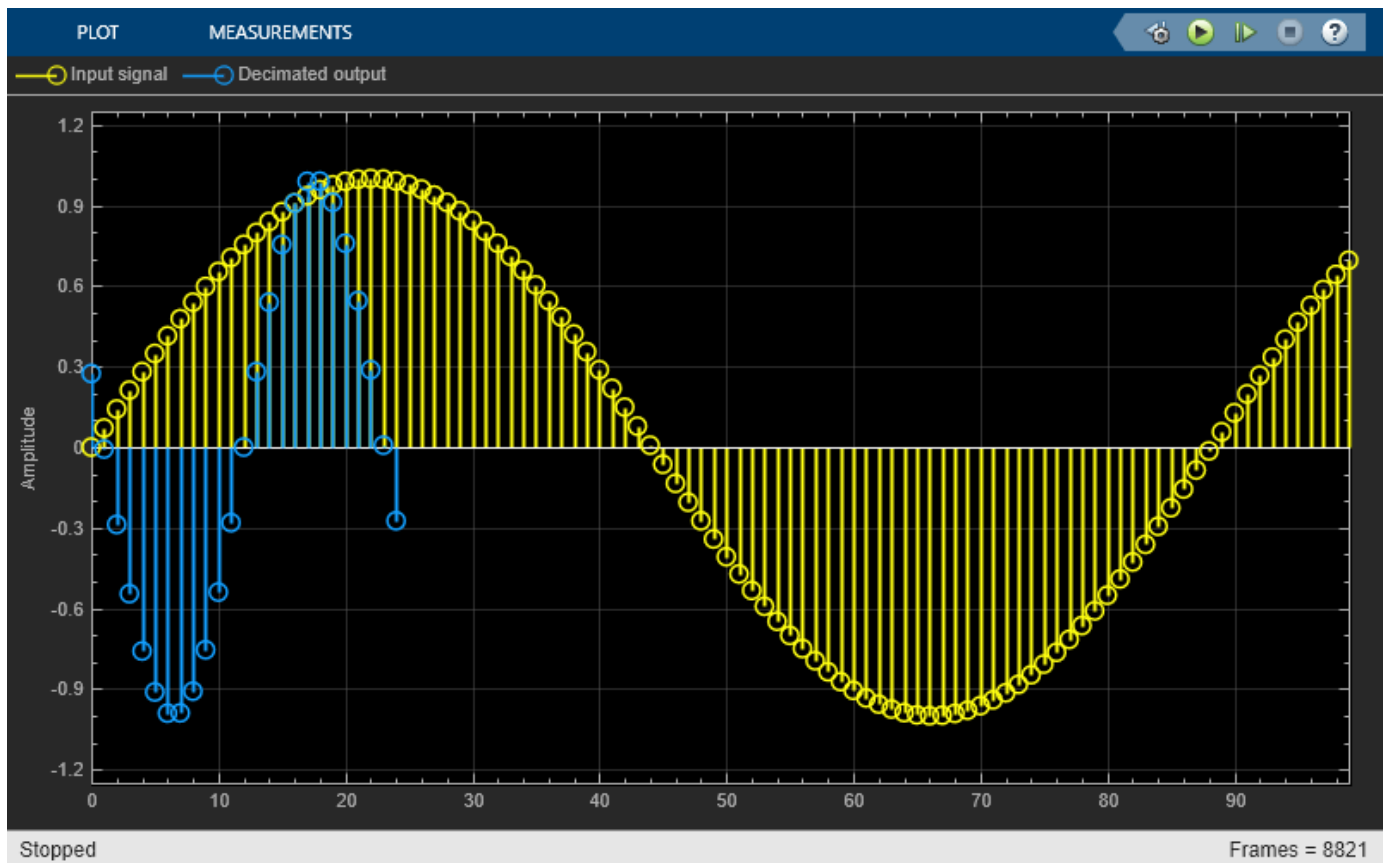
Decimate a sinusoidal signal using the Variable FIR Decimation block. You can vary the filter coefficients in the block dialog or through an input port while the simulation is running.

Open the model. The input is a sinusoidal signal with a frequency of 500 Hz, sample time of 1/44100 s, and contains 100 samples per frame. Pass this signal through the Variable FIR Decimation block. The **Maximum decimation factor** parameter in the Variable FIR Decimation block is set to 24. Specify the decimation factor and the filter coefficients through the **M** and the **coeffs** ports, respectively. The decimation factor is 4 and the filter coefficients are generated using the `designMultirateFIR(1,24)` function. This function generates an effective anti-aliasing lowpass filter with a normalized cutoff frequency no greater than 1/24.

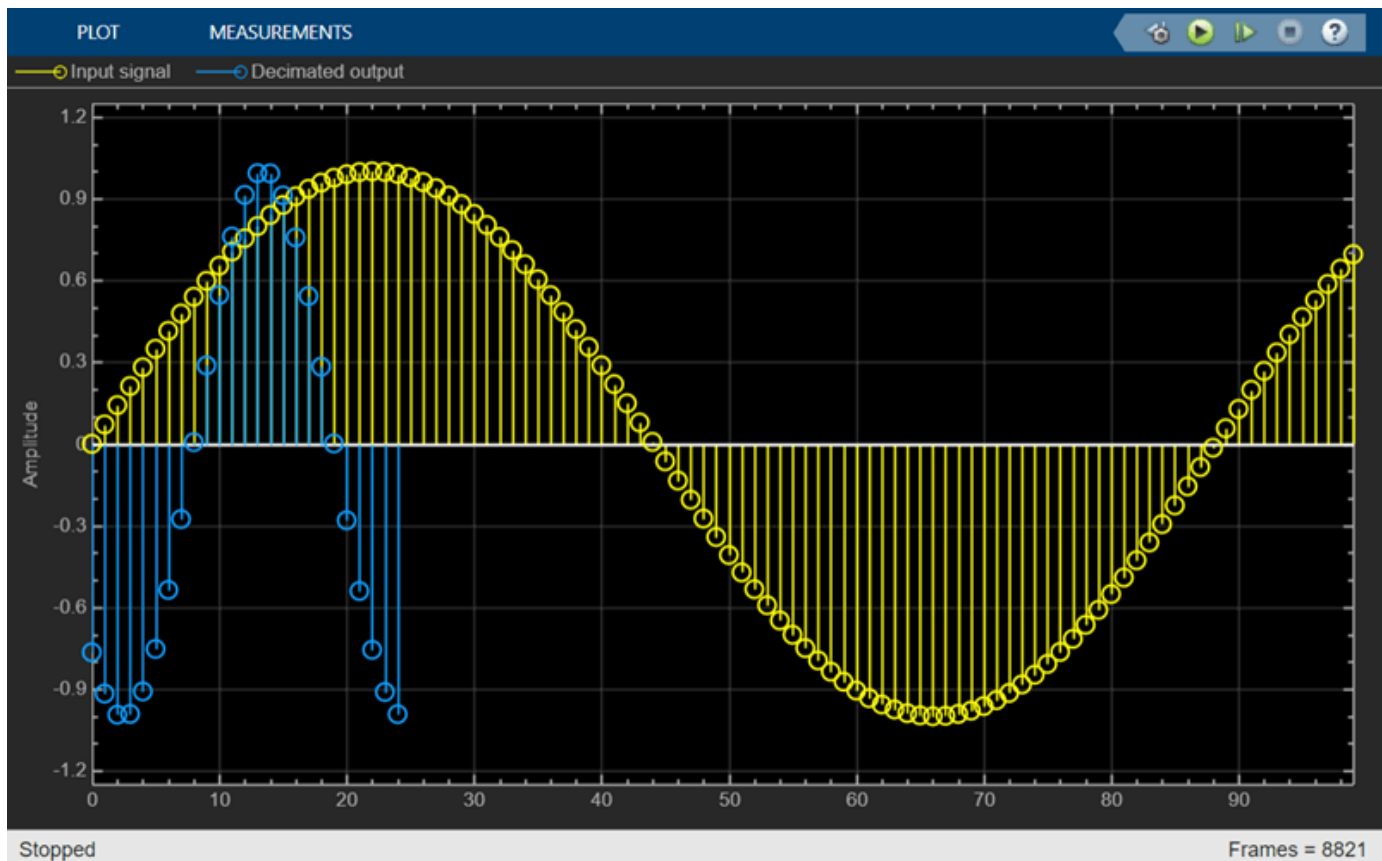
You can vary the filter coefficients using the Manual Switch.



Run the model. The Array Plot block shows the input signal and the decimated output in the display.



While the simulation is running, change the filter coefficients by clicking the Manual Switch. On the second branch, the `fir1` function generates the coefficients of a lowpass filter that has a similar passband frequency response and the same number of coefficients as the first filter. Note that you cannot change the number of filter coefficients while the simulation is running.



## Ports

### Input

**u** — Data input  
vector | matrix

Specify the data input as a vector or a matrix of size  $P$ -by- $Q$ . The block treats each column of the input signal as a separate channel. If the input is a two-dimensional signal, the first dimension represents the channel length (or frame size) and the second dimension represents the number of channels. If the input is a one-dimensional signal, then the block interprets it as a single channel.

The block accepts variable-size input signals, that is, you can change the size of each input channel during simulation, but you cannot change the number of channels.

Data Types: `single` | `double`

**M** — Decimation factor input port  
positive integer

Specify the decimation factor  $M$  as a positive integer that is an integer factor of the maximum decimation factor. For example, if you set **Maximum decimation factor** to 24, then the possible values for decimation factor are 1, 2, 3, 4, 6, 8, 12, and 24.

You can tune the decimation factor while the simulation is running. For an example, see “Decimate Sinusoidal Signal with Tunable Decimation Factor” on page 2-1488.

**Dependencies**

To enable this port, select the **Specify decimation factor from input port** parameter.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**coeffs** — Prototype filter coefficients  
vector

Specify the lowpass FIR filter coefficients in descending powers of  $z$  as a vector.

The transfer function  $H(z)$  of the FIR filter is given by:

You can generate the FIR filter coefficient vector,  $\mathbf{b} = [b_0, b_1, \dots, b_N]$ , using one of the DSP System Toolbox filter design functions such as `designMultirateFIR`, `firnyquist`, `firhalfband`, `firgr`, or `firceqrip`.

Compute the filter coefficients based on the maximum decimation factor  $M_{max}$  instead of the decimation factor  $M$  that you specify during simulation. For example, if  $M_{max}$  is 24 and  $M$  is 8, and you used `designMultirateFIR` as the design function, compute the filter coefficients using `designMultirateFIR(1,24)` instead of `designMultirateFIR(1,8)`.

To act as an effective anti-aliasing filter, the coefficients usually correspond to a lowpass filter with a normalized cutoff frequency no greater than  $1/M_{max}$ . To design such a filter, use the `designMultirateFIR` function.

The block internally initializes all filter states to zero.

You can change the filter coefficients during simulation but the number of filter coefficients must remain constant. For an example, see “Specify Filter Coefficients Through Input Port” on page 2-1491.

**Dependencies**

To enable this port, set the **Coefficient source** parameter to `Input port`.

Data Types: `single` | `double`

**Output**

**y** — Decimated output  
vector | matrix

Decimated output, returned as a vector or a matrix. The size of the output signal is same as the size of the input signal. However, the block computes a maximum of  $\text{ceil}(P/M)$ -by- $Q$  output samples, where  $P$  is the input frame size,  $Q$  is the number of input channels, and  $M$  is the decimation factor that you specify during simulation.

The complexity of the output signal depends on the complexity of the input signal and the complexity of the filter coefficients. See this table for more details.

| Input Signal | Filter Coefficients | Output Signal |
|--------------|---------------------|---------------|
| Real         | Complex             | Complex       |
| Real         | Real                | Real          |

| Input Signal | Filter Coefficients | Output Signal |
|--------------|---------------------|---------------|
| Complex      | Complex             | Complex       |
| Complex      | Real                | Complex       |

Data Types: single | double

## Parameters

**Maximum decimation factor** — Maximum decimation factor

24 (default) | positive integer

Specify the maximum decimation factor  $M_{max}$  as a positive integer. The decimation factor you specify through the input port **M** must be a factor of the value you specify in this parameter.

**Specify decimation factor from input port** — Specify decimation factor from input port

on (default) | off

When you select this parameter, you can specify the decimation factor through the input port **M**. When you clear this parameter, you can specify the decimation factor in the block dialog box through the **Decimation factor** parameter.

**Decimation factor** — Decimation factor

24 (default) | positive integer

Specify the decimation factor  $M$  as a positive integer that is an integer factor of the maximum decimation factor  $M_{max}$ . For example, if you set **Maximum decimation factor** to 24, then the possible values for **Decimation factor** are 1, 2, 3, 4, 6, 8, 12, and 24.

You can tune the decimation factor while the simulation is running.

**Tunable:** Yes

### Dependencies

To enable this parameter, clear the **Specify decimation factor from input port** parameter.

**Coefficient source** — FIR filter coefficient source

Auto (default) | Dialog parameter | Input port

Specify the FIR filter coefficient source as one of the following:

- **Auto** -- The block designs an FIR decimator using the decimation factor in the **Decimation factor** parameter. The `designMultirateFIR` function designs the filter and returns the coefficients used by the block.

For more information on the filter design, see Orfanidis [1].

- **Dialog parameter** -- Specify the filter coefficients through the **Prototype filter coefficients** parameter in the block dialog box.

- Input port -- Specify the filter coefficients through the **coeffs** input port.

### Prototype filter coefficients — Prototype filter coefficients

`designMultirateFIR(1,24)` (default) | vector

Specify the lowpass FIR filter coefficients in descending powers of  $z$  as a vector. By default, `designMultirateFIR(1,24)` computes the filter coefficients with the maximum decimation factor  $M_{max}$  of 24.

The transfer function  $H(z)$  of the FIR filter is given by:

You can generate the FIR filter coefficient vector,  $\mathbf{b} = [b_0, b_1, \dots, b_N]$ , using one of the DSP System Toolbox filter design functions such as `designMultirateFIR`, `firnyquist`, `firhalfband`, `firgr`, or `firceqrip`.

Compute the filter coefficients based on the maximum decimation factor  $M_{max}$  instead of the decimation factor  $M$  that you specify during simulation. For example, if  $M_{max}$  is 24 and  $M$  is 8, and you used `designMultirateFIR` as the design function, compute the filter coefficients using `designMultirateFIR(1,24)` instead of `designMultirateFIR(1,8)`.

To act as an effective anti-aliasing filter, the coefficients usually correspond to a lowpass filter with a normalized cutoff frequency no greater than  $1/M_{max}$ . To design such a filter, use the `designMultirateFIR` function.

The block internally initializes all filter states to zero.

**Tunable:** Yes

### Dependencies

To enable this parameter, set the **Coefficient source** parameter to Dialog parameter.

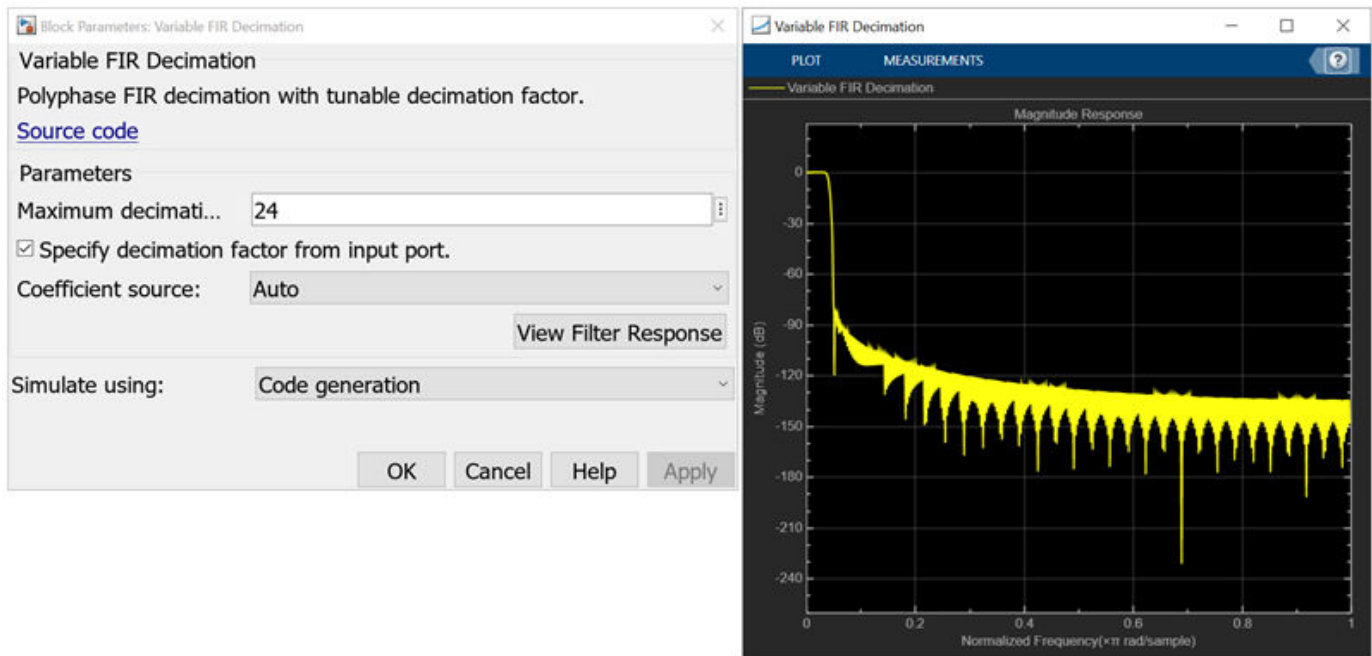
### View Filter Response — View Filter Response

button

Click this button to open the dynamic filter visualizer and display the magnitude response of the variable FIR decimation filter. The response is based on the parameters you select in the block dialog box. To update the magnitude response while the dynamic filter visualizer is running, modify the parameters in the dialog box and click **Apply**.

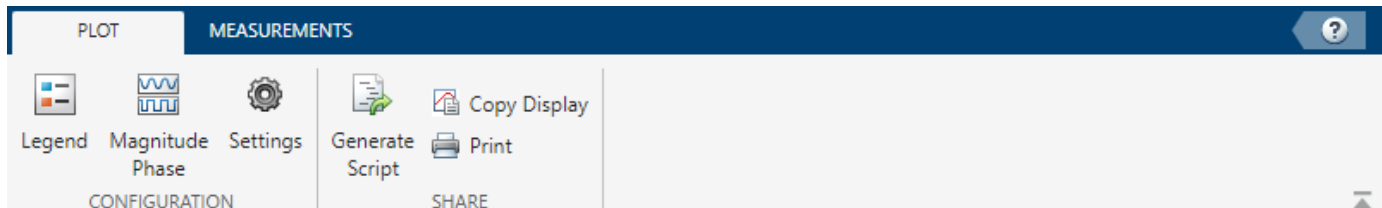
You cannot view the filter response when you set the **Coefficient source** parameter to Input port. To view the filter response, set the **Coefficient source** parameter to Auto or Dialog parameter.



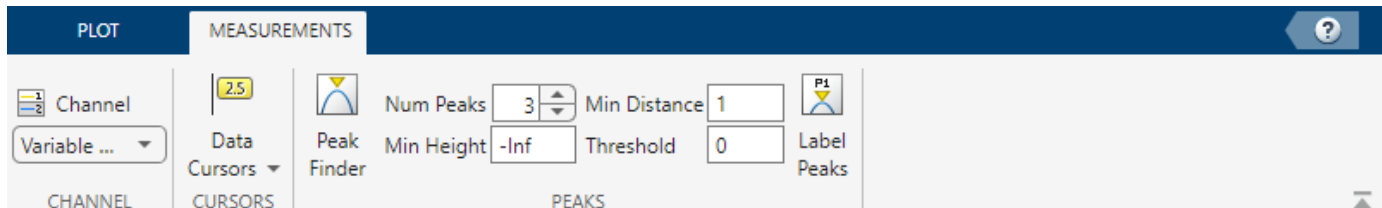


You can configure the plot settings and the signal measurements from the interface of the visualizer.

On the **Plot** tab, the **Configuration** section allows you to modify the plot settings. Click **Magnitude Phase** to display the magnitude and phase response of the filter.



On the **Measurements** tab, you can measure the signal statistics, place data cursors, and display the peak values of the selected signal.



For more details on the dynamic filter visualizer interface and its tools, see `dsp.DynamicFilterVisualizer`.

**Simulate using** — Simulate using

Code generation (default) | Interpreted execution

Specify the type of simulation to run. You can set this parameter to:

- **Code generation:** Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than **Interpreted execution**.
- **Interpreted execution:** Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than **Code generation**.

## Block Characteristics

|                                 |                 |
|---------------------------------|-----------------|
| <b>Data Types</b>               | double   single |
| <b>Direct Feedthrough</b>       | no              |
| <b>Multidimensional Signals</b> | no              |
| <b>Variable-Size Signals</b>    | yes             |
| <b>Zero-Crossing Detection</b>  | no              |

## More About

### Phase Continuity

Under certain conditions, the Variable FIR Decimation block maintains the phase across consecutive frames of data.

| Input Conditions                         | Downsample Factor Source   | Output Signal        | Phase Continuity |
|--|--|----------------------|------------------|
| Variable-size input signal               | Specify in block dialog box<br><br>Downsample factor does not change during simulation | Variable-size signal | Yes              |
| Fixed-size or variable-size input signal | Specify through input port<br><br>Downsample factor does not change during simulation  | Variable-size signal | Yes              |

Under certain conditions, the phase resets to 0 and is not maintained across consecutive frames of data.

| Input Conditions                         | Downsample Factor Source   | Output Signal        | Phase Continuity  |
|--|--|----------------------|-------------------|
| Fixed-size input signal                  | Specify in block dialog box<br><br>Downsample factor changes during simulation | Fixed-size signal    | Phase resets to 0 |
| Fixed-size or variable-size input signal | Specify through input port<br><br>Downsample factor changes during simulation  | Variable-size signal | Phase resets to 0 |

### Polyphase Subfilters

A polyphase implementation of an FIR decimator *splits* the lowpass FIR filter impulse response into  $M$  different subfilters, where  $M$  is the decimation factor that you specify during simulation. For more details on the polyphase implementation, see “Algorithms” on page 2-1500.

Let  $h(n)$  denote the FIR filter impulse response of length  $N+1$  and  $x(n)$  the input signal. Decimating the filter output by a factor of  $M$  is equivalent to the downsampled convolution:

$$y(n) = \sum_{l=0}^N h(l)x(nM - l)$$

The key to the efficiency of polyphase filtering is that specific input values are only multiplied by select values of the impulse response in the downsampled convolution. For example, letting  $M = 2$ , the input values  $x(0), x(2), x(4), \dots$  are combined only with the filter coefficients  $h(0), h(2), h(4), \dots$ , and the input values  $x(1), x(3), x(5), \dots$  are combined only with the filter coefficients  $h(1), h(3), h(5), \dots$ . By splitting the filter coefficients into two polyphase subfilters, no unnecessary computations are performed in the convolution. The outputs of the convolutions with the polyphase subfilters are interleaved and summed to yield the filter output.

The following code demonstrates how to construct the two polyphase subfilters for the default order 35 filter.

```
M = 2;
Num = fir1(35,0.4);
FiltLength = length(Num);
Num = flipud(Num(:));

if (rem(FiltLength, M) ~= 0)
    nzeros = M - rem(FiltLength, M);
    Num = [zeros(nzeros,1); Num]; % Appending zeros
end

len = length(Num);
nrows = len / M;
PolyphaseFilt = flipud(reshape(Num, M, nrows).');
```

The columns of `PolyphaseFilt` are subfilters containing the two *phases* of the filter in `Num`. For a general downsampling factor of  $M$ , there are  $M$  phases and therefore  $M$  subfilters.

## Algorithms

The FIR decimation filter is implemented efficiently using a polyphase structure. For more details on polyphase filters, see “Polyphase Subfilters” on page 2-1499.

To derive the polyphase structure, start with the transfer function of the FIR filter:

$$H(z) = b_0 + b_1z^{-1} + \dots + b_Nz^{-N}$$

$N+1$  is the length of the FIR filter.

You can rearrange this equation as follows:

$$H(z) = \begin{aligned} & (b_0 + b_{M_{\max}}z^{-M_{\max}} + b_{2M_{\max}}z^{-2M_{\max}} + \dots + b_{N-M_{\max}+1}z^{-(N-M_{\max}+1)}) + \\ & z^{-1}(b_1 + b_{M_{\max}+1}z^{-M_{\max}} + b_{2M_{\max}+1}z^{-2M_{\max}} + \dots + b_{N-M_{\max}+2}z^{-(N-M_{\max}+1)}) + \\ & \quad \vdots \\ & z^{-(M_{\max}-1)}(b_{M_{\max}-1} + b_{2M_{\max}-1}z^{-M_{\max}} + b_{3M_{\max}-1}z^{-2M_{\max}} + \dots + b_Nz^{-(N-M_{\max}+1)}) \end{aligned}$$

$M_{\max}$  is the number of polyphase components, and its value equals the maximum decimation factor that you specify.

You can write this equation as:

$$H(z) = E_0(z^{M_{\max}}) + z^{-1}E_1(z^{M_{\max}}) + \dots + z^{-(M_{\max}-1)}E_{M_{\max}-1}(z^{M_{\max}})$$

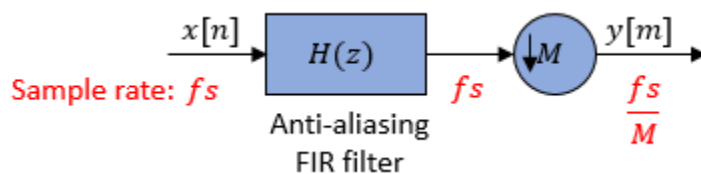
$E_0(z^{M_{\max}}), E_1(z^{M_{\max}}), \dots, E_{M_{\max}-1}(z^{M_{\max}})$  are the polyphase components of the FIR filter  $H(z)$ .

During simulation, the algorithm reconstructs the filter  $H(z)$  based on the decimation factor  $M$  that you specify during simulation. Rewriting  $H(z)$  in terms of the decimation factor  $M$ .

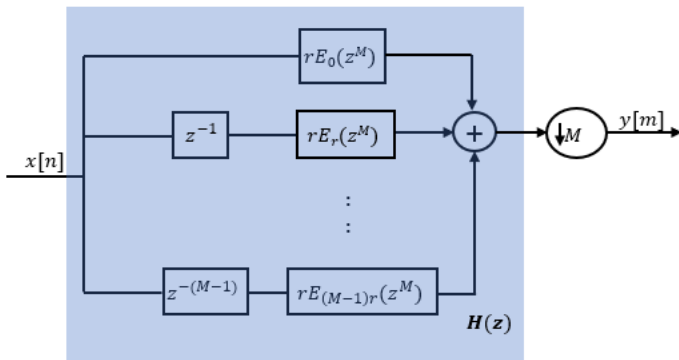
$$H(z) = rE_0(z^M) + z^{-1}rE_1(z^M) + \dots + z^{-(M-1)}rE_{(M-1)r}(z^M)$$

where,  $r = M_{\max}/M$ .

Conceptually, the FIR decimation filter contains a lowpass FIR filter followed by a downsampler.



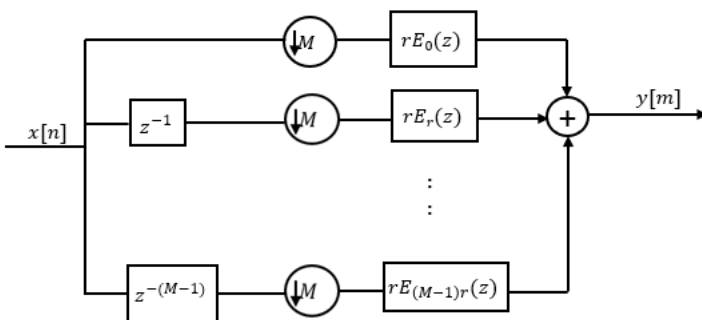
Replace  $H(z)$  with its polyphase representation.



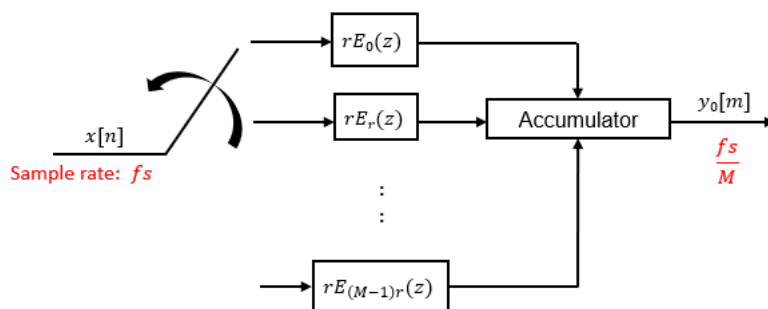
Here is the multirate noble identity for decimation.



Applying the noble identity for decimation moves the downsampling operation to before the filtering operation. This move enables you to filter the signal at a lower rate.



You can replace the delays and the decimation factor at the input with a commutator switch. The switch starts on the first branch 0 and moves in the counterclockwise direction as shown in this diagram. The accumulator at the output receives the processed input samples from each branch of the polyphase structure and accumulates these processed samples until the switch goes to branch 0. When the switch goes to branch 0, the accumulator outputs the accumulated value.



When the first input sample is delivered, the switch feeds this input to the branch 0 and the decimator computes the first output value. As more input samples come in, the switch moves in the counter clockwise direction through branches  $(M-1)r$ ,  $(M-2)r$ , and all the way up to branch 0, delivering one sample at a time to each branch. When the switch comes to branch 0, the decimator outputs the next set of output values. This process continues as data keeps coming in. Every time the switch comes to the branch 0, the decimator outputs  $y[m]$ . The decimator effectively outputs one sample for every  $M$  samples it receives. Hence the sample rate at the output of the FIR decimation filter is  $fs/M$ .

## Version History

Introduced in R2023a

## References

[1] Orfanidis, Sophocles J. *Introduction to Signal Processing*. Upper Saddle River, NJ: Prentice-Hall, 1996.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

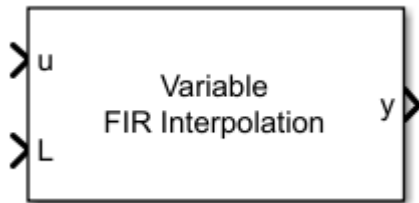
## See Also

### Blocks

Variable FIR Interpolation | FIR Decimation | FIR Halfband Decimator | IIR Halfband Decimator | CIC Compensation Decimator

# Variable FIR Interpolation

Polyphase FIR interpolation with tunable interpolation factor



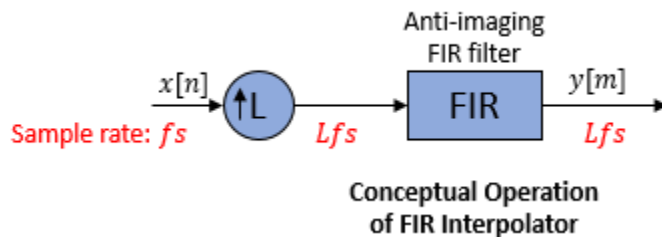
## Libraries:

DSP System Toolbox / Filtering / Multirate Filters

## Description

The Variable FIR Interpolation block performs an efficient polyphase FIR interpolation with a tunable interpolation factor. You can update the interpolation factor and the filter coefficients in the block dialog box or through an input port while the simulation is running.

Conceptually, the FIR interpolator (as shown in the schematic) consists of an upsampler followed by an FIR anti-imaging filter, which is usually an approximation of an ideal band-limited interpolation filter. To design an FIR anti-imaging filter, use the `designMultirateFIR` function. The upsampler upsamples each channel of the input to a higher rate by inserting  $L-1$  zeros between samples. The FIR filter that follows filters each channel of the upsampled data. The resulting discrete-time signal has a sample rate that is  $L$  times the original sample rate.



However, the actual block algorithm implements a direct-form FIR polyphase structure, an efficient equivalent of the combined system depicted in the diagram. For more details, see “Algorithms” on page 2-1512.

The block supports C and C++ code generation.

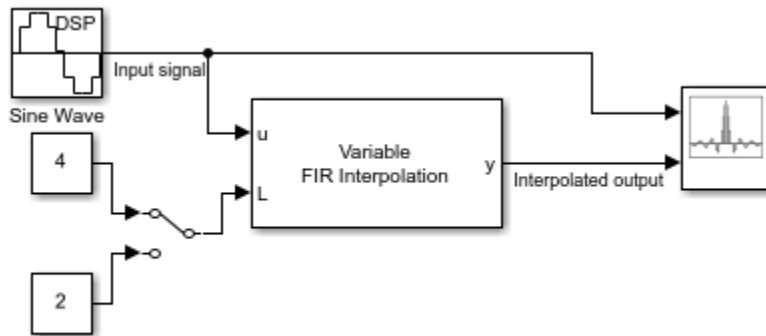
## Examples

### Interpolate Sinusoidal Signal with Tunable Interpolation Factor

Interpolate a sinusoidal signal by varying the interpolation factor using the Variable FIR Interpolation block. You can vary the interpolation factor in the block dialog box or through an input port while the simulation is running.

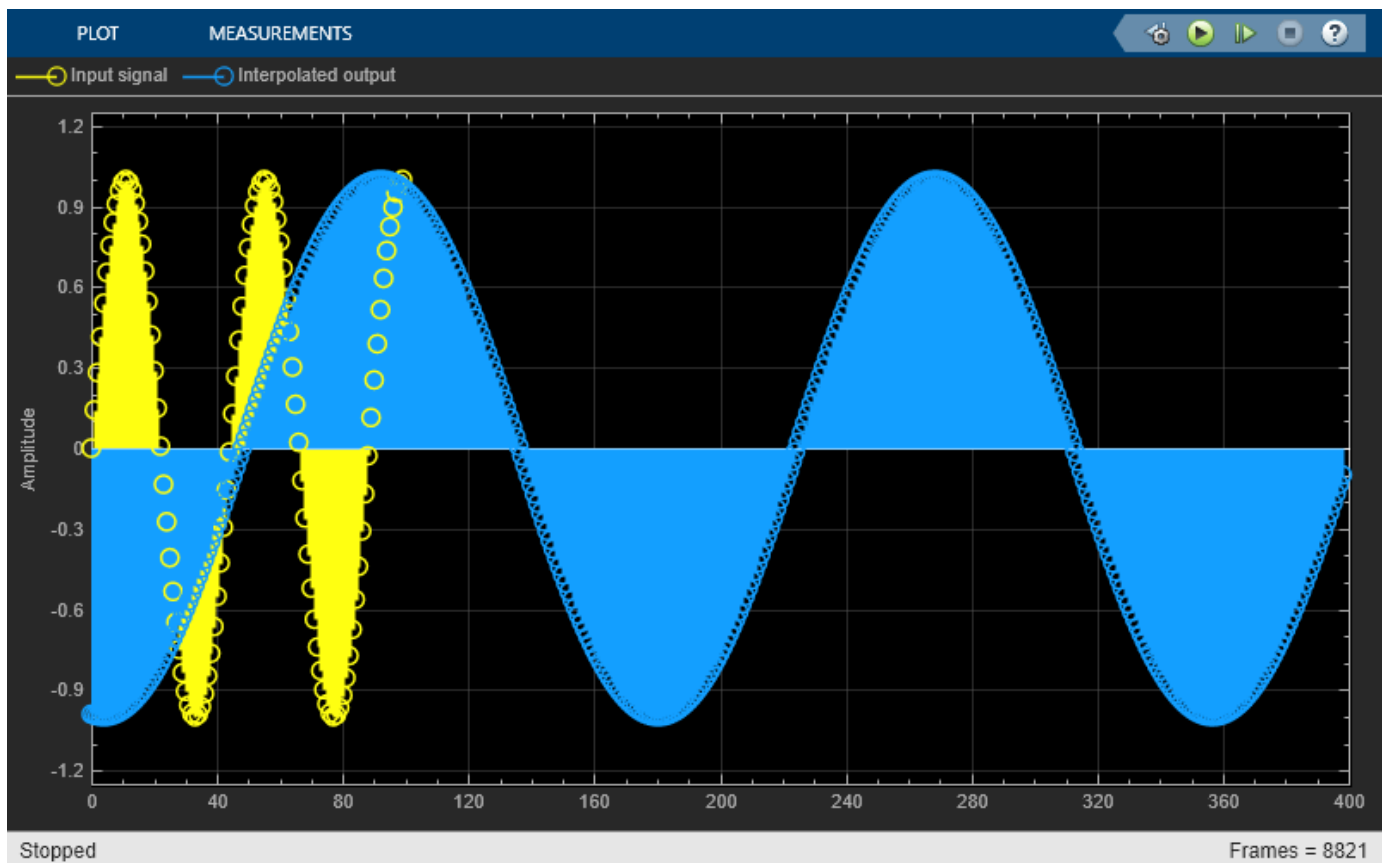
Open the model. The input is a sinusoidal signal with a frequency of 500 Hz, sample time of 1/44100 s, and contains 100 samples per frame. Pass this signal through the Variable FIR Interpolation block.

The **Maximum interpolation factor** parameter in the block is 24. The interpolation factor that you input through the port is 4.



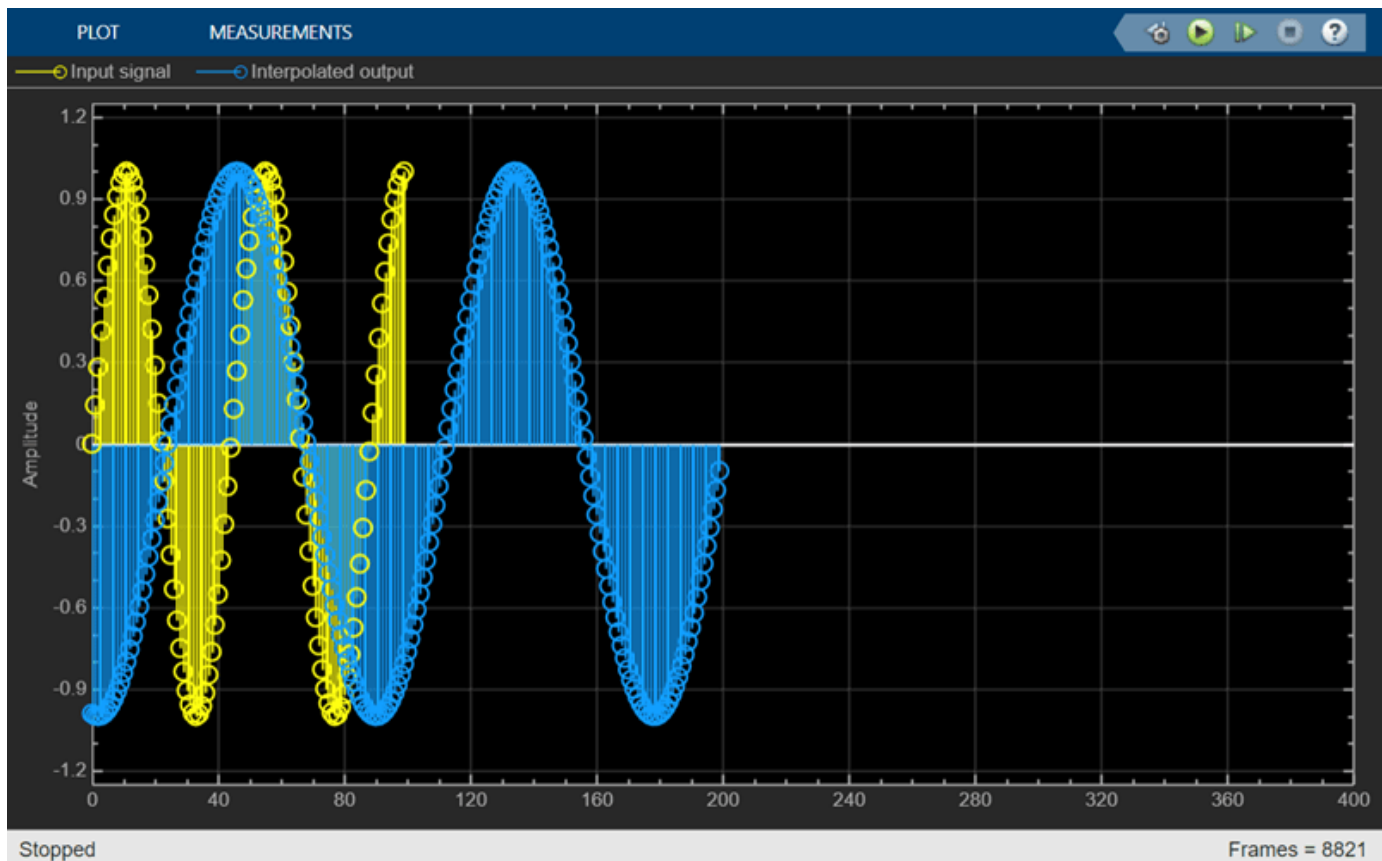
Copyright 2023 The MathWorks, Inc.

Run the model. The Array Plot block shows the input signal and the interpolated output on the display.



While the simulation is running, change the interpolation factor to 2 by clicking the Manual Switch. The span of the interpolated output updates in the Array Plot display. You can change the interpolation factor to any value that is an integer factor of the maximum interpolation factor of 24.





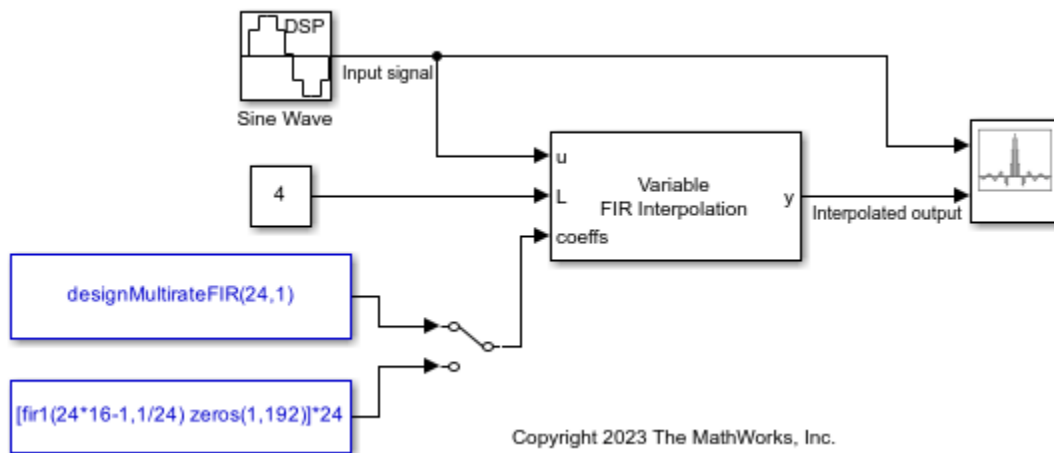
If you specify the interpolation factor in the block dialog box, you can tune the **Interpolation factor** parameter in the block dialog box while the simulation is running and the block updates the interpolated output accordingly.

### Specify Filter Coefficients Through Input Port

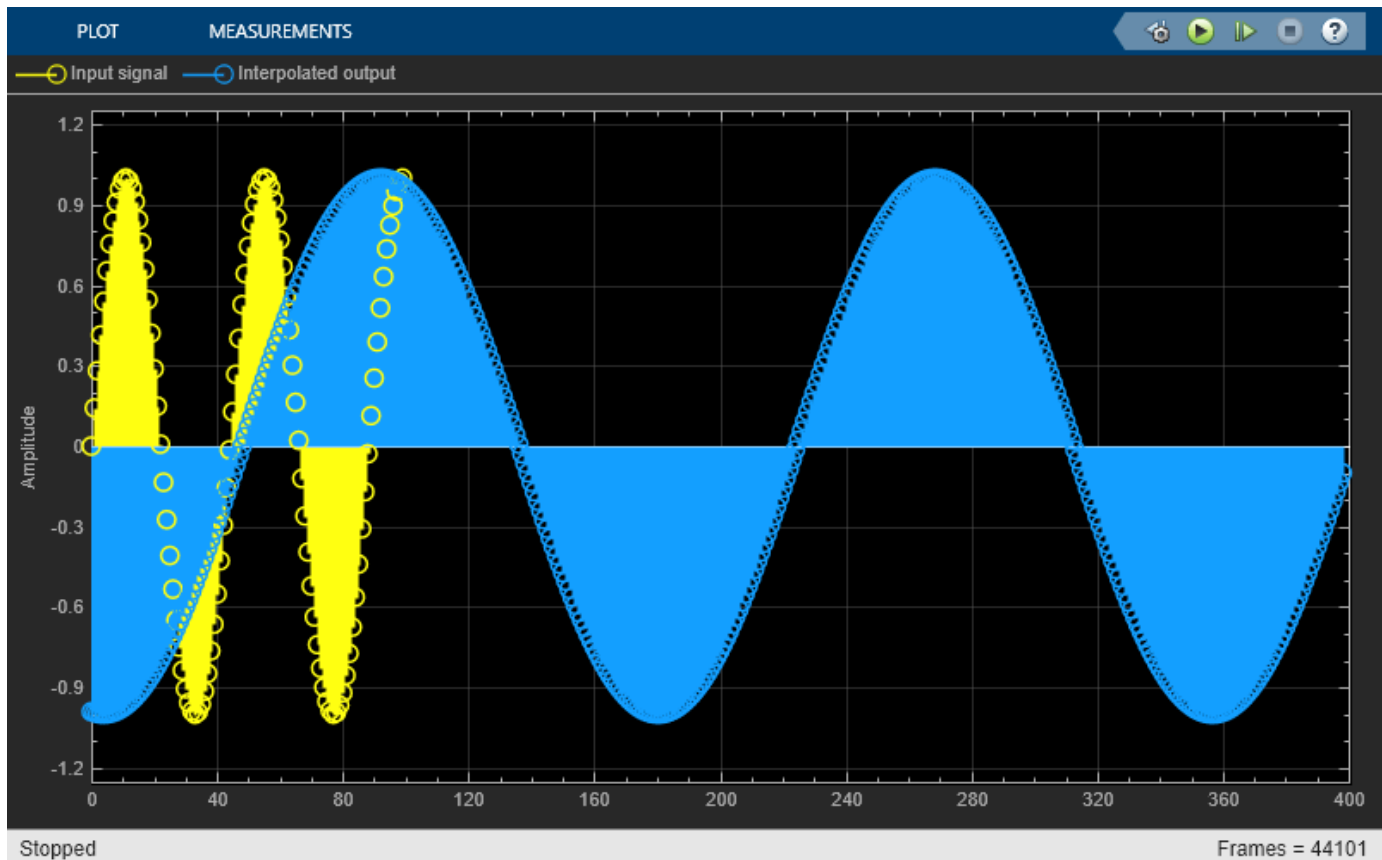
Interpolate a sinusoidal signal using the Variable FIR Interpolation block. You can vary the filter coefficients in the block dialog box or through an input port while the simulation is running.

Open the model. The input is a sinusoidal signal with a frequency of 500 Hz, sample time of 1/44100 s, and contains 100 samples per frame. Pass this signal through the Variable FIR Interpolation block. The **Maximum interpolation factor** parameter in the Variable FIR Interpolation block is set to 24. Specify the interpolation factor and the filter coefficients through the **L** and the **coeffs** ports, respectively. The interpolation factor is 4 and the filter coefficients are generated using the `designMultirateFIR(24,1)` function. This function generates an effective anti-imaging lowpass filter with a normalized cutoff frequency no greater than 1/24.

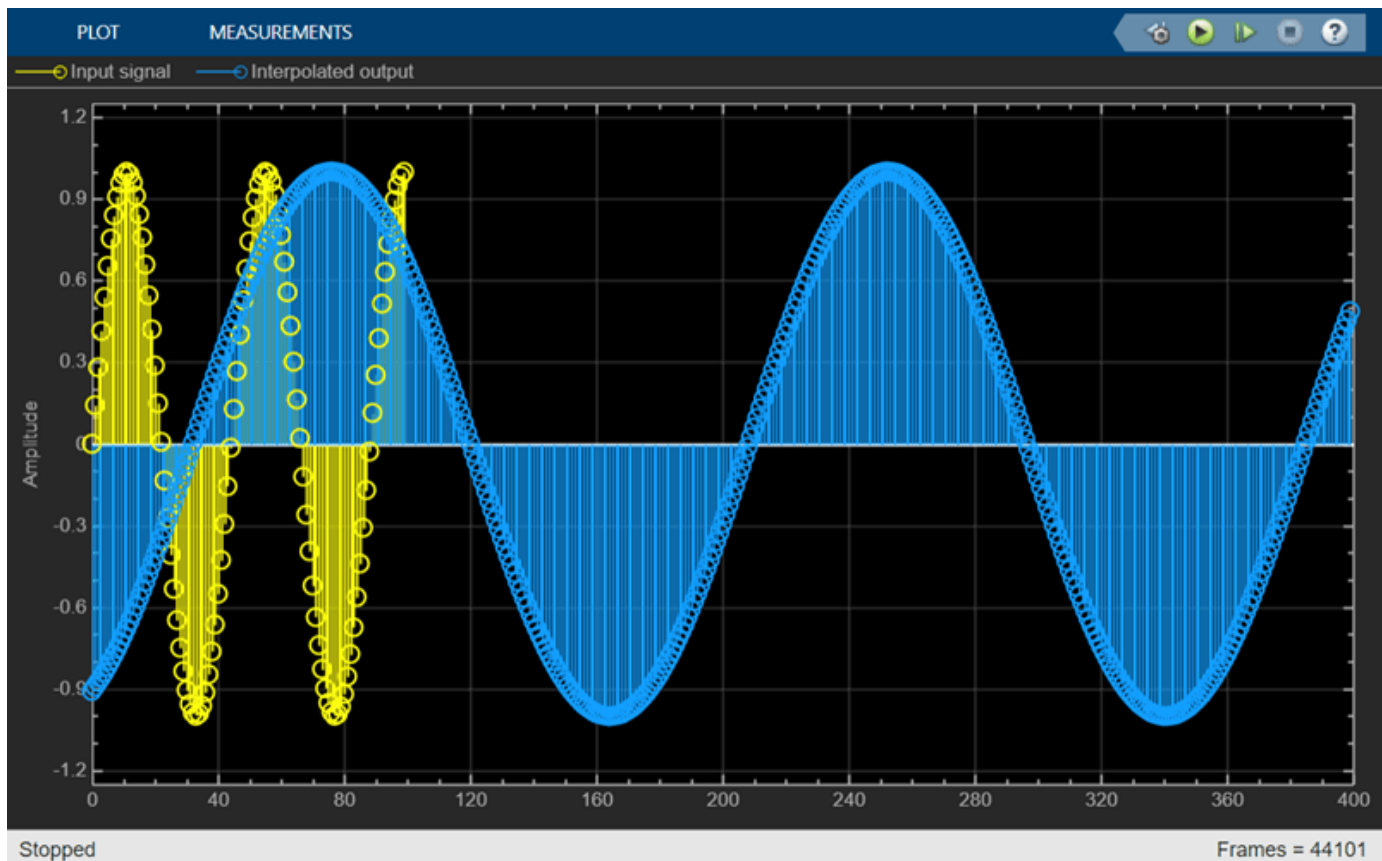
You can vary the filter coefficients using the Manual Switch.



Run the model. The Array Plot block shows the input signal and the interpolated output in the display.



While the simulation is running, change the filter coefficients by clicking the Manual Switch. On the second branch, the `fir1` function generates the coefficients of a lowpass filter that has a similar passband frequency response and the same number of coefficients as the first filter. Note that you cannot change the number of filter coefficients while the simulation is running.



## Ports

### Input

**u** — Data input  
vector | matrix

Specify the data input as a vector or a matrix. The block treats each column of the input signal as a separate channel. If the input is a two-dimensional signal, the first dimension represents the channel length (or frame size) and the second dimension represents the number of channels. If the input is a one-dimensional signal, then the block interprets it as a single channel.

The block accepts variable-size input signals, that is, you can change the size of each input channel during simulation, but you cannot change the number of channels.

Data Types: `single` | `double`

**L** — Interpolation factor input port  
positive integer

Specify the interpolation factor  $L$  as a positive integer that is an integer factor of the maximum interpolation factor. For example, if you set **Maximum interpolation factor** to 24, then the possible values for interpolation factor are 1, 2, 3, 4, 6, 8, 12, and 24.

You can tune the interpolation factor while the simulation is running. For an example, see “Interpolate Sinusoidal Signal with Tunable Interpolation Factor” on page 2-1503.

**Dependencies**

To enable this port, select the **Specify interpolation factor from input port** parameter.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**coeffs** — Prototype filter coefficients  
vector

Specify the numerator coefficients of the FIR filter transfer function  $H(z)$ .

$$H(z) = b_0 + b_1z^{-1} + \dots + b_Nz^{-N}$$

You can generate the FIR filter coefficient vector,  $\mathbf{b} = [b_0, b_1, \dots, b_N]$ , using one of the DSP System Toolbox filter design functions such as `designMultirateFIR`, `firnyquist`, `firhalfband`, `firgr` or `firceqrip`.

Compute the filter coefficients based on the maximum interpolation factor  $L_{max}$  instead of the interpolation factor  $L$  that you specify during simulation. For example, if  $L_{max}$  is 24 and  $L$  is 8, and you used `designMultirateFIR` as the design function, compute the filter coefficients using `designMultirateFIR(24,1)` instead of `designMultirateFIR(8,1)`.

To act as an effective anti-imaging filter, the coefficients usually correspond to a lowpass filter with a normalized cutoff frequency no greater than the reciprocal of the maximum interpolation factor. To design such a filter, use the `designMultirateFIR` function.

The block internally initializes all filter states to zero.

You can change the filter coefficients during simulation but the number of filter coefficients must remain constant. For an example, see “Specify Filter Coefficients Through Input Port” on page 2-1505.

**Dependencies**

To enable this port, set the **Coefficient source** parameter to `Input port`.

Data Types: `single` | `double`

**Output**

**y** — Interpolated output  
vector | matrix

Interpolated output, returned as a vector or a matrix. The size of the output signal is  $L_{max} \times$  input frame size-by- $N$ , where  $N$  is the number of input channels and  $L_{max}$  is the maximum interpolation factor. However, the block computes only  $L \times$  input frame size-by- $N$  samples. The remaining samples in the output signal are zero.

The complexity of the output signal depends on the complexity of the input signal and the complexity of the filter coefficients. See this table for more details.

| Input Signal | Filter Coefficients | Output Signal |
|--------------|---------------------|---------------|
| Real         | Complex             | Complex       |
| Real         | Real                | Real          |

| Input Signal | Filter Coefficients | Output Signal |
|--------------|---------------------|---------------|
| Complex      | Complex             | Complex       |
| Complex      | Real                | Complex       |

Data Types: single | double

## Parameters

**Maximum interpolation factor** — Maximum interpolation factor

24 (default) | positive integer

Specify the maximum interpolation factor  $L_{max}$  as a positive integer. The interpolation factor you specify through the input port **L** must be a factor of the value you specify in this parameter.

**Specify interpolation factor from input port** — Specify interpolation factor from input port

on (default) | off

When you select this parameter, you can specify the interpolation factor through the input port **L**. When you clear this parameter, you can specify the interpolation factor in the block dialog box through the **Interpolation factor** parameter.

**Interpolation factor** — Decimation factor

24 (default) | positive integer

Specify the interpolation factor  $L$  as a positive integer that is an integer factor of the maximum interpolation factor. For example, if you set **Maximum interpolation factor** to 24, then the possible values for **Interpolation factor** are 1, 2, 3, 4, 6, 8, 12, and 24.

You can tune the interpolation factor while the simulation is running.

**Tunable:** Yes

### Dependencies

To enable this parameter, clear the **Specify interpolation factor from input port** parameter.

**Coefficient source** — FIR filter coefficient source

Auto (default) | Dialog parameter | Input port

Specify the FIR filter coefficient source as one of the following:

- **Auto** -- The block designs an FIR interpolator using the interpolation factor in the **Interpolation factor** parameter. The `designMultirateFIR` function designs the filter and returns the coefficients used by the block.

For more information on the filter design, see Orfanidis [1].

- **Dialog parameter** -- Specify the filter coefficients through the **Prototype filter coefficients** parameter in the block dialog box.

- **Input port** -- Specify the filter coefficients through the **coeffs** input port.

**Prototype filter coefficients** — Prototype filter coefficients

`designMultirateFIR(24,1)` (default) | vector

Specify the numerator coefficients of the FIR filter transfer function  $H(z)$ .

$$H(z) = b_0 + b_1z^{-1} + \dots + b_Nz^{-N}$$

You can generate the FIR filter coefficient vector,  $\mathbf{b} = [b_0, b_1, \dots, b_N]$ , using one of the DSP System Toolbox filter design functions such as `designMultirateFIR`, `firnyquist`, `firhalfband`, `firgr` or `firceqrip`.

Compute the filter coefficients based on the maximum interpolation factor  $L_{max}$  instead of the interpolation factor  $L$  that you specify during simulation. For example, if  $L_{max}$  is 24 and  $L$  is 8, and you used `designMultirateFIR` as the design function, compute the filter coefficients using `designMultirateFIR(24,1)` instead of `designMultirateFIR(8,1)`.

To act as an effective anti-imaging filter, the coefficients usually correspond to a lowpass filter with a normalized cutoff frequency no greater than the reciprocal of the maximum interpolation factor. To design such a filter, use the `designMultirateFIR` function.

The block internally initializes all filter states to zero.

You can change the filter coefficients during simulation but the number of filter coefficients must remain constant.

**Tunable:** Yes

**Dependencies**

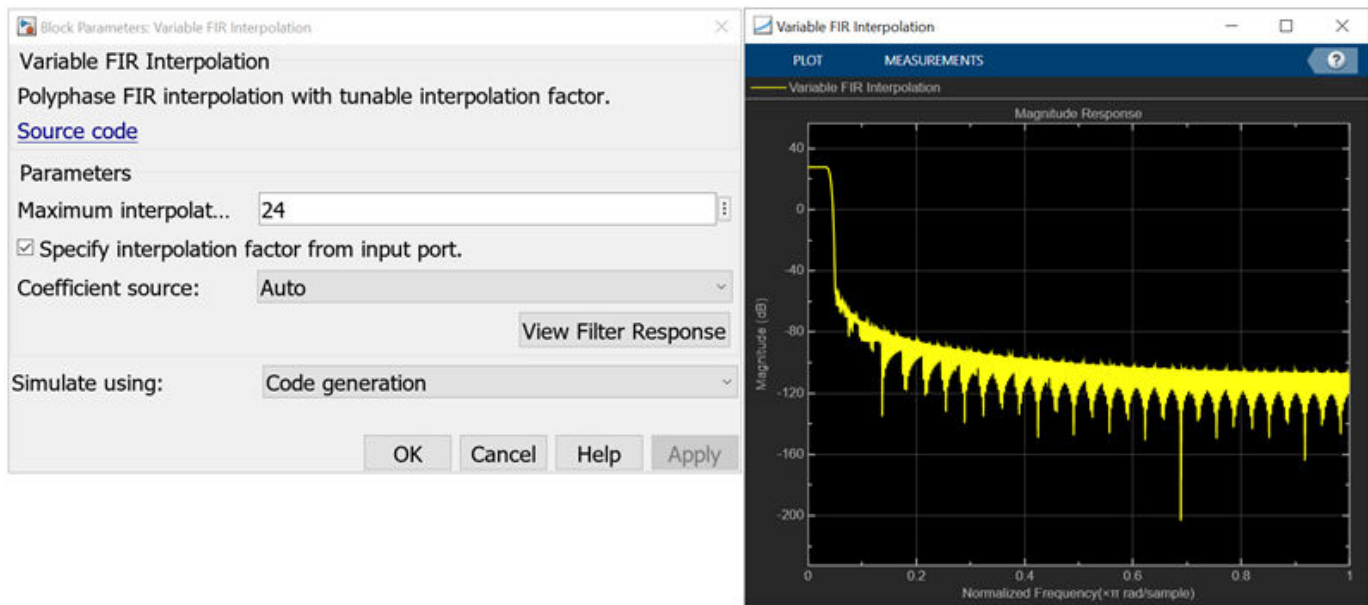
To enable this parameter, set the **Coefficient source** parameter to `Dialog` parameter.

**View Filter Response** — View Filter Response

button

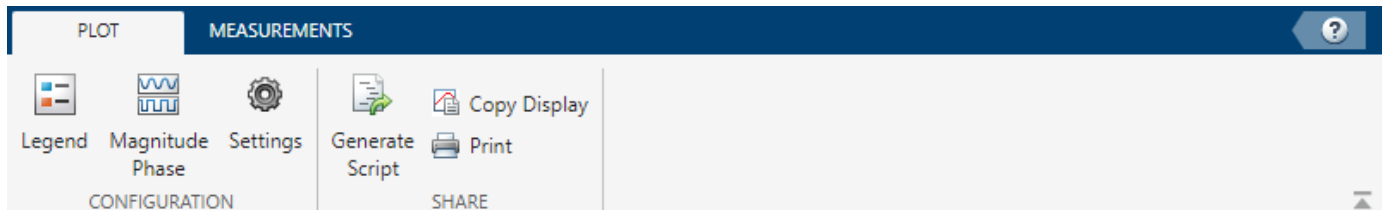
Click this button to open the dynamic filter visualizer and display the magnitude response of the variable FIR interpolation filter. The response is based on the parameters you select in the block dialog box. To update the magnitude response while the dynamic filter visualizer is running, modify the parameters in the dialog box and click **Apply**.

You cannot view the filter response when you set the **Coefficient source** parameter to `Input port`. To view the filter response, set the **Coefficient source** parameter to `Auto` or `Dialog` parameter.

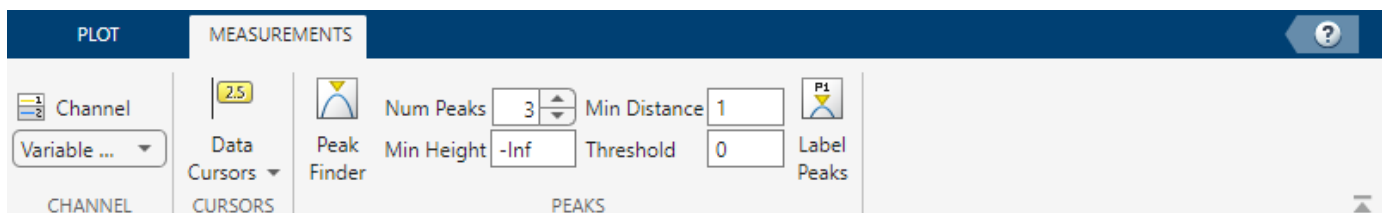


You can configure the plot settings and the signal measurements from the interface of the visualizer.

On the **Plot** tab, the **Configuration** section allows you to modify the plot settings. Click **Magnitude Phase** to display the magnitude and phase response of the filter.



On the **Measurements** tab, you can measure the signal statistics, place data cursors, and display the peak values of the selected signal.



For more details on the dynamic filter visualizer interface and its tools, see `dsp.DynamicFilterVisualizer`.

**Simulate using** — Simulate using

Code generation (default) | Interpreted execution

Specify the type of simulation to run. You can set this parameter to:

- **Code generation:** Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than Interpreted execution.
- **Interpreted execution:** Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than Code generation.

## Block Characteristics

|                                 |                 |
|---------------------------------|-----------------|
| <b>Data Types</b>               | double   single |
| <b>Direct Feedthrough</b>       | no              |
| <b>Multidimensional Signals</b> | no              |
| <b>Variable-Size Signals</b>    | yes             |
| <b>Zero-Crossing Detection</b>  | no              |

## Algorithms

The FIR interpolation filter is implemented efficiently using a polyphase structure.

To derive the polyphase structure, start with the transfer function of the FIR filter:

$$H(z) = b_0 + b_1z^{-1} + \dots + b_Nz^{-N}$$

$N+1$  is the length of the FIR filter.

You can rearrange this equation as follows:

$$\begin{aligned}
 H(z) = & \left( b_0 + b_{L_{\max}}z^{-L_{\max}} + b_{2L_{\max}}z^{-2L_{\max}} + \dots + b_{N-L_{\max}+1}z^{-(N-L_{\max}+1)} \right) + \\
 & z^{-1} \left( b_1 + b_{L_{\max}+1}z^{-L_{\max}} + b_{2L_{\max}+1}z^{-2L_{\max}} + \dots + b_{N-L_{\max}+2}z^{-(N-L_{\max}+1)} \right) + \\
 & \vdots \\
 & z^{-(L_{\max}-1)} \left( b_{L_{\max}-1} + b_{2L_{\max}-1}z^{-L_{\max}} + b_{3L_{\max}-1}z^{-2L_{\max}} + \dots + b_Nz^{-(N-L_{\max}+1)} \right)
 \end{aligned}$$

$L_{\max}$  is the number of polyphase components, and its value equals the maximum interpolation factor that you specify.

You can write this equation as:

$$H(z) = E_0(z^{L_{\max}}) + z^{-1}E_1(z^{L_{\max}}) + \dots + z^{-(L_{\max}-1)}E_{(L_{\max}-1)}(z^{L_{\max}})$$

$E_0(z^{L_{\max}}), E_1(z^{L_{\max}}), \dots, E_{L_{\max}-1}(z^{L_{\max}})$  are polyphase components of the FIR filter  $H(z)$ .

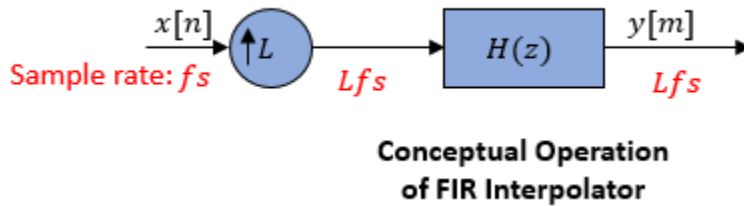
During simulation, the algorithm reconstructs the filter  $H(z)$  based on the interpolation factor  $L$  that you specify during simulation. Rewriting  $H(z)$  in terms of the interpolation factor  $L$ .

$$H(z) = E_0(z^L) + z^{-1}E_r(z^L) + \dots + z^{-(L-1)}E_{(L-1)r}(z^L)$$

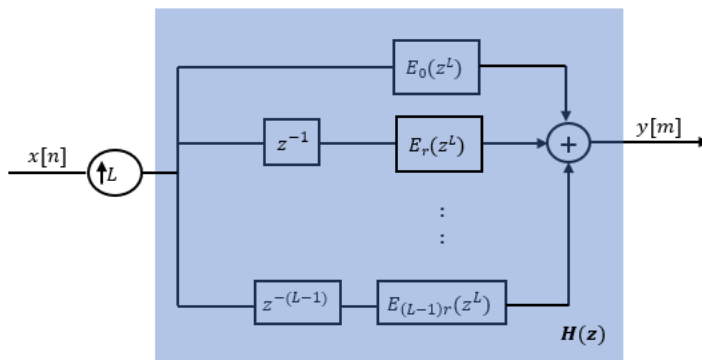


where,  $r = L_{max}/L$ .

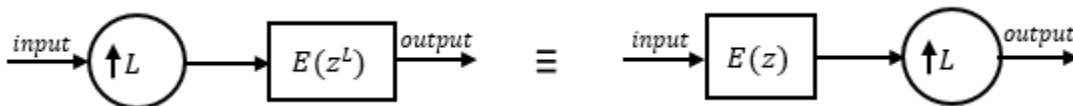
Conceptually, the FIR interpolation filter contains an upsampler followed by an FIR lowpass filter  $H(z)$ .



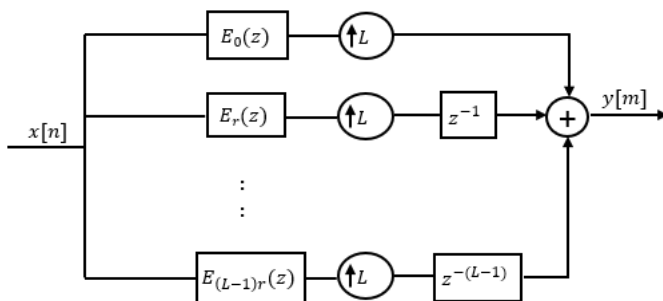
Replace  $H(z)$  with its polyphase representation.



Here is the multirate noble identity for interpolation.

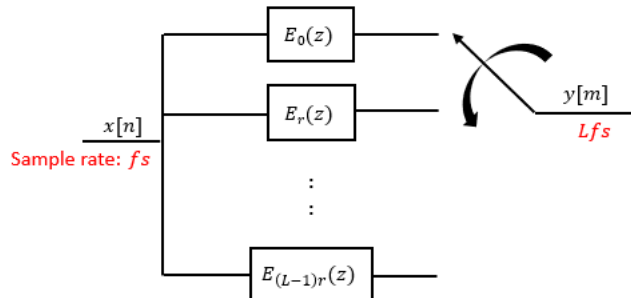


Applying the noble identity for interpolation moves the upsampling operation to after the filtering operation. This move enables you to filter the signal at a lower rate.



You can replace the upsampling operator, delay block, and adder with a commutator switch. The switch starts on the first branch 0 and moves in the counterclockwise direction, each time receiving

one sample from each branch. The interpolator effectively outputs  $L$  samples for every one input sample it receives. Hence the sample rate at the output of the FIR interpolation filter is  $Lfs$ .



## Version History

Introduced in R2023a

## References

[1] Orfanidis, Sophocles J. *Introduction to Signal Processing*. Upper Saddle River, NJ: Prentice-Hall, 1996.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

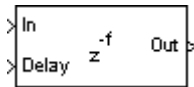
## See Also

### Blocks

Variable FIR Decimation | FIR Interpolation | FIR Halfband Interpolator | IIR Halfband Interpolator | CIC Compensation Interpolator

# Variable Fractional Delay

Delay input by time-varying fractional number of sample periods



## Libraries:

DSP System Toolbox / Signal Operations

## Description

The Variable Fractional Delay block delays the input signal by a specified number of fractional samples along each channel of the input. The block can also concurrently compute multiple delayed versions (taps) of the same signal. For an example, see “Delay Signal Using Multitap Fractional Delay”.

When the delay has a fractional value, the block interpolates the input signal to obtain new samples at noninteger sampling intervals. You can set **Interpolation mode** parameter to one of **Linear**, **FIR**, or **Farrow**. The block supports time-varying delay values. That is, the delay value can vary within a frame from sample to sample.

The block assumes that the input values at the **Delay** port are between  $D_{min}$  and  $D_{max}$ , where  $D_{min}$  appears in the **Valid delay range** section on the **Main** tab of the block dialog, and  $D_{max}$  is the value of the **Maximum delay (Dmax) in samples** parameter. The block clips delay values less than  $D_{min}$  to  $D_{min}$  and delay values greater than  $D_{max}$  to  $D_{max}$ .

You must consider additional factors when selecting valid Delay values for the **FIR** and **Farrow** interpolation modes. For details, see “Algorithms” on page 2-1528.

## Ports

### Input

**In** — Data input  
vector | matrix

Specify the data input as a vector or matrix. The data input must have the same data type as the delay input.

This block supports variable-size input signal. That is, you can change the number of input rows during the simulation. However, the number of channels must remain constant.

Example: [1 2 3 4;5 1 4 2;2 6 2 3;1 2 3 2;3 4 5 6;1 2 3 1]

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fixed point

**Delay** — Delay input  
scalar | vector | matrix | *N*-D array

Specify the delay input as a scalar, vector, matrix, or *N*-D array. The delay can be an integer or a fractional value. The block interpolates the signal to obtain new samples at noninteger sampling intervals. The delay input must have the same data type as the data input.

This block supports variable-size delay signal. That is, you can change one or both of the dimensions of the delay signal during simulation. However, the block must make sure that the resulting number of output channels remains constant throughout the simulation.

When the **Input processing** parameter is set to `Columns as channels (frame based)`, the table below shows the effect of the dimension of the delay input on the data input. For an example, see “Delay Signal Using Multitap Fractional Delay”.

| Data Input   | Delay Input        | Output             | Effect of Delay Input on Data Input   |
|--|--------------------|--------------------|---|
| $N$ (unoriented, one channel)                              | scalar             | Unoriented ( $N$ ) | One delay value applied to the input channel  |
| $N$ (unoriented, one channel)                              | Unoriented ( $N$ ) | Unoriented ( $N$ ) | Delay value varies within the frame from sample to sample   |
| $N$ (unoriented, one channel)                              | 1-by- $P$          | $N$ -by- $P$       | $P$ taps. Each column in the output is a delayed version of the input. The delay value is specified by the corresponding element in the delay input vector. |
| $N$ (unoriented, one channel)                              | $N$ -by- $P$       | $N$ -by- $P$       | $P$ taps. In addition, delay varies within each frame from sample to sample.  |
| $N$ -by-1 (one channel with frame size equal to $N$ )      | scalar             | $N$ -by-1          | One delay value applied to the input channel  |
| $N$ -by-1 (one channel with frame size equal to $N$ )      | Unoriented ( $N$ ) | $N$ -by-1          | Delay value varies within the frame from sample to sample   |
| $N$ -by-1 (one channel with frame size equal to $N$ )      | $N$ -by-1          | $N$ -by-1          | Delay value varies within the frame from sample to sample   |
| $N$ -by-1 (one channel with frame size equal to $N$ )      | 1-by- $P$          | $N$ -by- $P$       | $P$ taps. Each column in the output is a delayed version of the input. The delay value is specified by the corresponding element in the delay input vector. |
| $N$ -by-1 (one channel with frame size equal to $N$ )      | $N$ -by- $P$       | $N$ -by- $P$       | $P$ taps. In addition, delay varies within each frame from sample to sample.  |
| $N$ -by- $L$ ( $L$ channels with frame size equal to $N$ ) | scalar             | $N$ -by- $L$       | One delay value applied to all input channels   |

| <b>Data Input</b>  | <b>Delay Input</b>    | <b>Output</b>         | <b>Effect of Delay Input on Data Input</b>   |
|--|-----------------------|-----------------------|--|
| $N$ -by- $L$ ( $L$ channels with frame size equal to $N$ ) | 1-by- $L$             | $N$ -by- $L$          | Unique delay value for each input channel  |
| $N$ -by- $L$ ( $L$ channels with frame size equal to $N$ ) | $N$ -by-1             | $N$ -by- $L$          | Delay value varies within the frame from sample to sample. Same set of delay values for all channels.                                    |
| $N$ -by- $L$ ( $L$ channels with frame size equal to $N$ ) | $N$ -by- $L$          | $N$ -by- $L$          | Delay value varies within the frame from sample to sample. Different delay values for each input channel.                                |
| $N$ -by- $L$ ( $L$ channels with frame size equal to $N$ ) | 1-by-1-by- $P$        | $N$ -by- $L$ -by- $P$ | $L$ channels. $P$ taps per channel. Same delay for all channels.   |
| $N$ -by- $L$ ( $L$ channels with frame size equal to $N$ ) | 1-by- $L$ -by- $P$    | $N$ -by- $L$ -by- $P$ | $L$ channels. $P$ taps per channel. Taps vary across channels.   |
| $N$ -by- $L$ ( $L$ channels with frame size equal to $N$ ) | $N$ -by-1-by- $P$     | $N$ -by- $L$ -by- $P$ | $L$ channels. $P$ taps per channel. Delay varies within the frame from sample to sample. Same set of delay values for each channel.      |
| $N$ -by- $L$ ( $L$ channels with frame size equal to $N$ ) | $N$ -by- $L$ -by- $P$ | $N$ -by- $L$ -by- $P$ | $L$ channels. $P$ taps per channel. Delay varies within the frame from sample to sample. Different set of delay values for each channel. |

When the **Input processing** parameter is set to `Elements as channels (sample based)`, the table below shows the effect of the dimension of the delay input on the data input.

| <b>Data Input</b>                                     | <b>Delay Input</b> | <b>Output</b>      | <b>Effect of Delay Input on Data Input</b>                |
|---|--------------------|--------------------|---|
| $N$ (unoriented, one channel)                         | scalar             | Unoriented ( $N$ ) | One delay value applied to the input channel              |
| $N$ (unoriented, one channel)                         | Unoriented ( $N$ ) | Unoriented ( $N$ ) | Delay value varies within the frame from sample to sample |
| $N$ -by-1 (one channel with frame size equal to $N$ ) | scalar             | $N$ -by-1          | One delay value applied to the input channel              |

| Data Input  | Delay Input        | Output       | Effect of Delay Input on Data Input   |
|---|--------------------|--------------|---|
| $N$ -by-1 (one channel with frame size equal to $N$ )         | Unoriented ( $N$ ) | $N$ -by-1    | Delay value varies within the frame from sample to sample   |
| $N$ -by-1 (one channel with frame size equal to $N$ )         | $N$ -by-1          | $N$ -by-1    | Delay value varies within the frame from sample to sample   |
| $N$ -by- $L$ ( $L$ channels with $N$ samples in each channel) | scalar             | $N$ -by- $L$ | One delay value applied to all input channels   |
| $N$ -by- $L$ ( $L$ channels with $N$ samples in each channel) | 1-by- $L$          | $N$ -by- $L$ | Unique delay value for each input channel   |
| $N$ -by- $L$ ( $L$ channels with $N$ samples in each channel) | $N$ -by-1          | $N$ -by- $L$ | Delay value varies within the frame from sample to sample. Same set of delay values for all channels.     |
| $N$ -by- $L$ ( $L$ channels with $N$ samples in each channel) | $N$ -by- $L$       | $N$ -by- $L$ | Delay value varies within the frame from sample to sample. Different delay values for each input channel. |

Example: [2 3 4 5]

Example: [2.5]

Example: [5.6]

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fixed point

## Output

**Port\_1** — Delayed output

vector | matrix

Delayed output, returned as a vector or matrix. The data type and complexity of the output match the data type and complexity of the data input.

When the **Input processing** parameter is set to `Columns as channels (frame based)`, the table below shows the effect of the dimension of the delay input on the data input.

| Data Input                    | Delay Input | Output             | Effect of Delay Input on Data Input          |
|-------------------------------|-------------|--------------------|--|
| $N$ (unoriented, one channel) | scalar      | Unoriented ( $N$ ) | One delay value applied to the input channel |

| <b>Data Input</b>  | <b>Delay Input</b> | <b>Output</b>      | <b>Effect of Delay Input on Data Input</b>  |
|--|--------------------|--------------------|---|
| $N$ (unoriented, one channel)                              | Unoriented ( $N$ ) | Unoriented ( $N$ ) | Delay value varies within the frame from sample to sample   |
| $N$ (unoriented, one channel)                              | 1-by- $P$          | $N$ -by- $P$       | $P$ taps. Each column in the output is a delayed version of the input. The delay value is specified by the corresponding element in the delay input vector. |
| $N$ (unoriented, one channel)                              | $N$ -by- $P$       | $N$ -by- $P$       | $P$ taps. In addition, delay varies within each frame from sample to sample.  |
| $N$ -by-1 (one channel with frame size equal to $N$ )      | scalar             | $N$ -by-1          | One delay value applied to the input channel  |
| $N$ -by-1 (one channel with frame size equal to $N$ )      | Unoriented ( $N$ ) | $N$ -by-1          | Delay value varies within the frame from sample to sample   |
| $N$ -by-1 (one channel with frame size equal to $N$ )      | $N$ -by-1          | $N$ -by-1          | Delay value varies within the frame from sample to sample   |
| $N$ -by-1 (one channel with frame size equal to $N$ )      | 1-by- $P$          | $N$ -by- $P$       | $P$ taps. Each column in the output is a delayed version of the input. The delay value is specified by the corresponding element in the delay input vector. |
| $N$ -by-1 (one channel with frame size equal to $N$ )      | $N$ -by- $P$       | $N$ -by- $P$       | $P$ taps. In addition, delay varies within each frame from sample to sample.  |
| $N$ -by- $L$ ( $L$ channels with frame size equal to $N$ ) | scalar             | $N$ -by- $L$       | One delay value applied to all input channels   |
| $N$ -by- $L$ ( $L$ channels with frame size equal to $N$ ) | 1-by- $L$          | $N$ -by- $L$       | Unique delay value for each input channel   |
| $N$ -by- $L$ ( $L$ channels with frame size equal to $N$ ) | $N$ -by-1          | $N$ -by- $L$       | Delay value varies within the frame from sample to sample. Same set of delay values for all channels.   |

| Data Input   | Delay Input           | Output                | Effect of Delay Input on Data Input  |
|--|-----------------------|-----------------------|--|
| $N$ -by- $L$ ( $L$ channels with frame size equal to $N$ ) | $N$ -by- $L$          | $N$ -by- $L$          | Delay value varies within the frame from sample to sample. Different delay values for each input channel.                                |
| $N$ -by- $L$ ( $L$ channels with frame size equal to $N$ ) | 1-by-1-by- $P$        | $N$ -by- $L$ -by- $P$ | $L$ channels. $P$ taps per channel. Same delay for all channels.   |
| $N$ -by- $L$ ( $L$ channels with frame size equal to $N$ ) | 1-by- $L$ -by- $P$    | $N$ -by- $L$ -by- $P$ | $L$ channels. $P$ taps per channel. Taps vary across channels.   |
| $N$ -by- $L$ ( $L$ channels with frame size equal to $N$ ) | $N$ -by-1-by- $P$     | $N$ -by- $L$ -by- $P$ | $L$ channels. $P$ taps per channel. Delay varies within the frame from sample to sample. Same set of delay values for each channel.      |
| $N$ -by- $L$ ( $L$ channels with frame size equal to $N$ ) | $N$ -by- $L$ -by- $P$ | $N$ -by- $L$ -by- $P$ | $L$ channels. $P$ taps per channel. Delay varies within the frame from sample to sample. Different set of delay values for each channel. |

When the **Input processing** parameter is set to **Elements as channels (sample based)**, the table below shows the effect of the dimension of the delay input on the data input.

| Data Input  | Delay Input        | Output             | Effect of Delay Input on Data Input                       |
|---|--------------------|--------------------|---|
| $N$ (unoriented, one channel)                         | scalar             | Unoriented ( $N$ ) | One delay value applied to the input channel              |
| $N$ (unoriented, one channel)                         | Unoriented ( $N$ ) | Unoriented ( $N$ ) | Delay value varies within the frame from sample to sample |
| $N$ -by-1 (one channel with frame size equal to $N$ ) | scalar             | $N$ -by-1          | One delay value applied to the input channel              |
| $N$ -by-1 (one channel with frame size equal to $N$ ) | Unoriented ( $N$ ) | $N$ -by-1          | Delay value varies within the frame from sample to sample |
| $N$ -by-1 (one channel with frame size equal to $N$ ) | $N$ -by-1          | $N$ -by-1          | Delay value varies within the frame from sample to sample |



| Data Input  | Delay Input  | Output       | Effect of Delay Input on Data Input   |
|---|--------------|--------------|---|
| $N$ -by- $L$ ( $L$ channels with $N$ samples in each channel) | scalar       | $N$ -by- $L$ | One delay value applied to all input channels   |
| $N$ -by- $L$ ( $L$ channels with $N$ samples in each channel) | 1-by- $L$    | $N$ -by- $L$ | Unique delay value for each input channel   |
| $N$ -by- $L$ ( $L$ channels with $N$ samples in each channel) | $N$ -by-1    | $N$ -by- $L$ | Delay value varies within the frame from sample to sample. Same set of delay values for all channels.     |
| $N$ -by- $L$ ( $L$ channels with $N$ samples in each channel) | $N$ -by- $L$ | $N$ -by- $L$ | Delay value varies within the frame from sample to sample. Different delay values for each input channel. |

Example: [0 0 0 0;0 0 0 0;1 0 0 0;5 2 0 0;2 1 3 0;1 6 4 4]

Example: [0 0 0 0;0 0 0 0;0.5 1.0 1.5 2.0;3 1.5 3.5 3.0;3.5 3.5 3.0 2.5;1.5 4.0 2.5 2.5]

Example: [0 0 0 0;0 0 0 0;0 0 0 0;0 0 0 0;0.4 0.8 1.2 1.6]

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fixed point

## Parameters

**Interpolation mode** — Method of interpolation

Linear (default) | FIR | Farrow

Specify the method of interpolation. Using this method, the block interpolates the signal to obtain new samples at noninteger sampling intervals.

- **Linear** -- Linear interpolation. In this mode, the block stores the  $\mathbf{D}_{\max}+1$  most recent samples the **In** port receives for each channel.  $\mathbf{D}_{\max}$  is the value you specify in the **Maximum delay (Dmax) in samples** parameter.
- **FIR** -- Polyphase FIR interpolation. In this mode, the block stores the  $\mathbf{D}_{\max}+\mathbf{P}+1$  most recent samples the **In** port receives for each channel.  $\mathbf{P}$  is the value you specify in the **Interpolation filter half-length (P)** parameter.
- **Farrow** -- LaGrange method. In this mode, the block stores the  $\mathbf{D}_{\max}+\frac{\mathbf{N}}{2}+1$  most recent samples the **In** port receives for each channel.  $\mathbf{N}$  is the value you specify in the **Farrow filter length (N)** parameter.

For more details on these methods, see “Algorithms” on page 2-1528.

**Interpolation filter half-length (P)** — Half length of interpolation filter

4 (default) | positive integer in the range [1 65535]

Half-length of the FIR interpolation filter. For periodic signals, a larger value of this property, which indicates a higher order filter, yields a better estimate of the delayed output sample. A property value of 4 to 6, which corresponds to a 7th-order to 11th-order filter, is usually adequate.

Example: 6

Example: 10

#### Dependencies

This parameter applies only when you set **Interpolation mode** to FIR.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point`

**Interpolation points per input sample** — Number of interpolation points per input sample  
10 (default) | positive integer in the range [2, 65,535]

Number of interpolation points per input sample at which a unique FIR interpolation filter is computed.

Example: 20

Example: 5

#### Dependencies

This parameter applies only when you set **Interpolation mode** to FIR.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**Normalized input bandwidth (0 to 1)** — Normalized input bandwidth  
1 (default) | real scalar in the range (0, 1]

Normalized input bandwidth at which to constrain the interpolated output samples. A value of 1 equals the Nyquist frequency, or half the sampling frequency,  $F_s$ . Use this property to take advantage of the bandlimited frequency content of the input. For example, if the input signal does not have frequency content above  $F_s/4$ , you can specify a value of 0.5.

Example: 0.5

Example: 0.8

#### Dependencies

This parameter applies only when you set **Interpolation mode** to FIR.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point`

**Farrow filter length (N)** — Length of Farrow filter  
4 (default) | integer greater than or equal to 2

Length of the FIR filter implemented using the Farrow structure. If the length equals 2, the filter performs linear interpolation.

Example: 4

Example: 10

### Dependencies

This parameter applies only when you set **Interpolation mode** to Farrow.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

**Maximum delay (Dmax) in samples** — Maximum delay  
100 (default) | integer in the range [0 65535]

Maximum delay the block can produce,  $D_{max}$ . Input delay values exceeding this maximum are clipped to  $D_{max}$ .

Example: 200

Example: 500

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

### Input processing — Method to process the input

`Columns as channels (frame based)` (default) | `Elements as channels (sample based)`

Specify how the block should process the input. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` (default) — When you select this option, the block treats each column of the input as a separate channel. The block treats each of the  $R$  input columns as independent channels containing  $M_i$  sequential time samples.

The input to the **Delay** port,  $v$ , contains floating-point values that specify the number of sample intervals to delay the current input.

The input to the **Delay** port can be a scalar value to uniformly delay every sample in every channel. It can also be a length- $M$  column vector, containing one delay for each sample in the input frame. The block applies the set of delays contained in the vector identically to every channel of a multichannel input. The **Delay** port entry can also be a length- $R$  row vector, containing one delay for each channel. Finally, the **Delay** port entry can be an  $M$ -by- $R$  matrix, containing a different delay for each corresponding element of the input.

For example, if  $v$  is the  $M_i$ -by-1 matrix  $[v(1) \ v(2) \ \dots \ v(M_i)]'$ , the earliest sample in the current frame is delayed by  $v(1)$  fractional sample intervals, the following sample in the frame is delayed by  $v(2)$  fractional sample intervals, and so on. The block applies the set of fractional delays contained in  $v$  identically to every channel of a multichannel input.

- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel. The block treats each element of the  $N$ -D input array,  $u$ , as an independent channel. The input to the **Delay** port,  $v$ , must either be an  $N$ -D array of the same size and dimension as the input  $u$ , or be a scalar value, such that  $D_{min} \leq v \leq D_{max}$ .

For example, consider an  $M$ -by- $R$  input matrix. The block treats each of the  $M \times R$  matrix elements as independent channels. The input to the **Delay** port can be an  $M$ -by- $R$  matrix of floating-point values in the range  $D_{min} \leq v \leq D_{max}$  that specifies the number of sample intervals to delay each

channel of the input, or it can be a scalar floating-point value,  $D_{min} \leq v \leq D_{max}$ , by which to equally delay all channels.

In sample-based processing mode, the block treats an unoriented vector input as an  $M$ -by-1 matrix. In this mode, the output is also an unoriented vector.

**InitialConditions** — Initial values in the memory

0 (default) | scalar | 1-by- $R$ -by- $D$  array | 1-by- $R$ -by- $(D+L)$  array

Specify the values within the block's memory at the start of the simulation. The dimensions of this parameter can vary depending on whether you want fixed or time-varying initial conditions. The block treats each of the  $R$  input columns as a frame containing  $M$  sequential time samples from an independent channel.

For an  $M$ -by- $R$  input matrix,  $u$ , you can set this parameter as follows:

- To specify fixed initial conditions, set this parameter to a scalar value. The block initializes every sample of every channel in memory using the value you specify.
- The dimensions you specify for time-varying initial conditions depend on the interpolation method. To specify different time-varying initial conditions for each channel, set this parameter as follows:
  - If you set the **Interpolation mode** to Linear, set the **Initial conditions** to an array of size 1-by- $R$ -by- $D$ , where  $D$  is the value in **Maximum delay (Dmax) in samples** parameter.
  - If you set the **Interpolation mode** to FIR or Farrow, set the **Initial conditions** to an array of size 1-by- $R$ -by- $(D+L)$ , where  $D$  is the value of the maximum delay. For FIR interpolation,  $L$  is the value of the interpolation filter half length. For Farrow interpolation,  $L$  equals floor of half the value of the farrow filter length (floor( farrow filter length/2)).

Example: 1

Example: randn(1,3,104)

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32

**Disable direct feedthrough by increasing minimum possible delay by one** — Disable direct feedthrough

off (default) | on

Select this box to disable direct feedthrough by increasing the minimum possible delay value. When you set the **Input processing** parameter to Columns as channels (frame based), the block increases the minimum possible delay value by *frame size* - 1. Similarly, when you set the **Input processing** parameter to Elements as channels (sample based), the block increases the minimum possible delay value by one sample.

Checking this box allows you to use the Variable Fractional Delay block in feedback loops.

**For small input delay values** — Action to take for small input delay values

Clip to the minimum value necessary for centered kernel (default) | Use off-centered kernel | Switch to linear interpolation if kernel cannot be centered

Specify the block's behavior when the input delay values are too small to center the kernel.

You can specify how the block handles input delay values that are too small for the kernel to be centered using one of the following choices:

- In both FIR and Farrow interpolation modes, you can select **Clip to the minimum value necessary for centered kernel**. This option forces the block to increase  $D_{min}$  to the smallest value necessary to keep the kernel centered.
- In FIR interpolation mode, you can select **Switch to linear interpolation if kernel cannot be centered**. This option forces the block to preserve the value of  $D_{min}$  and compute all interpolated values using **Linear** interpolation.
- In Farrow interpolation mode, you can select **Use off-centered kernel**. This option forces the block to preserve the value of  $D_{min}$  and compute the interpolated values using a farrow filter with an off-centered kernel.

### Dependencies

This parameter applies only when **Interpolation mode** is set to FIR or Farrow.

**Valid delay range (in samples)** — Range of valid delay values

[0 100] (default) | [ $D_{min}$   $D_{max}$ ]

This parameter is read-only.

The delay range values [ $D_{min}$   $D_{max}$ ] are calculated (in samples) by the block based on the current parameter settings.  $D_{min}$  is the smallest possible valid delay value (in samples). The block clips all input delay values less than  $D_{min}$  to  $D_{min}$ .  $D_{max}$  is the maximum valid delay value (in samples). The block clips all input delay values greater than  $D_{max}$  to  $D_{max}$ .

When the **Interpolation mode** is set to one of the following:

- **Linear** --  $D_{min}$  equal 0.  $D_{max}$  equals the value you specify in the **Maximum delay (Dmax) in samples** parameter.
- **FIR** --  $D_{min}$  equals  $P - 1$ , where  $P$  is the value you specify in **Interpolation filter half-length (P)**.  $D_{max}$  equals the value you specify in the **Maximum delay (Dmax) in samples** parameter.
- **Farrow** --  $D_{min}$  equals  $N/2 - 1$ , where  $N$  is the value you specify in **Farrow filter length (N)**.  $D_{max}$  equals the value you specify in the **Maximum delay (Dmax) in samples** parameter.

Example: [1 100]

Example: [2 100]

Example: [3 100]

### Fixed-Point Properties

#### Fixed-Point Properties

**Rounding mode** — Rounding method for fixed-point operations

Zero (default) | Ceiling | Convergent | Floor | Nearest | Round | Simplest

Specify the rounding mode for fixed-point operations as one of the following:

- Zero
- Ceiling

- Convergent
- Floor
- Nearest
- Round
- Simplest

For more details, see rounding mode.

**Saturate on integer overflow** — Method of overflow action  
off (default) | on

When you select this parameter, the block saturates the result of its fixed-point operation. When you clear this parameter, the block wraps the result of its fixed-point operation. For details on saturate and wrap, see overflow mode for fixed-point operations.

**Coefficients** — Data type of the coefficients  
Same word length as input (default) | Specify word length

Specify the data type of the filter coefficients as one of the following:

- `Same word length as input` -- The word length of the filter coefficients matches that of the input to the block. The fraction length of the coefficients is automatically set to the binary-point only scaling that provides you with the best precision possible given the value and word length of the coefficients.
- `Specify word length` -- Specify the word length of the coefficients, in bits. In this mode, the fraction length of the coefficients is automatically set to the binary-point only scaling that provides you with the best precision possible given the value and word length of the coefficients.

For more information on the coefficients data type this block uses, see the “Fixed Point” on page 2-1531 section.

**Product output** — Data type of the product output  
Same as first input (default) | Binary point scaling

Specify the data type of the product output as one of the following:

- `Same as first input` -- The block specifies the product output data type to be the same as that of the data input.
- `Binary point scaling` -- Specify the word length and the fraction length of the product output, in bits.

For more information on the product output data type, see “Multiplication Data Types” and the “Fixed Point” on page 2-1531 section.

**Accumulator** — Data type of accumulation operation  
Same as product output (default) | Same as first input | Binary point scaling

Specify the data type of an accumulation operation as one of the following:

- `Same as product output` -- The block specifies the accumulator data type to be the same as that of the product output data type.

- `Same as first input` -- The block specifies the accumulator data type to be the same as that of the data input.
- `Binary point scaling` -- Specify the word length and the fraction length of the accumulator output, in bits.

For more information on the accumulator data type this block uses, see the “Fixed Point” on page 2-1531.

**Product output polyval** — Data type of the product polynomial value

`Same as first input (default) | Binary point scaling`

Specify the data type of the product polynomial value as one of the following:

- `Same as first input` -- The block specifies the product polynomial value data type to be the same as that of the data input.
- `Binary point scaling` -- Specify the word length and the fraction length of the product output polynomial, in bits.

For more information on the product polynomial value data type this block uses, see the “Fixed Point” on page 2-1531 section.

#### Dependencies

This property applies when you set **Interpolation mode** to Farrow.

**Accumulator polyval** — Data type of the accumulator polynomial value

`Same as first input (default) | Binary point scaling`

Specify the data type of the accumulator polynomial value as one of the following:

- `Same as first input` -- The block specifies the accumulator polynomial value data type to be the same as that of the data input.
- `Binary point scaling` -- Specify the word length and the fraction length of the accumulator polynomial value, in bits.

For more information on the accumulator polynomial value data type that this block uses, see the “Fixed Point” on page 2-1531 section.

#### Dependencies

This property applies when you set **Interpolation mode** to Farrow.

**Multiplicand polyval** — Data type of multiplicand polynomial value

`Same as first input (default) | Binary point scaling`

Specify the data type of the multiplicand polynomial value as one of the following:

- `Same as first input` -- The block specifies the multiplicand polynomial value data type to be the same as that of the data input.
- `Binary point scaling` -- Specify the word length and the fraction length of the multiplicand polynomial value, in bits.

For more information on the multiplicand polynomial value data type this block uses, see the “Fixed Point” on page 2-1531 section.

### Dependencies

This property applies when you set **Interpolation mode** to Farrow.

**Output** — Data type of block output

Same as accumulator (default) | Same as first input | Binary point scaling

Specify the data type of the block output as one of the following:

- `Same as accumulator` -- The block specifies the output data type to be the same as that of the accumulator output data type.
- `Same as first input` -- The block specifies the output data type to be the same as that of the data input.
- `Binary point scaling` -- Specify the word length and the fraction length of the block output, in bits.

For more information on the output data type this block uses, see the “Fixed Point” on page 2-1531 section.

**Lock data type settings against changes by the fixed-point tools** — Prevent fixed-point tools from overriding data types

off (default) | on

Select this parameter to prevent the fixed-point tools from overriding the data types you specify in the block dialog box.

### Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | double   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no                                      |
| <b>Multidimensional Signals</b> | no                                      |
| <b>Variable-Size Signals</b>    | yes                                     |
| <b>Zero-Crossing Detection</b>  | no                                      |

### Algorithms

The delay value specified at the **Delay** port serves as an index into the block's memory,  $U$ , which stores, at a minimum, the  $D_{\max}+1$  most recent samples received at the **In** port for each channel. For example, an integer delay of 5 on a scalar input sequence retrieves and outputs the fifth most recent input sample from the block's memory,  $U(6)$ . The block computes fractional delays by interpolating between stored samples. The block uses a linear, FIR, or farrow interpolation method to interpolate signal values at noninteger sample intervals.

#### Linear Interpolation Mode

For noninteger delays, at each sample time, the linear interpolation method uses the two samples in memory nearest to the specified delay to compute a value for the sample at that time.



For a vector data input, the output vector,  $y$ , is computed using the following relation:

$$\begin{aligned} v_i &= \text{floor}(v) \\ v_f &= v - v_i \\ y(i) &= U(i - v_i - 1) * v_f + U(i - v_i) * (1 - v_f) \end{aligned}$$

where,

- $i$  -- Index of the current sample
- $v$  -- Fractional delay
- $v_i$  -- Integer part of the delay
- $v_f$  -- Fractional part of the delay
- $U$  -- Input data vector
- $y$  -- Output data vector
- $U(i - v_i)$ ,  $U(i - v_i - 1)$  -- Two samples in memory nearest to the specified delay
- $i - v_i$  -- Distance, in samples, between the current index and the nearest point in the interpolation line.

The variable fractional delay stores the  $D_{max} + 1$  most recent samples received at the input for each channel, where  $D_{max}$  is the maximum delay specified.  $U$  represents the stored samples.

### FIR Interpolation Mode

In the FIR interpolation mode, the block stores the  $D_{max} + P + 1$  most recent samples received at the input for each channel, where  $P$  is the specified interpolation filter half-length.

In this mode, the block provides a discrete set of fractional delays:

$$v + \frac{i}{L}, \quad v \geq P - 1, \quad i = 0, 1, \dots, L - 1$$

If  $v$  is less than  $P - 1$ , the behavior depends on the **For small input delay values** parameter. You can specify the block's behavior when the input delay value is too small to center the kernel (less than  $P - 1$ ), by setting the **For small input delay values** parameter:

- **Clip to the minimum value necessary for centered kernel** -- The FIR interpolation method remains in use. The small input delay values are clipped to the smallest value necessary to center the kernel.
- **Switch to linear interpolation if kernel cannot be centered** -- Fractional delays are computed using linear interpolation when the input delay value is less than  $P - 1$ .

In the FIR interpolation mode, the algorithm implements a polyphase structure to compute a value for each sample at the specified delay. Each arm of the structure corresponds to a different delay value. The output computed for each sample corresponds to the output of the arm with a delay value nearest to the specified input delay. Therefore, only a discrete set of delays is actually possible. The number of coefficients in each of the  $L$  filter arms of the polyphase structure is  $2P$ . In most cases, using values of  $P$  between 4 and 6 provides you with reasonably accurate interpolation values.

The `designMultirateFIR` function designs the FIR interpolation filter.

For example, when you set the following values:

- Interpolation filter half-length ( $P$ ) to 4
- Interpolation points per input sample to 10
- Normalized input bandwidth to 1
- Stopband attenuation to 80 dB

The filter coefficients are given by:

```
b = designMultirateFIR(10,1,4,80);
```

The algorithm then implements this filter as a polyphase structure.

Increasing the filter half length ( $P$ ) increases the accuracy of the interpolation, but also increases the number of computations performed per input sample. The amount of memory needed to store the filter coefficients increases too. Increasing the interpolation points per sample ( $L$ ) increases the number of representable discrete delay points, but also increases the simulation's memory requirements. The computational load per sample is not affected.

The normalized input bandwidth from 0 to 1 allows you to take advantage of the bandlimited frequency content of the input. For example, if you know that the input signal does not have frequency content above  $F_s/4$ , you can specify 0.5 normalized bandwidth to constrain the frequency content of the output to that range.

---

**Note** You can consider each of the  $L$  interpolation filters to correspond to one output phase of an upsample-by- $L$  FIR filter. Therefore, the normalized input value improves the stopband in critical regions and relaxes the stopband requirements in frequency regions without signal energy.

---

### Farrow Interpolation Mode

In the farrow interpolation mode, the block stores the  $D_{max}+N/2+1$  most recent samples received at the input for each channel, where  $N$  is the specified farrow filter length.

The algorithm uses the LaGrange method to interpolate values.

To increase the minimum possible delay value, select the **Disable direct feedthrough by increasing minimum possible delay by one** check box. Checking this box prevents algebraic loops from occurring when you use the block inside a feedback loop.

To specify the behavior when the input delay value is too small to center the kernel (less than  $\frac{N}{2} - 1$ ), use the Farrow small delay action setting.

- **Clip to the minimum value necessary for centered kernel** -- The block clips small input delay values to the smallest value necessary to keep the kernel centered. This increases  $D_{min}$  but yields more accurate interpolation values.
- **Use off-centered kernel** -- The fractional delays are computed using a Farrow filter with an off-centered kernel. This mode does not increase  $D_{min}$ , but the results for input delay values less than  $\frac{N}{2} - 1$  are less accurate than the results achieved by keeping the kernel centered.

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

The diagrams in the following sections show the data types used within the Variable Fractional Delay for fixed-point signals.

Although you can specify most of these data types, the following data types are computed internally by the block and cannot be directly specified on the block dialog box.

| Data Type             | Word Length                               | Fraction Length         |
|-----------------------|---|-------------------------|
| vf data type          | Same word length as the coefficients      | Same as the word length |
| HoldInteger data type | Same word length as the input delay value | 0 bits                  |
| Integer data type     | 32 bits                                   | 0 bits                  |

**Note** When the input is fixed point, all internal data types are signed fixed point.

To compute the integer ( $v_i$ ) and fractional ( $v_f$ ) parts of the input delay value ( $v$ ), the Variable Fractional Delay block uses the following equations:

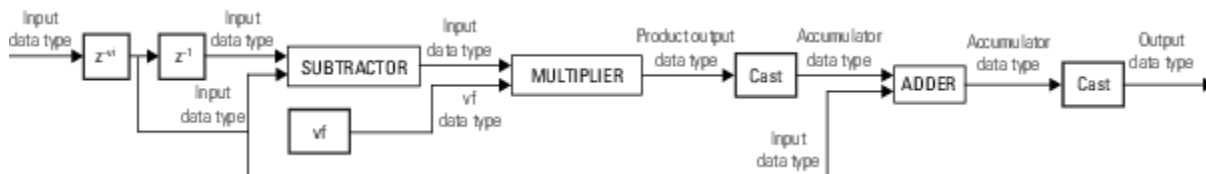
$$D_{min} < v < D_{max} \Rightarrow \begin{cases} v_i = \text{floor}(v) \\ v_f = v - v_i \end{cases}$$

$$v \leq D_{min} \Rightarrow \begin{cases} v_i = D_{min} \\ v_f = 0 \end{cases}$$

$$v \geq D_{max} \Rightarrow \begin{cases} v_i = D_{max} \\ v_f = 0 \end{cases}$$

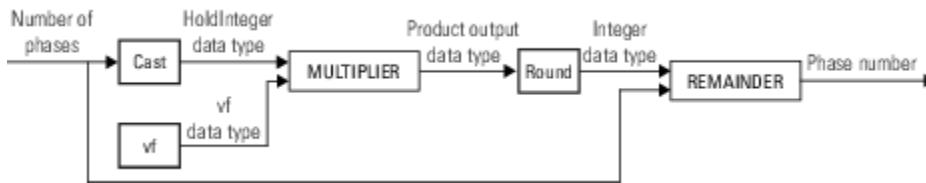
### Linear Interpolation Mode

The following diagram shows the fixed-point data types used by the Linear interpolation mode of the Variable Fractional Delay block.

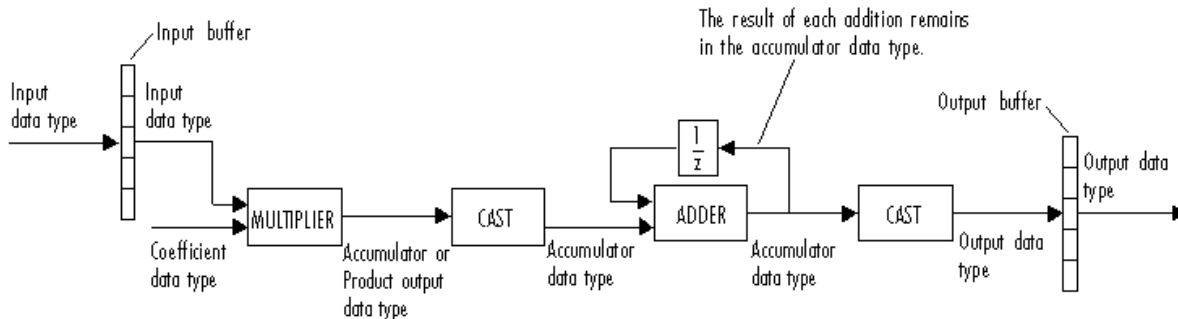


### FIR Interpolation Mode

The following diagram illustrates how the Variable Fractional Delay block selects the arm of the polyphase filter structure that most closely matches the fractional delay value ( $v_f$ ).



The following diagram shows the fixed-point data types used by the variable fractional delay algorithm in the FIR interpolation mode.



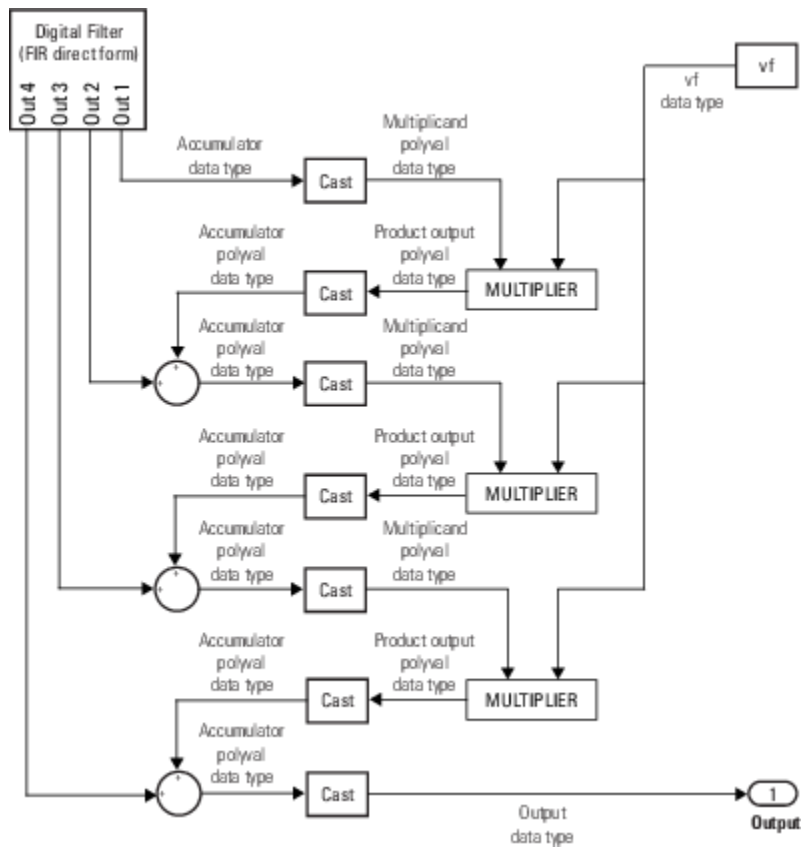
You can set the coefficient, product output, accumulator, and output data types in the block. This diagram shows that input data is stored in the input buffer with the same data type and scaling as the input. The block stores filtered data and any initial conditions in the output buffer using the output data type and scaling that you set.

When at least one of the inputs to the multiplier is real, the output of the multiplier is in the product output data type. When both inputs to the multiplier are complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication, see “Multiplication Data Types”.

### Farrow Interpolation Mode

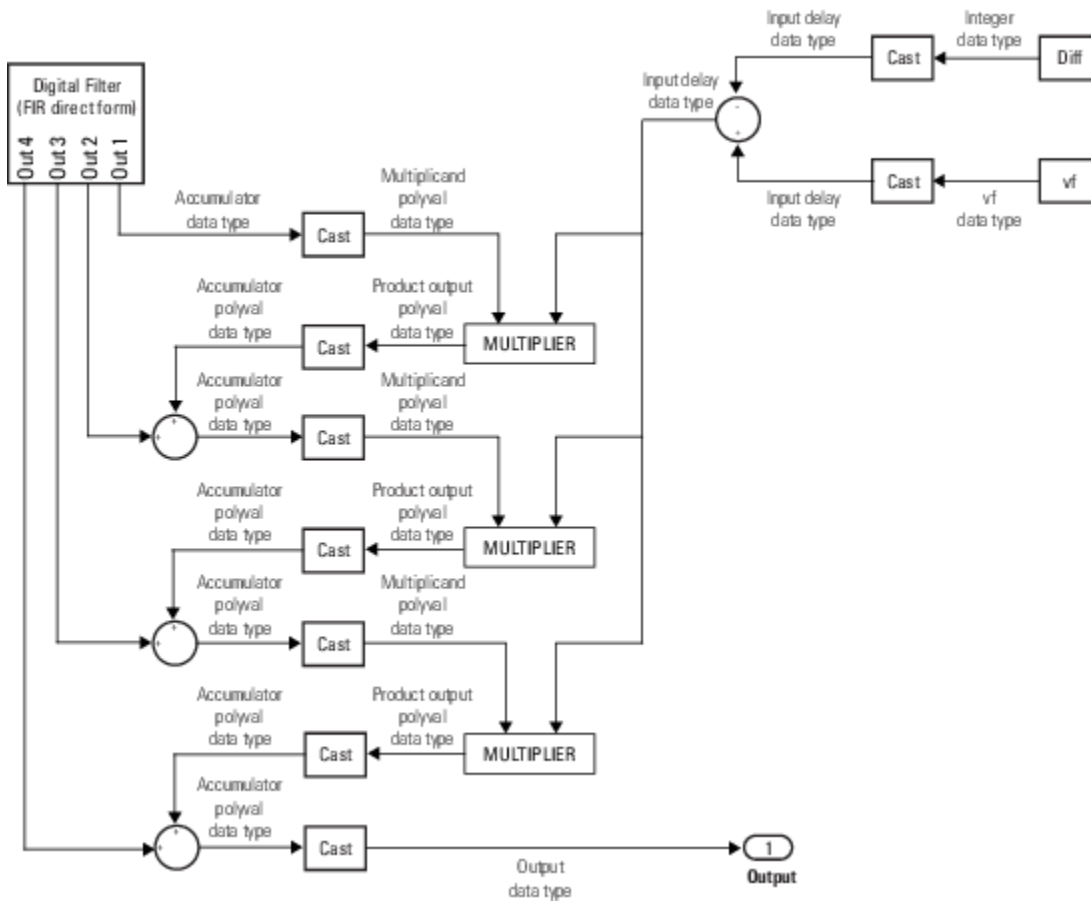
The following diagram shows the fixed-point data types used by the Farrow interpolation mode when:

- Farrow filter length is set to 4
- Farrow small delay action is set to Clip to the minimum value necessary for centered kernel



The following diagram shows the fixed-point data types used by the Farrow interpolation mode when:

- Farrow filter length is set to 4.
- Farrow small delay action is set to Use off-centered kernel.



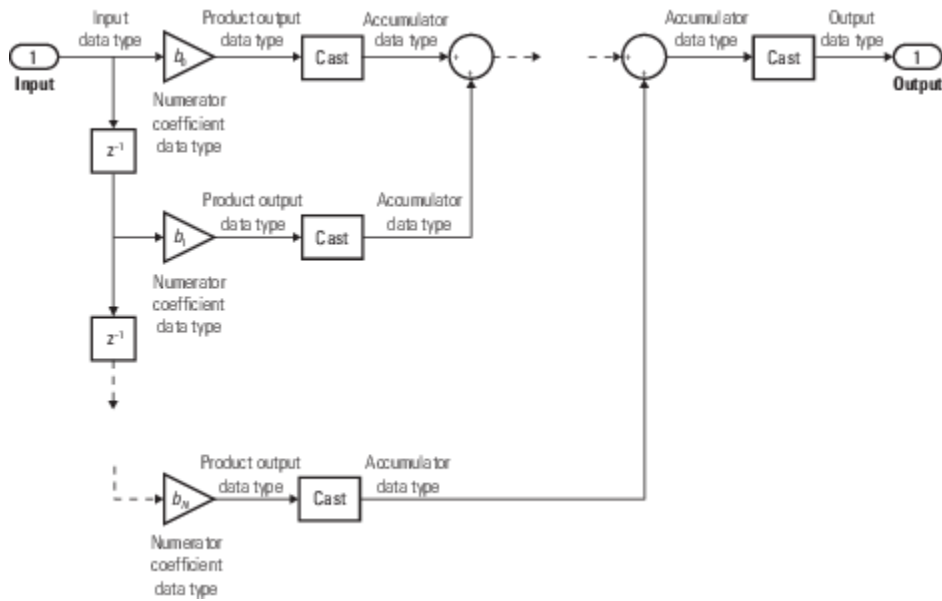
Diff is computed from the integer part of the delay value ( $v_i$ ) and the farrow filter length ( $N$ ) according to the following equation:

$$Diff = v_i - \left(\frac{N - 1}{2}\right)$$

$$Diff \geq 0 \Rightarrow Diff = 0$$

$$Diff < 0 \Rightarrow Diff = - Diff$$

The following diagram shows the fixed-point data types used by the Digital Filter's FIR direct form filter.



## See Also

### Objects

`dsp.VariableFractionalDelay`

### Blocks

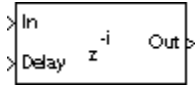
Delay | Unit Delay | Variable Integer Delay

### Topics

“Fractional Delay Filters Using Farrow Structures”

## Variable Integer Delay (Obsolete)

Delay input by time-varying integer number of sample periods



### Library

dspobslib

### Description

---

**Note** The Variable Integer Delay block has been replaced with the Simulink Variable Integer Delay block. Existing instances of the DSP block will continue to operate, but certain functionality will be disabled in future releases. See “Functionality being removed or replaced for blocks and System objects”.

---

The Variable Integer Delay block delays the discrete-time input at the In port by the integer number of sample intervals specified by the input to the Delay port. The sample rate of the input signal at the Delay port must be the same as the sample rate of the input signal at the In port. When these sample rates are not the same, you need to insert a Zero-Order Hold or Rate Transition block in order to make the sample rates identical. When you set the **Input processing** parameter to **Elements as channels (sample based)**, the delay for an N-D input can be a scalar value to uniformly delay every sample in every channel, or a matrix containing one delay value for each channel of the input. When you set the **Input processing** parameter to **Columns as channels (frame based)**, the delay can be a scalar value to uniformly delay every sample in every channel, a vector containing one delay value for each sample in the input frame, or a vector containing one delay value for each channel in the input frame.

The delay values should be in the range of 0 to  $D$ , where  $D$  is the **Maximum delay**. Delay values greater than  $D$  or less than 0 are clipped to those respective values and noninteger delays are rounded to the nearest integer value.

The Variable Integer Delay block differs from the Delay block in the following ways.

| Variable Integer Delay Block   | Delay Block   |
|--|---|
| The delay is provided as an input to the Delay port.   | You specify the delay as a parameter setting in the dialog box.   |
| Delay can vary with time; for example, when the block performs frame-based processing, the $n$ th element's delay in the first input frame can differ from the $n$ th element's delay in the second input frame. | Delay cannot vary with time; for example, when the block performs frame-based processing, the $n$ th element's delay is the same for every input frame. |



| Variable Integer Delay Block  | Delay Block   |
|---|---|
| When you use the Variable Integer Delay block in a feedback loop, you must check the <b>Disable direct feedthrough by increasing minimum possible delay by one</b> check box. This prevents the occurrence of an algebraic loop when the delay of the Variable Integer Delay block is driven to zero. | You can use the Delay block to break an algebraic loop. |

### Sample-Based Processing

When you set the **Input processing** parameter to **Elements as channels (sample based)**, the Variable Integer Delay block supports N-D input arrays. When the input is an  $M$ -by- $N$ -by- $P$  array, the block treats each of the  $M*N*P$  elements as independent channels, and applies the delay at the Delay port to each channel.

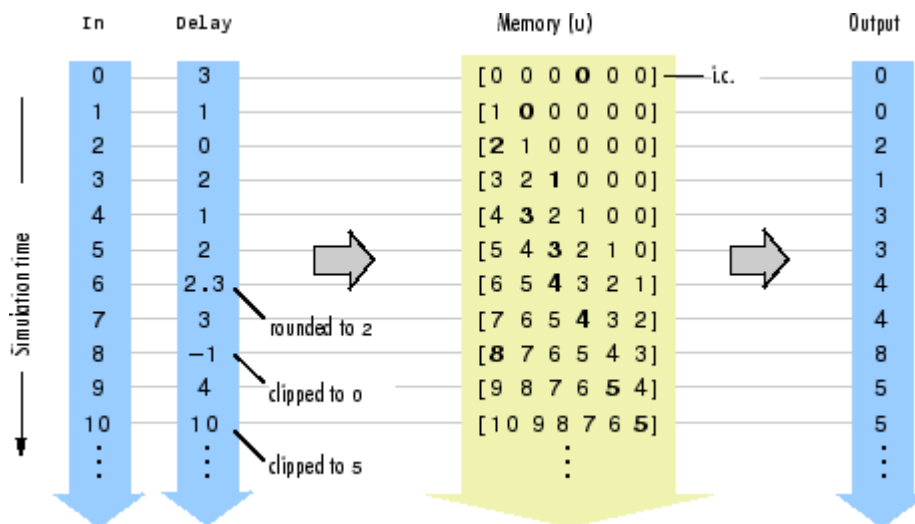
The Variable Integer Delay block stores the  $D+1$  most recent samples received at the In port for each channel. At each sample time the block outputs the stored sample(s) indexed by the input to the Delay port.

For example, when the input to the In port,  $u$ , is a scalar signal, the block stores a vector,  $U$ , of the  $D+1$  most recent signal samples. When the current input sample is  $U(1)$ , the previous input sample is  $U(2)$ , and so on, then the block's output is

```
y = U(v+1); % Equivalent MATLAB code
```

where  $v$  is the input to the Delay port. A delay value of 0 ( $v=0$ ) causes the block to pass through the sample at the In port in the same simulation step that it is received. The block's memory is initialized to the **Initial conditions** value at the start of the simulation (see below).

The next figure shows the block output for a scalar ramp sequence at the In port, a **Maximum delay** of 5, an **Initial conditions** of 0, and a variety of different delays at the Delay port.



The current input at each time step is immediately stored in memory as  $U(1)$ . This allows the current input to be available at the output for a delay of 0 ( $v=0$ ).

The **Initial conditions** parameter specifies the values in the block's memory at the start of the simulation. Unlike the Delay block, the Variable Integer Delay block does not have a fixed initial delay period during which the initial conditions appear at the output. Instead, the initial conditions are propagated to the output only when they are indexed in memory by the value at the Delay port. Both fixed and time-varying initial conditions can be specified in a variety of ways to suit the dimensions of the input sequence.

### Fixed Initial Conditions

The settings in this section specify fixed initial conditions. For a fixed initial condition, the block initializes each of  $D$  samples in memory to the value entered in the **Initial conditions** parameter. A fixed initial condition in sample-based mode can be specified in one of the following ways:

- Scalar value with which to initialize every sample of every channel in memory. For a general  $M$ -by- $N$  input and the parameter settings in this figure,

the block initializes 100  $M$ -by- $N$  matrices in memory with zeros.

- Array of size  $M$ -by- $N$ -by- $D$ . In this case, you can specify different fixed initial conditions for each channel. See the Array bullet in “Time-Varying Initial Conditions” on page 2-1538 below for details.

### Time-Varying Initial Conditions

The following settings specify time-varying initial conditions. For a time-varying initial condition, the block initializes each of  $D$  samples in memory to one of the values entered in the **Initial conditions** parameter. This allows you to specify a unique output value for each sample in memory. A time-varying initial condition in sample-based mode can be specified in one of the following ways:

- Vector containing  $D$  elements with which to initialize memory samples  $U(2:D+1)$ , where  $D$  is the **Maximum delay**. For a scalar input and the parameters in the next figure, the block initializes  $U(2:6)$  with values  $[-1, -1, -1, 0, 1]$ .

- Array of dimension  $M$ -by- $N$ -by- $D$  with which to initialize memory samples  $U(2:D+1)$ , where  $D$  is the **Maximum delay** and  $M$  and  $N$  are the number of rows and columns, respectively, in the input matrix. For a 2-by-3 input and the following parameters, the block initializes memory locations  $U(2:5)$  with values

$$\mathbf{U}(2) = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}, \quad \mathbf{U}(3) = \begin{bmatrix} 2 & 2 & 2 \\ 2 & 2 & 2 \end{bmatrix}, \quad \mathbf{U}(4) = \begin{bmatrix} 3 & 3 & 3 \\ 3 & 3 & 3 \end{bmatrix}, \quad \mathbf{U}(5) = \begin{bmatrix} 4 & 4 & 4 \\ 4 & 4 & 4 \end{bmatrix}$$

An  $M$ -by- $N$ -by- $P$ -by- $D$  array can be entered for the **Initial Conditions** parameter when the input is an  $M$ -by- $N$ -by- $P$  array. The  $(M,N,P,T)$ th sample of the **Initial Conditions** matrix provides the initial condition value for the  $(M,N,P)$ th channel of the input matrix at delay =  $D-t+1$  samples.

### Frame-Based Processing

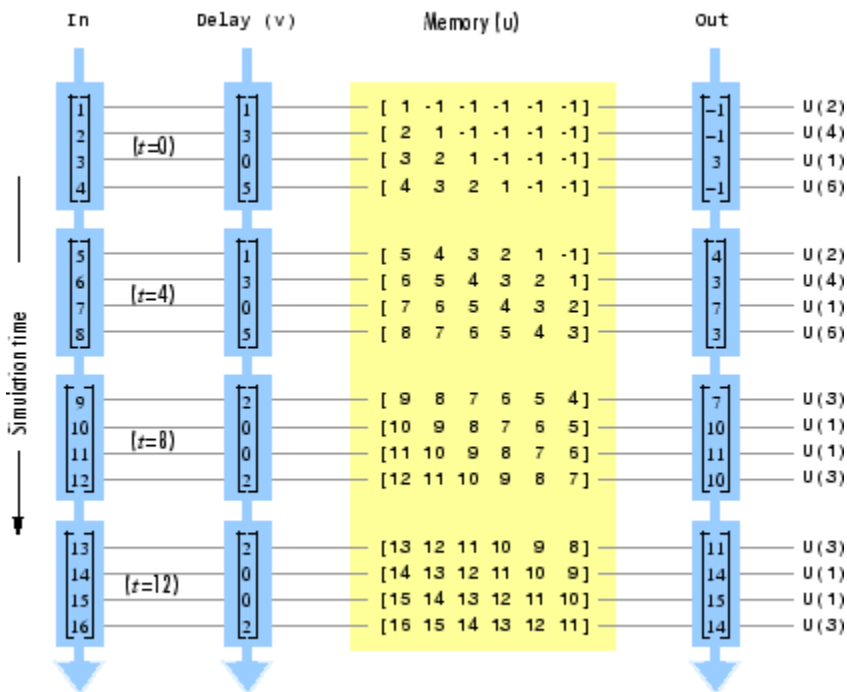
When you set the **Input processing** parameter to `Columns as channels (frame based)`, the input can be an  $M$ -by- $N$  matrix. The block treats each of the  $N$  input columns as independent channels containing  $M$  sequential time samples.

In this mode, the input at the Delay port can be a scalar value to uniformly delay every sample in every channel. It can also be a length- $M$  column vector containing one delay value for each sample in the input frame(s). The set of delays contained in the vector is applied identically to every channel of a multichannel input. The Delay port entry can also be a length- $N$  row vector, containing one delay for each channel. Finally, the Delay port entry can be an  $M$ -by- $N$  matrix, containing a different delay for each corresponding element of the input.

Vector  $v$  does not specify when the samples in the current input frame will appear in the output. Rather,  $v$  indicates which previous input samples (stored in memory) should be included in the current output frame. The first sample in the current output frame is the input sample  $v(1)$  intervals earlier in the sequence, the second sample in the current output frame is the input sample  $v(2)$  intervals earlier in the sequence, and so on.

The illustration below shows how this works for an input with a sample period of 1 and frame size of 4. The **Maximum delay** ( $D_{max}$ ) is 5, and the **Initial conditions** parameter is set to -1. The delay input changes from  $[1 \ 3 \ 0 \ 5]$  to  $[2 \ 0 \ 0 \ 2]$  after the second input frame. The samples in each output frame are the values in memory indexed by the elements of  $v$ :

$$\begin{aligned} y(1) &= U(v(1)+1) \\ y(2) &= U(v(2)+1) \\ y(3) &= U(v(3)+1) \\ y(4) &= U(v(4)+1) \end{aligned}$$



The **Initial conditions** parameter specifies the values in the block's memory at the start of the simulation. Both fixed and time-varying initial conditions can be specified.

### Fixed Initial Conditions

The settings shown in this section specify fixed initial conditions. For a fixed initial condition, the block initializes each of  $D$  samples in memory to the value entered in the **Initial conditions** parameter. A fixed initial condition in frame-based mode can be one of the following:

- Scalar value with which to initialize every sample of every channel in memory. For a general  $M$ -by- $N$  input with the parameter settings below, the block initializes five samples in memory with zeros.

|                          |   |
|--------------------------|---|
| Maximum delay (samples): | 5 |
| Initial conditions:      | 0 |

- Array of size 1-by- $N$ -by- $D$ . In this case, you can specify different fixed initial conditions for each channel. See the Array bullet in “Time-Varying Initial Conditions” on page 2-1540 below for details.

### Time-Varying Initial Conditions

The following setting specifies a time-varying initial condition. For a time-varying initial condition, the block initializes each of  $D$  samples in memory to one of the values entered in the **Initial conditions** parameter. This allows you to specify a unique output value for each sample in memory. When the block is performing frame-based processing, you can specify a time-varying initial condition in the following ways:

- Vector containing  $D$  elements. In this case, all channels have the same set of time-varying initial conditions specified by the entries of the vector. For the ramp input  $[1:100; 1:100]'$  with a

frame size of 4, delay of 5, and the following parameter settings, the block outputs the following sequence of frames at the start of the simulation:

$$\begin{bmatrix} -1 & -1 \\ -2 & -2 \\ -3 & -3 \\ -4 & -4 \end{bmatrix}, \begin{bmatrix} -5 & -5 \\ 1 & 1 \\ 2 & 2 \\ 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 \\ 5 & 5 \\ 6 & 6 \\ 7 & 7 \end{bmatrix}, \dots$$

- Array of size 1-by- $N$ -by- $D$ . In this case, you can specify different time-varying initial conditions for each channel. For the ramp input `[1:100; 1:100]'` with a frame size of 4, delay of 5, and the following parameter settings, the block outputs the following sequence of frames at the start of the simulation:

$$\begin{bmatrix} -1 & -11 \\ -2 & -22 \\ -3 & -33 \\ -4 & -44 \end{bmatrix}, \begin{bmatrix} -5 & -55 \\ 1 & 1 \\ 2 & 2 \\ 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 \\ 5 & 5 \\ 6 & 6 \\ 7 & 7 \end{bmatrix}, \dots$$

By specifying a 1-by- $N$ -by- $D$  initial condition array such that each 1-by- $N$  vector entry is identical, you can implement different fixed initial conditions for each channel.

## Examples

See “Basic Algorithmic Delay” in the *DSP System Toolbox User's Guide*.

## Parameters

### Maximum delay

The maximum delay that the block can produce for any sample. Delay input values exceeding this maximum are clipped at the maximum.

### Initial conditions

The values with which the block's memory is initialized.

### Input processing

Specify how the block should process the input. You can set this parameter to one of the following options:

- **Columns as channels (frame based)** — When you select this option, the block treats each column of the input as a separate channel.
- **Elements as channels (sample based)** — When you select this option, the block treats each element of the input as a separate channel.

---

**Note** The Inherited (this choice will be removed - see release notes) option will be removed in a future release. See “Frame-Based Processing” in the *DSP System Toolbox Release Notes* for more information.

---

### Disable direct feedthrough by increasing minimum possible delay by one

Select this box to disable direct feedthrough by adding one to the minimum possible delay value. When you set the **Input processing** parameter to Columns as channels (frame based), the block increases the minimum possible delay value by *frame size* - 1. Similarly, when you set the **Input processing** parameter to Elements as channels (sample based), the block increases the minimum possible delay value by one sample.

Checking this box allows you to use the Variable Integer Delay block in feedback loops.

## Supported Data Types

| Port  | Supported Data Types   |
|-------|--|
| In    | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed and unsigned)</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul> |
| Delay | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul>   |
| Out   | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed and unsigned)</li> <li>• Boolean</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul> |

## See Also

Delay

DSP System Toolbox

Variable Fractional Delay

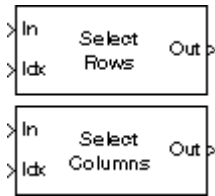
DSP System Toolbox

## Version History

Introduced in R2014b

# Variable Selector

Select subset of rows or columns from input



## Libraries:

DSP System Toolbox / Signal Management / Indexing

DSP System Toolbox HDL Support / Signal Management

## Description

The Variable Selector block extracts a subset of rows or columns from the  $M$ -by- $N$  input matrix  $u$  at each input port. You specify the number of input and output ports in the **Number of input signals** parameter.

When an element of the indexing vector references a nonexistent row or column of the input, the block implements the action that you specify by using the **Invalid index** parameter.

When the indexing vector elements are of the Boolean data type, the block performs logical indexing. Select **Fill empty spaces in outputs (for logical indexing)** to access the **Fill values** parameter. These values are appended to the output to make it as long as the input elements.

---

**Note** The Variable Selector block always copies the selected input rows or columns to a contiguous block of memory (unlike the Simulink Selector block).

---

## Ports

### Input

**In $N$**  —  $N$ th input signal

scalar | vector | matrix

$N$ th input signal, where  $N$  is the number of inputs specified in the “Number of input signals” on page 2-0 parameter.

When the input is an unoriented vector, the “Select” on page 2-0 parameter is ignored and the output is an unoriented vector of length  $L$  containing those elements specified by the length- $L$  indexing vector.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | Boolean | fixed point | enumerated

**Idx** — Index vector

scalar | vector

Specify a scalar value or vector containing the indices of the input rows or columns that appear in the output matrix.

**Dependencies**

This port appears only when you set the “Selector mode” on page 2-0 to **Variable**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point` | `enumerated`

**Output**

**OutN** — *N*th Output signal

`scalar` | `vector` | `matrix`

*N*th output signal, corresponding to the *N*th input signal, where *N* is the number of inputs specified in the “Number of input signals” on page 2-0 parameter. The output is returned as a scalar, vector, or matrix as the same type, size, and complexity as the corresponding input.

For variable and fixed indexing modes, the row selection operation is equivalent to

```
y = u(idx,:) % Equivalent MATLAB code
```

and the column selection operation is equivalent to

```
y = u(:,idx) % Equivalent MATLAB code
```

where `idx` is the length-*L* indexing vector. The row selection output size is *L*-by-*N* and the column selection output size is *M*-by-*L*. Input rows or columns can appear any number of times in the output, or not at all.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point` | `enumerated`

**Parameters**

**Number of input signals** — Number of inputs

1 (default) | positive integer

Specify the number of input signals as a positive integer. An input port is created on the block for each input signal.

**Select** — Select dimension

`Rows` (default) | `Columns`

Specify the dimension of the input to select, `Rows` or `Columns`.

When **Select** is set to:

- `Rows` -- The block extracts rows from each input matrix.
- `Columns` -- The block extracts columns from each input matrix.

When the input is an unoriented vector, the **Select** parameter is ignored. The output is an unoriented vector of length *L* containing those elements specified by the length-*L* indexing vector.

**Selector mode** — Indexing mode

`Variable` (default) | `Fixed`



Specify the type of indexing operation to perform, **Variable** or **Fixed**.

When **Selector mode** is set to:

- **Variable** -- The length- $L$  vector input to the “Idx” on page 2-0 port selects  $L$  rows or columns of each input to pass through to the output. You can update the elements of the indexing vector at each sample time, but the vector length must remain the same throughout the simulation.
- **Fixed** -- The **Idx** port is disabled. The length- $L$  vector that you specified in the “Elements” on page 2-0 parameter selects  $L$  rows or columns of each input to pass through to the output. The **Elements** parameter is tunable, so you can change the values of the indexing vector elements at any time during the simulation but the vector length must remain the same.

**Elements** — Selected elements

[1 3] (default) | scalar | vector

Specify a vector containing the indices of the input rows or columns that appear in the output matrix.

**Tunable:** Yes

**Dependencies**

This parameter appears only when you set the “Selector mode” on page 2-0 to **Fixed**.

**Index mode** — Index mode

Zero-based (default) | One-based

When set to **One-based**, an index value of 1 refers to the first row or column of the input. When set to **Zero-based**, an index value of 0 refers to the first row or column of the input.

**Invalid index** — Error handling

Clip Index (default) | Clip and Warn | Generate Error

Specify how the block handles an invalid index value. You can select one of these options:

- **Clip index** — Clip the index to the nearest valid value and do not issue an alert.

For example, if the block receives a 64-by-4 input and the “Select” on page 2-0 parameter is set to:

- **Rows** -- The block clips an index of 72 to 64.
- **Columns** -- The block clips an index of 72 to 4.

In both cases, the block clips an index of -2 to 1.

- **Clip and warn** — Clip the index to the nearest valid value and display a warning message at the MATLAB command line.
- **Generate error** — Display an error dialog box and terminate the simulation.

**Tunable:** Yes

**Fill empty spaces in outputs (for logical indexing)** — Fill in empty spaces in outputs  
on (default) | off

When the indexing vector elements are of a Boolean data type, the block performs logical indexing. Logical indexing can cause empty spaces in the output. Select this parameter to designate values to be appended to the output in the “Fill values” on page 2-0 parameter.

**Fill values** — Fill values  
0 (default) | scalar value | integer

Specify the fill values when the block performs logical indexing.

#### Dependencies

This parameter appears only when you select the “Fill empty spaces in outputs (for logical indexing)” on page 2-0 parameter.

### Block Characteristics

|                                 |  |
|---------------------------------|--|
| <b>Data Types</b>               | Boolean   double   enumerated   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no   |
| <b>Multidimensional Signals</b> | no   |
| <b>Variable-Size Signals</b>    | no   |
| <b>Zero-Crossing Detection</b>  | no   |

### Version History

Introduced before R2006a

### Extended Capabilities

#### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Generated code relies on the memcpy or memset function (string.h) under certain conditions.

#### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

#### HDL Architecture

This block has one default HDL architecture.

**HDL Block Properties**

|                                  |  |
|----------------------------------|--|
| <b>ConstrainedOutputPipeline</b> | Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder). |
| <b>InputPipeline</b>             | Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).                 |
| <b>OutputPipeline</b>            | Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).               |

**Complex Data Support**

This block supports code generation for complex signals.

**Fixed-Point Conversion**

Design and simulate fixed-point systems using Fixed-Point Designer™.

**See Also****Blocks**

Multiplex Selector | Permute Matrix | Selector | Submatrix

# Variance

Variance of input or sequence of inputs



**Libraries:**  
DSP System Toolbox / Statistics

## Description

The Variance block computes the unbiased variance of each row or column of the input, or along vectors of a specified dimension of the input. It can also compute the variance of the entire input. You can specify the dimension using the **Find the variance value over** parameter. The Variance block can also track the variance in a sequence of inputs over a period of time. To track the variance in a sequence of inputs, select the **Running variance** parameter.

---

**Note** The **Running** mode in the Variance block will be removed in a future release. To compute the running variance in Simulink, use the Moving Variance block instead.

---

## Ports

### Input

**In** — Data input  
vector | matrix |  $N$ -D array

The block accepts real-valued or complex-valued multichannel and multidimensional inputs.

This port is unnamed until you select the **Running variance** parameter and set the **Reset port** parameter to any option other than **None**.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fixed point

**Rst** — Reset port  
scalar

Specify the event that causes the block to reset the running variance. The sample time of the **Rst** input must be a positive integer multiple of the input sample time.

### Dependencies

To enable this port, select the **Running variance** parameter and set the **Reset port** parameter to any option other than **None**.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean

### Output

**Port\_1** — Variance along the specified dimension  
scalar | vector | matrix |  $N$ -D array

When you do not select the **Running variance** parameter, the block computes the variance in each row or column of the input, or along vectors of a specified dimension of the input. It can also compute the variance of the entire input at each individual sample time. Each element in the output array  $y$  is the variance of the corresponding column, row, or entire input. The output array  $y$  depends on the setting of the **Find the variance value over** parameter.

Consider a three-dimensional input signal of size  $M$ -by- $N$ -by- $P$ . When you set **Find the variance value over** to:

- **Entire input** — The output at each sample time is a scalar that contains the variance of the  $M$ -by- $N$ -by- $P$  input matrix.
- **Each row** — The output at each sample time consists of an  $M$ -by-1-by- $P$  array, where each element contains the variance of each vector over the second dimension of the input. For an  $M$ -by- $N$  matrix input, the output at each sample time is an  $M$ -by-1 column vector.
- **Each column** — The output at each sample time consists of a 1-by- $N$ -by- $P$  array, where each element contains the variance of each vector over the first dimension of the input. For an  $M$ -by- $N$  matrix input, the output at each sample time is a 1-by- $N$  row vector.

In this mode, the block treats length- $M$  unoriented vector inputs as  $M$ -by-1 column vectors.

- **Specified dimension** — The output at each sample time depends on the value of the **Dimension** parameter. If you set the **Dimension** to 1, the output is the same as when you select **Each column**. If you set the **Dimension** to 2, the output is the same as when you select **Each row**. If you set the **Dimension** to 3, the output at each sample time is an  $M$ -by- $N$  matrix containing the variance of each vector over the third dimension of the input.

When you select **Running variance**, the block tracks the variance of each channel in a time sequence of inputs. In this mode, you must also specify a value for the **Input processing** parameter. When you set **Input processing** to:

- **Elements as channels (sample based)** — The block treats each element of the input as a separate channel. For a three-dimensional input signal of size  $M$ -by- $N$ -by- $P$ , the block outputs an  $M$ -by- $N$ -by- $P$  array. Each element  $y_{ijk}$  of the output contains the variance of the element  $u_{ijk}$  for all inputs since the last reset.

When a reset event occurs, the running variance  $y_{ijk}$  in the current frame is reset to the element  $u_{ijk}$ .

- **Columns as channels (frame based)** — The block treats each column of the input as a separate channel. This option does not support input signals with more than two dimensions. For a two-dimensional input signal of size  $M$ -by- $N$ , the block outputs an  $M$ -by- $N$  matrix. Each element  $y_{ij}$  of the output contains the variance of the elements in the  $j$ th column of all inputs since the last reset, up to and including the element  $u_{ij}$  of the current input.

When a reset event occurs, the running variance for each channel becomes the variance of all the samples in the current input frame, up to and including the current input sample.

The data type of the output matches the data type of the input.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fixed point

## Parameters

### Main Tab

**Running variance** — Option to select running variance  
off (default) | on

When you select the **Running variance** parameter, the block tracks the variance value of each channel in a time sequence of inputs.

**Find the variance value over** — Dimension over which variance is computed  
Each column (default) | Entire input | Each row | Specified dimension

- **Each column** — The block outputs the variance over each column.
- **Each row** — The block outputs the variance over each row.
- **Entire input** — The block outputs the variance over the entire input.
- **Specified dimension** — The block outputs the variance over the dimension specified in the **Dimension** parameter.

### Dependencies

To enable this parameter, clear the **Running variance** parameter.

**Dimension** — Custom dimension  
1 (default) | scalar

Specify the dimension (one-based value) of the input signal over which the variance is computed. The value of this parameter must be greater than 0 and less than or equal to the number of dimensions in the input signal.

### Dependencies

To enable this parameter, set **Find the variance value over** to **Specified dimension**.

**Input processing** — Method to process the input in running mode  
Columns as channels (frame based) (default) | Elements as channels (sample based)

- **Columns as channels (frame based)** — The block treats each column of the input as a separate channel. This option does not support input signals with more than two dimensions. For a two-dimensional input signal of size  $M$ -by- $N$ , the block outputs an  $M$ -by- $N$  matrix. Each element  $y_{ij}$  of the output contains the variance of the elements in the  $j$ th column of all inputs since the last reset, up to and including the element  $u_{ij}$  of the current input.

When a reset event occurs, the running variance for each channel becomes the variance of all the samples in the current input frame, up to and including the current input sample.

- **Elements as channels (sample based)** — The block treats each element of the input as a separate channel. For a three-dimensional input signal of size  $M$ -by- $N$ -by- $P$ , the block outputs an  $M$ -by- $N$ -by- $P$  array. Each element  $y_{ijk}$  of the output contains the variance of the element  $u_{ijk}$  for all inputs since the last reset.

When a reset event occurs, the running variance  $y_{ijk}$  in the current frame is reset to the element  $u_{ijk}$ .

### Variable-Size Inputs

When your inputs are of variable size, and you select the **Running variance** parameter, then:

- If you set the **Input processing** parameter to **Elements as channels (sample based)**, the state is reset.
- If you set the **Input processing** parameter to **Columns as channels (frame based)**, then:
  - When the input size difference is in the number of channels (number of columns), the state is reset.
  - When the input size difference is in the length of channels (number of rows), the state is not reset, and the running operation is carried out as usual.

### Dependencies

To enable this parameter, select the **Running variance** parameter.

### Reset port — Reset event

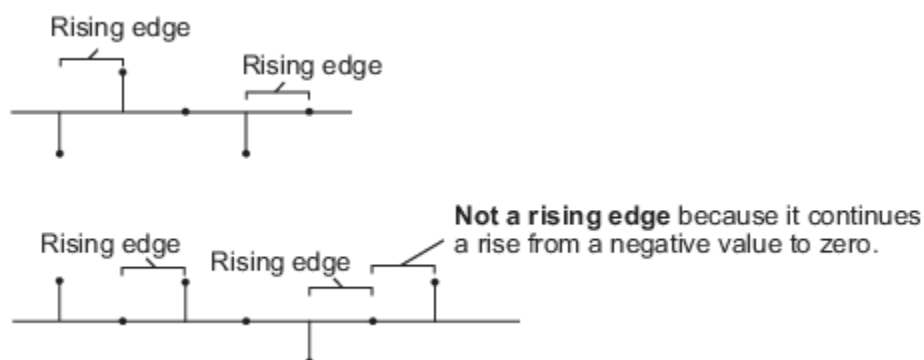
None (default) | Rising edge | Falling edge | Either edge | Non-zero sample

The block resets the running variance whenever a reset event is detected at the optional **Rst** port. The reset sample time must be a positive integer multiple of the input sample time.

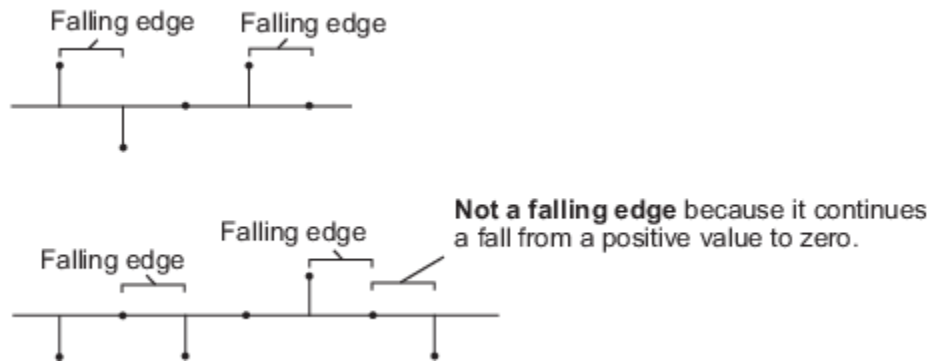
When a reset event occurs while the **Input processing** parameter is set to **Elements as channels (sample based)**, the running variance for each channel is initialized to the value in the corresponding channel of the current input. Similarly, when the **Input processing** parameter is set to **Columns as channels (frame based)**, the running variance for each channel becomes the variance of all the samples in the current input frame, up to and including the current input sample.

Use this parameter to specify the reset event.

- **None** — Disables the **Rst** port.
- **Rising edge** — Triggers a reset operation when the **Rst** input does one of the following:
  - Rises from a negative value to either a positive value or zero.
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero.



- **Falling edge** — Triggers a reset operation when the **Rst** input does one of the following:
  - Falls from a positive value to a negative value or zero.
  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero.



- **Either edge** — Triggers a reset operation when the **Rst** input is a Rising edge or Falling edge.
- **Non-zero sample** — Triggers a reset operation at each sample time, when the **Rst** input is not zero.

---

**Note** When running simulations in the Simulink multitasking mode, reset signals have a one-sample latency. Therefore, when the block detects a reset event, there is a one-sample delay at the reset port rate before the block applies the reset. For more information on latency and the Simulink tasking modes, see “Excess Algorithmic Delay (Tasking Latency)” and “Time-Based Scheduling and Code Generation” (Simulink Coder).

---

### Dependencies

To enable this parameter, select the **Running variance** parameter.

### Data Types Tab

---

**Note** To use these parameters, the data input must be fixed point. For all other inputs, the parameters on the **Data Types** tab are ignored.

---

### Rounding mode — Method of rounding operation

Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Specify the rounding mode for fixed-point operations. For more details, see rounding mode.

### Saturate on integer overflow — Method of overflow action

off (default) | on

When you select this parameter, the block saturates the result of its fixed-point operation. When you clear this parameter, the block wraps the result of its fixed-point operation. For details on saturate and wrap, see overflow mode for fixed-point operations.



**Input-squared product output** — Data type of the input-squared termSame as `input` (default) | Binary point scaling

The squares of the input elements are stored in the **Input-squared product output** data type. If the input is complex, the squares of the real and imaginary parts of the input are stored in this data type. For more details, see “Fixed Point” on page 2-1555.

You can set this parameter to:

- `Inherit`: Same as `input` — The data type is same as the input data type.
- `Binary point scaling` — The **Input-squared product output** data type uses binary point scaling. If you select this option, the block displays the fields to specify the **Word length** and **Fraction length**. The **Signedness** is inherited from the input.

**Input-sum-squared product** — Data type of the input-sum-squared termSame as `input-squared product` (default) | Binary point scaling

The squares of the sum of the input elements are stored in the **Input-sum-squared product** data type. If the input is complex, the squares of the sum of the real parts and the squares of the sum of the imaginary parts are stored in this data type. For more details, see “Fixed Point” on page 2-1555.

You can set this parameter to:

- `Same as input-squared product` — The data type is the same as the input squared-product data type.
- `Binary point scaling` — The **Input-sum-squared product** data type uses binary point scaling. If you select this option, the block displays the fields to specify the **Word length** and **Fraction length**. The **Signedness** is inherited from the input.

**Accumulator** — Accumulator data typeSame as `input-squared product` (default) | Same as `input` | Binary point scaling

**Accumulator** specifies the data type of the output of an accumulation operation in the Variance block. See “Fixed Point” on page 2-1555 for illustrations depicting the use of the accumulator data type in this block.

You can set this parameter to:

- `Same as input-squared product` — The accumulator data type is the same as the input-squared product data type.
- `Same as input` — The accumulator data type is the same as the input data type.
- `Binary point scaling` — The **Accumulator** data type uses binary point scaling. If you select this option, the block displays the fields to specify the **Word length** and **Fraction length**. The **Signedness** is inherited from the input.

**Output** — Output data typeSame as `input-squared product` (default) | Same as `accumulator` | Same as `input` | Binary point scaling

**Output** specifies the data type of the output of the Variance block. See “Fixed Point” on page 2-1555 for information about the use of the output data type in this block. You can set it to:

- Same as `input-squared product` — The output data type is the same as the input-squared product data type.
- Same as `accumulator` — The output data type is the same as the accumulator data type.
- Same as `input` — The output data type is the same as the input data type.
- `Binary point scaling` — The **Output** data type uses binary point scaling. If you select this option, the block displays the fields to specify the **Word length** and **Fraction length**. The **Signedness** is inherited from the input.

**Lock data type settings against changes by the fixed-point tools** — Prevent fixed-point tools from overriding data types

off (default) | on

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block.

## Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | double   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no                                      |
| <b>Multidimensional Signals</b> | no                                      |
| <b>Variable-Size Signals</b>    | yes                                     |
| <b>Zero-Crossing Detection</b>  | no                                      |

## More About

### Variance

The variance of a discrete-time signal is the square of the standard deviation of the signal. Variance gives a measure of deviation of the signal from its mean value.

For purely real or imaginary input,  $u$ , of size  $M$ -by- $N$ , the variance is given by:

$$y = \sigma^2 = \frac{\sum_{i=1}^M \sum_{j=1}^N |u_{ij}|^2 - \frac{\left| \sum_{i=1}^M \sum_{j=1}^N u_{ij} \right|^2}{M*N}}{M*N - 1}.$$

where,

- $u_{ij}$  is the input data element at indices  $i, j$ .
- $M$  is the length of the  $j$ th column.
- $N$  is the number of columns.

For complex inputs, the variance is given by the following equation:

$$\sigma^2 = \sigma_{\text{Re}}^2 + \sigma_{\text{Im}}^2$$

where,

- $\sigma_{\text{Re}}^2$  is the variance of the real part of the complex input.
- $\sigma_{\text{Im}}^2$  is the variance of the imaginary part of the complex input.

## Algorithms

### Variance

When you clear the **Running variance** parameter in the block and specify a dimension, the block produces results identical to the MATLAB `var` function when it is called as `y = var(u,theta,D)`, where,

- `u` is the data input.
- `D` is the dimension.
- `y` is the variance along the specified dimension.

When this block calculates the variance along the entire input, the result is identical to calling the `var` function as `y = var(u(:))`.

For a complex input signal, the variance is the sum of the variances of the real and imaginary parts.

$$\sigma^2 = \sigma_{\text{Re}}^2 + \sigma_{\text{Im}}^2$$

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

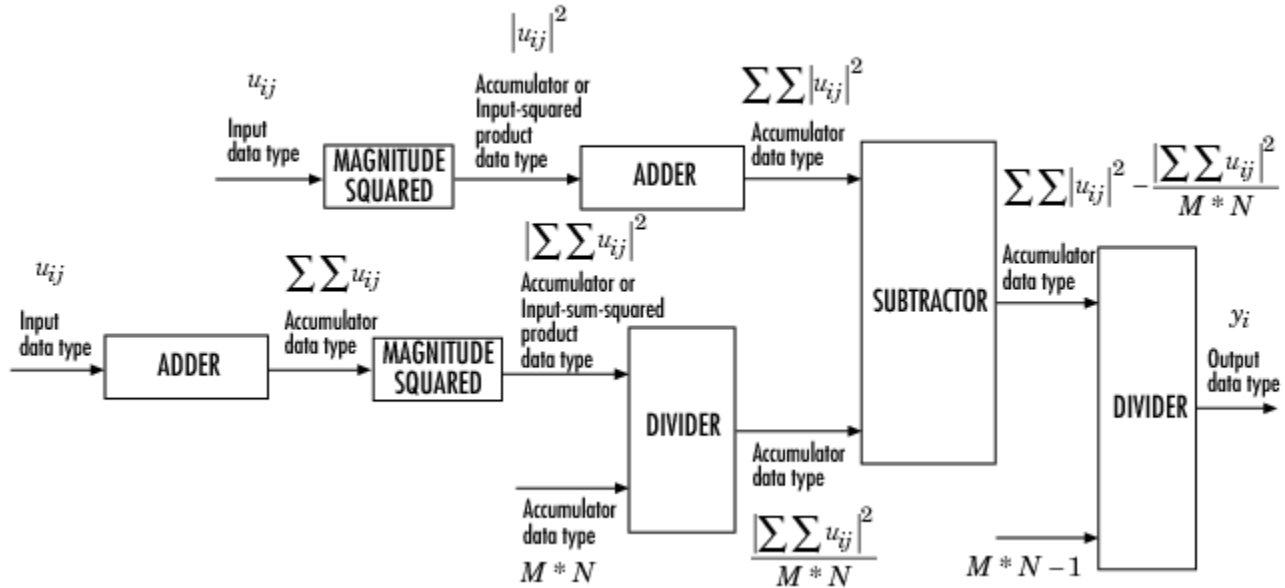
### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

For purely real or imaginary input  $u$  of size  $M$ -by- $N$ , the variance is given by:

$$y = \sigma^2 = \frac{\sum_{i=1}^M \sum_{j=1}^N |u_{ij}|^2 - \frac{\left| \sum_{i=1}^M \sum_{j=1}^N u_{ij} \right|^2}{M*N}}{M*N - 1}.$$

The following diagram shows the data types used within the Variance block when the input is fixed-point.



For complex inputs, the variance is given by the following equation:

$$\sigma^2 = \sigma_{\text{Re}}^2 + \sigma_{\text{Im}}^2$$

## See Also

### Functions

var

### Objects

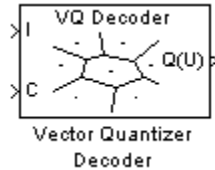
dsp.MovingVariance

### Blocks

Moving Variance

# Vector Quantizer Decoder

Find vector quantizer codeword that corresponds to given, zero-based index value



## Library

Quantizers

dspquant2

## Description

The Vector Quantizer Decoder block associates each input index value with a codeword, a column vector of quantized output values defined in the **Codebook values** parameter. When you input multiple index values into this block, the block outputs a matrix of quantized output vectors. This matrix is created by horizontally concatenating the codeword vectors that correspond to each index value.

You can select how you want to enter the codebook values using the **Source of codebook** parameter. When you select **Specify via dialog**, you can type the codebook values into the block parameters dialog box. Select **Input port** and port C appears on the block. The block uses the input to port C as the **Codebook values** parameter.

The **Codebook values** parameter is a  $k$ -by- $N$  matrix of values, where  $k \geq 1$  and  $N \geq 1$ . Each column of this matrix is a codeword vector, and each codeword vector corresponds to an index value. The index values are zero based; therefore, the first codeword vector corresponds to an index value of 0, the second codeword vector corresponds to an index value of 1, and so on.

The input to this block is a vector of index values, where  $0 \leq \text{index} < N$  and  $N$  is the number of columns of the codebook matrix. Use the **Action for out of range index value** parameter to determine how the block behaves when an input index value is out of this range. When you want any index values less than 0 to be set to 0 and any index values greater than or equal to  $N$  to be set to  $N-1$ , select **Clip**. When you want to be warned when any index values less than 0 are set to 0 and any index values greater than or equal to  $N$  are set to  $N-1$ , select **Clip and warn**. When you want the simulation to stop and display an error when the index values are out of range, select **Error**.

## Data Type Support

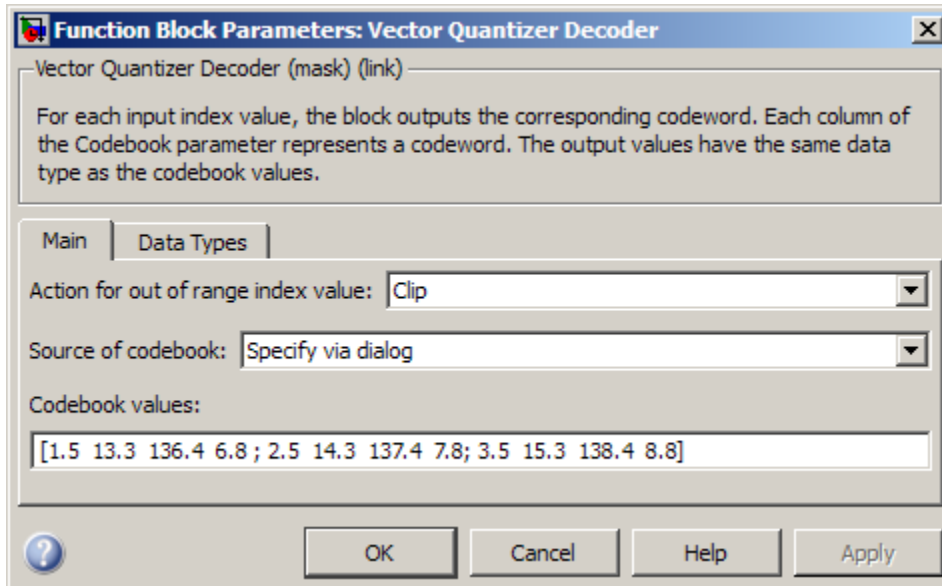
The input to the block can be the index values and the codebook values. The data type of the index input to the block at port I can be `uint8`, `uint16`, `uint32`, `int8`, `int16`, or `int32`. The data type of the codebook values can be `double`, `single`, or `Fixed-point`.

The output of the block is the quantized output values. These quantized output values always have the same data type as the codebook values. When the codebook values are specified via an input port,

the block assigns the same data type to the Q(U) output port. When the codebook values are specified via the dialog, use the **Codebook and output data type** parameter to specify the data type of the Q(U) output port. The data type of the codebook and quantized output can be Same as input, double, single, Fixed-point, User-defined, or Inherit via back propagation.

## Dialog Box

The **Main** pane of the Vector Quantizer Decoder block dialog appears as follows.



### Action for out of range index value

Choose the behavior of the block when an input index value is out of range, where  $0 \leq \text{index} < N$  and  $N$  is the length of the codebook vector. Select **Clip** when you want any index values less than 0 to be set to 0 and any index values greater than or equal to  $N$  to be set to  $N-1$ . Select **Clip and warn** when you want to be warned when any index values less than 0 are set to 0 and any index values greater than or equal to  $N$  are set to  $N-1$ . Select **Error** when you want the simulation to stop and display an error when the index values are out of range.

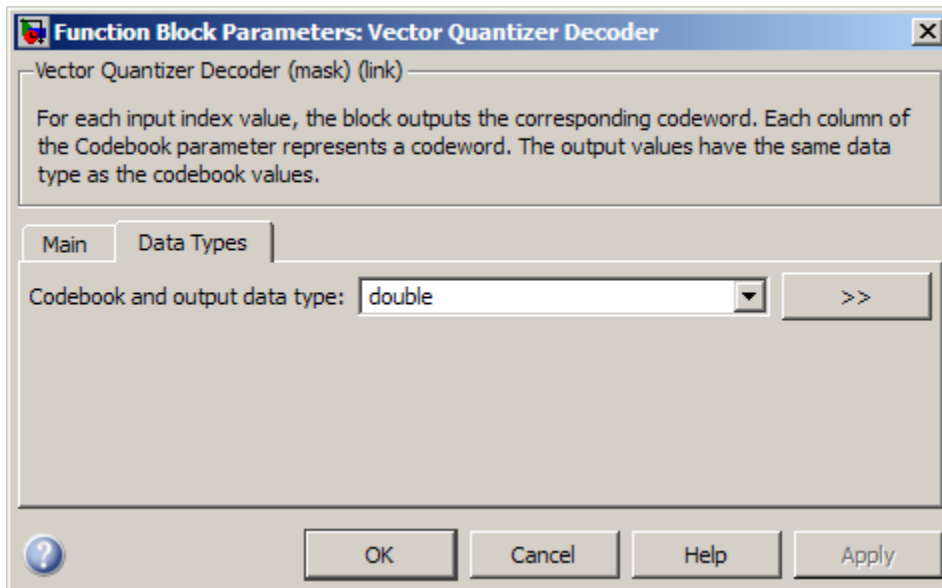
### Source of codebook

Choose **Specify via dialog** to type the codebook values into the block parameters dialog box. Select **Input port** to specify the codebook values using the block's input port, C.

### Codebook values

Enter a  $k$ -by- $N$  matrix of quantized output values, where  $1 \leq k$  and  $1 \leq N$ . Each column of your matrix corresponds to an index value. This parameter is visible if, from the **Source of codebook** list, you select **Specify via dialog**.


The **Data Types** pane of the Vector Quantizer Decoder block dialog appears as follows.



### Codebook and output data type

Specify the data type of the codebook and quantized output values. You can select one of the following:

- A rule that inherits a data type, for example, `Inherit: Same as input`.
- A built in data type, such as `double`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Control Data Types of Signals” (Simulink) in *Simulink User's Guide* (Simulink) for more information.

This parameter is available only when you set the **Source of codebook** parameter to `Specify via dialog`. If you set the **Source of codebook** parameter to `Input port`, the output values have the same data type as the input codebook values.

### Codebook and Output Data Type Assistant Parameters

#### Mode

Select how you would like to specify the data type properties of the **Codebook and output data type**. You can choose:

- **Inherit** — Lets you specify a rule for inheriting a data type, for example, `Inherit: Same as input`
- **Built in**— Lets you specify a built in data type, for example, `double`
- **Fixed point** — Lets you specify the fixed-point attributes of the data type.
- **Expression** — Lets you specify an expression that evaluates to a valid data type, for example, `fixdt(1,16)`

**Signedness**

Specify the **Signedness** for the **Codebook and output data type**.

**Scaling**

Specify the **Scaling** for the **Codebook and output data type**.

For more information see “Scaling” in the DSP System Toolbox User's Guide.

**Word length**

Specify the **Word length** for the **Codebook and output data type**.

**Fraction length**

Specify the **Fraction length** for the **Codebook and output data type**.

**Data type override**

Specify the data type override mode. You can select one of the following options:

- **Inherit** — Inherits the data type override setting specified for the model.
- **Off** — Ignores the data type override setting specified for the model and uses the fixed-point data type you specify

This parameter appears only when you set the **Mode** parameter to **Built in** or **Fixed Point**. For more information, see “Specify Data Types Using Data Type Assistant” (Simulink).

**References**

Gersho, A. and R. Gray. *Vector Quantization and Signal Compression*. Boston: Kluwer Academic Publishers, 1992.

**Supported Data Types**

| Port | Supported Data Types   |
|------|--|
| I    | <ul style="list-style-type: none"> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul>   |
| C    | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point</li> <li>• 8-, 16-, and 32-bit signed integers</li> </ul> |



| Port | Supported Data Types  |
|------|---|
| Q(U) | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point</li> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul> |

## See Also

|                          |                    |
|--------------------------|--------------------|
| Quantizer                | Simulink           |
| Scalar Quantizer Decoder | DSP System Toolbox |
| Uniform Encoder          | DSP System Toolbox |
| Uniform Decoder          | DSP System Toolbox |
| Vector Quantizer Encoder | DSP System Toolbox |

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Generated code relies on the `memcpy` or `memset` function (`string.h`) under certain conditions.

### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

## Vector Quantizer Design

(Removed) Design vector quantizer using Vector Quantizer Design Tool (VQDTool)

---

**Note** The Vector Quantizer Design block has been removed.

---

### Library

Quantizers

dspquant2

### Description

Double-click on the Vector Quantizer Design block to start VQDTool, a GUI that allows you to design and implement a vector quantizer. You can also start VQDTool by typing `vqdtool` at the MATLAB command prompt. Based on your specifications, VQDTool iteratively calculates the codebook values that minimize the mean squared error between the training set and the codebook until the stopping criteria for the design process is satisfied. The block uses the resulting codebook values to implement your vector quantizer.

For the **Training Set** parameter, enter a  $k$ -by- $M$  matrix of values you want to use to train the quantizer codebook. The variable  $k$ , where  $k \geq 1$ , is the length of each training vector. It also represents the dimension of your quantizer. The variable  $M$ , where  $M \geq 2$ , is the number of training vectors. This data can be created using a MATLAB function, such as the default value `randn(10,1000)`, or it can be any variable defined in the MATLAB workspace.

You have two choices for the **Source of initial codebook** parameter. Select **Auto-generate** to have the block choose the values of the initial codebook. In this case, the block picks  $N$  random training vectors as the initial codebook, where  $N$  is the **Number of levels** parameter and  $N \geq 2$ . When you select **User defined**, enter the initial codebook values in the **Initial codebook** field. The initial codebook matrix must have the same number of rows as the training set. Each column of the codebook is a codeword, and your codebook must have at least two codewords.

For the given training set and initial codebook, the block performs an iterative process, using the Generalized Lloyd Algorithm (GLA), to design a final codebook. For each iteration of the GLA, the block first associates each training vector with its nearest codeword by calculating the distortion. You can specify one of the two possible methods for calculating distortion using the **Distortion measure** parameter.

When you select **Squared error** for the **Distortion measure** parameter, the block finds the nearest codeword by calculating the squared error (unweighted). Consider the codebook  $CB = [CW_1 \ CW_2 \ \dots \ CW_N]$ . This codebook has  $N$  codewords; each codeword has  $k$  elements. The  $i$ -th codeword is defined as  $CW_i = [a_{1i} \ a_{2i} \ \dots \ a_{ki}]$ . The training set has  $M$  columns and is defined as  $U = [U_1 \ U_2 \ \dots \ U_M]$ , where the  $p$ -th training vector is  $U_p = [u_{1p} \ u_{2p} \ \dots \ u_{kp}]'$ . The squared error (unweighted) is calculated using the equation

$$D = \sum_{j=1}^k (a_{ji} - u_{jp})^2$$

When you select **Weighted squared error** for the **Distortion measure** parameter, enter a vector or matrix for the **Weighting factor** parameter. When the weighting factor is a vector, its length must be equal to the number of rows in the training set. This weighting factor is used for each training vector. When the weighting factor is a matrix, it must be the same size as the training set matrix. The block finds the nearest codeword by calculating the weighted squared error. If the weighting factor for the  $p$ -th column of the training vector,  $U_p$ , is defined as  $W_p = [w_{1p} \ w_{2p} \ \dots \ w_{kp}]'$ , then the weighted squared error is defined by the equation

$$D = \sum_{j=1}^k w_{jp}(a_{ji} - u_{jp})^2$$

Once the block has associated all the training vectors with their nearest codeword vectors, the block calculates the mean squared error for the codebook and checks to see if the stopping criteria for the process has been satisfied.

The two possible options for the **Stopping criteria** parameter are **Relative threshold** and **Maximum iteration**. When you want the design process to stop when the fractional drop in the squared error is below a certain value, select **Relative threshold**. Then, type the maximum acceptable fractional drop in the **Relative threshold** field. The fraction drop in the squared error is defined as

$$\frac{\text{error at previous iteration} - \text{error at current iteration}}{\text{error at previous iteration}}$$

When you want the design process to stop after a certain number of iterations, choose **Maximum iteration**. Then, enter the maximum number of iterations you want the block to perform in the **Maximum iteration** field. For **Stopping criteria**, you can also choose **Whichever comes first** and enter **Relative threshold** and **Maximum iteration** values. The block stops iterating as soon as one of these conditions is satisfied.

When a training vector has the same distortion for two different codeword vectors, the algorithm uses the **Tie-breaking rule** parameter to determine which codeword vector the training vector is associated with. When you want the training vector to be associated with the lower indexed codeword, select **Lower indexed codeword**. To associate the training vector with the higher indexed codeword, select **Higher indexed codeword**.

With each iteration, the block updates the codeword values in order to minimize the distortion. The **Codebook update method** parameter defines the way the block calculates these new codebook values.

---

**Note** If, for the **Distortion measure** parameter, you choose **Squared error**, the **Codebook update method** parameter is set to **Mean**.

---

If, for the **Distortion measure** parameter, you choose **Weighted squared error** and you choose **Mean** for the **Codebook update method** parameter, the new codeword vector is found as follows. Suppose there are three training vectors associated with one codeword vector. The training vectors are

$$TS_1 = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, TS_3 = \begin{bmatrix} 10 \\ 12 \end{bmatrix}, \text{ and } TS_7 = \begin{bmatrix} 11 \\ 12 \end{bmatrix}.$$

The new codeword vector is calculated as  $CW_{new} = \begin{bmatrix} \frac{1 + 10 + 11}{3} \\ \frac{2 + 12 + 12}{3} \end{bmatrix}$

where the denominator is the number of training vectors associated with this codeword. If, for the **Codebook update method** parameter, you choose **Centroid** and you specify the weighting factors

$$W_1 = \begin{bmatrix} 0.1 \\ 0.2 \end{bmatrix}, W_3 = \begin{bmatrix} 1 \\ 0.6 \end{bmatrix}, \text{ and } W_7 = \begin{bmatrix} 0.3 \\ 0.4 \end{bmatrix}, \text{ the new codeword vector is calculated as}$$

$$CW_{new} = \begin{bmatrix} \frac{(0.1)(1) + (1)(10) + (0.3)(11)}{0.1 + 1 + 0.3} \\ \frac{(0.2)(2) + (0.6)(12) + (0.4)(12)}{0.2 + 0.6 + 0.4} \end{bmatrix}$$

Click **Design and Plot** to design the quantizer with the parameter values specified on the left side of the GUI. The performance curve and the entropy of the quantizer are updated and displayed in the figures on the right side of the GUI.

---

**Note** You must click **Design and Plot** to apply any changes you make to the parameter values in the VQDTool dialog box.

---

The following is an example of how the block calculates the entropy of the quantizer at each iteration. Suppose you have a codebook with four codewords and a training set with 200 training vectors. Also suppose that, at the  $i$ -th iteration, 40 training vectors are associated with the first codeword, 60 training vectors are associated with the second codeword, 20 training vectors are associated with the third codeword, and 80 training vectors are associated with the fourth codeword. The probability that a training vector is associated with the first codeword is  $\frac{40}{200}$ . The probabilities that training vectors are associated with the second, third, and fourth codewords are  $\frac{60}{200}$ ,  $\frac{20}{200}$ , and  $\frac{80}{200}$ , respectively. The GUI uses these probabilities to calculate the entropy according to the equation

$$H = \sum_{i=1}^N -p_i \log_2 p_i$$

where  $N$  is the number of codewords. Based on these probabilities, the GUI calculates the entropy of the quantizer at the  $i$ -th iteration as

$$H = - \left( \frac{40}{200} \log_2 \frac{40}{200} + \frac{60}{200} \log_2 \frac{60}{200} + \frac{20}{200} \log_2 \frac{20}{200} + \frac{80}{200} \log_2 \frac{80}{200} \right)$$

$$H = 1.8464$$

VQDTool can export parameter values that correspond to the figures displayed in the GUI. Click the **Export Outputs** button, or press **Ctrl+E**, to export the **Final Codebook**, **Mean Square Error**, and **Entropy** values to the workspace, a text file, or a MAT-file.

In the **Model** section of the GUI, specify the destination of the block that will contain the parameters of your quantizer. For **Destination**, select **Current model** to create a block with your parameters in

the model you most recently selected. Type `gcs` in the MATLAB Command Window to display the name of your current model. Select `New model` to create a block in a new model file.

From the **Block type** list, select `Encoder` to design a Vector Quantizer Encoder block. Select `Decoder` to design a Vector Quantizer Decoder block. Select `Both` to design a Vector Quantizer Encoder block and a Vector Quantizer Decoder block.

In the **Encoder block name** field, enter a name for the Vector Quantizer Encoder block. In the **Decoder block name** field, enter a name for the Vector Quantizer Decoder block. When you have a Vector Quantizer Encoder and/or Decoder block in your destination model with the same name, select the **Overwrite target block** check box to replace the block's parameters with the current parameters. When you do not select this check box, a new Vector Quantizer Encoder and/or Decoder block is created in your destination model.

Click **Generate Model**. VQDTool uses the parameters that correspond to the current plots to set the parameters of the Vector Quantizer Encoder and/or Decoder blocks.

## Parameters

### Training Set

Enter the samples of the signal you would like to quantize. This data set can be a MATLAB function or a variable defined in the MATLAB workspace. The typical length of this data vector is  $1e5$ .

### Source of initial codebook

Select `Auto-generate` to have the block choose the initial codebook values. Choose `User defined` to enter your own initial codebook values.

### Number of levels

Enter the number of codeword vectors,  $N$ , in your codebook matrix, where  $N \geq 2$ .

### Initial codebook

Enter your initial codebook values. From the **Source of initial codebook** list, select `User defined` in order to activate this parameter. The codebook must have the same number of rows as the training set. You must provide at least two codeword vectors.

### Distortion measure

When you select `Squared error`, the block finds the nearest codeword by calculating the squared error (unweighted). When you select `Weighted squared error`, the block finds the nearest codeword by calculating the weighted squared error.

### Weighting factor

Enter a vector or matrix. The block uses these values to compute the weighted squared error. When the weighting factor is a vector, its length must be equal to the number of rows in the training set. This weighting factor is used for each training vector. When the weighting factor is a matrix, it must be the same size as the training set matrix. The individual weighting factors cannot be negative. The weighting factor vector or matrix cannot contain all zeros.

### Stopping criteria

Choose `Relative threshold` to enter the maximum acceptable fractional drop in the squared quantization error. Choose `Maximum iteration` to specify the number of iterations at which to stop. Choose `Whichever comes first` and the block stops the iteration process as soon as the relative threshold or maximum iteration value is attained.

**Relative threshold**

This parameter is available when you choose `Relative threshold` or `Whichever comes first` for the **Stopping criteria** parameter. Enter the value that is the maximum acceptable fractional drop in the squared quantization error.

**Maximum iteration**

This parameter is available when you choose `Maximum iteration` or `Whichever comes first` for the **Stopping criteria** parameter. Enter the maximum number of iterations you want the block to perform.

**Tie-breaking rules**

When a training vector has the same distortion for two different codeword vectors, select `Lower indexed codeword` to associate the training vector with the lower indexed codeword. Select `Higher indexed codeword` to associate the training vector with the higher indexed codeword.

**Codebook update method**

When you choose `Mean`, the new codeword vector is calculated by taking the average of all the training vector values that were associated with the original codeword vector. When you choose `Centroid`, the block calculates the new codeword vector by taking the weighted average of all the training vector values that were associated with the original codeword vector. Note that if, for the **Distortion measure** parameter, you choose `Squared error`, the **Codebook update method** parameter is set to `Mean`.

**Destination**

Choose `Current model` to create a Vector Quantizer block in the model you most recently selected. Type `gcs` in the MATLAB Command Window to display the name of your current model. Choose `New model` to create a block in a new model file.

**Block type**

Select `Encoder` to design a Vector Quantizer Encoder block. Select `Decoder` to design a Vector Quantizer Decoder block. Select `Both` to design a Vector Quantizer Encoder block and a Vector Quantizer Decoder block.

**Encoder block name**

Enter a name for the Vector Quantizer Encoder block.

**Decoder block name**

Enter a name for the Vector Quantizer Decoder block.

**Overwrite target block**

When you do not select this check box and a Vector Quantizer Encoder and/or Decoder block with the same block name exists in the destination model, a new Vector Quantizer Encoder and/or Decoder block is created in the destination model. When you select this check box and a Vector Quantizer Encoder and/or Decoder block with the same block name exists in the destination model, the parameters of these blocks are overwritten by new parameters.

**Generate Model**

Click this button and VQDTool uses the parameters that correspond to the current plots to set the parameters of the Vector Quantizer Encoder and/or Decoder blocks.

**Design and Plot**

Click this button to design a quantizer using the parameters on the left side of the GUI and to update the performance curve and entropy plots on the right side of the GUI.

You must click **Design and Plot** to apply any changes you make to the parameter values in the VQDTool GUI.

### Export Outputs

Click this button, or press **Ctrl+E**, to export the **Final Codebook**, **Mean Squared Error**, and **Entropy** values to the workspace, a text file, or a MAT-file.

### Supported Data Types

- Double-precision floating point

## Version History

Introduced before R2006a

### R2021b: Vector Quantizer Design block has been removed

The Vector Quantizer Design block has been removed.

### References

- [1] Gersho, A. and R. Gray. *Vector Quantization and Signal Compression*. Boston: Kluwer Academic Publishers, 1992.

### Extended Capabilities

#### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

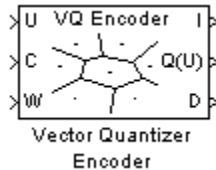
### See Also

#### Blocks

Quantizer | Scalar Quantizer Decoder | Uniform Encoder | Uniform Decoder | Vector Quantizer Decoder | Vector Quantizer Encoder

## Vector Quantizer Encoder

For given input, find index of nearest codeword based on Euclidean or weighted Euclidean distance measure



## Library

Quantizers

dspquant2

## Description

The Vector Quantizer Encoder block compares each input column vector to the codeword vectors in the codebook matrix. Each column of this codebook matrix is a codeword. The block finds the codeword vector nearest to the input column vector and returns its zero-based index. This block supports real floating-point and fixed-point signals on all input ports.

The block finds the nearest codeword by calculating the distortion. The block uses two methods for calculating distortion: Euclidean squared error (unweighted) and weighted Euclidean squared error. Consider the codebook,  $CB = [CW_1 \ CW_2 \ \dots \ CW_N]$ . This codebook has  $N$  codewords; each codeword has  $k$  elements. The  $i$ -th codeword is defined as a column vector,  $CW_i = [a_{1i} \ a_{2i} \ \dots \ a_{ki}]$ . The multichannel input has  $M$  columns and is defined as  $U = [U_1 \ U_2 \ \dots \ U_M]$ , where the  $p$ -th input column vector is  $U_p = [u_{1p} \ u_{2p} \ \dots \ u_{kp}]$ . The squared error (unweighted) is calculated using the equation

$$D = \sum_{j=1}^k (a_{ji} - u_{jp})^2$$

The weighted squared error is calculated using the equation

$$D = \sum_{j=1}^k w_j (a_{ji} - u_{jp})^2$$

where the weighting factor is defined as  $W = [w_1 \ w_2 \ \dots \ w_k]$ . The index of the codeword that is associated with the minimum distortion is assigned to the input column vector.

You can select how you want to enter the codebook values using the **Source of codebook** parameter. When you select **Specify via dialog**, you can type the codebook values into the block parameters dialog box. Select **Input port** and port C appears on the block. The block uses the input to port C as the **Codebook** parameter.

The **Codebook** parameter is an  $k$ -by- $N$  matrix of values, where  $k \geq 1$  and  $N \geq 1$ . Each input column vector is compared to this codebook. Each column of the codebook matrix is a codeword, and each



codeword has an index value. The first codeword vector corresponds to an index value of 0, the second codeword vector corresponds to an index value of 1, and so on. The codeword vectors must have the same number of rows as the input, U.

For the **Distortion measure** parameter, select **Squared error** when you want the block to calculate the distortion by evaluating the Euclidean distance between the input column vector and each codeword in the codebook. Select **Weighted squared error** when you want to use a weighting factor to emphasize or deemphasize certain input values.

For the **Source of weighting factor** parameter, select **Specify via dialog** to enter a weighting factor vector in the dialog box. Choose **Input port** to specify the weighting factor using port W.

Use the **Weighting factor** parameter to emphasize or deemphasize certain input values when calculating the distortion measure. For example, consider the  $p$ -th input column vector,  $U_p$ , as previously defined. When you want to neglect the effect of the first element of this vector, enter  $[0 \ 1 \ 1 \ \dots \ 1]$  as the **Weighting factor** parameter. This weighting factor is used to calculate the weighted squared error using the equation

$$D = \sum_{j=1}^k w_j (a_{ji} - u_{jp})^2$$

Because of the weighting factor used in this example, the weighted squared error is not affected by the first element of the input matrix. Therefore, the first element of the input column vector no longer impacts the choice of index value output by the Vector Quantizer Encoder block.

Use the **Index output data type** parameter to specify the data type of the index values output at port I. The data type of the index values can be `int8`, `uint8`, `int16`, `uint16`, `int32`, or `uint32`.

When an input vector is equidistant from two codewords, the block uses the **Tie-breaking rule** parameter to determine which index value the block chooses. When you want the input vector to be represented by the lower index valued codeword, select **Choose the lower index**. To represent the input column vector by the higher index valued codeword, select **Choose the higher index**.

Select the **Output codeword** check box to output at port Q(U) the codeword vectors that correspond to each index value. When the input is a matrix, the corresponding codeword vectors are horizontally concatenated into a matrix.

Select the **Output quantization error** check box to output at port D the quantization error that results when the block represents the input column vector by its nearest codeword. When the input is a matrix, the quantization error values are horizontally concatenated.

The Vector Quantizer Encoder block accepts real floating-point and fixed-point inputs. For more information on the data types accepted by each port, see “Data Type Support” on page 2-1569 or “Supported Data Types” on page 2-1572.

### Data Type Support

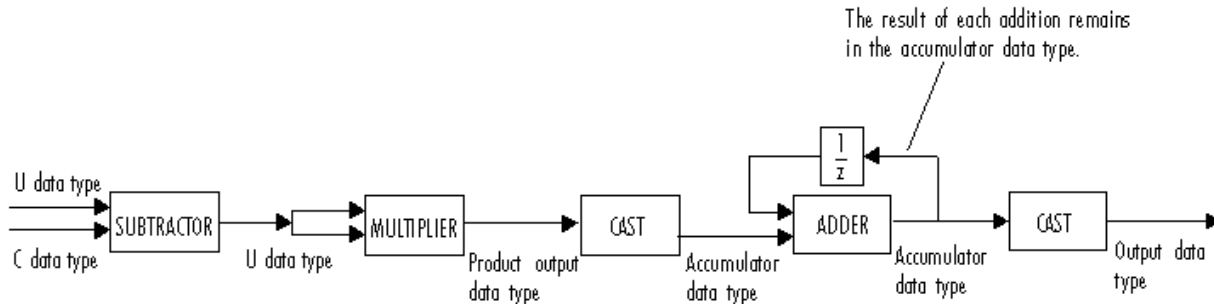
The input data values, codebook values, and weighting factor values are input to the block at ports U, C, and W, respectively. The data type of the input data values, codebook values, and weighting factor values can be `double`, `single`, or `Fixed-point`. The input data, codebook values, and weighting factor must be the same data type.

The outputs of the block are the index values, output codewords, and quantization error. Use the **Index output data type** parameter to specify the data type of the index output from the block at

port I. The data type of the index can be `int8`, `uint8`, `int16`, `uint16`, `int32`, or `uint32`. The data type of the output codewords and the quantization error can be `double`, `single`, or `Fixed-point`. The block assigns the data type of the output codewords and the quantization error based on the data type of the input data.

### Fixed-Point Data Types

The following diagram shows the data types used within the Vector Quantizer Encoder block for fixed-point signals.



You can set the product output, accumulator, and index output data types in the block dialog as discussed below.

## Dialog Box

### Main Tab

#### Source of codebook

Choose `Specify via dialog` to type the codebook values into the block parameters dialog box. Select `Input port` to specify the codebook values using the block's input port, `C`.

#### Codebook

Enter a  $k$ -by- $N$  matrix of values, where  $1 \leq k$  and  $1 \leq N$ , to which your input column vector or matrix is compared. This parameter is visible if, from the **Source of codebook** list, you select `Specify via dialog`.

#### Distortion measure

Select `Squared error` when you want the block to calculate the distortion by evaluating the Euclidean distance between the input column vector and each codeword in the codebook. Select `Weighted squared error` when you want the block to calculate the distortion by evaluating a weighted Euclidean distance using a weighting factor to emphasize or deemphasize certain input values.

#### Source of weighting factor

Select `Specify via dialog` to enter a value for the weighting factor in the dialog box. Choose `Input port` and specify the weighting factor using port `W` on the block. This parameter is visible if, for the **Distortion measure** parameter, you select `Weighted squared error`.

#### Weighting factor

Enter a vector of values. This vector must have length equal to the number of rows of the input, `U`. This parameter is visible if, for the **Source of weighting factor** parameter, you select `Specify via dialog`.

**Tie-breaking rule**

Set this parameter to determine the behavior of the block when an input column vector is equidistant from two codewords. When you want the input column vector to be represented by the lower index valued codeword, select `Choose the lower index`. To represent the input column vector by the higher index valued codeword, select `Choose the higher index`.

**Output codeword**

Select this check box to output the codeword vectors nearest to the input column vectors.

**Output quantization error**

Select this check box to output the quantization error value that results when the block represents the input column vector by the nearest codeword.

**Index output data type**

Select `int8`, `uint8`, `int16`, `uint16`, `int32`, or `uint32` as the data type of the index output at port I. To inherit the index output data type, select `Inherit via back propagation`.

**Data Types Tab****Rounding mode**

Specify the rounding mode for fixed-point operations as one of the following:

- Floor
- Ceiling
- Convergent
- Nearest
- Round
- Simplest
- Zero

For more details, see rounding mode.

**Overflow mode**

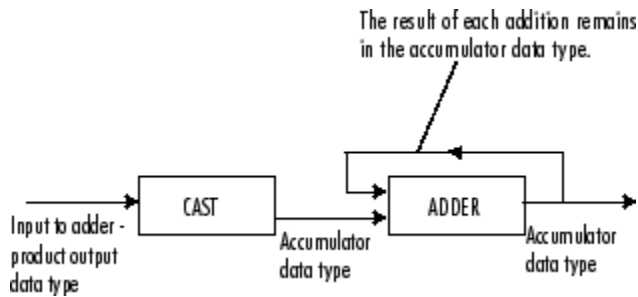
When you select this parameter, the block saturates the result of its fixed-point operation. When you clear this parameter, the block wraps the result of its fixed-point operation. For details on saturate and wrap, see overflow mode for fixed-point operations.

**Product output**

As depicted above, the output of the multiplier is placed into the product output data type and scaling. Use this parameter to specify how you would like to designate this product output word and fraction lengths.

- When you select `Same as input`, these characteristics match those of the input to the block.
- When you select `Binary point scaling`, you can enter the word length and the fraction length of the product output, in bits.

## Accumulator



As depicted above, inputs to the accumulator are cast to the accumulator data type. The output of the adder remains in the accumulator data type as each element of the input is added to it. Use this parameter to specify how you would like to designate the accumulator word and fraction lengths.

- When you select `Same as product output`, these characteristics match those of the product output.
- When you select `Same as input`, these characteristics match those of the input to the block.
- When you select `Binary point scaling`, you can enter the word length and the fraction length of the accumulator, in bits.

## References

Gersho, A. and R. Gray. *Vector Quantization and Signal Compression*. Boston: Kluwer Academic Publishers, 1992.

## Supported Data Types

| Port | Supported Data Types   |
|------|--|
| U    | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed only)</li> <li>• 8-, 16-, and 32-bit signed integers</li> </ul> |
| C    | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed only)</li> <li>• 8-, 16-, and 32-bit signed integers</li> </ul> |
| W    | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed only)</li> <li>• 8-, 16-, and 32-bit signed integers</li> </ul> |
| I    | <ul style="list-style-type: none"> <li>• 8-, 16-, and 32-bit signed integers</li> <li>• 8-, 16-, and 32-bit unsigned integers</li> </ul>   |

| Port | Supported Data Types   |
|------|--|
| Q(U) | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed only)</li> <li>• 8-, 16-, and 32-bit signed integers</li> </ul> |
| D    | <ul style="list-style-type: none"> <li>• Double-precision floating point</li> <li>• Single-precision floating point</li> <li>• Fixed point (signed only)</li> <li>• 8-, 16-, and 32-bit signed integers</li> </ul> |

## See Also

|                          |                    |
|--------------------------|--------------------|
| Quantizer                | Simulink           |
| Scalar Quantizer Decoder | DSP System Toolbox |
| Uniform Encoder          | DSP System Toolbox |
| Uniform Decoder          | DSP System Toolbox |
| Vector Quantizer Decoder | DSP System Toolbox |

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Generated code relies on the `memcpy` or `memset` function (`string.h`) under certain conditions.

### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

If the input is fixed point, it must be a signed integer or a signed fixed point value with a power-of-two slope and zero bias.

# Waterfall

(To be removed) View vectors of data over time

---

**Note** The Waterfall block will be removed in a future release. Use a different scope instead, see “Scopes and Data Logging”.

---



**Libraries:**

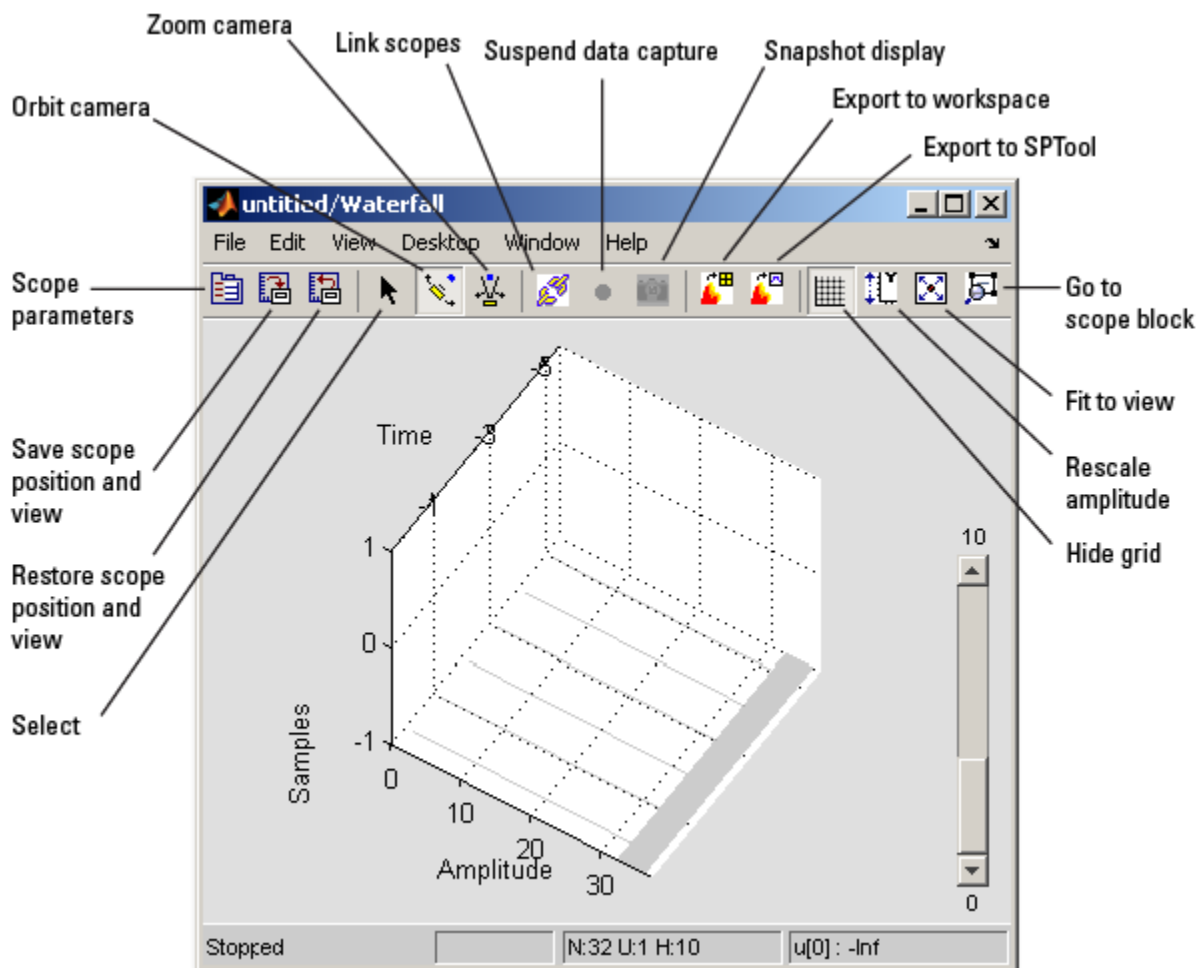
DSP System Toolbox / Obsolete Blocks

## Description

The Waterfall block displays multiple vectors of data at one time. These vectors represent the input data at consecutive sample times.

The data is displayed in 3D in the Waterfall window. By default, the  $x$ -axis represents amplitude, the  $y$ -axis represents samples, and the  $z$ -axis represents time. You can adjust the number of sample vectors that the block displays, move and resize the Waterfall window, and modify block parameter values during the simulation. The Waterfall window has toolbar buttons that enable you to zoom in on the displayed data, suspend the data capture, freeze the scope display, save the scope position, and export data to the workspace.

The toolbar buttons are labeled on the Waterfall window as it appears when you double-click a Waterfall block.



## Ports

### Input

**Port\_1** — Signal or signals to visualize  
 scalar | vector | matrix | array

The Waterfall block accepts most numeric data types as input. However, the input is converted to double-precision before the block processes the data. The Waterfall block displays only real-valued, double-precision vectors of data.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

### Parameters

You can control the display and behavior of the Waterfall window using the Parameters dialog box. To open the Parameters dialog box, click the Scope parameters button.

You can alter the Waterfall parameters while the simulation is running. However, when you make changes to values in text boxes, you must click **Enter** or click outside the text box for the block to accept your changes.

## Display

These parameters control the Waterfall display.

**Display traces** — Number of data vectors to display  
6 (default) | positive integer

Number of data vectors to display in the Waterfall window, specified as a positive integer.

### Programmatic Use

Control this parameter from the command-line using `set_param` and `get_param`.

**Block Parameter:** NumTraces

**Type:** positive scalar specified as a character vector

**Update interval** — Number of vectors to store  
1 (default) | positive integer

Number of vectors the block should store before it displays them to the window, specified as a positive integer.

### Programmatic Use

Control this parameter from the command-line using `set_param` and `get_param`.

**Block Parameter:** UpdateInterval

**Type:** positive scalar specified as a character vector

**Colormap** — Colors of display  
autumn (default) | gray | hot | pink | jet | copper | red | white

Colors for the displayed data, specified as a valid colormap.

### Programmatic Use

Control this parameter from the command-line using `set_param` and `get_param`.

**Block Parameter:** CMapStr

**Type:** character vector

**Type:** valid colormap name, see `map`

**Transparency** — Transparency of oldest and newest data vectors  
newest: opaque, oldest: transparent (default)

Transparency of the oldest and newest data vectors, specified with the sliders. The leftmost position makes the data vector transparent. The rightmost position makes the data vector opaque. The intermediate data vectors transition between the two chosen transparency values.

### Programmatic Use

Control this parameter from the command-line using `set_param` and `get_param`.

**Block Parameter:** TNewest and TOldest

**Type:** number between [0,1] specified as a character vector

## Axes

These parameters control the axes in the Waterfall window.



**Y Min** — Minimum value of y-axis

-1 (default) | integer

Minimum value of the y-axis specified as an integer.

**Programmatic Use**

Control this parameter from the command-line using `set_param` and `get_param`.

**Block Parameter:** YMin

**Type:** character vector

**Y Max** — Maximum value of y-axis

1 (default) | integer

Maximum value of the y-axis specified as an integer

**Programmatic Use**

Control this parameter from the command-line using `set_param` and `get_param`.

**Block Parameter:** YMax

**Type:** character vector

**Axis color** — Color of axes



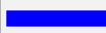

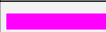
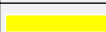


'w' (default) | RGB triplet | specified color name | hexadecimal color code

Color of the axes, specified as a RGB triplet, specified color name, or hexadecimal color code.








RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range  $[0, 1]$ ; for example,  $[0.4 \ 0.6 \ 0.7]$ .
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from 0 to F. The values are not case sensitive. Thus, the color codes '#FF8800', '#ff8800', '#F80', and '#f80' are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

| Color Name | Short Name | RGB Triplet | Hexadecimal Color Code | Appearance  |
|------------|------------|-------------|------------------------|---|
| "red"      | "r"        | [1 0 0]     | "#FF0000"              |  |
| "green"    | "g"        | [0 1 0]     | "#00FF00"              |  |
| "blue"     | "b"        | [0 0 1]     | "#0000FF"              |  |
| "cyan"     | "c"        | [0 1 1]     | "#00FFFF"              |  |
| "magenta"  | "m"        | [1 0 1]     | "#FF00FF"              |  |
| "yellow"   | "y"        | [1 1 0]     | "#FFFF00"              |  |
| "black"    | "k"        | [0 0 0]     | "#000000"              |  |
| "white"    | "w"        | [1 1 1]     | "#FFFFFF"              |  |

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

| RGB Triplet            | Hexadecimal Color Code | Appearance  |
|------------------------|------------------------|---|
| [0 0.4470 0.7410]      | "#0072BD"              |  |
| [0.8500 0.3250 0.0980] | "#D95319"              |  |
| [0.9290 0.6940 0.1250] | "#EDB120"              |  |
| [0.4940 0.1840 0.5560] | "#7E2F8E"              |  |
| [0.4660 0.6740 0.1880] | "#77AC30"              |  |
| [0.3010 0.7450 0.9330] | "#4DBEEE"              |  |
| [0.6350 0.0780 0.1840] | "#A2142F"              |  |

### Programmatic Use

Control this parameter from the command-line using `set_param` and `get_param`.

**Block Parameter:** `AxisColor`

**Type:** character vector within a character vector

**Example:** `set_param(block, 'AxisColor', char('b'))`

**X Axis** — x-axis label

`Samples (default) | string`

Enter the x-axis label.

### Programmatic Use

Control this parameter from the command-line using `set_param` and `get_param`.

**Block Parameter:** `XLabel`

**Type:** character vector

**Y Axis** — y-axis label

`Amplitude (default) | string`

Enter the y-axis label.

### Programmatic Use

Control this parameter from the command-line using `set_param` and `get_param`.

**Block Parameter:** `YLabel`

**Type:** character vector

**Z Axis** — z-axis label

`Time (default) | string`

Enter the z-axis label.

### Programmatic Use

Control this parameter from the command-line using `set_param` and `get_param`.

**Block Parameter:** `ZLabel`

**Type:** character vector

### Data History

These parameters control how many input data vectors the Waterfall block stores. They also control how the data is exported to the MATLAB workspace.

**History traces** — Number of vectors to store  
10 (default) | positive integer

Number of vectors (traces) that you want the block to store, specified as a positive integer.

**Programmatic Use**

Control this parameter from the command-line using `set_param` and `get_param`.

**Block Parameter:** `HistoryLength`

**Type:** positive integer specified as a character vector

**When the buffer is full** — Behavior of full buffer  
`Overwrite` (default) | `Suspend` | `Extend`

Use this parameter to control the behavior of the block when the buffer is full:

- `Overwrite` — The old data is replaced with the new data.
- `Suspend` — The block stops storing data in the buffer; but the simulation continues to run.
- `Extend` — The block extends the buffer so that it can continue to store all the input data.

**Programmatic Use**

Control this parameter from the command-line using `set_param` and `get_param`.

**Block Parameter:** `HistoryFull`

**Type:** character vector

**Data logging** — Export data from block  
`Selected` (default) | `All visible` | `All history`

Use this parameter to control which data is exported from the block:

- `Selected` — The selected data vector is exported.
- `All visible` — All of the data vectors displayed in the Waterfall window are exported.
- `All history` — All of the data vectors stored in the history buffer of the block are exported.

**Programmatic Use**

Control this parameter from the command-line using `set_param` and `get_param`.

**Block Parameter:** `ExportMode`

**Type:** character vector

**Export variable** — Variable name of exported data  
`ExportData` (default) | valid MATLAB variable name

Name of the variable that represents your data in the MATLAB workspace.

**Programmatic Use**

Control this parameter from the command-line using `set_param` and `get_param`.

**Block Parameter:** `MLExportName`

**Type:** character vector

**Export at end of simulation** — Automatically export data  
`off` (default) | `on`

Select this check box to automatically export the data to the MATLAB workspace when the simulation stops.

### Programmatic Use

Control this parameter from the command-line using `set_param` and `get_param`.

**Block Parameter:** `AutoExport`

**Values:** 'off' | 'on'

### Triggering

These parameters control when the Waterfall block starts and stops capturing data.

#### Begin recording — When to start capturing data

Immediately (default) | After T seconds | After N inputs | User-defined

This parameter controls when the Waterfall block starts capturing data:

- **Immediately** — The Waterfall window captures the input data as soon as the simulation starts.
- **After T seconds** — The **Time, T** parameter appears in the dialog box. Enter the number of seconds the block should wait before it begins capturing data.
- **After N inputs** — The **Count, N** parameter appears in the dialog box. Enter the number of inputs the block should receive before it begins capturing data.
- **User-defined** — The **Function name** parameter appears in the dialog box. Enter the name of a MATLAB function that defines when the block should begin capturing data. For more information about how you define this function, see “Scope Trigger Function”.

### Programmatic Use

Control this parameter from the command-line using `set_param` and `get_param`.

**Block Parameter:** `TrigStartMode` | `TrigStartT` | `TrigStartN` | `TrigStartFcn`

**Type:** character vector

#### Stop recording — When to stop capturing data

Never (default) | After T seconds | After N inputs | User-defined

This parameter controls when the Waterfall block stops capturing data:

- **Never** — The block captures the input data as long as the simulation is running.
- **After T seconds** — The **Time, T** parameter appears in the dialog box. Enter the number of seconds the block should wait before it stops capturing data.
- **After N inputs** — The **Count, N** parameter appears in the dialog box. Enter the number of inputs the block should receive before it stops capturing data.
- **User-defined** — The **Function name** parameter appears in the dialog box. Enter the name of a MATLAB function that defines when the block should stop capturing data. For more information about how you define this function, see “Scope Trigger Function”.

### Programmatic Use

Control this parameter from the command-line using `set_param` and `get_param`.

**Block Parameter:** `TrigStopMode` | `TrigStopT` | `TrigStopN` | `TrigStopFcn`

**Type:** character vector

**Re-arm trigger** — When to capture data

Never (default) | After T seconds | After N inputs | User-defined

This parameter controls when the Waterfall block begins waiting to capture data.

- **Never** — The Waterfall Scope block starts and stops capturing data as defined by the **Begin recording** and **Stop recording** parameters.
- **After T seconds** — The **Time, T** parameter appears in the dialog box. Enter the number of seconds the block should wait before it begins waiting to capture data.
- **After N inputs** — The **Count, N** parameter appears in the dialog box. Enter the number of inputs the block should receive before it begins waiting to capture data.
- **User-defined** — The **Function name** parameter appears in the dialog box. Enter the name of a MATLAB function that defines when the block should begin waiting to capture data. For more information about how you define this function, see “Scope Trigger Function”.

**Dependency**

To enable this parameter, set **Stop recording** to After T seconds, After N inputs, or User-defined.

**Programmatic Use**

Control this parameter from the command-line using `set_param` and `get_param`.

**Block Parameter:** TrigRearmMode | TrigRearmT | TrigRearmN | TrigRearmFcn

**Type:** character vector

**Transform**

These parameters transform the input data to the Waterfall block. The result of the transform is displayed in the Waterfall window.

The block assumes that the input to the block corresponds to the **Transform** parameter you select. For example, when you choose **Complex-> Angle**, the block assumes that the input is complex. But the block does not produce an error when the input is not complex. Therefore, you must verify the format of your input data to guarantee that a meaningful result is displayed in the Waterfall window.

**Transform** — Transformation to apply to input

None (default) | Amplitude-&gt; dB | Complex-&gt; Mag Lin | Complex-&gt; Mag dB | ...

Choose a transform that you would like to apply to the input of the Waterfall block:

- **None** — The input is displayed as it is received by the block.
- **Amplitude-> dB** — The block converts the input amplitude into decibels.
- **Complex-> Mag Lin** — The block converts the complex input into linear magnitude.
- **Complex-> Mag dB** — The block converts the complex input into magnitude in decibels.
- **Complex-> Angle** — The block converts the complex input into phase.
- **FFT-> Mag Lin Fs/2** — The block takes the linear magnitude of the FFT input and plots it from 0 to the Nyquist frequency.
- **FFT-> Mag dB Fs/2** — The block takes the magnitude of the FFT input, converts it to decibels, and plots it from 0 to the Nyquist frequency.
- **FFT-> Angle Fs/2** — The block converts the FFT input into phase and plots it from 0 to the Nyquist frequency.

- Power -> dB — The block converts the input power into decibels.
- User-defined fcn — see “Function” on page 2-0 .
- User-defined expr — see “Expression” on page 2-0 .

### Programmatic Use

Control this parameter from the command-line using `set_param` and `get_param`.

**Block Parameter:** XFormMode

**Type:** character vector

**Function** — Custom transformation function

abs (default)

Enter a function that you would like to apply to the input of the Waterfall block. For more information about how you define this function, see “Scope Transform Function”.

### Dependency

To enable this parameter, set **Transform** to User-defined fcn.

### Programmatic Use

Control this parameter from the command-line using `set_param` and `get_param`.

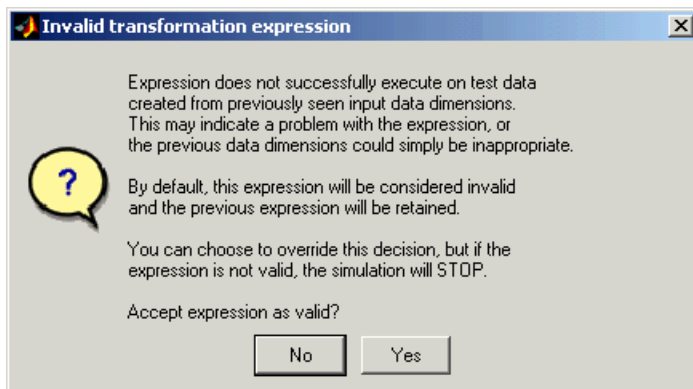
**Block Parameter:** XFormFcn

**Type:** character vector

**Expression** — Custom transformation expression

(u+1)/2 (default) | expression with real-valued output

Enter an expression that you would like to apply to the input of the Waterfall block. The result of this expression must be real-valued. When you write the expression, be sure to include only one unknown variable. The block assumes this unknown variable represents the input to the block. When the block believes your expression is invalid, this window appears.



When you click **No**, your expression is not applied to the input. When you click **Yes** and your expression is invalid, your simulation stops and Simulink displays an error.

### Dependency

To enable this parameter, set **Transform** to User-defined expr.

**Programmatic Use**

Control this parameter from the command-line using `set_param` and `get_param`.

**Block Parameter:** XFormExpr

**Type:** character vector

**Block Characteristics**

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | double   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no                                      |
| <b>Multidimensional Signals</b> | no                                      |
| <b>Variable-Size Signals</b>    | no                                      |
| <b>Zero-Crossing Detection</b>  | no                                      |

**Version History**

**Introduced before R2006a**

**R2020b: Waterfall block will be removed**

*Not recommended starting in R2020b*

The Waterfall block will be removed in a future release. Use a different scope instead, see “Scopes and Data Logging”.

**Extended Capabilities****C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

This block can be used for simulation visibility in systems that generate code, but is not included in the generated code.

**HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

This block can be used for simulation visibility in subsystems that generate HDL code, but is not included in the hardware implementation.

**PLC Code Generation**

Generate Structured Text code using Simulink® PLC Coder™.

This block can be used for simulation visibility in systems that generate code, but is not included in the generated code.

**Fixed-Point Conversion**

Design and simulate fixed-point systems using Fixed-Point Designer™.

This block accepts fixed-point input, but converts it to `double` for display.

**See Also**

Scope | Time Scope | Spectrum Analyzer



# Wavelet Scattering

Model wavelet scattering network in Simulink



## Libraries:

DSP System Toolbox / Feature Extractors

## Description

The Wavelet Scattering block creates a framework for wavelet time scattering in the Simulink environment. Use this block to derive low-variance features from real-valued data and then use those features in machine learning and deep learning applications. The block uses predefined wavelet filters to compute the scalogram and applies an averaging filter to the scalogram for feature extraction. For more information, see “Wavelet Scattering” (Wavelet Toolbox). To perform wavelet scattering in MATLAB, use the `waveletScattering` function.

The Wavelet Scattering block requires Wavelet Toolbox.

## Ports

### Input

**Input** — Input signal

vector | matrix

Specify the input signal as a vector or a matrix. The block treats each column in the matrix as a separate channel. For example, the block treats a 1-by- $N_{chann}$  row vector as  $N_{chann}$  separate channels.

When the input data type is `single`, the block performs internal operations in single-precision. The input data type determines the precision of the scattering decomposition in the filter banks.

The input frame length does not have to match the signal length specified in the block dialog box. The block uses a buffer to feed the wavelet scattering algorithm with frames of length **Signal length (samples)**. If the input frame size does not match the signal length, the block runs in multirate mode.

Data Types: `single` | `double`

### Output

**Output** — Output signal

matrix | 3-D array

Output signal, returned as a matrix or a 3-D array.

For a column vector input, the output dimensions are  $N_{path}$ -by- $N_{scat}$ , where  $N_{path}$  is the number of scattering paths and  $N_{scat}$  is the number of scattering coefficients in each path, or the resolution of the scattering coefficients.

For matrix inputs, the output dimensions are  $N_{path}$ -by- $N_{scat}$ -by- $N_{chann}$ , where  $N_{chann}$  is the number of columns in the input.

Data Types: `single` | `double`

## Parameters

**Signal length (samples)** — Signal length in samples  
1024 (default) | positive integer  $\geq 16$

Specify the signal length in samples as a positive integer  $\geq 16$ .

The input frame length does not have to match the signal length. The block uses a buffer to feed the wavelet scattering algorithm with frames of length **Signal length**. If the input frame size does not match the signal length, the block runs in multirate mode.

Data Types: `double`

**Overlap length (Samples)** — Overlap length in samples  
0 (default) | positive integer

Specify the overlap (in samples) between consecutive frames of a signal as a nonnegative integer.

Data Types: `double`

**Specify sample rate** — Specify sample rate  
off (default) | on

Specify the sample rate on the block dialog using the **Sample rate (Hz)** parameter. If you do not select this check box, the frequencies are in cycles/sample and the Nyquist frequency is  $\frac{1}{2}$ .

Data Types: `Boolean`

**Sample rate (Hz)** — Sample rate  
16e3 (default) | positive scalar

Specify the sample rate in Hz as a positive scalar.

## Dependencies

To enable this parameter, select the **Specify sample rate** parameter.

Data Types: `double`

**Invariance scale (samples)** — Scattering transform invariance scale in samples  
512 (default) | positive integer

Specify the scattering transform invariance scale in samples as a positive integer. Invariance scale specifies the translation invariance of the scattering transform.

If you do not specify the sample rate, invariance scale is measured in samples. If you specify the sample rate, invariance scale is measured in seconds.

Invariance scale cannot exceed the signal length. If signal length is 1000 samples and you do not specify a sample rate, the maximum value of the invariance scale is 1000 samples.

**Dependencies**

To enable this parameter, clear the **Specify sample rate** parameter.

Data Types: double

**Invariance scale (s)** — Scattering transform invariance scale in seconds  
0.032 (default) | positive scalar

Specify the scattering transform invariance scale in samples as a positive scalar. Invariance scale specifies the translation invariance of the scattering transform.

If you do not specify the sample rate, invariance scale is measured in samples. If you specify the sample rate, invariance scale is measured in seconds. By default, invariance scale is one-half the signal length.

Invariance scale cannot exceed the signal length. If signal length is 1000 samples and you specify a sample rate of 200 Hz, then the maximum value of the invariance scale is 5.

**Dependencies**

This parameter appears only when you select the **Specify sample rate** parameter.

Data Types: double

**Quality factors** — Quality factors for scattering filter banks  
[8,1] (default) | positive integer | vector of monotonically decreasing positive integers

Specify the quality factors for the scattering filter banks as a positive integer or a vector of monotonically decreasing positive integers. A filter bank quality factor is the number of wavelet filters per octave. Quality factors cannot exceed 32 and must be greater than or equal to 1.

By default, the wavelet scattering network the block creates has two filter banks. The first filter bank has a quality factor of eight wavelets per octave and the second filter bank has a quality factor of 1 wavelet per octave.

Data Types: double

**Reduce the number of scattering paths** — Optimize scattering transform  
off (default) | on

Specify whether to optimize the scattering transform and reduce the number of scattering paths.

When you select this parameter, the algorithm reduces the number of scattering paths to compute based on a bandwidth consideration. The scattering transform excludes scattering paths of order 2 and greater that do not satisfy this criterion:

The center frequency minus  $\frac{1}{2}$  the 3-dB bandwidth of the wavelet filter in the  $(i+1)^{\text{th}}$  filter bank must overlap 0 (DC) plus  $\frac{1}{2}$  the 3-dB bandwidth of the wavelet filter in the  $i^{\text{th}}$  filter bank.

If this criterion is not satisfied, the higher-order path is excluded. Selecting the **Reduce the number of scattering paths** parameter can significantly reduce the number of scattering paths and computational complexity of the scattering transform for most networks.

Data Types: Boolean

**Oversampling factor** — Oversampling factor  
0 (default) | nonnegative integer | Inf

Specify the oversampling factor as a nonnegative integer or Inf.

The oversampling factor specifies the factor by which the number of scattering coefficients per signal increase on a  $\log_2$  scale. By default, the **Oversampling factor** is set to 0, which corresponds to critically downsampling the coefficients. To obtain a fully undecimated scattering transform, set **Oversampling factor** to Inf.

Setting **Oversampling factor** to a value that results in more coefficients than samples is equivalent to setting **Oversampling factor** to Inf. Increasing the **Oversampling factor** significantly increases the computational complexity and memory requirements of the scattering transform.

If you set **Oversampling factor** to 2, the scattering transform returns  $2^2$  times as many coefficients for each scattering path with respect to the critically sampled number.

Data Types: double

**Boundary signal extension** — Signal extension method  
periodic (default) | reflection

Specify the signal extension method to apply at the boundary as:

- **periodic** — Extend signal periodically to length  $2^{\text{ceil}(\log_2(N))}$ , where  $N$  is the signal length.
- **reflection** — Extend signal by reflection to length  $2^{\text{ceil}(\log_2(2N))}$ , where  $N$  is the signal length.

The signal is extended to match the length of the wavelet filters. The length of the filters are powers of two.

The signal extension method is for internal operations. Results are downsampled back onto the scale of the original signal before being returned.

### Scattering coefficients

**Include zeroth-order coefficients** — Include zeroth-order coefficients  
off (default) | on

Specify whether to include the zeroth-order coefficients.

Data Types: Boolean

**Normalization** — Coefficient normalization  
none (default) | parent

Specify the type of normalization to apply to the scattering coefficients as **none** or **parent**. If specified as **parent**, scattering coefficients of order greater than 0 are normalized by their parents along the scattering path.

**Transform** — Transformation on coefficients  
none (default) | log

Specify the type of transformation to apply to the scattering coefficients as none or log.

## Block Characteristics

|                                 |                 |
|---------------------------------|-----------------|
| <b>Data Types</b>               | double   single |
| <b>Direct Feedthrough</b>       | no              |
| <b>Multidimensional Signals</b> | no              |
| <b>Variable-Size Signals</b>    | no              |
| <b>Zero-Crossing Detection</b>  | no              |

## Version History

Introduced in R2022b

## References

- [1] Andén, Joakim, and Stéphane Mallat. "Deep Scattering Spectrum." *IEEE Transactions on Signal Processing* 62, no. 16 (August 2014): 4114–28. <https://doi.org/10.1109/TSP.2014.2326991>.
- [2] Mallat, Stéphane. "Group Invariant Scattering." *Communications on Pure and Applied Mathematics* 65, no. 10 (October 2012): 1331–98. <https://doi.org/10.1002/cpa.21413>.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block does not support code generation if the dynamic memory allocation is off. To turn on the dynamic memory allocation, run this command in the MATLAB command prompt.

```
set_param(gcs, 'MATLABDynamicMemAlloc', 'on')
```

## See Also

### Functions

waveletScattering

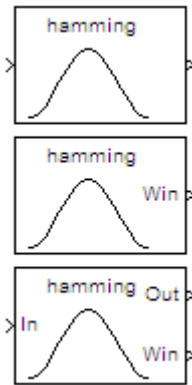
### Topics

"Wavelet Scattering" (Wavelet Toolbox)

"Fault Detection Using Wavelet Scattering and Recurrent Deep Networks" (Wavelet Toolbox)

## Window Function

Compute and apply window to input signal



### Libraries:

DSP System Toolbox / Signal Operations

## Description

The Window Function block has three modes of operation that enable you to apply a window, or compute a window, or compute and apply a window to an input signal. You can select the mode via the **Operation** parameter. In each mode, the block first creates a window vector  $w$  by sampling the window specified in the **Window type** parameter.

## Ports

### Input

**In** — Input signal

vector | matrix |  $N$ -D array

Input signal, specified as a vector, matrix, or an  $N$ -D array. When the input is fixed point, it can be signed only.

This port is enabled only when you set **Operation** to either `Apply window to input` or `Generate and apply window`.

This port is unnamed when you set **Operation** to `Apply window to input`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `fixed point`

### Output

**Out** — Output

vector | matrix |  $N$ -D array

Output signal, returned as a vector, matrix, or an  $N$ -D array.

The **Out** port is enabled only when **Operation** is set to:

- **Apply window to input** -- The block computes an  $M$ -by-1 window vector  $w$  and applies it to the  $M$ -by- $N$  input, where  $M$  is the number of rows in the input signal.
- **Generate and apply window** -- The block generates an  $M$ -by-1 window vector  $w$  and applies it to the input.

The output signal has the same dimensions as the input signal. For more details, see “Operation” on page 2-0 .

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `fixed point`

**Win** — Window output vector

Window output, returned as a vector. When **Operation** is set to **Apply window to input** or **Generate and apply window**, the window vector is of the same length as the number of rows in the input signal. When **Operation** is set to **Generate window**, the length of window vector is determined by the **Window length** parameter.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `fixed point`

## Parameters

### Main

**Operation** — Modes of operation

`Apply window to input (default)` | `Generate window` | `Generate and apply window`

The Window Function block has three modes of operation that you can select through the **Operation** parameter. In each mode, the block first creates a window vector  $w$  by sampling the window specified in the **Window type** parameter at  $M$  discrete points. The operation modes are:

- **Apply window to input**

In this mode, the block computes an  $M$ -by-1 window vector  $w$  and applies it to the input. The output  $y$  always has the same dimension as the input.

When the input is an  $M$ -by- $N$  matrix  $u$ , the window is multiplied element-wise with each of the  $N$  channels in the input matrix  $u$ . This is equivalent to the following MATLAB code:

```
y = repmat(w,1,N) .* u % Equivalent MATLAB code
```

The window is always applied to the first dimension:

$$y(i, j, \dots, k) = w(i) * u(i, j, \dots, k) \quad i = 1, \dots, M, \quad j = 1, \dots, N, \quad \dots, \quad k = 1, \dots, P$$

A length- $M$  unoriented vector input is treated as an  $M$ -by-1 vector.

- **Generate window**

In this mode, the block generates an unoriented window vector  $w$  with length  $M$  specified by the **Window length** parameter. The **In** port is disabled for this mode.

- **Generate and apply window**

In this mode, the block generates an  $M$ -by-1 window vector  $w$  and applies it to the input. The block produces two outputs:

- At the **Out** port, the block produces the result of the multiplication  $y$ , which has the same dimension as the input.
- At the **Win** port, the block produces the  $M$ -by-1 window vector  $w$ .

When the input is an  $M$ -by- $N$  matrix  $u$ , the window is multiplied element-wise with each of the  $N$  channels in the input matrix  $u$ . This is equivalent to the following MATLAB code:

```
y = repmat(w,1,N) .* u % Equivalent MATLAB code
```

The window is always applied to the first dimension:

$$y(i, j, \dots, k) = w(i) * u(i, j, \dots, k) \quad i = 1, \dots, M, \quad j = 1, \dots, N, \quad \dots, \quad k = 1, \dots, P$$

A length- $M$  1-D vector input is treated as an  $M$ -by-1 vector.

#### Window type — Type of window

Hamming (default) | Bartlett | Blackman | ...

This table lists the available window types.

| Window Type | Description  |
|-------------|--|
| Bartlett    | Computes a Bartlett window.<br>$w = \text{bartlett}(M)$  |
| Blackman    | Computes a Blackman window.<br>$w = \text{blackman}(M)$  |
| Boxcar      | Computes a rectangular window.<br>$w = \text{rectwin}(M)$  |
| Chebyshev   | Computes a Chebyshev window with stopband ripple $R$ .<br>$w = \text{chebwin}(M, R)$   |
| Hamming     | Computes a Hamming window.<br>$w = \text{hamming}(M)$  |
| Hann        | Computes a Hann window (also known as a Hanning window).<br>$w = \text{hann}(M)$   |
| Hanning     | Obsolete. This window type is included only for compatibility with older models. Use the Hann <b>Window type</b> instead of Hanning whenever possible. |
| Kaiser      | Computes a Kaiser window with the Kaiser parameter $\beta$ .<br>$w = \text{kaiser}(M, \beta)$  |
| Taylor      | Computes a Taylor window.<br>$w = \text{taylorwin}(M)$   |
| Triang      | Computes a triangular window.<br>$w = \text{triang}(M)$  |



| Window Type  | Description  |
|--------------|--|
| User Defined | <p>Computes the user-defined window function specified by the entry in the <b>Window function name</b> parameter, <code>usrwin</code>.</p> <pre>w = usrwin(M) % Window takes no extra parameters w = usrwin(M,x<sub>1</sub>,...,x<sub>n</sub>) % Window takes extra parameters {x<sub>1</sub> ... x<sub>n</sub>}</pre> |

**Sampling** — Type of sampling  
Symmetric (default) | Periodic

Specify the window sampling for generalized-cosine windows.

For the generalized-cosine windows (Blackman, Hamming, Hann, and Hanning), the **Sampling** parameter determines whether the window samples are computed in a periodic or a symmetric manner. For example, when **Sampling** is set to `Symmetric`, a Hamming window of length  $M$  is computed as:

```
w = hamming(M) % Symmetric (aperiodic) window
```

When **Sampling** is set to `Periodic`, the same window is computed as:

```
w = hamming(M+1) % Periodic (asymmetric) window
w = w(1:M)
```

Tunable (Simulink) in simulation only.

#### Dependencies

This parameter is visible only when you select Blackman, Hamming, Hann, or Hanning for the **Window type** parameter.

**Sample mode** — Sample mode  
Continuous (default) | Discrete

Specify the sample mode for the block, `Continuous` or `Discrete`, when it is in the `Generate window mode`. In the `Apply window to input` and `Generate and apply window modes`, the block inherits the sample mode from its driving block.

#### Dependencies

This parameter is visible only when you select `Generate window` for the **Operation** parameter.

**Sample time** — Sample time  
1 (default) | real scalar

Specify the sample time for the block when **Operation** is set to `Generate window` and **Sample mode** is set to `Discrete`. When **Operation** is set to `Apply window to input` and `Generate and apply window`, the block inherits the sample time from its driving block.

#### Dependencies

This parameter is visible only when you set **Sample mode** to `Discrete`.

**Window length** — Length of window  
64 (default) | positive integer

Specify the length of the window to apply.

**Dependencies**

This parameter is visible only when you select **Generate window** for the **Operation** parameter. Otherwise, the window vector length is computed to match the length of the first dimension of the input.

**Stopband attenuation in dB** — Stopband attenuation in dB  
50 (default) | nonnegative scalar

Specify the level of stopband attenuation  $R_s$  in decibels.

Tunable (Simulink) in simulation only.

**Dependencies**

This parameter is visible only when you select **Chebyshev** for the **Window type** parameter.

**Beta** — Kaiser window  $\beta$  parameter  
10 (default) | real scalar

Specify the **Kaiser** window  $\beta$  parameter. Increasing  $\beta$  widens the mainlobe and decreases the amplitude of the window sidelobes in the window's frequency magnitude response.

Tunable (Simulink) in simulation only.

**Dependencies**

This parameter is visible only when you select **Kaiser** for the **Window type** parameter.

**Number of sidelobes** — Number of sidelobes  
4 (default) | positive integer

Specify the number of sidelobes as a scalar integer value greater than zero.

**Dependencies**

This parameter is visible only when you select **Taylor** for the **Window type** parameter.

**Maximum sidelobe level relative to mainlobe (dB)** — Maximum sidelobe level relative to mainlobe  
-30 (default) | scalar less than or equal to zero

Specify in decibels the maximum sidelobe level relative to the mainlobe. This parameter must be a scalar less than or equal to zero. The default value of -30 produces sidelobes with peaks 30 dB down from the mainlobe peak.

**Dependencies**

This parameter is visible only when you select Taylor for the **Window type** parameter.

**Window function name** — Window function name

hamming (default) | character vector

Specify the name of the user-defined window function to be calculated by the block. The output window vector returned by the custom window function must be of data type **double**.

**Dependencies**

This parameter is visible only when you select User defined for the **Window type** parameter.

**Specify additional arguments to the hamming function** — Specify additional arguments to the window function

off (default) | on

Select to enable the **Cell array of additional arguments** parameter, when the user-defined window requires parameters other than the window length.

**Dependencies**

This parameter is visible only when you select User defined for the **Window type** parameter.

**Cell array of additional arguments** — Additional arguments

{'symmetric'} (default) | cell array

Specify the extra parameters required by the user-defined window function, besides the window length.

**Dependencies**

This parameter is only available when you select the **Specify additional arguments to the hamming function** parameter. The entry must be a cell array.

**Data Types**

When the **Operation** parameter on the **Main** tab is set to Generate window, the following parameters appear.

**Window data type** — Window data type

double (default) | single | Fixed-point | User-defined | Inherit via back propagation

Specify the window data type in one of the following ways:

- double
- single
- Fixed-point -- To specify the window data type and scaling in the **Signed**, **Word length**, **Set fraction length in output to**, and **Fraction length** parameters.
- User-defined -- To specify the window data type and scaling in the **User-defined data type**, **Set fraction length in output to**, and **Fraction length** parameters.

- **Inherit via back propagation** -- To set the window data type and scaling to match the following block.

**Signed** — Window output is signed  
on (default) | off

Select to output a signed fixed-point signal. Otherwise, the signal is unsigned.

**Dependencies**

This parameter appears only when you set **Window data type** to Fixed-point.

**Word length** — Word length  
16 (default) | positive integer in the range [2 128]

Specify the word length of the fixed-point window data type in bits.

**Dependencies**

This parameter is visible only when you set **Window data type** to Fixed-point.

**User-defined data type (e.g. sfix(16), float('single'))** — user-defined data type  
sfix(16) (default) | built-in data type | fixed-point data type

Specify any built-in or fixed-point data type. You can specify fixed-point data types using the `fixdt` function.

**Dependencies**

This parameter is only visible when you set **Window data type** to User-defined.

**Set fraction length in output to** — Scaling of fixed-point window data type  
Best precision (default) | User-defined

Specify the scaling of the fixed-point window data type by using either of these methods:

- Choose **Best precision** to have the window data type scaling automatically set such that the output signal has the best possible precision.
- Choose **User-defined** to specify the window data type scaling in the **Fraction length** parameter.

**Dependencies**

This parameter is visible only when you set **Window data type** to Fixed-point or User-defined, and when the specified window data type is a fixed-point data type.

**Fraction length** — User-defined fraction length  
15 (default) | integer

Specify the fraction length of the fixed-point window data type in bits.

**Dependencies**

This parameter is visible only when you set **Window data type** to Fixed-point or User-defined, and when you set **Set fraction length in output to** to User-defined.

When the **Operation** parameter on the **Main** tab is set to either Apply window to input or Generate and apply window, the following parameters appear.

**Rounding mode** — Rounding mode

Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Select the rounding mode for fixed-point operations.

The window vector  $w$  does not obey this parameter. It always rounds to Nearest.

---

**Note** The **Rounding mode** and **Overflow mode** settings have no effect on numerical results when both of these conditions exist:

- **Product output** is Inherit via internal rule
- **Output** is Same as product output

With these data type settings, the block is effectively operating in a full-precision mode.

---

**Overflow mode** — Overflow mode

Wrap (default) | Saturate

Select the overflow mode for fixed-point operations.

The window vector  $w$  does not obey this parameter. It is always saturated.

**Window** — Window data type

Same word length as input (default) | Specify word length | Binary point scaling | Slope and bias scaling

Choose how you specify the word length and fraction length of the window vector  $w$ .

When you select:

- **Same word length as input** -- The word length of the window vector elements is the same as the word length of the input. The fraction length is automatically set to the best precision possible.
- **Specify word length** -- You can enter the word length of the window vector elements in bits. The fraction length is automatically set to the best precision possible.
- **Binary point scaling** -- You can enter the word length and the fraction length of the window vector elements in bits.
- **Slope and bias scaling** -- You can enter the word length, in bits, and the slope of the window vector elements. This block requires power-of-two slope and a bias of zero.

The window vector does not obey the **Rounding mode** and **Overflow mode** parameters. It is always saturated and rounded to Nearest.

**Product output** — Product output data type

Inherit via internal rule (default) | Same as input | Binary point scaling | Slope and bias scaling

Use this parameter to specify how you want to designate the product output word and fraction lengths.

When you select:

- **Inherit via internal rule** -- The product output word length and fraction length are calculated automatically. For information on how the product output word and fraction lengths are calculated when an internal rule is used, see “Inherit via Internal Rule”.
- **Same as input** -- These characteristics match those of the input to the block.
- **Binary point scaling** -- You can enter the word length and the fraction length of the product output in bits.
- **Slope and bias scaling** -- You can enter the word length in bits and the slope of the product output. This block requires power-of-two slope and a bias of zero.

**Output** — Output data type

Same as product output (default) | Same as input | Binary point scaling | Slope and bias scaling

Choose how you specify the word length and fraction length of the output of the block.

When you select:

- **Same as product output** -- These characteristics match those of the product output.
- **Same as input** -- These characteristics match those of the input to the block.
- **Binary point scaling** -- You can enter the word length and the fraction length of the output in bits.
- **Slope and bias scaling** -- You can enter the word length in bits and the slope of the output. This block requires power-of-two slope and a bias of zero.

**Lock data type settings against changes by the fixed-point tools** — Prevent fixed-point tools from overriding data types

off (default) | on

Select this parameter to prevent the fixed-point tools from overriding the data types you specify in the block dialog box.

**Block Characteristics**

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | double   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no                                      |
| <b>Multidimensional Signals</b> | no                                      |
| <b>Variable-Size Signals</b>    | yes                                     |

|                                |    |
|--------------------------------|----|
| <b>Zero-Crossing Detection</b> | no |
|--------------------------------|----|

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Generated code relies on the `memcpy` or `memset` function (`string.h`) under certain conditions.

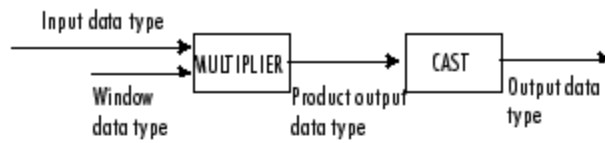
### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

If the input is fixed point, it must be a signed integer or a signed fixed point value with a power-of-two slope and zero bias.

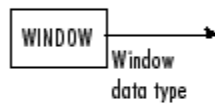
The following diagram shows the data types used within the Window Function block for fixed-point signals for each of the three operating modes.

**Apply window to input**



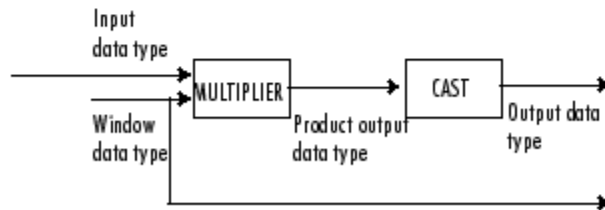
The input data type comes from the driving block. You can set the window, product output, and output data types in the block dialog. In this mode, the window vector is not output from the block.

**Generate window**



In this mode, the block acts as a source. The window vector is output in the window data type you specify in the block dialog.

**Generate and apply window**



The input data type comes from the driving block. You can set the window, product output, and output data types in the block dialog. In this mode, the window vector is output from the block.

You can set the window, product output, and output data types in the block dialog box.

**See Also**

**Functions**

bartlett | blackman | rectwin | chebwin | hamming | hann | kaiser | taylorwin | triang

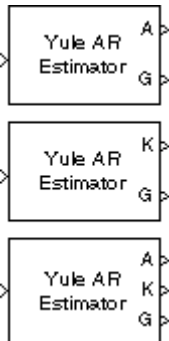
**Blocks**

FFT



# Yule-Walker AR Estimator

Compute estimate of autoregressive (AR) model parameters using Yule-Walker method



## Libraries:

DSP System Toolbox / Estimation / Parametric Estimation

## Description

The Yule-Walker AR Estimator block uses the Yule-Walker AR method, also called the autocorrelation method, to fit an autoregressive (AR) model to the windowed input data by minimizing the forward prediction error in the least squares sense. This process results in the Yule-Walker equations, which the block solves using the Levinson-Durbin recursion. The block outputs are always nonsingular.

## Ports

### Input

**Input** — Input  
vector | matrix

Specify the input data as a vector or a matrix. The block assumes the input data to be the output of an AR system driven by white noise.

When you specify a matrix, the block treats each column of the matrix as a channel. If the input is a row vector of length  $N$ , then the block treats the input as having  $N$  different channels. If the input is an unoriented vector, the block treats the input as a single channel.

### Dependency

To use an input that is a row vector, set **Output(s)** to A.

Data Types: single | double

### Output

**A** — Normalized estimate of the AR model polynomial coefficients  
vector | matrix

Normalized estimate of the AR model polynomial coefficients  $A(z)$ , returned as a vector or a matrix. If the output is a vector, then it is of length  $p+1$  in descending powers of  $z$ . If the output is a matrix, then each column of the matrix is of length  $p+1$  and contains the normalized estimate of AR model coefficients in descending powers of  $z$ .

The block computes the estimate of these coefficients independently for each successive input frame.

$$H(z) = \frac{\sqrt{G}}{A(z)} = \frac{\sqrt{G}}{1 + a(2)z^{-1} + \dots + a(p+1)z^{-p}}$$

where,

- $H(z)$  -- Transfer function of the estimated AR model
- $G$  -- Square of the model gain
- $A(z)$  -- Polynomial coefficients of the AR model

#### Dependency

To enable this port, set the **Output(s)** parameter to A or A and K.

Data Types: single | double

#### **K** — Reflection coefficients

column vector | unoriented vector | matrix

Reflection coefficients (which are a secondary result of the Levinson recursion), returned as a column vector, an unoriented vector of length  $p$ , or a matrix of size  $p$ -by- $N$ , where  $N$  is the number of input channels. For each input channel, port **K** outputs a length- $p$  column whose elements are the AR model reflection coefficients.

#### Dependency

To enable this port, set the **Output(s)** parameter to K or A and K.

To output reflection coefficients  $K$ , specify the input to be a row vector, unoriented vector, or a matrix.

Data Types: single | double

#### **G** — Square of model gain

scalar | vector

Square of the gain of the estimated AR model for each input channel, returned as a scalar or a vector of length  $N$ , where  $N$  is the number of input channels. The port **G** outputs a scalar for each channel.

Here is the equation for the transfer function of the estimated AR model which shows the gain and the polynomial coefficients of the model:

$$H(z) = \frac{\sqrt{G}}{A(z)} = \frac{\sqrt{G}}{1 + a(2)z^{-1} + \dots + a(p+1)z^{-p}}$$

where,

- $H(z)$  -- Transfer function of the estimated AR model
- $G$  -- Square of the model gain
- $A(z)$  -- Polynomial coefficients of the AR model

Data Types: single | double

## Parameters

**Output(s)** — Output of AR model coefficients

A (default) | A and K | K

Specify whether the block outputs model coefficients (A), reflection coefficients (K), or both (A and K).

**Inherit estimation order from input dimensions** — Inherit estimation order from input dimensions

on (default) | off

When you select the **Inherit estimation order from input dimensions** parameter, the block sets the order  $p$  of the all-pole model to a value that is one less than the length of each input channel. Otherwise, the order is the value you specify in the **Estimation order** parameter.

**Estimation order** — Order of AR model

4 (default) | nonnegative integer

Specify the order of the AR model  $p$  as a nonnegative integer.

To guarantee a valid output, you must set the **Estimation order** parameter to be a scalar less than or equal to half the input channel length.

### Dependencies

To enable this parameter, clear the **Inherit estimation order from input dimensions** parameter.

## Block Characteristics

|                                 |                 |
|---------------------------------|-----------------|
| <b>Data Types</b>               | double   single |
| <b>Multidimensional Signals</b> | No              |
| <b>Variable-Size Signals</b>    | No              |

## More About

### AR( $p$ ) Model

In an AR model of order  $p$ , the current output is a linear combination of the past  $p$  outputs plus a white noise input.

The weights on the  $p$  past outputs minimize the mean squared prediction error of the autoregression. If  $y(n)$  is the current value of the output and  $x(n)$  is a zero-mean white noise input, the AR( $p$ ) model is

$$\sum_{k=0}^p a(k)y(n-k) = x(n).$$

### Reflection Coefficients

The reflection coefficients are the partial autocorrelation coefficients scaled by -1.

The reflection coefficients indicate the time dependence between  $y(n)$  and  $y(n - k)$  after subtracting the prediction based on the intervening  $k - 1$  time steps.

### Compare AR Model Parameter Estimation Methods

This table compares the features of the Burg AR Estimator block to the Covariance AR Estimator, Modified Covariance AR Estimator, and the Yule-Walker AR Estimator blocks.

The Yule-Walker AR Estimator and Burg AR Estimator blocks return similar results for large frame sizes.

|                                      | <b>Burg AR Estimator</b>   | <b>Covariance AR Estimator</b>                                    | <b>Modified Covariance AR Estimator</b>   | <b>Yule-Walker AR Estimator</b>  |
|--------------------------------------|--|---|---|--|
| <b>Characteristics</b>               | Does not apply window to data  | Does not apply window to data                                     | Does not apply window to data   | Applies window to data   |
|                                      | Minimizes the forward and backward prediction errors in the least squares sense, with the AR coefficients constrained to satisfy the L-D recursion | Minimizes the forward prediction error in the least squares sense | Minimizes the forward and backward prediction errors in the least squares sense | Minimizes the forward prediction error in the least squares sense (also called "autocorrelation method")         |
| <b>Advantages</b>                    | Always produces a stable model   |   |   | Always produces a stable model   |
| <b>Disadvantages</b>                 |  | Can produce unstable models                                       | Can produce unstable models   | Performs relatively poorly for short data records  |
| <b>Conditions for Nonsingularity</b> |  | Order must be less than or equal to 1/2 the input frame size      | Order must be less than or equal to 2/3 the input frame size                    | Because of the biased estimate, the autocorrelation matrix is guaranteed to positive-definite, hence nonsingular |

## Version History

Introduced before R2006a

## References

- [1] Kay, S. M. *Modern Spectral Estimation: Theory and Application*. Englewood Cliffs, NJ: Prentice-Hall, 1988.
- [2] Marple, S. L., Jr., *Digital Spectral Analysis with Applications*. Englewood Cliffs, NJ: Prentice-Hall, 1987.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Generated code relies on the `memcpy` or `memset` function (`string.h`) under certain conditions.

## See Also

### Functions

`aryule`

### Blocks

Burg AR Estimator | Covariance AR Estimator | Modified Covariance AR Estimator | Yule-Walker Method

## Yule-Walker Method

Power spectral density estimate using Yule-Walker method



### Libraries:

DSP System Toolbox / Estimation / Power Spectrum Estimation

### Description

The Yule-Walker Method block estimates the power spectral density (PSD) of the input using the Yule-Walker AR method. This method, also called the autocorrelation method, fits an autoregressive (AR) model to the windowed input data by minimizing the forward prediction error in the least squares sense. This process leads to the Yule-Walker equations, which the Levinson-Durbin recursion solves.

The block computes the spectrum from the FFT of the estimated AR model parameters.

### Ports

#### Input

**Input** — Input

column vector | unoriented vector

Specify the input as a column vector or an unoriented vector. This input represents a frame of consecutive time samples from a single-channel signal.

Data Types: `single` | `double`

#### Output

**Output** — Power spectral density estimate

column vector

Power spectral density estimate of the signal at  $N_{fft}$  equally spaced frequency points, returned as a column vector. The frequency points are in the range  $[0, F_s)$ , where  $F_s$  is the sampling rate of the signal.

The block outputs are always nonsingular.

Data Types: `single` | `double`

### Parameters

**Inherit estimation order from input dimensions** — Inherit estimation order from input dimensions

`off` (default) | `on`

When you select **Inherit estimation order from input dimensions** parameter, the block sets the order of the all-pole model (estimation order) to a value that is one less than the length of the input

frame size. Otherwise, the **Estimation order** parameter value specifies the order. The block computes the spectrum from the FFT of the estimated AR model parameters.

#### **Estimation order** — Order of AR model

6 (default) | nonnegative integer

Specify the estimation order of the AR model as a nonnegative integer. To guarantee a valid output, the **Estimation order** parameter must be less than or equal to 1/2 the input vector length.

#### **Dependencies**

To enable this parameter, clear the **Inherit estimation order from input dimensions** parameter.

#### **Inherit FFT length from estimation order** — Inherit FFT length from estimation order

off (default) | on

When you select the **Inherit FFT length from estimation order** parameter, the FFT length  $N_{fft}$  is one greater than the estimation order. To specify the number of points on which to perform the FFT, clear the **Inherit FFT length from estimation order** parameter. You can then specify the FFT length as a power of 2 using the **FFT length** parameter. The block zero-pads or wraps the input to  $N_{fft}$  before computing the FFT.

#### **FFT length** — FFT length

256 (default) | positive integer greater than or equal to 2

Enter the number of data points  $N_{fft}$  on which to perform the FFT as a positive integer greater than or equal to 2. When  $N_{fft}$  is larger than the input frame size, the block zero-pads each frame as needed. When  $N_{fft}$  is smaller than the input frame size, the block wraps each frame as needed.

#### **Dependencies**

To enable this parameter, clear the **Inherit FFT length from input dimensions** parameter.

#### **Inherit sample time from input** — Inherit sample time from input

on (default) | off

When you select the **Inherit sample time from input** parameter, the block computes the frequency data from the sample period of the input signal. For the block to produce a valid output, the following conditions must hold:

- The input to the block is the original signal with no samples added or deleted (by insertion of zeros, for example).
- The sample period of the time-domain signal in the simulation equals the sample period of the original time series.

If these conditions do not hold, clear the **Inherit sample time from input** check box. You can then specify a sample time using the **Sample time of original time series** parameter.

#### **Sample time of original time series** — Sample time of original time-domain signal

1 (default) | positive scalar

Specify the sample time of the original time-domain signal as a positive scalar.

### Dependencies

To enable this parameter, clear the **Inherit sample time from input** parameter.

## Block Characteristics

|                                 |                 |
|---------------------------------|-----------------|
| <b>Data Types</b>               | double   single |
| <b>Multidimensional Signals</b> | No              |
| <b>Variable-Size Signals</b>    | No              |

## More About

### Compare Power Spectral Density Estimation Methods

The Burg Method and Yule-Walker Method blocks return similar results for large frame sizes.

This table compares the features of the Burg Method block to the Covariance Method, Modified Covariance Method, and the Yule-Walker Method blocks.

|                        | <b>Burg</b>  | <b>Covariance</b>   | <b>Modified Covariance</b>  | <b>Yule-Walker</b>   |
|------------------------|--|---|---|--|
| <b>Characteristics</b> | Does not apply window to data  | Does not apply window to data   | Does not apply window to data   | Applies window to data   |
|                        | Minimizes the forward and backward prediction errors in the least squares sense, with the AR coefficients constrained to satisfy the L-D recursion | Minimizes the forward prediction error in the least squares sense                   | Minimizes the forward and backward prediction errors in the least squares sense | Minimizes the forward prediction error in the least squares sense (also called autocorrelation method) |
| <b>Advantages</b>      | High resolution for short data records   | Better resolution than Yule-Walker for short data records (more accurate estimates) | High resolution for short data records  | Performs as well as other methods for large data records   |
|                        | Always produces a stable model   | Able to extract frequencies from data consisting of $p$ or more pure sinusoids      | Able to extract frequencies from data consisting of $p$ or more pure sinusoids  | Always produces a stable model   |
|                        |  |   | Does not suffer spectral line-splitting   |  |



|                                      | <b>Burg</b>  | <b>Covariance</b>  | <b>Modified Covariance</b>                                   | <b>Yule-Walker</b>  |
|--------------------------------------|--|--|--|---|
| <b>Disadvantages</b>                 | Peak locations highly dependent on initial phase                                       | Can produce unstable models                                  | Can produce unstable models                                  | Performs relatively poorly for short data records   |
|                                      | Can suffer spectral line-splitting for sinusoids in noise, or when order is very large | Frequency bias for estimates of sinusoids in noise           | Peak locations slightly dependent on initial phase           | Frequency bias for estimates of sinusoids in noise  |
|                                      | Frequency bias for estimates of sinusoids in noise                                     |  | Minor frequency bias for estimates of sinusoids in noise     |   |
| <b>Conditions for Nonsingularity</b> |  | Order must be less than or equal to 1/2 the input frame size | Order must be less than or equal to 2/3 the input frame size | Because of the biased estimate, the autocorrelation matrix is guaranteed to be positive-definite, hence nonsingular |

## Version History

Introduced before R2006a

## References

- [1] Kay, S. M. *Modern Spectral Estimation: Theory and Application*. Englewood Cliffs, NJ: Prentice-Hall, 1988.
- [2] Marple, S. L. Jr., *Digital Spectral Analysis with Applications*. Englewood Cliffs, NJ: Prentice-Hall, 1987.
- [3] Orfanidis, S. J. *Introduction to Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1995.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

## See Also

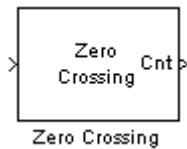
Burg Method | Covariance Method | Levinson-Durbin | Autocorrelation LPC | Short-Time FFT | Yule-Walker AR Estimator

## Topics

“Spectral Analysis”

## Zero Crossing

Count number of times signal crosses zero in single time step



### Libraries:

DSP System Toolbox / Signal Operations

## Description

The Zero Crossing block concludes that a signal in a given channel has passed through zero if it meets any of the following criteria, where  $x_i$  is the current value of the signal,  $x_{i-1}$  is the previous value, and so on:

- $x_i < 0$  and  $x_{i-1} > 0$
- $x_i > 0$  and  $x_{i-1} < 0$
- For a positive integer  $L$ ,  $x_i < 0$ ,  $x_{i-1} = 0$ , and  $x_{i-L-1} > 0$ , where  $0 \leq l \leq L$ .
- For a positive integer  $L$ ,  $x_i > 0$ ,  $x_{i-1} = 0$ , and  $x_{i-L-1} < 0$ , where  $0 \leq l \leq L$ .

For the first value of the input signal,  $x_{i-1}$  and  $x_{i-2}$  are zero. The block outputs the number of times the signal crosses zero in a single time step at the **Cnt** port.

## Ports

### Input

**Port\_1** — Data input  
vector | matrix

The input signal specified as a real-valued vector or matrix.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

### Output

**Cnt** — Number of zero crossings  
scalar | row vector

The block outputs the number of zero crossings in the data input as one of these:

- Scalar -- When the data input is a column vector, the block outputs a scalar value representing the number of zero crossings in the data input.
- Row vector -- When the data input is a matrix, the block outputs a row vector with each element in the row vector representing the number of zero crossings in the corresponding column of the data input.

Data Types: uint32

## Parameters

**Input processing** — Method to process input signals

Columns as channels (frame based) (default) | Elements as channels (sample based)

Specify how the block should process the input.

- Columns as channels (frame based) — When you select this option, the block treats each column of the input as a separate channel.
- Elements as channels (sample based) — When you select this option, the block treats each element of the input as a separate channel.

## Block Characteristics

|                                 |   |
|---------------------------------|---|
| <b>Data Types</b>               | double   fixed point   integer   single |
| <b>Direct Feedthrough</b>       | no                                      |
| <b>Multidimensional Signals</b> | no                                      |
| <b>Variable-Size Signals</b>    | no                                      |
| <b>Zero-Crossing Detection</b>  | no                                      |

## Version History

Introduced before R2006a

## Extended Capabilities

### C/C++ Code Generation

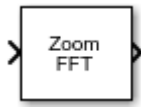
Generate C and C++ code using Simulink® Coder™.

## See Also

Hit Crossing

## Zoom FFT

High-resolution FFT of a portion of a spectrum



**Libraries:**  
DSP System Toolbox / Transforms

### Description

The Zoom FFT block computes the fast Fourier Transform (FFT) of a signal over a portion of frequencies in the Nyquist interval. By setting an appropriate decimation factor  $D$ , and sampling rate  $F_s$ , you can choose the bandwidth of frequencies to analyze  $BW$ , where  $BW = F_s/D$ . You can also select a specific range of frequencies to analyze in the Nyquist interval by choosing the center frequency of the desired band.

The resolution of a signal is the ratio of  $F_s$  and the FFT length ( $L$ ). Using zoom FFT, you can retain the same resolution you would achieve with a full-size FFT on your original signal by computing a small FFT on a shorter signal. The shorter signal comes from decimating the original signal. The savings come from being able to compute a much shorter FFT while achieving the same resolution. For a decimation factor of  $D$ , the new sampling rate,  $F_{sd}$ , is  $F_s/D$ , and the new frame size (and FFT length) is  $L_d = L/D$ . The resolution of the decimated signal is  $F_{sd}/L_d = F_s/L$ . To achieve a higher resolution of the shorter band, use the original FFT length,  $L$ , instead of the decimated FFT length,  $L_d$ .

### Ports

#### Input

**x** — Input data  
vector | matrix

Input data for which the block computes zoom FFT, specified as a vector or a matrix.

The block treats each column of the input signal as a separate channel. If the input is a two-dimensional signal, the first dimension represents the channel length (or frame size) and the second dimension represents the number of channels. If the input is a one-dimensional signal, then the block interprets it as a single-channel signal.

When you input a variable-size signal (frame length changes during simulation), the frame length of the signal can be arbitrary, that is, the input frame length does not have to be a multiple of the decimation factor. When you input a fixed-size signal (frame length does not change during simulation), the frame length can be arbitrary only when you select the **Allow arbitrary frame length for fixed-size input signals** parameter.

This port is unnamed until you select the **Specify center frequency from input port** parameter and click **Apply**.

Data Types: single | double

**Fc** — Center frequency input  
real scalar

Center frequency of the desired band in Hz, specified as a real scalar in the range  $(- \text{SampleRate}/2, \text{SampleRate}/2)$ . *SampleRate* is either inherited from the input signal or specified through the **Input sample rate (Hz)** parameter.

#### Dependency

To enable this port, select the **Specify center frequency from input port** check box and click **Apply**.

Data Types: single | double

#### Output Arguments

**Port\_1** — Zoom FFT output  
vector | matrix

Zoom FFT output, returned as a vector or matrix.

If you select the **Inherit FFT Length from input dimensions** check box, the dimensions of the output signal depend on the dimensions of the input signal, the decimation factor of the Zoom FFT block, and whether you select the **Allow arbitrary frame length for fixed-size input signals** parameter.

This table provides more details on the dimensions of the output signal when you input a fixed-size input signal.

#### Fixed-Size Input Signal

| Input Signal  | Output Signal Dimensions  |
|---|---|
| $P$ -by-1 column vector, where $P$ is a multiple of the decimation factor $D$     | $(P/D)$ -by-1   |
| $P$ -by-1 column vector, where $P$ is not a multiple of the decimation factor $D$ | $\text{ceil}(P/D)$ -by-1 when you select <b>Allow arbitrary frame length for fixed-size input signals</b> .<br><br>If you do not select <b>Allow arbitrary frame length for fixed-size input signals</b> , the block errors.    |
| $P$ -by- $Q$ matrix, where $P$ is a multiple of the decimation factor $D$         | $(P/D)$ -by- $Q$  |
| $P$ -by- $Q$ matrix, where $P$ is not a multiple of the decimation factor $D$     | $\text{ceil}(P/D)$ -by- $Q$ when you select <b>Allow arbitrary frame length for fixed-size input signals</b> .<br><br>If you do not select <b>Allow arbitrary frame length for fixed-size input signals</b> , the block errors. |

This table gives more details on the dimensions of the output signal when you input a variable-size input signal. When you input a variable-size signal (frame length changes during simulation), the **Allow arbitrary frame length for fixed-size input signals** is visible on the block dialog box but does not have any impact on the input frame length. You can input a variable-size signal of any frame

length even if you do not select the **Allow arbitrary frame length for fixed-size input signals** parameter.

### Variable-Size Input Signal

| Input Signal | Output Signal Dimensions    |
|--------------|-----------------------------|
| $P$ -by- $Q$ | $\text{ceil}(P/D)$ -by- $Q$ |

If you clear the **Inherit FFT Length from input dimensions** check box, the output frame size equals the value you specify in the **FFT length( $L$ )** parameter.

| Input Signal | Output Signal Dimensions |
|--------------|--------------------------|
| $P$ -by- $Q$ | $L$ -by- $Q$             |

The data type of the output is same as the data type of the input.

Data Types: `single` | `double`

## Parameters

**Decimation factor** — Decimation factor  
2 (default) | positive integer

Decimation factor, specified as a positive integer. This value specifies the factor by which the block reduces the bandwidth of the input signal. The number of rows in the input signal must be a multiple of the decimation factor.

Example: 4

Example: 8

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**Specify center frequency from input port** — Flag to specify center frequency from input port  
'off' (default) | 'on'

When you select this option and click **Apply**, the input port **Fc** appears on the block icon. You can pass the center frequency through this input port as a scalar.

**Center frequency (Hz)** — Center frequency  
0 (default) | real scalar

Center frequency of the desired band in Hz, specified as a real scalar in the range  $(- \text{SampleRate}/2, \text{SampleRate}/2)$ . *SampleRate* is the input sample rate either inherited from the input or specified through the **Input sample rate (Hz)** parameter.

Example: 0.5

Example: 10

### Dependencies

This parameter applies when you clear the **Specify center frequency from input port** check box.

Data Types: `single` | `double`

**Inherit FFT Length from input dimensions** — Flag to inherit FFT length from input dimensions  
'on' (default) | 'off'

When you select this option, the FFT length is the ratio of the input frame size (number of rows in the input) and the **Decimation factor**.

**FFT length** — FFT length  
64 (default) | positive scalar

FFT length, specified as a positive integer. The FFT length must be greater than or equal to the ratio of the frame size (number of input rows) and the **Decimation factor**.

Example: 24

Example: 52

#### Dependencies

This parameter applies when you clear the **Inherit FFT Length from input dimensions** check box.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**Inherit Sample rate from input** — Flag to inherit sample rate from input  
'off' (default) | 'on'

When you clear this check box, the block inherits the sample rate from the input signal.

**Input sample rate (Hz)** — Input sample rate  
44100 (default) | positive real scalar

Input sample rate in Hz, specified as positive real scalar.

Example: 44100

Example: 48000

**Allow arbitrary frame length for fixed-size input signals** — Allow arbitrary frame length for fixed-size input signals

off (default) | on

Specify whether fixed-size input signals (whose size does not change during simulation) can have an arbitrary frame length, where the frame length does not have to be a multiple of the decimation factor. The block uses this parameter only for fixed-size input signals and ignores it if the input has a variable-size.

When the input signal is a variable-size signal, the signal can have arbitrary frame length, that is, the frame length does not have to be a multiple of the decimation factor.

For fixed-size input signals, if you:

- Select the **Allow arbitrary frame length for fixed-size input signals** parameter, the frame length of the signal does not have to be a multiple of the decimation factor. If the input is not a multiple of the decimation factor, then the output is generally a variable-size signal. Therefore, to support arbitrary input size, the block must also support variable-size operations, which you can enable by selecting the **Allow arbitrary frame length for fixed-size input signals** parameter.
- Clear the **Allow arbitrary frame length for fixed-size input signals** parameter, the input frame length must be a multiple of the decimation factor.

**Simulate using** — Type of simulation to run

Interpreted execution (default) | Code generation

Type of simulation to run. You can set this parameter to:

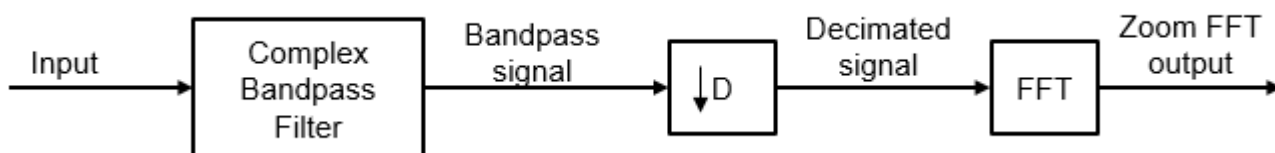
- **Interpreted execution:** Simulate model using the MATLAB interpreter. This option shortens startup time.
- **Code generation:** Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations as long as the model does not change. This option requires additional startup time but provides faster subsequent simulations.

## Block Characteristics

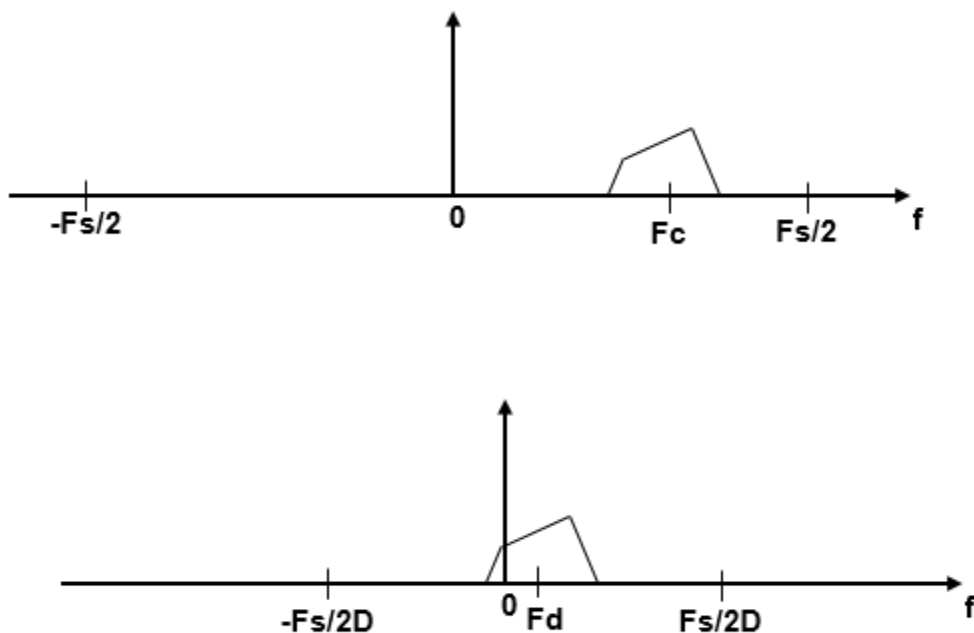
|                                 |                 |
|---------------------------------|-----------------|
| <b>Data Types</b>               | double   single |
| <b>Multidimensional Signals</b> | No              |
| <b>Variable-Size Signals</b>    | Yes             |

## Algorithms

The zoom FFT algorithm leverages bandpass filtering before computing the FFT of the signal. The concept of bandpass filtering is that suppose you are interested in the band  $[F1, F2]$  of the original input signal, sampled at the rate  $F_s$  Hz. If you pass this signal through a complex (one-sided) bandpass filter centered at  $F_c = (F1 + F2)/2$ , with the bandwidth  $BW = F2 - F1$ , and then downsample the signal by a factor of  $D = \text{floor}(F_s/BW)$ , the desired band comes down to the baseband.







If  $F_c$  cannot be expressed in the form of  $k \times F_s/D$ , where  $k$  is an integer, then the shifted, decimated spectrum is not centered at DC. In this case, the center frequency gets translated to  $F_d$ .

$$F_d = F_c - (F_s/D) \times \text{floor}((D \times F_c + F_s/2)/F_s)$$

The complex bandpass filter is obtained by first designing a lowpass filter prototype and then multiplying the lowpass coefficients with a complex exponential. This algorithm uses a multirate, multistage FIR filter as the lowpass filter prototype. To obtain the bandpass filter, the coefficients of each stage are frequency shifted. The decimation factor is the cumulative decimation factor of each stage. The complex bandpass filter followed by the decimator are implemented using an efficient polyphase structure. For more details on the design of the complex bandpass filter from the multirate multistage FIR filter prototype, see “Zoom FFT” and “Complex Bandpass Filter Design”.

## Version History

Introduced in R2017b

### R2023a: Support for arbitrary input frame length

This block supports input signals with arbitrary frame lengths when the:

- Input signal is a fixed-size signal (frame length does not change during simulation) and you select the **Allow arbitrary frame length for fixed-size input signals** parameter.
- Input signal is a variable-size signal (frame length changes during simulation).

When this block supports an input signal with an arbitrary frame length, the input frame length does not have to be a multiple of the decimation factor and the output is a variable-size signal.

## References

[1] Harris, F.J. *Multirate Signal Processing for Communication Systems*. Prentice Hall, 2004, pp. 208-209.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

## See Also

### Objects

`dsp.ZoomFFT` | `dsp.FFT` | `dsphdl.FFT` | `dsp.IFFT` | `dsphdl.IFFT`

### Blocks

`FFT` | `FFT` | `Magnitude FFT` | `Short-Time FFT`

### Topics

“Zoom FFT”

“Complex Bandpass Filter Design”

# Analysis Methods for Filter System Objects

---

## Analysis Methods for Filter System Objects

| Method  | Description   |
|---|---|
| Single-rate and Multirate Analysis  |   |
| fvtool  | Filter visualization tool                                     |
| info  | Filter information  |
| freqz   | Frequency response of a discrete-time filter                  |
| phasez  | Phase response of a discrete-time filter                      |
| zerophase   | Zero-phase response of a discrete-time filter                 |
| grpdelay  | Group delay of a discrete-time filter                         |
| phasedelay  | Phase delay of a discrete-time filter                         |
| impz  | Impulse response of a discrete-time filter                    |
| impzlength  | Impulse response length                                       |
| stepz   | Step response of a discrete-time filter                       |
| zplane  | Pole/Zero plot  |
| cost  | Cost estimate   |
| measure   | Measure filter response                                       |
| order   | Filter order  |
| firtype   | Determine the type (1-4) of a linear phase FIR filter         |
| coeffs  | Return filter coefficients                                    |
| Multirate Analysis  |   |
| polyphase   | Polyphase decomposition of multirate filters                  |
| gain  | Gain of a Cascaded Integrator-Comb (CIC) filter               |
| Second-order Sections   |   |
| scale   | Scale second-order sections                                   |
| scalecheck  | Check scale of second-order sections                          |
| scaleopts   | Create options object for sos scaling                         |
| cumsec  | Vector of cumulative second-order section filters             |
| reorder   | Reorder second-order sections                                 |
| sos   | Convert IIR filter to Biquad filter                           |
| Code Generation   |   |
| realizemdl  | Filter realization diagram                                    |
| Fixed-point (supported by FIRFilter, IIRFilter, AllpoleFilter, and BiquadFilter only) |   |
| noisepsd  | Power spectral density of filter output due to roundoff noise |
| noisepsdopts  | Create options object for noisepsd computation                |
| freqrespest   | Frequency response estimate via filtering                     |

| <b>Method</b> | <b>Description</b>                                    |
|---------------|---|
| freqrespopts  | Create options object for frequency response estimate |
| Other         |   |
| isallpass     | True for allpass discrete-time filter                 |
| islinphase    | True for linear discrete-time filter                  |
| ismaxphase    | True if maximum phase                                 |
| isminphase    | True if minimum phase                                 |
| isreal        | True for discrete-time filter with real coefficients  |
| isstable      | True if the filter is stable                          |
| isfir         | True if the filter is FIR                             |
| issos         | True if filter is in second order sections form       |
| specifyall    | Specify all fixed-point properties                    |



# System Objects

---

## powermeter

Measure power and CCDF of the power of voltage signal

### Description

The `powermeter` System object computes the power measurements of a voltage signal. When you set the `ComputeCCDF` property to `true`, the object also calculates the complementary cumulative distribution function (CCDF) of the power of a voltage signal. The CCDF measurements that the object outputs are relative power and probability (in percentage). The power measurements include average power, peak power, and peak-to-average power ratio.

For more details on how the object computes the power measurements and the CCDF measurements, see “Algorithms” on page 4-11.

To measure the power and the CCDF of the power of a voltage signal:

- 1 Create the `powermeter` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
meter = powermeter  
meter = powermeter(Len,Overlap,Name=Value)  
meter = powermeter(Name=Value)
```

### Description

`meter = powermeter` returns a `powermeter` system object that computes power, peak-to-average power ratio (PAPR), and the complementary cumulative distribution function (CCDF) of the power of voltage signal. The CCDF helps find the probability that the instantaneous signal power exceeds a specified level above the average signal power.

`meter = powermeter(Len,Overlap,Name=Value)` sets the `WindowLength` property to `Len` and the `OverlapLength` property to `Overlap`.

To enable this syntax, set the `ComputeCCDF` property to `false`.

`meter = powermeter(Name=Value)` returns a `powermeter` system object with each specified property set to the specified value. Enclose each property name in quotes. You can use this syntax with the previous input argument.



## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see *System Design in MATLAB Using System Objects*.

### Measurement — Desired power measurement

'Average power' (default) | 'Peak power' | 'Peak-to-average power ratio' | 'All'

Desired power measurement, specified as 'Average power', 'Peak power', 'Peak-to-average power ratio' or 'All'.

For more details on how the object computes these measurements, see “Algorithms” on page 4-11.

### ReferenceLoad — Reference load

1 (default) | positive scalar

Reference load used by the power meter to compute the power values in ohms, specified as a positive scalar.

**Tunable:** Yes

Data Types: `single` | `double`

### ComputeCCDF — Compute CCDF

false (default) | true

Specify whether to calculate CCDF as:

- `false` -- The object does not calculate the CCDF measurements. The object computes the power measurements using the “Sliding Window Method” on page 4-11.
- `true` -- The object computes the CCDF measurements.
  - All input samples since object creation or reset are used to compute statistics. The power measurements are stationary.
  - The `WindowLength` and `OverlapLength` properties become read-only and are set to `Inf` and `0` respectively.

Data Types: `logical`

### PowerRange — Power range

50 (default) | positive scalar

The x-axis range of the computed CCDF curves in dB, specified as a positive scalar. The computed CCDF curves end at the maximum relative power, namely, peak-to-average power ratio (PAPR) of the signal, and start at `PAPR - PowerRange`. For the CCDF capability of the `powermeter` object, relative power is the power in dB by which the instantaneous signal power is above the average signal power.

### Dependencies

To enable this property, set the `ComputeCCDF` property to `true`.

Data Types: `single` | `double`

### **PowerResolution — Power Resolution in dB**

0.1 (default) | positive scalar

The x-axis resolution of the computed CCDF curves in dB, specified as a positive scalar.

#### **Dependencies**

To enable this property, set the `ComputeCCDF` property to `true`.

Data Types: `single` | `double`

### **WindowLength — Window length**

256 (default) | nonnegative integer

Sliding window length over which the measurement is computed, specified as a nonnegative integer.

#### **Dependencies**

If the `ComputeCCDF` property is `true`, then `WindowLength` property is set to `Inf` and is read-only.

Data Types: `single` | `double`

### **OverlapLength — Overlap length between windows**

`WindowLength - 1` (default) | nonnegative integer

Overlap length between sliding windows, specified as a nonnegative integer. The value of overlap length varies in the range  $[0, \text{WindowLength} - 1]$ . If not specified, the overlap length is `WindowLength - 1`.

#### **Dependencies**

If the `ComputeCCDF` property is `true`, then `OverlapLength` property is set to 0 and is read-only.

Data Types: `single` | `double`

### **OutputPowerUnits — Output power units**

'dBm' (default) | 'dBW' | 'Watts'

Units of the measured power values, specified as 'dBm', 'dBW', or 'Watts'.

## **Usage**

### **Syntax**

```
avgpwr = meter(x)
peakpwr = meter(x)
papr = meter(x)
[avgpwr, peakpwr, papr] = meter(x)
```

### **Description**

`avgpwr = meter(x)` computes the average power of the input signal `x` when the `Measurement` property is set to 'Average power'. Each column of `x` is treated as an independent channel. The object computes the average power of each channel of the input signal independently.

`peakpwr = meter(x)` computes the peak power of the input signal `x` when the `Measurement` property is set to `'Peak power'`. Each column of `x` is treated as an independent channel. The object computes the peak power of each channel of the input signal independently.

`papr = meter(x)` computes the peak-to-average power ratio of the input signal `x` when the `Measurement` property is set to `'Peak-to-average power ratio'`. Each column of `x` is treated as an independent channel. The object computes the peak-to-average power ratio of each channel of the input signal independently.

`[avgpwr, peakpwr, papr] = meter(x)` computes the average power, peak power, and the peak-to-average power ratio of the input signal `x` when the `Measurement` property is set to `'All'`. Each column of `x` is treated as an independent channel. The object computes the power measurements of each channel of the input signal independently.

### Input Arguments

#### **x** — Input voltage signal

vector | matrix

Input voltage signal, specified as a vector or a matrix in volts. If `x` is a matrix, the object treats each column as an independent channel and computes the power measurements along each channel.

The object also accepts variable-size inputs. That is, once the object is locked, you can change the size of each input channel, but you cannot change the number of channels.

Data Types: `single` | `double`

### Output Arguments

#### **avgpwr** — Average power

scalar | vector | matrix

Average power of the voltage signal, returned as a scalar, vector or a matrix, with the units determined by the `PowerUnits` property.

If you set the `ComputeCCDF` property to `false`, the object computes the moving average power using the “Sliding Window Method” on page 4-11.

When you input a signal of size  $m$ -by- $n$  to the object when the `ComputeCCDF` property is set to `false`, the output has an upper bound size of `ceil(m/hop size)`-by- $n$ . Hop size is window length – overlap length. In other cases, the output has a size of  $m$ -by- $n$ .

When you generate code from this object, the variable-size behavior of the output in the generated code depends on the input frame length and whether the size of the input signal is fixed or variable. For more details, see “Code Generation” on page 4-14.

If you set the `ComputeCCDF` property to `true`, the object computes the stationary average power of the entire signal along each channel. In this case, the size of the output is 1-by- $M$ , where  $M$  is the number of channels in the input signal.

For details on how the object computes the average power, see “Average Power” on page 4-11.

Data Types: `single` | `double`

#### **peakpwr** — Peak power

scalar | vector | matrix

Peak power of the voltage signal, returned as a scalar, vector or a matrix, with the units determined by the `PowerUnits` property.

If you set the `ComputeCCDF` property to `false`, the object computes the moving peak power using the “Sliding Window Method” on page 4-11.

When you input a signal of size  $m$ -by- $n$  to the object when the `ComputeCCDF` property is set to `false`, the output has an upper bound size of  $\text{ceil}(m/\text{hop size})$ -by- $n$ . Hop size is window length – overlap length. In other cases, the output has a size of  $m$ -by- $n$ .

When you generate code from this object, the variable-size behavior of the output in the generated code depends on the input frame length and whether the size of the input signal is fixed or variable. For more details, see “Code Generation” on page 4-14.

If you set the `ComputeCCDF` property to `true`, the object computes the stationary peak power of the entire signal along each channel. In this case, the size of the output is 1-by- $M$ , where  $M$  is the number of channels in the input signal.

For details on how the object computes the peak power, see “Peak Power” on page 4-12.

Data Types: `single` | `double`

### **papr** — Peak-to-average power ratio

scalar | vector | matrix

Peak-to-average power ratio of the voltage signal, returned as a scalar, vector or a matrix.

If you set the `ComputeCCDF` property to `false`, the object computes the moving peak-to-average power ratio using the “Sliding Window Method” on page 4-11.

When you input a signal of size  $m$ -by- $n$  to the object when the `ComputeCCDF` property is set to `false`, the output has an upper bound size of  $\text{ceil}(m/\text{hop size})$ -by- $n$ . Hop size is window length – overlap length. In other cases, the output has a size of  $m$ -by- $n$ .

When you generate code from this object, the variable-size behavior of the output in the generated code depends on the input frame length and whether the size of the input signal is fixed or variable. For more details, see “Code Generation” on page 4-14.

If you set the `ComputeCCDF` property to `true`, the object computes the stationary peak-to-average power ratio of the entire signal along each channel. In this case, the size of the output is 1-by- $M$ , where  $M$  is the number of channels in the input signal.

For details on how the object computes the peak-to-average power ratio, see “Peak-to-Average Power Ratio” on page 4-12.

Data Types: `single` | `double`

## **Object Functions**

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Specific to powermeter

|               |   |
|---------------|---|
| plotCCDF      | Plot CCDF curves  |
| ccdf          | Get coordinates of CCDF curves                            |
| relativePower | Use CCDF to find relative power for specified probability |
| probability   | Use CCDF to find probability for specified relative power |

## Common to All System Objects

|         |  |
|---------|--|
| step    | Run System object algorithm  |
| release | Release resources and allow changes to System object property values and input characteristics |
| reset   | Reset internal states of System object   |

## Examples

### Compute Power Measurements of Voltage Signal

Compute the power measurements of a noisy sinusoidal signal using a power meter. These measurements include average power, peak power, and peak-to-average power ratio.

Assume the maximum voltage of the signal to be 100 V. The instantaneous values of a sinusoidal waveform are given by the equation  $V_i = V_{max} \times \sin(2\pi ft)$ , where  $V_i$  is the instantaneous value,  $V_{max}$  is the maximum voltage of the signal, and  $f$  is the frequency of the signal in Hz.

### Initialization

The input signal is a sum of two sine waves with frequencies set to 1 kHz and 10 kHz, respectively. The frame length and the sampling frequency of the generated signal is 512 samples and 44.1 kHz, respectively.

To measure the power in this signal, create a `powermeter` object. Set `Measurement` to 'All'. This setting enables the object to measure the average power, peak power, and peak-to-average power ratio. Set the length of the sliding window to 16 samples and the reference load to 50 Ohms. Use this object to measure the power in dBm units. Visualize the power measurements using the `timescope` object.

```

FrameLength = 512;
Fs = 44.1e3;
A = 100;
sine1 = dsp.SinWave(Amplitude=A, ...
    Frequency=1e3, ...
    SampleRate=44.1e3, ...
    SamplesPerFrame=FrameLength);
sine2 = dsp.SinWave(Amplitude=A, ...
    Frequency=10e3, ...
    SampleRate=44.1e3, ...
    SamplesPerFrame=FrameLength);
pm = powermeter(16,Measurement="All", ...
    ReferenceLoad=50, ...
    PowerUnits="dBm");
scope = timescope(NumInputPorts=4,SampleRate=Fs, ...
    TimeSpanSource="property", ...
    TimeSpan=96, ...
    YLabel="dBm", ...

```

```

        YLimits=[-30 90]);
title = 'Power Measurements';
scope.ChannelNames = {'Average power', ...
    'Peak power', 'Peak-to-average power ratio', ...
    'Expected Average Power'};
scope.Title = title;

```

### Compute the Power Measurements

Add zero-mean white Gaussian noise with a standard deviation of 0.001 to the sum of sine waves. Vary the amplitude of the sine waves. Measure the average power, peak power, and the peak-to-average power ratio of this noisy sinusoidal signal that has a varying amplitude. For details on how the object measures these power values, see “Algorithms” on page 4-11. Compare the measured values to the expected value of the average power.

The expected value of the average power  $P$  of the noisy sinusoidal signal is given by the following equation.

$$P = \frac{A_1^2}{2R} + \frac{A_2^2}{2R} + \text{var}(\text{noise}),$$

where,

- $A_1$  is the amplitude of the first sinusoidal signal.
- $A_2$  is the amplitude of the second sinusoidal signal.
- $R$  is the reference load in ohms.

In dBm, the expected power is computed using the following equation:

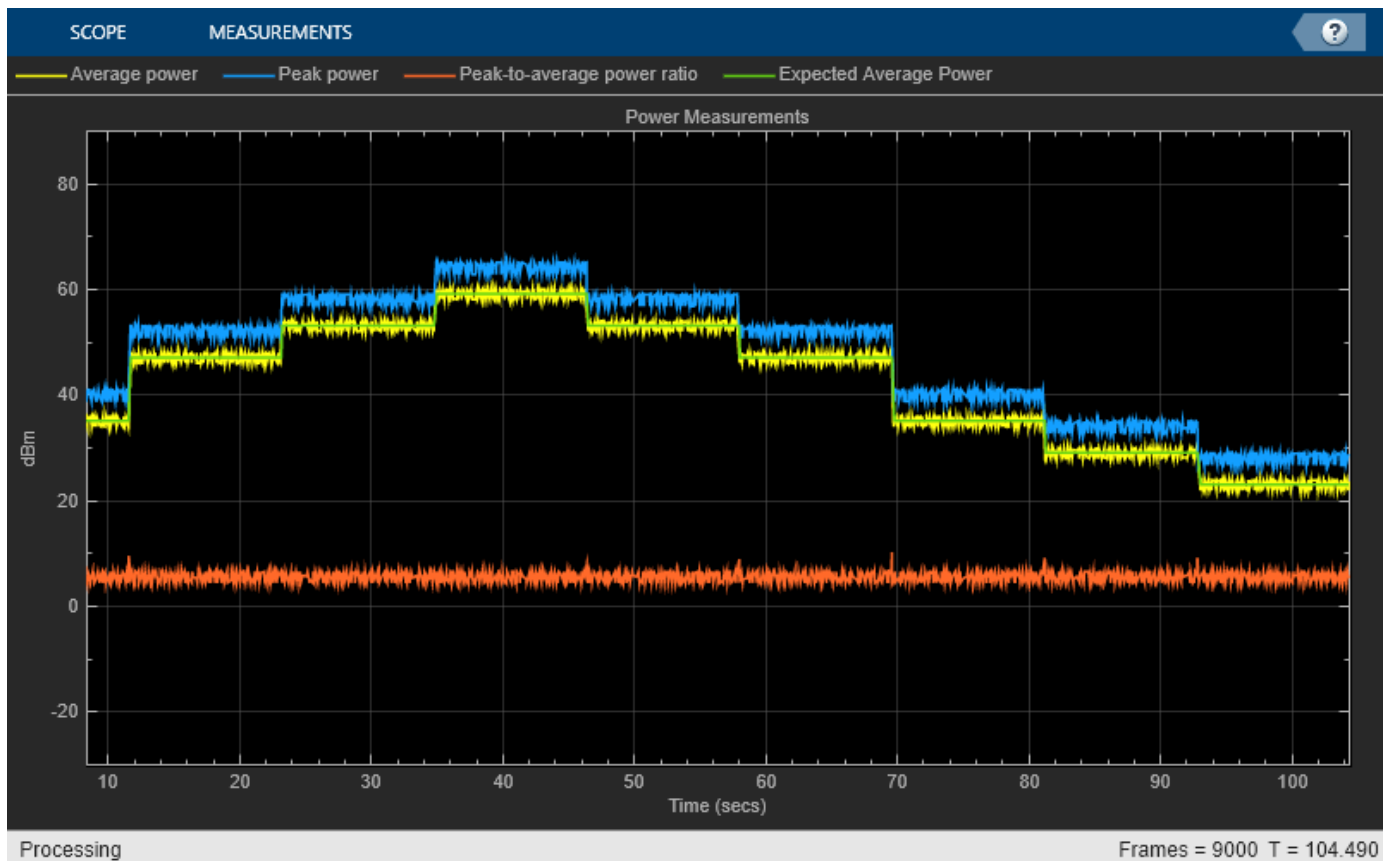
$$\text{expPwr}_{\text{dBm}} = 10\log_{10}(P) + 30.$$

Compare the expected value with the value computed by the object. All values are in dBm. These values match very closely. To verify, view the computed measurements using the `timescope` object.

```

Vect = [1/8 1/2 1 2 1 1/2 1/8 1/16 1/32];
for index = 1:length(Vect)
    V = Vect(index);
    for i = 1:1000
        x = V*sine1()+V*sine2()+0.001.*randn(FrameLength,1);
        P = (((V*A)^2)/100)+(((V*A)^2)/100)+(0.001)^2;
        expPwr = (10*log10(P)+30)*ones(FrameLength,1);
        [avgPwr,pkPwr,papr] = pm(x);
        scope(avgPwr,pkPwr,papr,expPwr);
    end
end
end

```



### Compute and Plot CCDF Measurements of Voltage Signal

Compute the CCDF measurements of a voltage signal. The CCDF measurements include the relative power and the probability. Relative power is the amount of power in dB by which the instantaneous signal power is above the average signal power. Probability in percentage refers to the probability that the instantaneous signal power is above the average signal power by the relative power in dB.

Initialize a powermeter object and set the ComputeCCDF property to true. The input to the object is a random voltage signal.

```
x = complex(rand(10000,1)-0.5,rand(10000,1)-0.5);
pm = powermeter(ComputeCCDF=true)
```

```
pm =
  powermeter with properties:
    Measurement: 'Average power'
    ReferenceLoad: 1
    ComputeCCDF: true
    PowerRange: 50
    PowerResolution: 0.1000
    OutputPowerUnits: 'dBm'

  Read-only properties:
    WindowLength: Inf
```

```
OverlapLength: 0
```

By default, the powermeter object computes the average power of the signal in dBm. The reference load is 1 ohm.

```
averagePower = pm(x)
```

```
averagePower = 22.2189
```

Using the probability function, find the probability that the instantaneous power of the signal is 3 dB above the average power. The relative power in this case is 3 dB.

```
prob = probability(pm,3)
```

```
prob = 7.6009
```

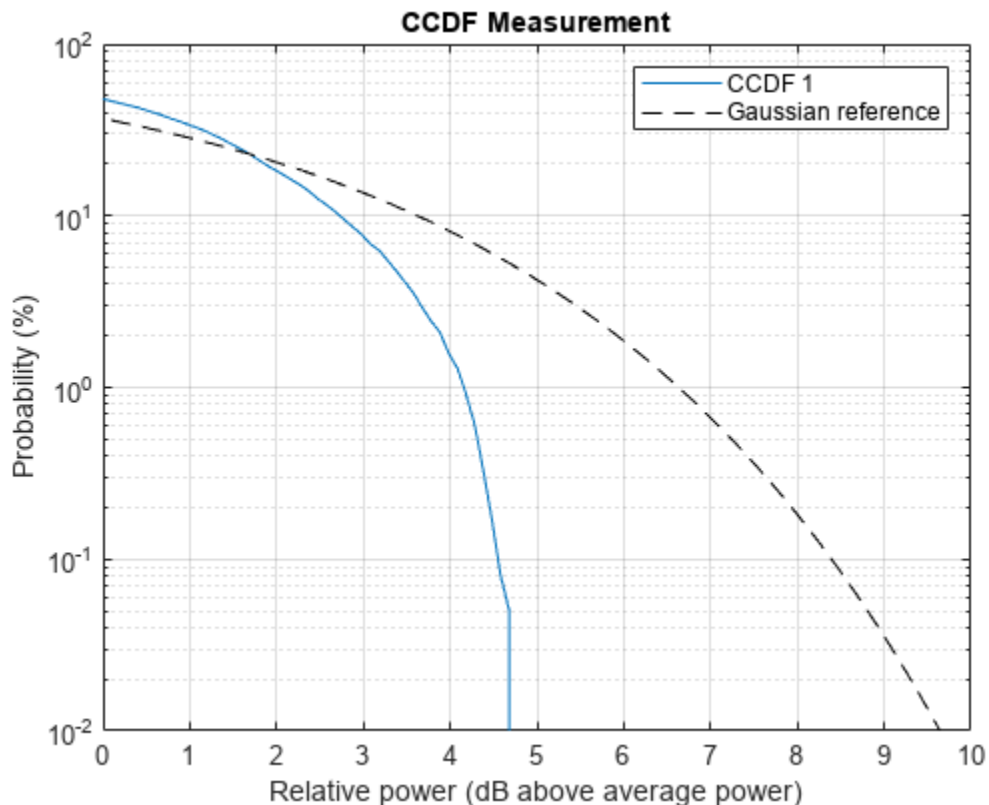
Compute the relative power using the relativePower function. Verify that the relative power for the computed probability is indeed 3 dB.

```
relpwr = relativePower(pm,prob)
```

```
relpwr = 3.0000
```

Using the plotCCDF function, plot the CCDF curve. Set the reference curve to Gaussian.

```
plotCCDF(pm,GaussianReference=true)
```





## Algorithms

### Sliding Window Method

When you do not configure the power meter to compute the CCDF measurements, the power meter computes the moving power measurements using the sliding window method.

In the sliding window method, the block computes the power measurement over a finite duration of the signal. The window length defines the length of the data over which the algorithm computes the power value. The window moves as new data comes in. The output for each input sample is the measurement over the current sample and  $Len - 1$  previous samples.  $Len$  is the length of the sliding window in samples. To compute the first output sample, the algorithm waits until it receives the hop size number of input samples. Hop size is defined as window length - overlap length. Remaining samples in the window are considered to be zero. As an example, if the window length is 5 and the overlap length is 2, then the algorithm waits until it receives 3 samples of input to compute the first sample of the output. After generating the first output, it generates the subsequent output samples for every hop size number of input samples.

For a more detailed example, see “Sliding Window Method”.

If the window is large, the power that the block computes is closer to the stationary power of the data. For data that does not change rapidly, use a long window to get a smoother measurement. For data that changes fast, use a smaller window.

When you configure the power meter to compute the CCDF measurements, the algorithm computes the stationary power of the data. It sets the window length to  $Inf$  and the overlap length to  $0$ , making both the parameters read-only.

### Average Power

When you do not configure the power meter to compute the CCDF measurements, the power meter computes the moving average power of the voltage signal across each channel using the Sliding Window Method on page 4-11. When you do configure the power meter to compute the CCDF measurements, the power meter computes the stationary average power of the voltage signal across each channel.

These equations give the average power in dBm, dBW, and in Watts units.

$$AvgPower_{dBm} = 10\log_{10}(Avg(|x|^2)/R) + 30$$

$$AvgPower_{dBW} = 10\log_{10}(Avg(|x|^2)/R)$$

$$AvgPower_{Watts} = Avg(|x|^2)/R$$

where,

- $x$  is the input voltage signal.
- $R$  is the reference load (in ohms) that the block uses to compute the power value.
- $Avg$  represents the moving average power when the power meter does not compute the CCDF measurements. The `powermeter` object in MATLAB uses the `dsp.MovingAverage` object and the Power Meter block in Simulink uses the Moving Average block.

When the power meter does compute the CCDF measurements, *Avg* represents the stationary average power across each channel.

### Peak Power

When you do not configure the power meter to compute the CCDF measurements, the power meter computes the moving peak power of the voltage signal across each channel using the Sliding Window Method on page 4-11. When you do configure the power meter to compute the CCDF measurements, the power meter computes the stationary peak power of the voltage signal across each channel.

These equations give the peak power in dBm, dBW, and in Watts units.

$$PeakPower_{dBm} = 10\log_{10}(Max(|x|^2)/R) + 30$$

$$PeakPower_{dBW} = 10\log_{10}(Max(|x|^2)/R)$$

$$PeakPower_{Watts} = Max(|x|^2)/R$$

where,

- *x* is the input voltage signal.
- *R* is the reference load (in ohms) that the block uses to compute the power value.
- *Max* represents the moving peak power when the power meter does not compute the CCDF measurements. The `powermeter` object in MATLAB uses the `dsp.MovingMaximum` object and the Power Meter block in Simulink uses the Moving Maximum block.

When the power meter does compute the CCDF measurements, *Max* represents the stationary peak power across each channel.

### Peak-to-Average Power Ratio

When you do not configure the power meter to compute the CCDF measurements, the power meter computes the moving peak-to-average power ratio of the voltage signal across each channel using the Sliding Window Method on page 4-11. When you do configure the power meter to compute the CCDF measurements, the power meter computes the stationary peak-to-average power ratio of the voltage signal across each channel.

These equations give the peak-to-average power ratio in dBm, dBW, and in Watts units.

$$pkAvgPwr_{dBm} = 10\log_{10}(Max(|x|^2)/Avg(|x|^2))$$

$$pkAvgPwr_{dBW} = 10\log_{10}(Max(|x|^2)/Avg(|x|^2))$$

$$pkAvgPwr_{Watts} = Max(|x|^2)/Avg(|x|^2)$$

where,

- *x* is the input voltage signal.
- *Avg* represents the moving average power when the power meter does not compute the CCDF measurements. The `powermeter` object in MATLAB uses the `dsp.MovingAverage` object and the Power Meter block in Simulink uses the Moving Average block.

When the power meter does compute the CCDF measurements, *Avg* represents the stationary average power across each channel.

- *Max* represents the moving peak power when the power meter does not compute the CCDF measurements. The `powermeter` object in MATLAB uses the `dsp.MovingMaximum` object and the Power Meter block in Simulink uses the Moving Maximum block.

When the power meter does compute the CCDF measurements, *Max* represents the stationary peak power across each channel.

## Relative Power

The power meter computes the relative power only when you configure the algorithm to compute the CCDF measurements. The algorithm uses a window of infinite duration to compute the relative power.

Relative power is the amount of power in dB by which the instantaneous signal power is above the average signal power. The block calculates the relative power using these equations:

If output power is in dBm,

$$relPower = 10\log_{10}(|x|^2/R) + 30 - AvgPower_{dBm}$$

If output power is in dBW,

$$relPower = 10\log_{10}(|x|^2/R) - AvgPower_{dBW}$$

If output power is in Watts,

$$relPower = (|x|^2/R) - AvgPower_{Watts}$$

where,

- $x$  is the input voltage signal.
- $R$  is the reference load (in ohms) that the block uses to compute the power value.
- $|x|^2/R$  is the instantaneous signal power in Watts.
- *AvgPower* is the average power of the voltage signal. For more details on how the algorithm computes this measurement, see "Average Power" on page 2-1183.

## Probability

The power meter computes the probability only when you configure the algorithm to compute the CCDF measurements. Probability in percentage refers to the probability that the instantaneous signal power is above the average signal power by the relative power in dB.

## Version History

Introduced in R2021a

### R2022b: Change in variable-size behavior for output signal in generated code

Starting in R2022b, if you generate code from this object with the `ComputeCCDF` property set to `false`, and if you input a signal with a frame length that is a multiple of the hop size (window length

– overlap length), this object generates a fixed-size output signal in the generated code. For more details, see “Code Generation” on page 4-14.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- See “System Objects in MATLAB Code Generation” (MATLAB Coder).
- When you set the `ComputeCCDF` property to `false` and generate code from this object, the variable-size behavior of the output depends on the input frame length and whether the size of the input signal is fixed or variable.

See this table for more details.

| Input Signal  | Input Size   | Output Signal   |
|---------------|--------------|---|
| Fixed-size    | $m$ -by- $n$ | When the input frame length is a multiple of the hop size, the output signal has a fixed-size of $(m/\text{hop size})$ -by- $n$ .<br><br>When input frame length is not a multiple of the hop size, the output signal is variable-sized and has an upper bound of $\text{ceil}(m/\text{hop size})$ -by- $n$ . |
| Variable-size | $m$ -by- $n$ | Output is a variable-size signal.<br><br>Output has an upper bound size of $\text{ceil}(m/\text{hop size})$ -by- $n$ .  |

## See Also

### Functions

`plotCCDF` | `ccdf` | `relativePower` | `probability`

### Objects

`dsp.MovingAverage` | `dsp.MovingMaximum` | `dsp.SpectrumEstimator` | `spectrumAnalyzer`

### Blocks

Power Meter

### Topics

“Compute Average Power of 256 QAM Signal in MATLAB” (Communications Toolbox)

# dsp.AdaptiveLatticeFilter

**Package:** dsp

Adaptive lattice filter

## Description

The `dsp.AdaptiveLatticeFilter` System object computes output, error, and coefficients using a lattice-based FIR adaptive filter.

To implement the adaptive FIR filter object:

- 1 Create the `dsp.AdaptiveLatticeFilter` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
alf = dsp.AdaptiveLatticeFilter
alf = dsp.AdaptiveLatticeFilter(len)
alf = dsp.AdaptiveLatticeFilter(Name,Value)
```

### Description

`alf = dsp.AdaptiveLatticeFilter` returns a lattice-based FIR adaptive filter System object, `alf`. This System object computes the filtered output and the filter error for a given input and desired signal.

`alf = dsp.AdaptiveLatticeFilter(len)` returns an `AdaptiveLatticeFilter` System object with the `Length` property set to `len`.

`alf = dsp.AdaptiveLatticeFilter(Name,Value)` returns an `AdaptiveLatticeFilter` System object with each specified property set to the specified value. Enclose each property name in single quotes. Unspecified properties have default values.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

**Method — Method to calculate filter coefficients**

'Least-squares Lattice' (default) | 'QR-decomposition Least-squares Lattice' | 'Gradient Adaptive Lattice'

Specify the method used to calculate filter coefficients as one of 'Least-squares Lattice', 'QR-decomposition Least-squares Lattice', 'Gradient Adaptive Lattice'. The default value is 'Least-squares Lattice'. For algorithms used to implement these three different methods, refer to [1] [2]. This property is nontunable.

**Length — Length of filter coefficients vector**

32 (default) | positive integer

Specify the length of the FIR filter coefficients vector as a positive integer value. This property is nontunable.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**ForgettingFactor — Least-squares lattice forgetting factor**

1 (default) | positive scalar

Specify the least-squares lattice forgetting factor as a scalar positive numeric value less than or equal to 1. Setting this value to 1 denotes infinite memory during adaptation.

**Tunable:** Yes

**Dependencies**

This property applies only if the Method property is set to 'Least-squares Lattice' or 'QR-decomposition Least-squares Lattice'.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**StepSize — Joint process step size of gradient adaptive filter**

0.1 (default) | positive scalar

Specify the joint process step size of the gradient adaptive lattice filter as a positive numeric scalar less than or equal to 1.

**Tunable:** Yes

**Dependencies**

This property applies only if the Method property is set to 'Gradient Adaptive Lattice'.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical

**Offset — Offset for denominator of StepSize normalization term**

1 (default) | nonnegative scalar

Specify an offset value for the denominator of the StepSize normalization term as a nonnegative numeric scalar. A nonzero offset helps avoid a divide-by-near-zero condition when the input signal amplitude is very small.

**Tunable:** Yes

**Dependencies**

This property applies only if the `Method` property is set to 'Gradient Adaptive Lattice'.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**ReflectionStepSize — Reflection process step size**

`StepSize` property value (default) | positive scalar

Specify the reflection process step size of the gradient adaptive lattice filter as a scalar numeric value between 0 and 1, both inclusive. The default value is the `StepSize` property value.

**Tunable:** Yes

**Dependencies**

Use this property only if the `Method` property is set to 'Gradient Adaptive Lattice'.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

**AveragingFactor — Averaging factor of energy estimator**

`1 - StepSize` (default) | positive scalar

Specify the averaging factor as a positive numeric scalar less than 1. Use this property to compute the exponentially windowed forward and backward prediction error powers for the coefficient updates. The default is the value of `1 - StepSize`.

**Tunable:** Yes

**Dependencies**

This property applies only if the `Method` property is set to 'Gradient Adaptive Lattice'.

Data Types: `single` | `double`

**InitialPredictionErrorPower — Initial prediction error power**

`1.0` (default) | positive scalar

Specify the initial values for the prediction error vectors as a scalar positive numeric value.

If the `Method` property is set to 'Least-squares Lattice' or 'QR-decomposition Least-squares Lattice', the default value is `1.0`. If the `Method` property is set to 'Gradient Adaptive Lattice', the default value is `0.1`.

**Tunable:** Yes

Data Types: `single` | `double`

**InitialCoefficients — Initial coefficients of filter**

`0` (default) | scalar | vector

Specify the initial values of the FIR adaptive filter coefficients as a scalar or a vector of length equal to the value of the `Length` property.

**Tunable:** Yes

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**LockCoefficients — Locked status of coefficient updates**`false (default) | true`

Specify whether to lock the filter coefficient values. By default, the value of this property is `false`, and the object continuously updates the filter coefficients. If this property is set to `true`, the filter coefficients do not update and their values remain the same.

**Tunable:** Yes

**Dependencies**

This property is applicable only if the `Method` property is set to `'Gradient Adaptive Lattice'`.

**Usage****Syntax**`[y,err] = alf(x,d)`**Description**

`[y,err] = alf(x,d)` filters the input `x`, using `d` as the desired signal, and returns the filtered output in `y` and the filter error in `err`. The System object estimates the filter weights needed to minimize the error between the output signal and the desired signal. You can access these coefficients by accessing the `Coefficients` property of the object. This can be done only after calling the object. For example, to access the optimized coefficients of the `alf` filter, call `alf.Coefficients` after you pass the input and desired signal to the object.

**Input Arguments****x — Data input**`scalar | column vector`

The signal to be filtered by the adaptive lattice filter. The input, `x`, and the desired signal, `d`, must have the same size and data type.

The input can be a variable-size signal. You can change the number of elements in the column vector even when the object is locked. The System object locks when you call the object.

Data Types: `single` | `double`

**d — Desired signal**`scalar | column vector`

The adaptive lattice filter adapts its coefficients to minimize the error, `err`, and converge the input signal `x` to the desired signal `d` as closely as possible.

The input, `x`, and the desired signal, `d`, must have the same size and data type.

The desired signal can be a variable-size signal. You can change the number of elements in the column vector even when the object is locked. The System object locks when you call the object.

Data Types: `single` | `double`

Complex Number Support: Yes



## Output Arguments

### **y** — Filtered output

scalar | column vector

Filtered output, returned as a scalar or a column vector. The object adapts its filter coefficients to converge the input signal  $x$  to match the desired signal  $d$ . The filter outputs the converged signal.

Data Types: `single` | `double`

### **err** — Difference between output and desired signal

scalar | column vector

Difference between the output signal  $y$  and the desired signal  $d$ , returned as a scalar or a column vector. The objective of the adaptive lattice filter is to minimize this error. The object adapts its coefficients to converge towards optimal filter coefficients that produce an output signal that matches closely with the desired signal. To access the adaptive lattice filter coefficients, call `alf.Coefficients` after you pass the input and desired signal to the object algorithm.

Data Types: `single` | `double`

Complex Number Support: Yes

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Specific to dsp.AdaptiveLatticeFilter

`mseSim` Estimated mean squared error for adaptive filters

## Common to All System Objects

`step` Run System object algorithm

`release` Release resources and allow changes to System object property values and input characteristics

`reset` Reset internal states of System object

## Examples

### QPSK Adaptive Equalization Using Adaptive Lattice Filter

Create the QPSK signal and the noise, filter them to obtain the received signal, and delay the received signal to obtain the desired signal.

```
D = 16;
b = exp(1i*pi/4)*[-0.7 1];
a = [1 -0.7];
ntr = 1000;
s = sign(randn(1,ntr+D)) + 1i*sign(randn(1,ntr+D));
n = 0.1*(randn(1,ntr+D) + 1i*randn(1,ntr+D));
r = filter(b,a,s) + n;
```

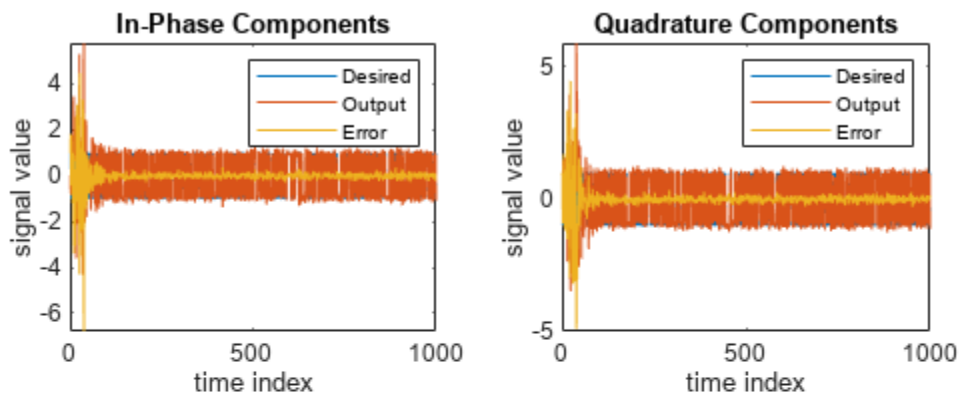
```
x = r(1+D:ntr+D);
d = s(1:ntr);
```

Use the Adaptive Lattice Filter to compute the filtered output and the filter error for the input and desired signal.

```
lam = 0.995;
del = 1;
alf = dsp.AdaptiveLatticeFilter('Length', 32, ...
    'ForgettingFactor', lam, 'InitialPredictionErrorPower', del);
[y,e] = alf(x,d);
```

Plot the In-Phase and the Quadrature components of the desired, output, and the error signals.

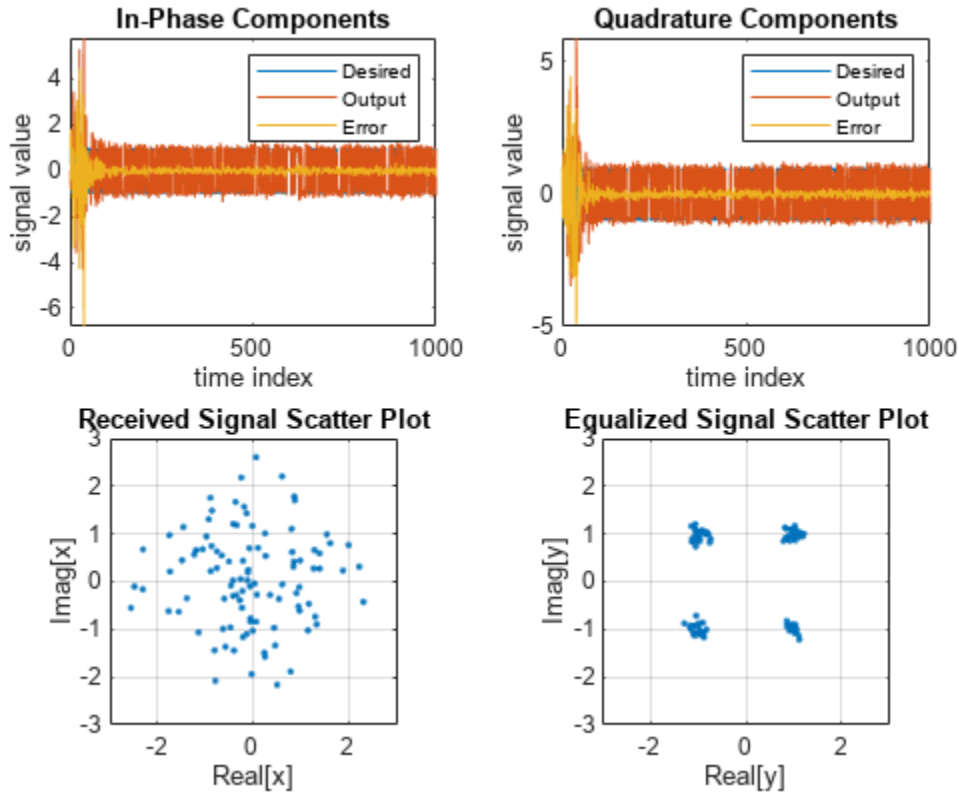
```
subplot(2,2,1); plot(1:ntr,real([d;y;e]));
title('In-Phase Components');
legend('Desired','Output','Error');
xlabel('time index'); ylabel('signal value');
subplot(2,2,2); plot(1:ntr,imag([d;y;e]));
title('Quadrature Components');
legend('Desired','Output','Error');
xlabel('time index'); ylabel('signal value');
```



Plot the received and equalized signals' scatter plots.

```
subplot(2,2,3); plot(x(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Received Signal Scatter Plot'); axis('square');
xlabel('Real[x]'); ylabel('Imag[x]'); grid on;
```

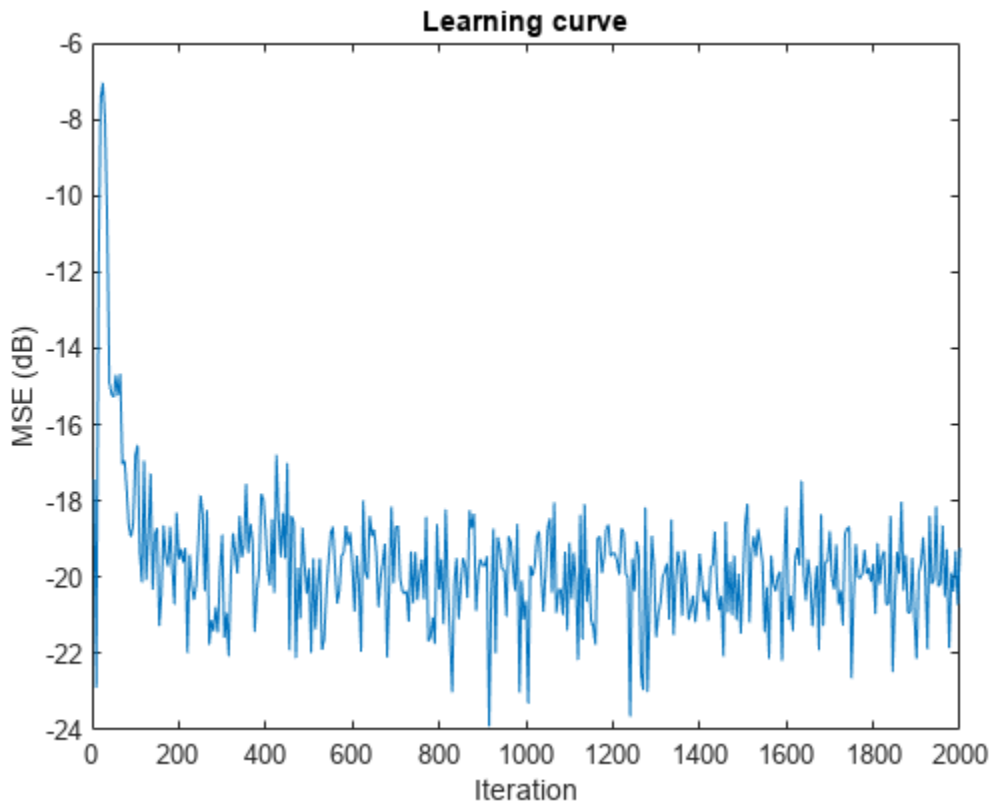
```
subplot(2,2,4); plot(y(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Equalized Signal Scatter Plot'); axis('square');
xlabel('Real[y]'); ylabel('Imag[y]'); grid on;
```



### System Identification of FIR Filter Using Adaptive Lattice Filter

```
ha = fir1(31,0.5);
% FIR system to be identified
fir = dsp.FIRFilter('Numerator',ha);
iir = dsp.IIRFilter('Numerator',sqrt(0.75),...
'Denominator',[1 -0.5]);
x = iir(sign(randn(2000,25)));
% Observation noise signal
n = 0.1*randn(size(x));
% Desired signal
d = fir(x)+n;
% Filter length
l = 32;
% Decimation factor for analysis
% and simulation results
m = 5;
ha = dsp.AdaptiveLatticeFilter(l);
[simmse,meanWsim,Wsim,traceKsim] = msesim(ha,x,d,m);
plot(m*(1:length(simmse)),10*log10(simmse));
xlabel('Iteration');
```

```
ylabel('MSE (dB)');  
% Plot the learning curve used for  
% adaptive lattice filter used in system identification  
title('Learning curve')
```



## Version History

Introduced in R2013b

## References

- [1] Griffiths, Lloyd J. "A Continuously Adaptive Filter Implemented as a Lattice Structure". *Proceedings of IEEE Int. Conf. on Acoustics, Speech, and Signal Processing*, Hartford, CT, pp. 683-686, 1977 .
- [2] Haykin, S. *Adaptive Filter Theory*, 4th Ed. Upper Saddle River, NJ: Prentice Hall, 1996.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

## **See Also**

### **Objects**

`dsp.LMSFilter` | `dsp.RLSFilter` | `dsp.AffineProjectionFilter` |  
`dsp.FrequencyDomainAdaptiveFilter` | `dsp.FilteredXLMSFilter` | `dsp.FIRFilter`

# dsp.AffineProjectionFilter

**Package:** dsp

Compute output, error and coefficients using affine projection (AP) Algorithm

## Description

The `dsp.AffineProjectionFilter` System object filters each channel of the input using AP filter implementations.

To filter each channel of the input:

- 1 Create the `dsp.AffineProjectionFilter` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
apf = dsp.AffineProjectionFilter
apf = dsp.AffineProjectionFilter(len)
apf = dsp.AffineProjectionFilter(Name,Value)
```

### Description

`apf = dsp.AffineProjectionFilter` returns an adaptive FIR filter System object, `apf`. This System object computes the filtered output and the filter error for a given input and desired signal using the affine projection (AP) algorithm.

`apf = dsp.AffineProjectionFilter(len)` returns an affine projection filter object with the `Length` property set to `len`.

`apf = dsp.AffineProjectionFilter(Name,Value)` returns an affine projection filter object with each specified property set to the specified value. Enclose each property name in single quotes. Unspecified properties have default values.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

**Method — Method to calculate filter coefficients**

Direct Matrix Inversion (default) | Recursive Matrix Update | Block Direct Matrix Inversion

Specify the method used to calculate filter coefficients as Direct Matrix Inversion, Recursive Matrix Update, Block Direct Matrix Inversion. This property is nontunable.

**Length — Length of filter coefficients vector**

32 (default) | positive integer

Specify the length of the FIR filter coefficients vector as a scalar positive integer value. This property is nontunable.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**ProjectionOrder — Projection order of affine projection algorithm**

2 (default) | positive integer

Specify the projection order of the affine projection algorithm as a scalar positive integer value greater than or equal to 2. This property defines the size of the input signal covariance matrix. This property is nontunable.

Data Types: double

**StepSize — Affine projection step size**

1 (default) | nonnegative scalar

Specify the affine projection step size factor as a scalar nonnegative numeric value between 0 and 1, both inclusive. Setting the step size equal to one provides the fastest convergence during adaptation.

**Tunable:** Yes

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical

**InitialCoefficients — Initial coefficients of filter**

0 (default) | scalar | vector

Specify the initial values of the FIR adaptive filter coefficients as a scalar or a vector of length equal to the Length property value.

**Tunable:** Yes

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**InitialOffsetCovariance — Initial values of offset input covariance matrix**

1 (default) | scalar | square matrix

Specify the initial values for the offset input covariance matrix. This property must be either a scalar positive numeric value or a positive-definite square matrix with each dimension equal to the ProjectionOrder property value. If it is a scalar value, the OffsetCovariance property is initialized to a diagonal matrix with the diagonal elements equal to that scalar value. If it is a square matrix, the OffsetCovariance property is initialized to the value of that square matrix.

**Tunable:** Yes

**Dependencies**

This property is applicable only if the `Method` property is set to `Direct Matrix Inversion` or `Block Direct Matrix Inversion`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**InitialInverseOffsetCovariance — Initial values of offset input covariance matrix inverse**

20 (default) | scalar | square matrix

Specify the initial values for the offset input covariance matrix inverse. This property must be either a scalar positive numeric value or a positive-definite square matrix with each dimension equal to the `ProjectionOrder` property value. If it is a scalar value, the `InverseOffsetCovariance` property is initialized to a diagonal matrix with each diagonal element equal to that scalar value. If it is a square matrix, the `InverseOffsetCovariance` property is initialized to the values of that square matrix.

**Tunable:** Yes

**Dependencies**

This property is applicable only if the `Method` property is set to `Recursive Matrix Update`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**InitialCorrelationCoefficients — Initial correlation coefficients**

0 (default) | scalar | vector

Specify the initial values of the correlation coefficients of the FIR filter as a scalar or a vector of length equal to `ProjectionOrder - 1`.

**Tunable:** Yes

**Dependencies**

This property is applicable only if the `Method` property is set to `Recursive Matrix Update`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**LockCoefficients — Lock coefficient updates**

false (default) | true

Specify whether the filter coefficient values should be locked. When you set this property to `true`, the filter coefficients are not updated and their values remain the same. The default value is `false` (filter coefficients are continuously updated).

**Tunable:** Yes

**Usage****Syntax**

```
[y,err] = apf(x,d)
```



## Description

`[y,err] = apf(x,d)` filters the input `x`, using `d` as the desired signal, and returns the filtered output in `y` and the filter error in `err`. The System object estimates the filter weights needed to minimize the error between the output signal and the desired signal. You can access these coefficients by accessing the `Coefficients` property of the object. This can be done only after calling the object. For example, to access the optimized coefficients of the `apf` filter, call `apf.Coefficients` after you pass the input and desired signal to the object.

## Input Arguments

### **x** — Data input

scalar | column vector

The signal to be filtered by the affine projection filter. The input, `x`, and the desired signal, `d`, must have the same size and data type.

The input can be a variable-size signal. You can change the number of elements in the column vector even when the object is locked. The System object locks when you call the object to run its algorithm.

Data Types: `single` | `double`

### **d** — Desired signal

scalar | column vector

The affine projection filter adapts its coefficients to minimize the error, `err`, and converge the input signal `x` to the desired signal `d` as closely as possible.

The input, `x`, and the desired signal, `d`, must have the same size and data type.

The desired signal can be a variable-size signal. You can change the number of elements in the column vector even when the object is locked. The System object locks when you call the object.

Data Types: `single` | `double`

Complex Number Support: Yes

## Output Arguments

### **y** — Filtered output

scalar | column vector

Filtered output, returned as a scalar or a column vector. The object adapts its filter coefficients to converge the input signal `x` to match the desired signal `d`. The filter outputs the converged signal.

Data Types: `single` | `double`

### **err** — Difference between output and desired signal

scalar | column vector

Difference between the output signal `y` and the desired signal `d`, returned as a scalar or a column vector. The objective of the affine projection filter is to minimize this error. The object adapts its coefficients to converge towards optimal filter coefficients that produce an output signal that matches closely with the desired signal. To access the affine projection filter coefficients, call `apf.Coefficients` after you pass the input and desired signal to the object.

Data Types: `single` | `double`

Complex Number Support: Yes

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Specific to `dsp.AffineProjectionFilter`

`mse` Estimated mean squared error for adaptive filters

## Common to All System Objects

`step` Run System object algorithm

`release` Release resources and allow changes to System object property values and input characteristics

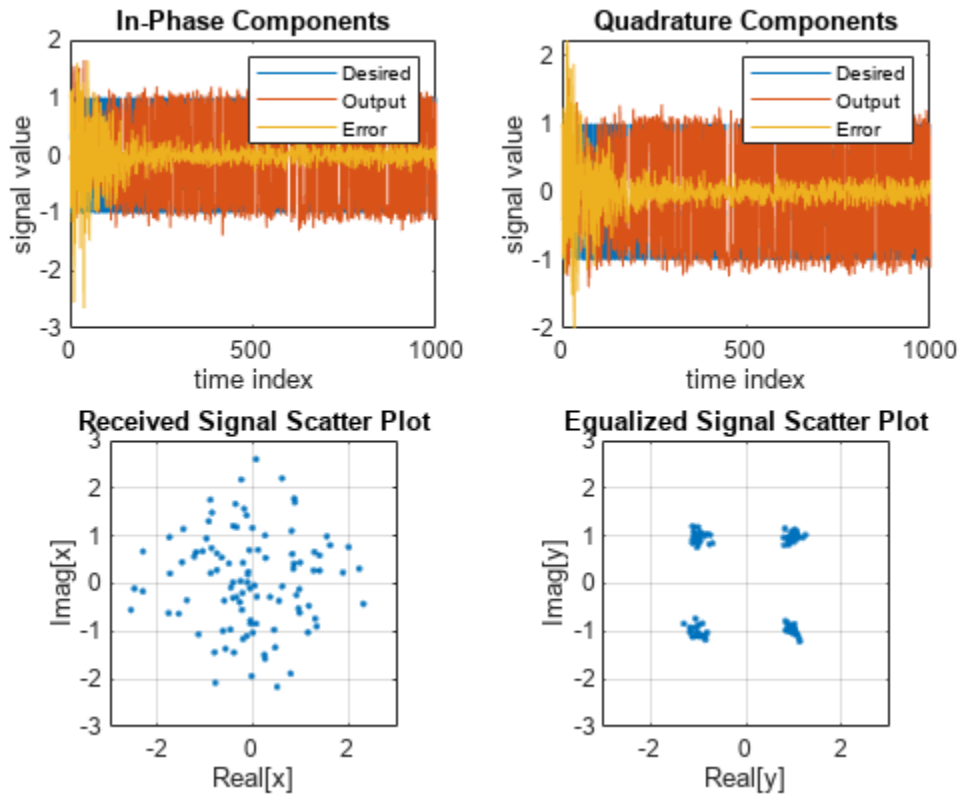
`reset` Reset internal states of System object

## Examples

### QPSK Adaptive Equalization

QPSK Adaptive Equalization Using a 32-Coefficient FIR Filter (1000 Iterations)

```
D = 16; % Number of samples of delay
b = exp(1i*pi/4)*[-0.7 1]; % Numerator coefficients of channel
a = [1 -0.7]; % Denominator coefficients of channel
ntr = 1000; % Number of iterations
s = sign(randn(1,ntr+D)) + 1i*sign(randn(1,ntr+D)); % Baseband signal
n = 0.1*(randn(1,ntr+D)) + 1i*randn(1,ntr+D); % Noise signal
r = filter(b,a,s)+n; % Received signal
x = r(1+D:ntr+D); % Input signal (received signal)
d = s(1:ntr); % Desired signal (delayed QPSK signal)
mu = 0.1; % Step size
po = 4; % Projection order
offset = 0.05; % Offset for covariance matrix
apf = dsp.AffineProjectionFilter('Length', 32, ...
    'StepSize', mu, 'ProjectionOrder', po, ...
    'InitialOffsetCovariance', offset);
[y,e] = apf(x,d);
subplot(2,2,1); plot(1:ntr,real([d;y;e]));
title('In-Phase Components');
legend('Desired','Output','Error');
xlabel('time index'); ylabel('signal value');
subplot(2,2,2); plot(1:ntr,imag([d;y;e]));
title('Quadrature Components');
legend('Desired','Output','Error');
xlabel('time index'); ylabel('signal value');
subplot(2,2,3); plot(x(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Received Signal Scatter Plot'); axis('square');
xlabel('Real[x]'); ylabel('Imag[x]'); grid on;
subplot(2,2,4); plot(y(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Equalized Signal Scatter Plot'); axis('square');
xlabel('Real[y]'); ylabel('Imag[y]'); grid on;
```



### System Identification of FIR Filter Using Affine Projection Filter

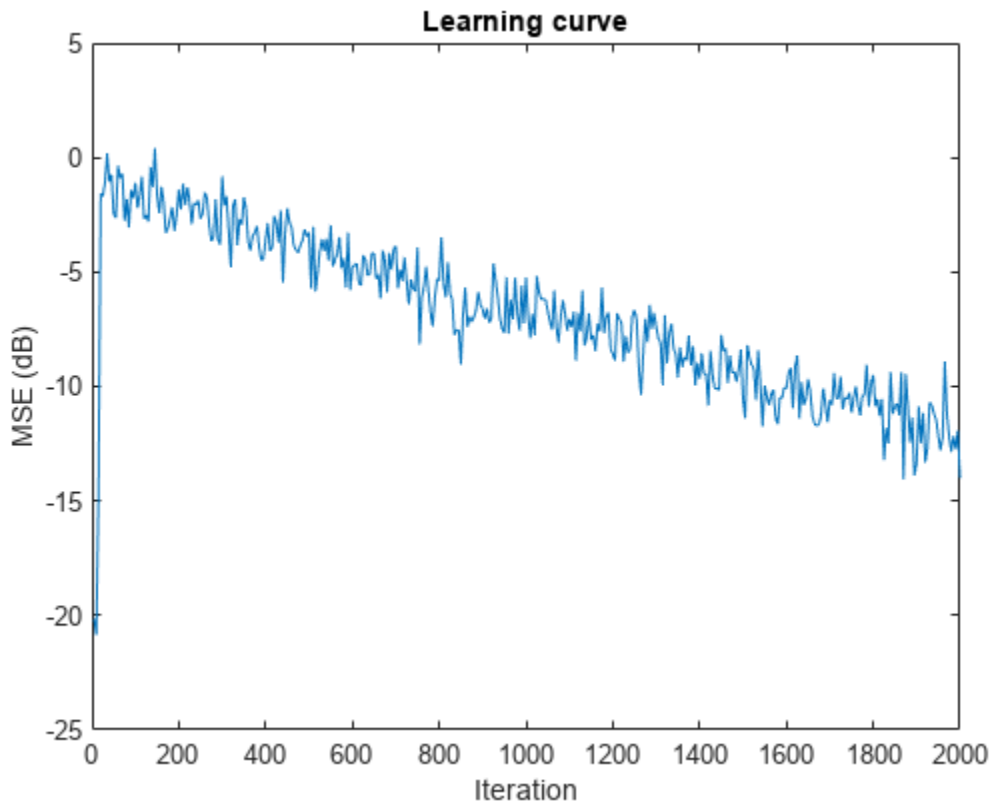
```

ha = fir1(31,0.5);
% FIR system to be identified
fir = dsp.FIRFilter('Numerator',ha);
iir = dsp.IIRFilter('Numerator',sqrt(0.75),...
    'Denominator',[1 -0.5]);
x = iir(sign(randn(2000,25)));
% Observation noise signal
n = 0.1*randn(size(x));
% Desired signal
d = fir(x)+n;
% Filter length
l = 32;
% Affine Projection filter Step size.
mu = 0.008;
% Decimation factor for analysis
% and simulation results
m = 5;

apf = dsp.AffineProjectionFilter(l,'StepSize',mu);
[simmse,meanWsim,Wsim,traceKsim] = msesim(apf,x,d,m);
plot(m*(1:length(simmse)),10*log10(simmse));
xlabel('Iteration'); ylabel('MSE (dB)');
% Plot the learning curve for affine projection filter

```

```
% used in system identification  
title('Learning curve')
```



## Algorithms

The affine projection algorithm (APA) is an adaptive scheme that estimates an unknown system based on multiple input vectors [1]. It is designed to improve the performance of other adaptive algorithms, mainly those that are LMS based. The affine projection algorithm reuses old data resulting in fast convergence when the input signal is highly correlated, leading to a family of algorithms that can make trade-offs between computation complexity with convergence speed [2].

The following equations describe the conceptual algorithm used in designing AP filters:

$$\mathbf{U}ap(n) = \begin{pmatrix} u(n) & \dots & u(n-L) \\ \vdots & \ddots & \vdots \\ u(n-N) & \dots & u(n-L-N) \end{pmatrix} = (\mathbf{u}(n) \ \mathbf{u}(n-1) \ \dots \ \mathbf{u}(n-L))$$

$$\mathbf{y}ap(n) = \mathbf{U}^T ap(n) \mathbf{w}(n) = \begin{pmatrix} y(n) \\ \cdot \\ \cdot \\ \cdot \\ y(n-L) \end{pmatrix}$$

$$\mathbf{d}ap(n) = \begin{pmatrix} d(n) \\ \cdot \\ \cdot \\ \cdot \\ d(n-L) \end{pmatrix}$$

$$\mathbf{e}ap(n) = \mathbf{d}ap(n) - \mathbf{y}ap(n) = \begin{pmatrix} e(n) \\ \cdot \\ \cdot \\ \cdot \\ e(n-L) \end{pmatrix}$$

$$\mathbf{w}(n) = \mathbf{w}(n-1) + \mu \mathbf{U}ap(n) \mathbf{e}ap$$

where  $\mathbf{C}$  is either  $\varepsilon \mathbf{I}$  if the initial offset covariance is a scalar  $\varepsilon$ , or  $\mathbf{R}$  if the initial offset covariance is a matrix  $\mathbf{R}$ . The variables are as follows:

| Variable             | Description                                       |
|----------------------|---|
| $n$                  | The current time index                            |
| $u(n)$               | The input sample at step $n$                      |
| $\mathbf{U}_{ap}(n)$ | The matrix of the last $L+1$ input signal vectors |
| $\mathbf{w}(n)$      | The adaptive filter coefficients vector           |
| $y(n)$               | The adaptive filter output                        |
| $d(n)$               | The desired signal                                |
| $e(n)$               | The error at step $n$                             |
| $L$                  | The projection order                              |
| $N$                  | The filter order (i.e., filter length = $N+1$ )   |
| $\mu$                | The step size                                     |

## Version History

Introduced in R2013a

## References

- [1] K. Ozeki, T. Umeda, "An adaptive Filtering Algorithm Using an Orthogonal Projection to an Affine Subspace and its Properties", *Electron. Commun. Jpn.* 67-A(5), May 1984, pp. 19-27.

[2] Paulo S. R. Diniz, *Adaptive Filtering: Algorithms and Practical Implementation*, Second Edition. Boston: Kluwer Academic Publishers, 2002.

### **Extended Capabilities**

#### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

### **See Also**

#### **Objects**

`dsp.FIRFilter` | `dsp.LMSFilter` | `dsp.RLSFilter`

# dsp.AllpassFilter

**Package:** dsp

Single section or cascaded allpass filter

## Description

The `dsp.AllpassFilter` object filters each channel of the input using allpass filter implementations. To import this object into Simulink, use the MATLAB System block.

To filter each channel of the input:

- 1 Create the `dsp.AllpassFilter` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
Allpass = dsp.AllpassFilter
Allpass = dsp.AllpassFilter(Name,Value)
```

### Description

`Allpass = dsp.AllpassFilter` returns an allpass filter System object, `Allpass`, that filters each channel of the input signal independently using an allpass filter, with the default structure and coefficients.

`Allpass = dsp.AllpassFilter(Name,Value)` returns an allpass filter System object, `Allpass`, with each property set to the specified value.

### Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

### Structure — Internal allpass filter structure

Minimum multiplier (default) | Lattice | Wave Digital Filter

You can specify the internal allpass filter implementation structure as one of `Minimum multiplier` | `Lattice` | `Wave Digital Filter`. Each structure uses a different set of coefficients, independently stored in the corresponding object property.

**AllpassCoefficients — Allpass polynomial coefficients**`[-2(-1/2) 0.5] (default) | N-by-1 | N-by-2`

Specify the real allpass polynomial filter coefficients. Specify this property as either an *N*-by-1 or *N*-by-2 matrix of *N* first-order or second-order allpass sections. The default value defines a stable second-order allpass filter with poles and zeros located at  $\pm\pi/3$  in the Z plane.

**Tunable:** Yes

**Dependencies**

This property is applicable only when the `Structure` property is set to `Minimum multiplier`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**WDFCoefficients — Wave Digital Filter allpass coefficients**`[1/2, -2(1/2)/3] (default) | N-by-1 | N-by-2`

Specify the real allpass coefficients in the Wave Digital Filter form. Specify this property as either a *N*-by-1 or *N*-by-2 matrix of *N* first-order or second-order allpass sections. All elements must have absolute values less than or equal to 1. This value is a transformed version of the default value of `AllpassCoefficients`, computed using `allpass2wdf(AllpassCoefficients)`. These coefficients define the same stable second-order allpass filter as when `Structure` is set to `'Minimum multiplier'`.

**Tunable:** Yes

**Dependencies**

This property is only applicable when the `Structure` property is set to `Wave Digital Filter`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**LatticeCoefficients — Lattice allpass coefficients**`[-2(1/2)/3, 1/2] (default) | vector`

Specify the real or complex allpass coefficients as lattice reflection coefficients. Specify this property as either a row vector (single-section configuration) or a column vector. This value is a transformed and transposed version of the default value of `AllpassCoefficients`, computed using `transpose(tf2latc([1 h.AllpassCoefficients]))`. These coefficients define the same stable second-order allpass filter as when `Structure` is set to `'Lattice'`.

**Tunable:** Yes

**Dependencies**

This property is applicable only if the `Structure` property is set to `Lattice`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**TrailingFirstOrderSection — Indicate if last section is first order**`false (default) | true`

Indicate if last section is first order or second order. When you set `TrailingFirstOrderSection` to `true`, the last section is considered to be first-order, and the second element of the last row of the *N*-by-2 matrix is ignored. When you set `TrailingFirstOrderSection` to `false`, the last section is considered to be second-order.



## Usage

### Syntax

```
y = Allpass(x)
```

### Description

`y = Allpass(x)` filters the input signal `x` using an allpass filter to produce the output `y`. Each column of `x` is filtered independently as a separate channel over time.

### Input Arguments

#### **x** — Data input

vector | matrix

Data input, specified as a vector or a matrix. This object also accepts variable-size inputs. Once the object is locked, you can change the size of each input channel, but you cannot change the number of channels.

Data Types: `single` | `double`

### Output Arguments

#### **y** — Filtered output

vector | matrix

Filtered output, returned as a vector or a matrix. The size, data type, and complexity of the output signal matches that of the input signal.

Data Types: `double` | `single`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

### Specific to `dsp.AllpassFilter`

|                       |  |
|-----------------------|--|
| <code>freqz</code>    | Frequency response of discrete-time filter System object     |
| <code>fvtool</code>   | Visualize frequency response of DSP filters                  |
| <code>impz</code>     | Impulse response of discrete-time filter System object       |
| <code>info</code>     | Information about filter System object                       |
| <code>coeffs</code>   | Returns the filter System object coefficients in a structure |
| <code>cost</code>     | Estimate cost of implementing filter System object           |
| <code>grpdelay</code> | Group delay response of discrete-time filter System object   |

### Common to All System Objects

|                      |  |
|----------------------|--|
| <code>step</code>    | Run System object algorithm  |
| <code>release</code> | Release resources and allow changes to System object property values and input characteristics |

reset     Reset internal states of System object

## Examples

### Lowpass Filtering using Two Allpass Filters

Construct the Allpass Filters

```
Fs = 48000;     % in Hz
FL = 1024;
APF1 = dsp.AllpassFilter('AllpassCoefficients',...
    [-0.710525516540603   0.208818210000029]);
APF2 = dsp.AllpassFilter('AllpassCoefficients',...
    [-0.940456403667957   0.6;...
    -0.324919696232907   0],...
    'TrailingFirstOrderSection',true);
```

Construct the Transfer Function Estimator to estimate the transfer function between the random input and the Allpass filtered output

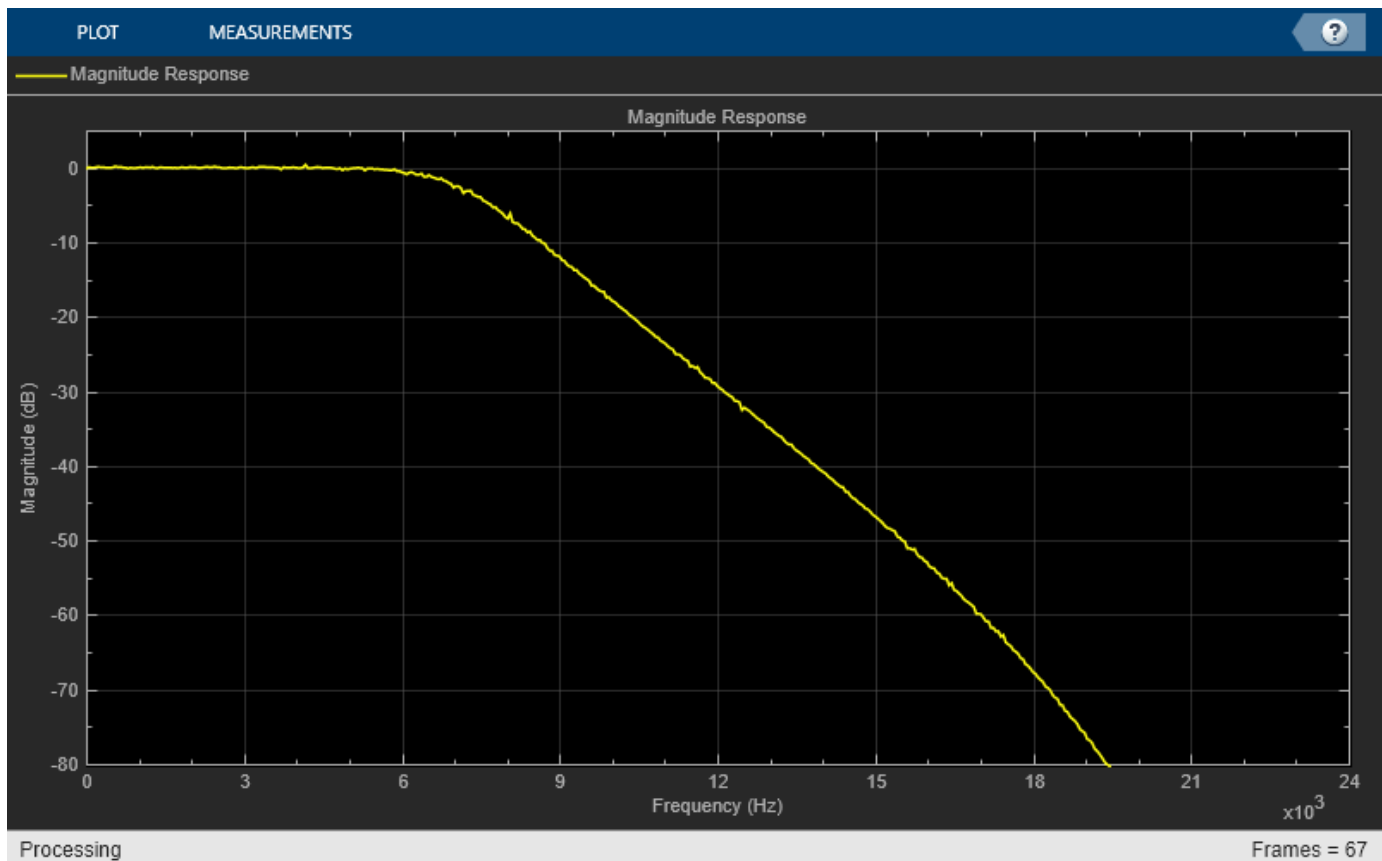
```
TFE = dsp.TransferFunctionEstimator('FrequencyRange',...
    'onesided','SpectralAverages',2);
```

Construct the ArrayPlot to plot the magnitude response

```
AP = dsp.ArrayPlot('PlotType','Line','YLimits', [-80 5],...
    'YLabel','Magnitude (dB)','SampleIncrement', Fs/FL,...
    'XLabel','Frequency (Hz)','Title','Magnitude Response',...
    'ShowLegend', true,'ChannelNames',{'Magnitude Response'});
```

Filter the Input and show the magnitude response of the estimated transfer function between the input and the filtered output

```
tic;
while toc < 5
    in = randn(FL,1);
    out = 0.5.*(APF1(in) + APF2(in));
    A = TFE(in, out);
    AP(db(A));
end
```



## Algorithms

The transfer function of an allpass filter is given by

$$H(z) = \frac{c(n) + c(n-1)z^{-1} + \dots + z^{-n}}{1 + c(1)z^{-1} + \dots + c(n)z^{-n}}$$

$c$  is allpass polynomial coefficients vector. The order,  $n$ , of the transfer function is the length of vector  $c$ .

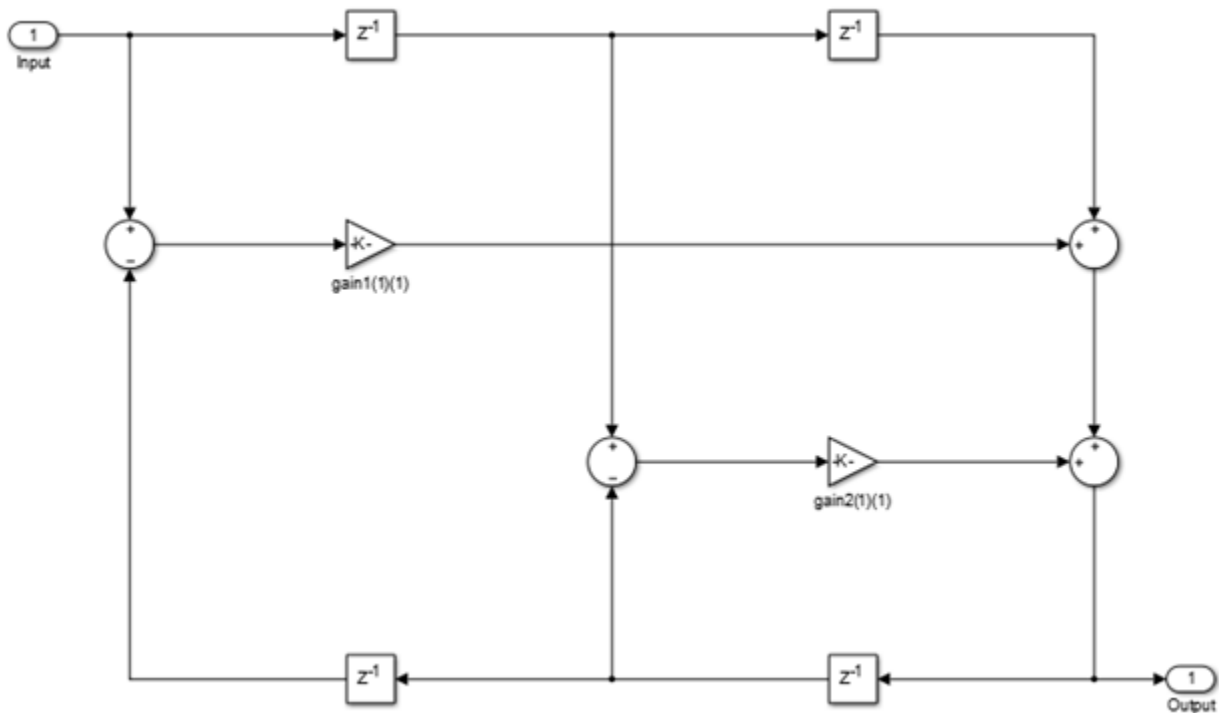
In the minimum multiplier form and wave digital form, the allpass filter is implemented as a cascade of either second-order (biquad) sections or first-order sections. When the coefficients are specified as an  $N$ -by-2 matrix, each row of the matrix specifies the coefficients of a second-order filter. The last element of the last row can be ignored based on the trailing first-order setting. When the coefficients are specified as an  $N$ -by-1 matrix, each element in the matrix specifies the coefficient of a first-order filter. The cascade of all the filter sections forms the allpass filter.

In the lattice form, the coefficients are specified as a vector.

These structures are computationally more economical and structurally more stable compared to the generic IIR filters, such as `df1`, `df1t`, `df2`, `df2t`. For all structures, the allpass filter can be a single-section or a multiple-section (cascaded) filter. The different sections can have different orders, but they are all implemented according to the same structure.

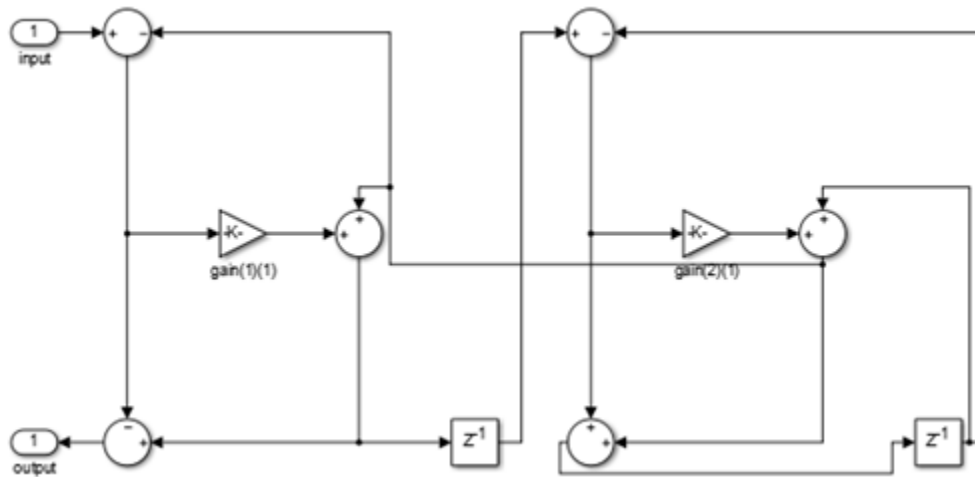
### Minimum Multiplier

This structure realizes the allpass filter with the minimum number of required multipliers, equal to the order  $n$ . It also uses  $2n$  delay units and  $2n$  adders. The multipliers use the specified coefficients, which are equal to the polynomial vector  $c$  in the allpass transfer function. In this second-order section of the minimum multiplier structure, the coefficients vector,  $c$ , is equal to  $[0.1 \ -0.7]$ .



### Wave Digital Filter

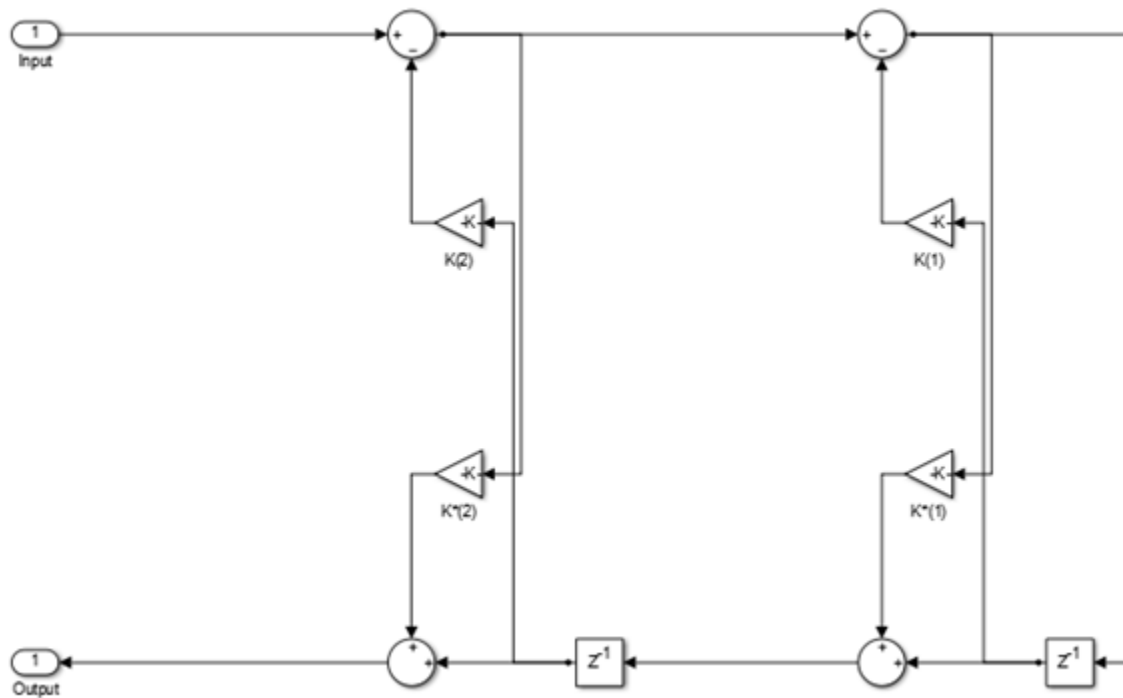
This structure uses  $n$  multipliers, but only  $n$  delay units, at the expense of requiring  $3n$  adders. To use this structure, specify the coefficients in wave digital filter (WDF) form. Obtain the WDF equivalent of the conventional allpass coefficients using `allpass2wdf(allpass_coefficients)`. To convert WDF coefficients into the equivalent allpass polynomial form, use `wdf2allpass(WDF_coefficients)`. In this second-order section of the WDF structure, the coefficients vector  $w$  is equal to `allpass2wdf([0.1 -0.7])`.



### Lattice

This lattice structure uses  $2n$  multipliers,  $n$  delay units, and  $2n$  adders. To use this structure, specify the coefficients as a vector.

You can obtain the lattice equivalent of the conventional allpass coefficients using `transpose(tf2latc(1, [1 allpass_coefficients]))`. In the following second-order section of the lattice structure, the coefficients vector is computed using `transpose(tf2latc(1, [1 0.1 -0.7]))`. Use these coefficients for a filter that is functionally equivalent to the minimum multiplier structure with coefficients `[0.1 -0.7]`.



## Version History

Introduced in R2013a

## References

- [1] Regalia, Philip A. and Mitra Sanjit K. and Vaidyanathan, P. P. (1988) "The Digital All-Pass Filter: AVersatile Signal Processing Building Block." *Proceedings of the IEEE*, Vol. 76, No. 1, 1988, pp. 19-37
- [2] M. Lutovac, D. Tasic, B. Evans, *Filter Design for Signal Processing Using MATLAB and Mathematica*. Upper Saddle River, NJ: Prentice Hall, 2001.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The System object supports code generation only when the `Structure` property is set to `Minimum multiplier` or `Lattice`.
- See "System Objects in MATLAB Code Generation" (MATLAB Coder).

## See Also

### Functions

`freqz` | `fvtool` | `impz` | `info` | `coeffs` | `cost` | `grpdelay` | `allpass2wdf`

### Objects

`dsp.SOSFilter` | `dsp.IIRFilter`

### Topics

"Analysis Methods for Filter System Objects" on page 3-2

# dsp.AllpoleFilter

**Package:** dsp

IIR Filter with no zeros

## Description

The `dsp.AllpoleFilter` object filters each channel of the input using allpole filter implementations.

To filter each channel of the input:

- 1 Create the `dsp.AllpoleFilter` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
allpole = dsp.AllpoleFilter
allpole = dsp.AllpoleFilter(Name,Value)
```

### Description

`allpole = dsp.AllpoleFilter` returns an allpole filter System object, `allpole`, which independently filters each channel of the input over successive calls to the algorithm. This System object uses a specified allpole filter implementation.

`allpole = dsp.AllpoleFilter(Name,Value)` returns an allpole filter System object, `allpole`, with each property set to the specified value.

### Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

### Structure — Filter structure

Direct form (default) | Direct form transposed | Lattice AR

Specify the filter structure as one of | `Direct form` | `Direct form transposed` | `Lattice AR`. Analysis methods are not supported for fixed-point processing if the structure is `Direct form` or `Direct form transposed`. This property is nontunable.

**Denominator — Filter denominator coefficients**

[1 0.1] (default) | row vector

Specify the denominator coefficients as a numeric row vector.

**Tunable:** Yes**Dependencies**

This property is applicable when the Structure property is set to one of `Direct form` | `Direct form transposed`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`**ReflectionCoefficients — Lattice filter coefficients**

[0.2 0.4] (default) | row vector

Specify the lattice filter coefficients as a numeric row vector.

**Tunable:** Yes**Dependencies**

This property is applicable when the Structure property is set to `Lattice AR`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`**InitialConditions — Initial conditions for the filter states**

0 (default) | scalar | vector | matrix

Specify the initial conditions of the filter states.

You can specify the initial conditions as a scalar, vector, or matrix. If you specify a scalar value, this System object initializes all delay elements in the filter to that value. You can also specify a vector whose length equals the number of delay elements in the filter. When you do so, each vector element specifies a unique initial condition for the corresponding delay element. The object applies the same vector of initial conditions to each channel of the input signal.

You can also specify a matrix with the same number of rows as the number of delay elements in the filter and one column for each channel of the input signal. In this case, each element specifies a unique initial condition for the corresponding delay element in the corresponding channel.

**Tunable:** YesData Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`**CoefficientsDataType — Denominator coefficients word- and fraction-length designations**

Same word length as input (default) | Custom

Specify the denominator coefficients fixed-point data type as one of `Same word length as input` | `Custom`. This property is nontunable.

**ReflectionCoefficientsDataType — Reflection coefficients word- and fraction-length designations**

Same word length as input (default) | Custom

Specify the reflection coefficients fixed-point data type as one of `Same word length as input` | `Custom`. This property is nontunable.



## Fixed-Point Properties

### ProductDataType — Product word- and fraction-length designations

Full precision (default) | Same as input | Custom

Specify the product fixed-point data type as one of | Full precision | Same as input | Custom |. This property is nontunable.

### AccumulatorDataType — Accumulator word- and fraction-length designations

Full precision (default) | Same as input | Same as product | Custom

Specify the accumulator fixed-point data type to one of | Full precision | Same as input | Same as product | Custom |. This property is nontunable.

### OutputDataType — Output word- and fraction-length designations

Same as input (default) | Same as accumulator | Custom

Specify the output fixed-point data type as one of | Same as accumulator | Same as input | Custom |. This property is nontunable.

### StateDataType — State word- and fraction-length designations

Same as accumulator (default) | Same as input | Custom

Specify the state fixed-point data type as one of | Same as input | Same as accumulator | Custom. This property is nontunable.

### CustomCoefficientsDataType — Custom denominator word- and fraction-lengths

numericType ([,16,15) (default) | numericType

Specify the denominator coefficients fixed-point type as an autosigned numericType (Fixed-Point Designer) object. This property is nontunable.

#### Dependencies

This property is applicable when the CoefficientsDataType property is Custom.

### CustomReflectionCoefficientsDataType — Custom reflection coefficients word- and fraction-lengths

numericType ([,16,15) (default) | numericType

Specify the denominator coefficients fixed-point type as an autosigned numericType (Fixed-Point Designer) object. This property is nontunable.

#### Dependencies

This property is applicable when the ReflectionCoefficientsDataType property is Custom.

### CustomProductDataType — Custom Product word- and fraction-lengths

numericType ([,32,30) (default) | numericType

Specify the product fixed-point type as an autosigned scaled numericType object. This property is nontunable.

#### Dependencies

This property applies when you set the ProductDataType property to Custom.

**CustomAccumulatorDataType — Custom accumulator word- and fraction-lengths**`numericType([],32,30)` (default) | `numericType`

Specify the accumulator fixed-point type as an autosigned scaled `numericType` object. This property is nontunable.

**Dependencies**

This property applies when you set the `AccumulatorDataType` property to `Custom`.

**CustomStateDataType — Custom state word- and fraction-lengths**`numericType([],16,15)` (default) | `numericType`

Specify the state fixed-point type as an autosigned scaled `numericType` object. This property is nontunable.

**Dependencies**

This property applies when you set the `StateDataType` property to `Custom`.

**CustomOutputDataType — Custom output word- and fraction-lengths**`numericType([],16,15)` (default) | `numericType`

Specify the output fixed-point type as an autosigned scaled `numericType` object. This property is nontunable.

**Dependencies**

This property applies when you set the `OutputDataType` property to `Custom`.

**Usage****Syntax**`y = allpole(x)`**Description**

`y = allpole(x)` filters the real or complex input signal `x` using an allpole filter to produce the output `y`.

**Input Arguments****x — Data input**`vector` | `matrix`

Data input, specified as a vector or a matrix. This object also accepts variable-size inputs. Once the object is locked, you can change the size of each input channel, but you cannot change the number of channels.

When the input data is of a fixed-point type, it must be signed. The allpole filter object operates on each channel of the input signal independently over successive calls to the algorithm.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fi`

## Output Arguments

### **y** — Filtered output

vector | matrix

Filtered output, returned as a vector or a matrix. The size, data type, and complexity of the output signal matches that of the input signal.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fi`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Specific to `dsp.AllpoleFilter`

|                       |  |
|-----------------------|--|
| <code>freqz</code>    | Frequency response of discrete-time filter System object         |
| <code>fvtool</code>   | Visualize frequency response of DSP filters                      |
| <code>impz</code>     | Impulse response of discrete-time filter System object           |
| <code>info</code>     | Information about filter System object                           |
| <code>coeffs</code>   | Returns the filter System object coefficients in a structure     |
| <code>cost</code>     | Estimate cost of implementing filter System object               |
| <code>grpdelay</code> | Group delay response of discrete-time filter System object       |
| <code>phasez</code>   | Phase response of discrete-time filter System object (unwrapped) |

## Common to All System Objects

|                      |  |
|----------------------|--|
| <code>step</code>    | Run System object algorithm  |
| <code>release</code> | Release resources and allow changes to System object property values and input characteristics |
| <code>reset</code>   | Reset internal states of System object   |

## Examples

### Lowpass Filter a Waveform with Two Frequencies

Use an Allpole filter to apply a lowpass filter to a waveform with two sinusoidal frequencies.

```
t = (0:1000)./8e3;
xin = sin(2*pi*1e3*t)+sin(2*pi*3e3*t);

src = dsp.SignalSource(xin', 4);
sink = dsp.SignalSink;
allpole = dsp.AllpoleFilter;
tt = (-25:25)';
xsinc = 0.4*sinc(0.4*tt);
asinc = lpc(xsinc,51);
allpole.Denominator = asinc;

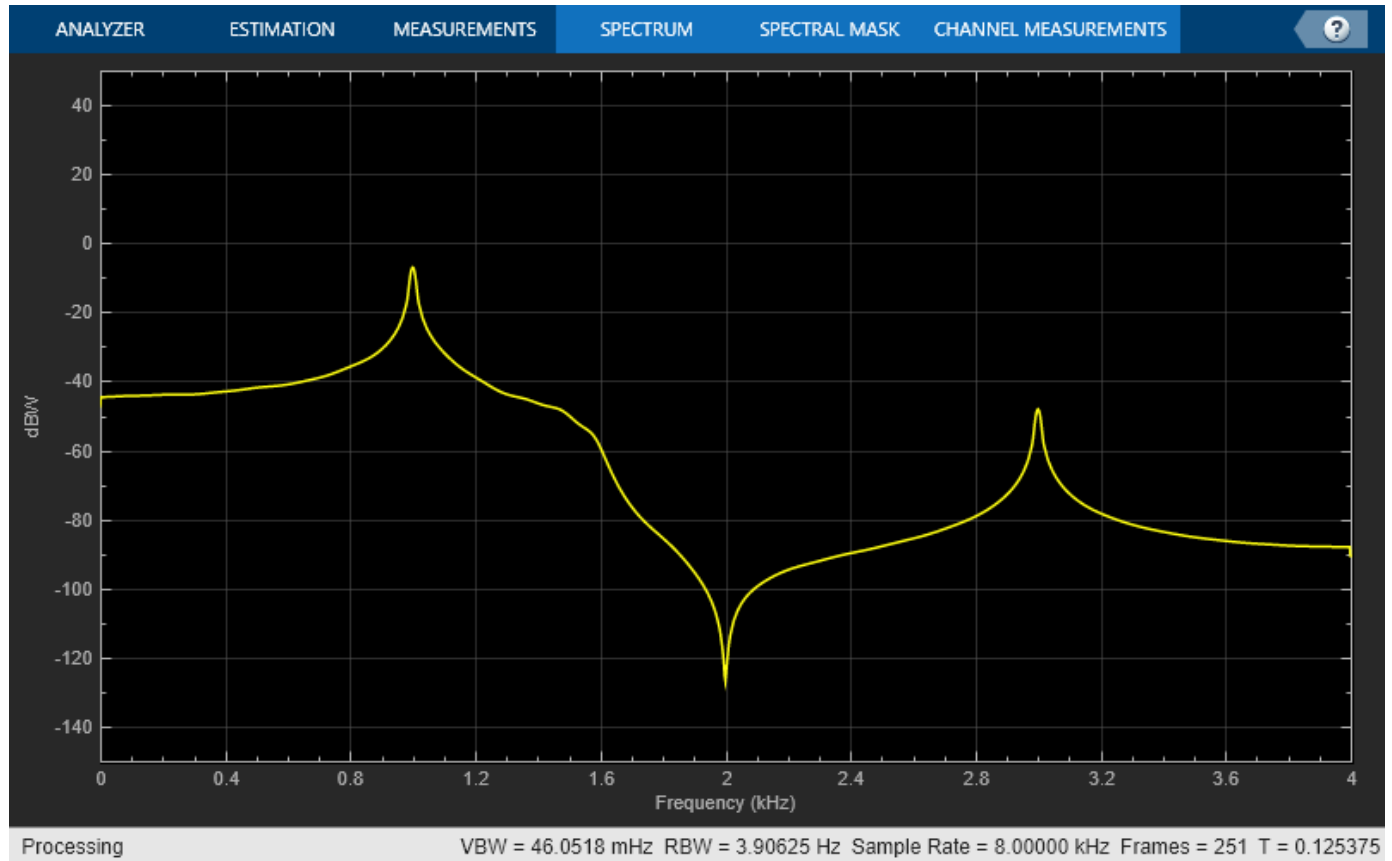
sa = spectrumAnalyzer('SampleRate',8e3,...
    'Method','welch',...
```

```

        'PlotAsTwoSidedSpectrum',false,...
        'OverlapPercent',80,'SpectrumUnits','dBW',...
        'YLimits',[-150 50]);

while ~isDone(src)
    input = src();
    filteredOutput = allpole(input);
    sink(filteredOutput);
    sa(filteredOutput)
end

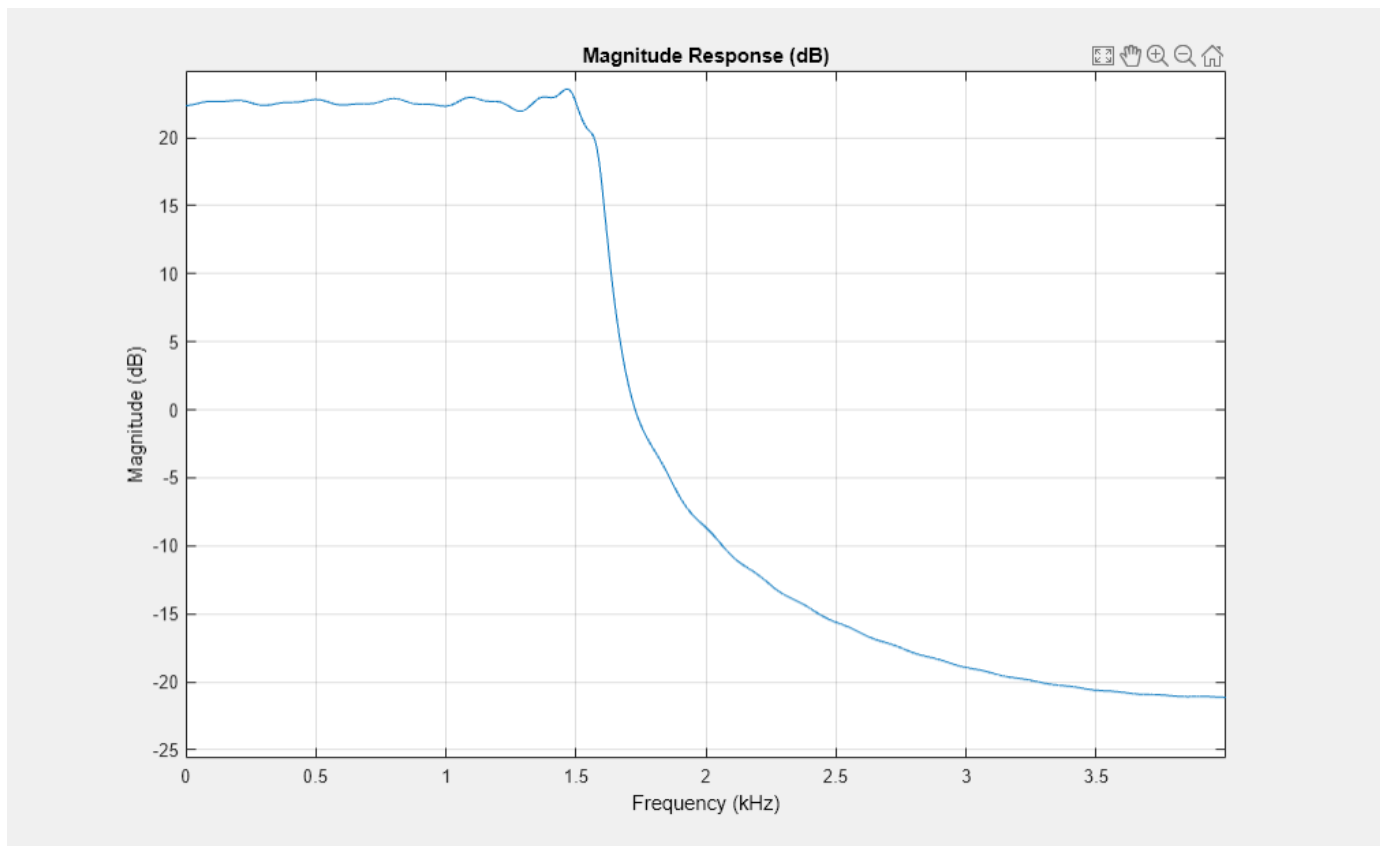
```



```

filteredResult = sink.Buffer;
fvtool(allpole,'Fs',8000)

```



## Algorithms

This object implements the algorithm, inputs, and outputs described on the Allpole Filter block reference page. The object properties correspond to the block parameters.

## Version History

**Introduced in R2012b**

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Only the Denominator property is tunable for code generation.
- See “System Objects in MATLAB Code Generation” (MATLAB Coder).

## **See Also**

### **Functions**

freqz | fvtool | impz | info | coeffs | cost | grpdelay

### **Objects**

dsp.SOSFilter | dsp.FIRFilter | dsp.IIRFilter

### **Topics**

“Analysis Methods for Filter System Objects” on page 3-2

# dsp.AnalyticSignal

**Package:** dsp

Analytic signals of discrete-time inputs

## Description

The `dsp.AnalyticSignal` System object computes analytic signals of discrete-time inputs. The real part of the analytic signal in each channel is a replica of the real input in that channel, and the imaginary part is the Hilbert transform of the input. In the frequency domain, the analytic signal doubles the positive frequency content of the original signal while zeroing-out negative frequencies and retaining the DC component. The object computes the Hilbert transform using an equiripple FIR filter.

To compute the analytic signal of a discrete-time input:

- 1 Create the `dsp.AnalyticSignal` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
anaSig = dsp.AnalyticSignal
anaSig = dsp.AnalyticSignal(order)
anaSig = dsp.AnalyticSignal(Name,Value)
```

### Description

`anaSig = dsp.AnalyticSignal` returns an analytic signal object, `anaSig`, that computes the complex analytic signal corresponding to each channel of a real  $M$ -by- $N$  input matrix.

`anaSig = dsp.AnalyticSignal(order)` returns an analytic signal object, `anaSig`, with the "FilterOrder" on page 4-0 property set to `order`.

`anaSig = dsp.AnalyticSignal(Name,Value)` returns an analytic signal object, `anaSig`, with each specified property set to the specified value.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

**FilterOrder** — Filter order used to compute Hilbert transform

100 (default) | scalar integer

Order of the equiripple FIR filter used in computing the Hilbert transform, specified as an even integer scalar greater than 3.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**Usage****Syntax**

```
y = anaSig(x)
```

**Description**

`y = anaSig(x)` computes the analytic signal,  $y$ , of the  $M$ -by- $N$  input matrix  $x$ , according to the equation

$$\mathbf{Y} = \mathbf{X} + jH\{\mathbf{X}\}$$

where  $j$  is the imaginary unit and  $H\{\mathbf{X}\}$  denotes the Hilbert transform.

Each of the  $N$  columns in  $x$  contains  $M$  sequential time samples from an independent channel. The method computes the analytic signal for each channel.

**Input Arguments****x** — Data input

vector | matrix

Data input, specified as a vector or a matrix.

Data Types: single | double

**Output Arguments****y** — Analytic signal output

vector | matrix

Analytic signal output, returned as a vector or a matrix.

Data Types: single | double

**Object Functions**

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

**Common to All System Objects**

```
step      Run System object algorithm
```



release Release resources and allow changes to System object property values and input characteristics  
 reset Reset internal states of System object

## Examples

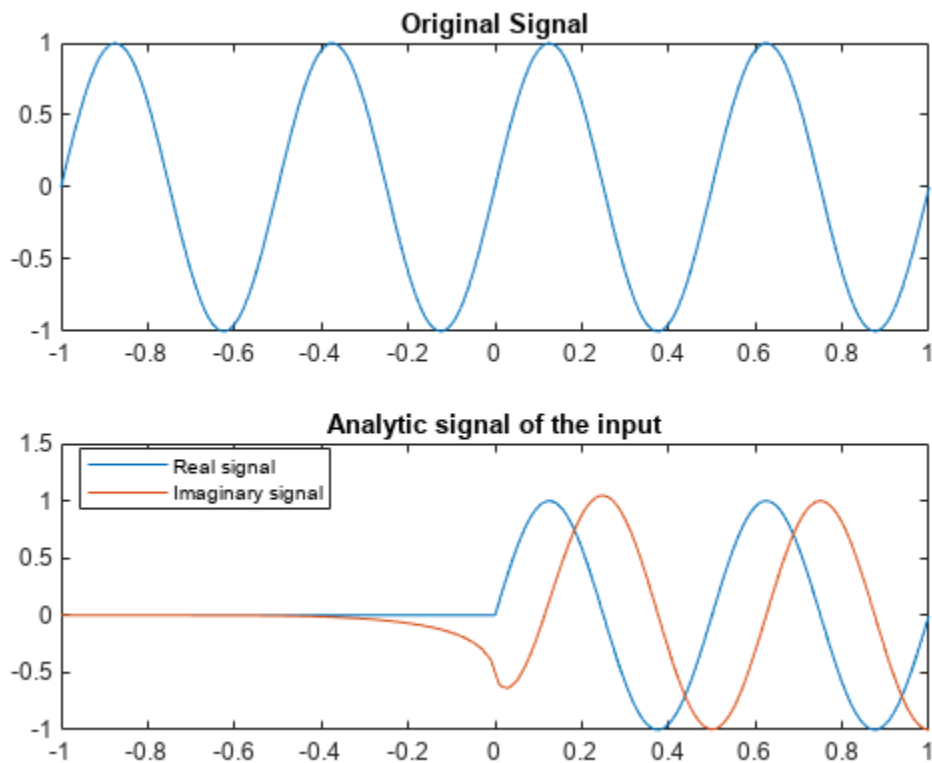
### Compute The Analytic Signal

Compute the analytic signal of a sinusoidal input.

```
t = (-1:0.01:1)';
x = sin(4*pi*t);
anaSig = dsp.AnalyticSignal(200);
y = anaSig(x);
```

View the analytic signal.

```
subplot(2,1,1);
plot(t, x)
title('Original Signal');
subplot(2,1,2), plot(t, [real(y) imag(y)]);
title('Analytic signal of the input')
legend('Real signal','Imaginary signal',...
  'Location','best');
```



## More About

### Analytic Signal

The analytic signal  $x = x_r + jx_i$ , where the real part  $x_r$  is the original data and the imaginary part  $x_i$  contains the Hilbert transform. The imaginary part is a version of the original real sequence with a 90° phase shift. Sines are therefore transformed to cosines, and conversely, cosines are transformed to sines. The Hilbert-transformed series has the same amplitude and frequency content as the original sequence. The transform includes phase information that depends on the phase of the original.

The Hilbert transform is useful in calculating instantaneous attributes of a time series, especially the amplitude and the frequency. The instantaneous amplitude is the amplitude of the complex Hilbert transform. The instantaneous frequency is the time rate of change of the instantaneous phase angle. For a pure sinusoid, the instantaneous amplitude and frequency are constant. The instantaneous phase, however, is a sawtooth, reflecting how the local phase angle varies linearly over a single cycle.

## Algorithms

The algorithm computes the Hilbert transform using an equiripple FIR filter of the specified order  $n$ . The linear phase filter is designed using the Remez exchange algorithm and imposes a delay of  $n/2$  on the input samples.

## Version History

Introduced in R2012a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

This object also supports SIMD code generation using Intel AVX2 technology under these conditions:

- Input signal is real-valued.
- Input signal has a data type of `single` or `double`.

The SIMD technology significantly improves the performance of the generated code.

## See Also

### Objects

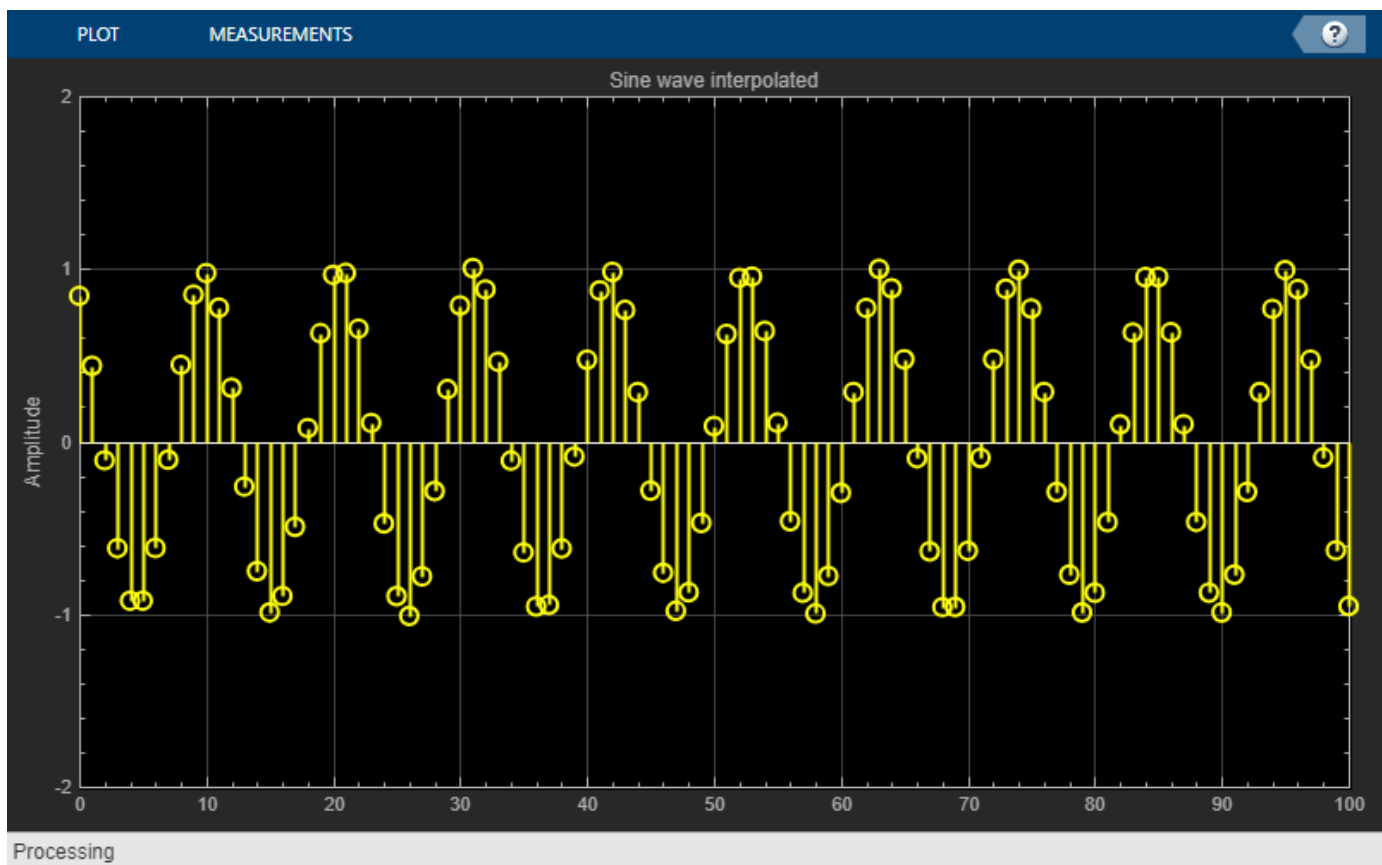
`dsp.FFT` | `dsp.IFFT`

# dsp.ArrayPlot

Display vectors or arrays

## Description

Display vectors or arrays where the data is uniformly spaced along the x-axis.



## Creation

### Syntax

```
scope = dsp.ArrayPlot  
scope = dsp.ArrayPlot(Name=Value)
```

### Description

`scope = dsp.ArrayPlot` creates an Array Plot object, `scope`.

`scope = dsp.ArrayPlot(Name=Value)` sets properties using one or more name-value pairs. For example, `scope = dsp.ArrayPlot(NumInputPorts=3)`

## Properties

Most properties can be changed from the `dsp.ArrayPlot` UI.

### Plot Configuration

#### **NumInputPorts** — Number of input ports

1 (default) | integer between [1, 96]

Number of input ports, specified as a positive integer. Each signal coming through a separate input becomes a separate channel in the scope. You must invoke the scope with the same number of inputs as the value of this property.

#### **XDataMode** — Source of the x-data spacing

"Sample increment and X-offset" (default) | "Custom"

Specify whether to use the `SampleIncrement` and `XOffset` property values to determine spacing, or specify your own custom spacing. If you specify "Custom", you also must specify the `CustomXData` property values.

You can set this property only when creating the object.

#### **Scope Window Use**

Open the **Plot** tab, click **Settings**, and set **X-Data Mode**.

Data Types: char | string

#### **CustomXData** — X-data values

empty vector (default) | vector

Specify the desired x-data values as a row or column vector of length equal to the frame length of the individual inputs. If you use the default (empty vector) value, the x-data is uniformly spaced and set to  $(0:L-1)$ , where  $L$  is the frame length.

You can set this property only when creating the object.

Example: `scope = dsp.ArrayPlot(XDataMode="Custom", CustomXData=logspace(0, log10(44100/2), 1024))`

#### **Scope Window Use**

Open the **Plot** tab, click **Settings**, and set **X-Data Mode** to **Custom** and specify **Custom X-Data**.

#### **Dependency**

To use this property, set `XDataMode` to "Custom".

#### **SampleIncrement** — Sample increment of input

1 (default) | finite numeric scalar

Specify the spacing between samples along the x-axis as a finite numeric scalar. The input signal is only y-axis data. x-axis data is set automatically based on the `XOffset` and `SampleIncrement` properties.

Example: When `XOffset` is 0 and `SampleIncrement` is 1, the x-axis values are set to 0, 1, 2, 3, 4, ...

Example: When `XOffset` is -1 and `SampleIncrement` is 0.25, the x-axis values are set to -1, -0.75, -0.5, -0.25, 0, ...

#### Scope Window Use

Open the **Plot** tab, click **Settings**, and set **Sample Increment**.

#### Dependency

To use this property, set `XDataMode` to 'Sample increment and X-offset'.

#### XOffset — Display offset of x-axis

0 (default) | scalar

Display offset of x-axis, specified as a numeric scalar. x-axis data is set automatically based on both the `SampleIncrement` and `XOffset` values. The x-offset represents the first value on the x-axis.

Example: When `XOffset` is 0 and `SampleIncrement` is 1, the x-axis values are set to 0, 1, 2, 3, 4, ...

Example: When `XOffset` is -1 and `SampleIncrement` is 0.25, the x-axis values are set to -1, -0.75, -0.5, -0.25, 0, ...

#### Scope Window Use

Open the **Plot** tab, click **Settings**, and set **X-Offset**.

#### Dependency

To use this property, set `XDataMode` to "Sample increment and X-offset".

#### XScale — Scale of x-axis

"Linear" (default) | "Log"

Specify whether the scale of the x-axis is "Linear" or "Log". If `XOffset` is a negative value, you cannot set this property to "Log".

#### Scope Window Use

Open the **Plot** tab, click **Settings**, and set **XScale**.

Data Types: char | string

#### YScale — Scale of y-axis

"Linear" (default) | "Log"

Specify whether the scale of the y-axis is "Linear" or "Log".

#### Scope Window Use

Open the **Plot** tab, click **Settings**, and set **YScale**.

Data Types: char | string

#### PlotType — Control type of plot

"Stem" (default) | "Line" | "Stairs"

Specify the type of plot to use for all the input signals displayed in the scope window:

- "Stem" - The scope displays the input signal as circles with vertical lines extending down to the x-axis at each of the sampled values.
- "Line" - The scope displays the input signal as lines connecting each of the sampled values.
- "Stairs" - The scope displays the input signal as a stair-step graph. A stair-step graph is made up of only horizontal lines and vertical lines. Each horizontal line represents the signal value for a discrete sample period and is connected to two vertical lines. Each vertical line represents a change in values occurring at a sample. Stair-step graphs are useful for drawing time history graphs of digitally sampled data.

### Scope Window Use

Open the **Settings** and set **Plot Type**.

### AxesScaling — Axes scaling mode




"OnceAtStop" (default) | "Auto" | "Manual" | "Updates"

Specify when the scope scales the axes. Valid values are:

- "Auto" — The scope scales the axes as needed to fit the data, both during and after simulation.
- "Manual" — The scope does not scale the axes automatically.
- "OnceAtStop" — The scope scales the axes when the simulation stops.
- "Updates" — The scope scales the axes once after a set number of visual updates. The number of updates is determined by the value of the `AxesScalingNumUpdates` property.

You can set this property only when creating the object.

### Scope Window Use

Hover over the array plot to see the zoom , pan , and autoscale  buttons. You can also zoom and pan using your mouse.

Data Types: `char` | `string`

### AxesScalingNumUpdates — Number of updates before scaling

100 (default) | real positive integer

Specify the number of updates before scaling as a real, positive scalar integer.

### Dependency

To enable this property, set `AxesScaling` to "Updates".

Data Types: `double`

### Measurements

#### MeasurementChannel — Channel for which to obtain measurements

1 (default) | positive integer

Channel for which to obtain measurements, specified as a positive integer in the range [1 *N*], where *N* is the number of input channels.

**Scope Window Use**

Click the **Measurements** tab on the Array Plot toolstrip. In the **Channel** section, select a **Channel**.

Data Types: double

**CursorMeasurements — Cursor measurements**

CursorMeasurementsConfiguration object

Cursor measurements to display screen or waveform cursors, specified as a CursorMeasurementsConfiguration object.

All CursorMeasurementsConfiguration properties are tunable.

**Scope Window Use**

Click the **Measurements** tab on the Array Plot toolstrip and modify the cursor measurements in the **Cursors** section.

**PeakFinder — Peak finder measurements**

PeakFinderConfiguration object

Peak finder measurements to compute and display the largest calculated peak values, specified as a PeakFinderConfiguration object.

All PeakFinderConfiguration properties are tunable.

**Scope Window Use**

Click the **Measurements** tab on the Array Plot toolstrip and modify the peak finder measurements in the **Peaks** section.

**SignalStatistics — Signal statistics measurements**

SignalStatisticsConfiguration object

Signal statistics measurements to compute and display signal statistics, specified as a SignalStatisticsConfiguration object.

All SignalStatisticsConfiguration properties are tunable.

**Scope Window Use**

Click the **Measurements** tab on the Array Plot toolstrip and modify the signal statistics measurements in the **Statistics** section.

**Visualization****Name — Window name**

'Array Plot' (default) | character vector | string scalar

Specify the name of the scope. This name appears as the title of the scope's figure window. To specify a title of a scope plot, use the **Title** property.

Data Types: char | string

**Position — Scope window position and size in pixels**

screen center (default) | [left bottom width height]

Specify, in pixels, the size and location of the scope window as a four-element vector of the form `[left bottom width height]`. By default, the scope window appears in the center of your screen with a width of 800 pixels and height of 450 pixels. The default values for this property may change depending on your screen resolution.


### MaximizeAxes — Maximize axes control

"Auto" (default) | "On" | "Off"

Specify whether to display the scope in maximized-axes mode. In this mode, the axes are expanded to fit into the entire display. To conserve space, labels do not appear in each display. Instead, tick-mark values appear on top of the plotted data. You can select one of the following options:

- "Auto" — The axes appear maximized in all displays only if the `Title` and `YLabel` properties are empty for every display. If you enter any value in any display for either of these properties, the axes are not maximized.
- "On" — The axes appear maximized in all displays. Any values entered into the `Title` and `YLabel` properties are hidden.
- "Off" — None of the axes appear maximized.

### Scope Window Use

Hover over the array plot to see the maximize axes button .

Data Types: `char` | `string`

### Title — Display title

' ' (default) | character vector | string scalar

Specify the display title as a character vector or string.

### Scope Window Use

Open the **Plot** tab, click **Settings**, and set **Title**.

Data Types: `char` | `string`

### ShowLegend — Show legend

false (default) | true

To show a legend with the input names, set this property to `true`.

From the legend, you can control which signals are visible. This control is equivalent to changing the visibility in the **Style** dialog box. In the scope legend, click a signal name to hide the signal in the scope. To show the signal, click the signal name again. To show only one signal, right-click the signal name. To show all signals, press **Esc**.

---

**Note** The legend only shows the first 20 signals. Any additional signals cannot be viewed or controlled from the legend.

---

### Scope Window Use

On the **Plot** tab, click **Legend**.



Data Types: `logical`

### **ChannelNames — Channel names**

empty cell (default) | cell array of character vectors | array of strings

Specify the input channel names as a cell array of character vectors or an array of strings. The names appear in the legend, **Settings**, and **Measurements** panels. If you do not specify names, the channels are labeled as Channel 1, Channel 2, etc.

#### **Dependency**

To see channel names, set `ShowLegend` to `true`.

Data Types: `char`

### **ShowGrid — Display grid**

`true` (default) | `false`

Set this property to `true` to show grid lines on the plot.

#### **Scope Window Use**

Open the **Plot** tab, click **Settings**, and select **Grid**.

### **PlotAsMagnitudePhase — Plot signal as magnitude and phase**

`false` (default) | `true`

- `true` - The scope plots the magnitude and phase of the input signal on two separate axes within the same active display.
- `false` - The scope plots the real and imaginary parts of the input signal on two separate axes within the same active display.

This property is useful for complex-valued input signals. Turning on this property affects the phase for real-valued input signals. When the amplitude of the input signal is nonnegative, the phase is 0 degrees. When the amplitude of the input signal is negative, the phase is 180 degrees.

#### **Scope Window Use**

On the **Plot** tab, select the **Magnitude Phase** button.

### **XLabel — x-axis label**

`""` (default) | character vector | string scalar

Specify the text for the scope to display below the x-axis.

#### **Scope Window Use**

Open the **Plot** tab, click **Settings**, and set **XLabel**.

Data Types: `char` | `string`

### **YLabel — y-axis label**

"Amplitude" (default) | character vector | string scalar

Specify the text for the scope to display to the left of the y-axis.

**Dependencies**

This property applies only when `PlotAsMagnitudePhase` is `false`. When `PlotAsMagnitudePhase` is `true`, the two y-axis labels are read-only values "Magnitude" and "Phase", for the magnitude plot and the phase plot, respectively.

**Scope Window Use**

Open the **Plot** tab, click **Settings**, and set **YLabel**.

Data Types: `char` | `string`

**YLimits — y-axis limits**

`[-10,10]` (default) | `[ymin, ymax]`

Specify the y-axis limits as a two-element numeric vector, `[ymin, ymax]`.

If `PlotAsMagnitudePhase` is `false`, the default is `[-10,10]`. If `PlotAsMagnitudePhase` is `true`, the default is `[0,10]`.

**Dependencies**

When `PlotAsMagnitudePhase` is `true`, this property specifies the y-axis limits of only the magnitude plot. The y-axis limits of the phase plot are always `[-180,180]`.

**Scope Window Use**

Open the **Plot** tab, click **Settings**, and set **Y-Axis Limits** as a two-element numeric vector.

**Usage****Syntax**

```
scope(signal)
scope(signal1,signal2,...,signalN)
```

**Description**

`scope(signal)` displays the signal in the Array Plot.

`scope(signal1,signal2,...,signalN)` displays multiple signals in the Array Plot. The signals can have a different number of channels and different frame lengths.

**Input Arguments****signal — Input signal or signals to visualize**

`scalar` | `vector` | `matrix`

Specify one or more input signals to visualize in the `dsp.ArrayPlot`. Signals can have a different number of channels and different frame lengths.

Example: `scope(signal1,signal2)`

**UI Customization**

To customize the style of signals on the array plot, open the **Settings** and use the bottom row of options to select a signal and modify the style, width, color, and marker type.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

## Object Functions

To use an object function, specify the object as the first input argument.

|                             |  |
|-----------------------------|--|
| <code>generateScript</code> | Generate MATLAB script to create scope with current settings                                   |
| <code>hide</code>           | Hide scope window  |
| <code>show</code>           | Display scope window   |
| <code>isVisible</code>      | Determine visibility of scope  |
| <code>step</code>           | Run System object algorithm  |
| <code>release</code>        | Release resources and allow changes to System object property values and input characteristics |
| <code>reset</code>          | Reset internal states of System object   |

If you want to restart the simulation from the beginning, call `reset` to clear the scope window displays. Do not call `reset` after calling `release`.

## Examples

### Plot a Gaussian Distribution

Create a new Array Plot object.

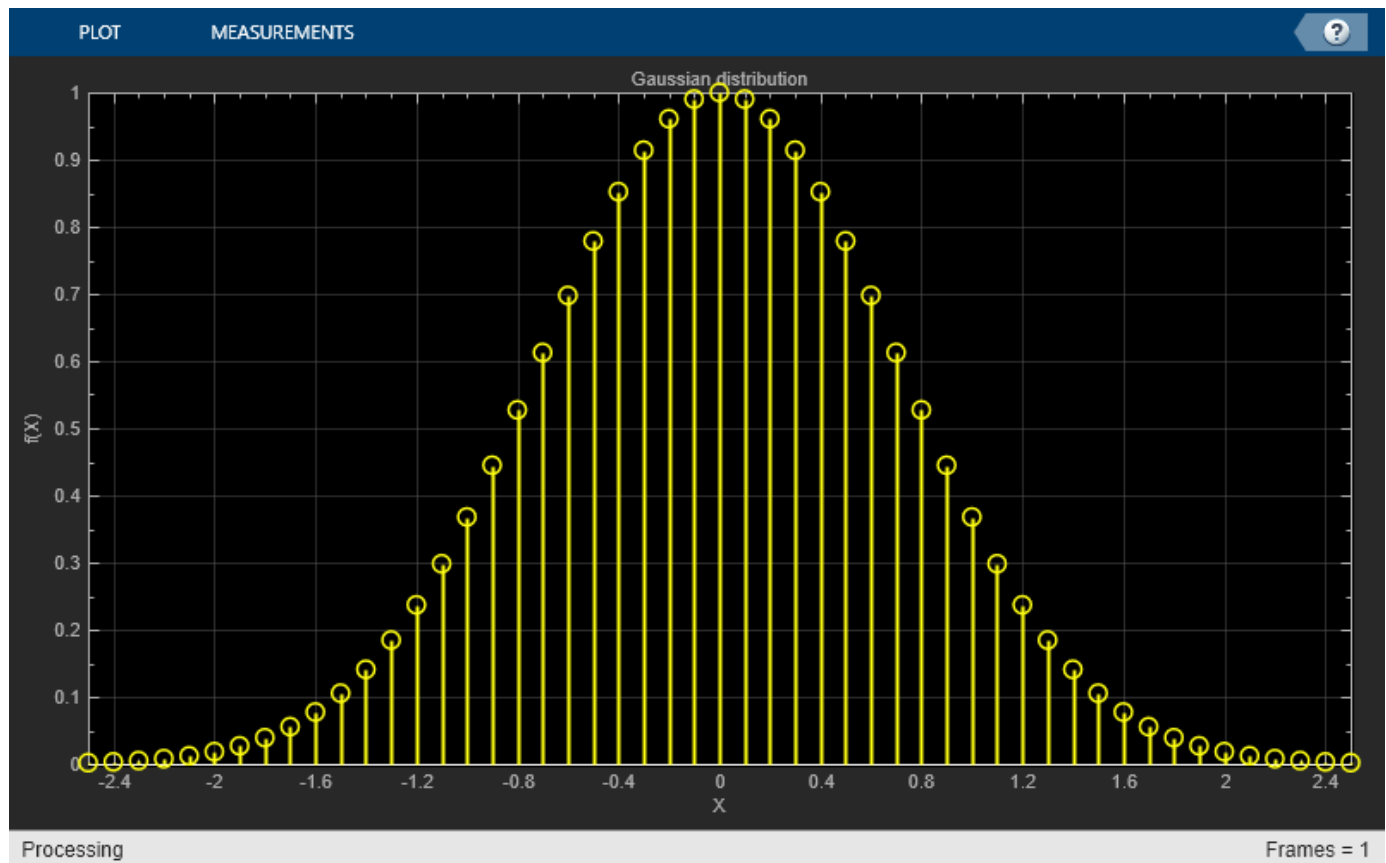
```
scope = dsp.ArrayPlot;
```

Configure the properties of the Array Plot object for a Gaussian distribution.

```
scope.YLimits = [0 1];
scope.XOffset = -2.5;
scope.SampleIncrement = 0.1;
scope.Title = "Gaussian distribution";
scope.XLabel = "X";
scope.YLabel = "f(X)";
```

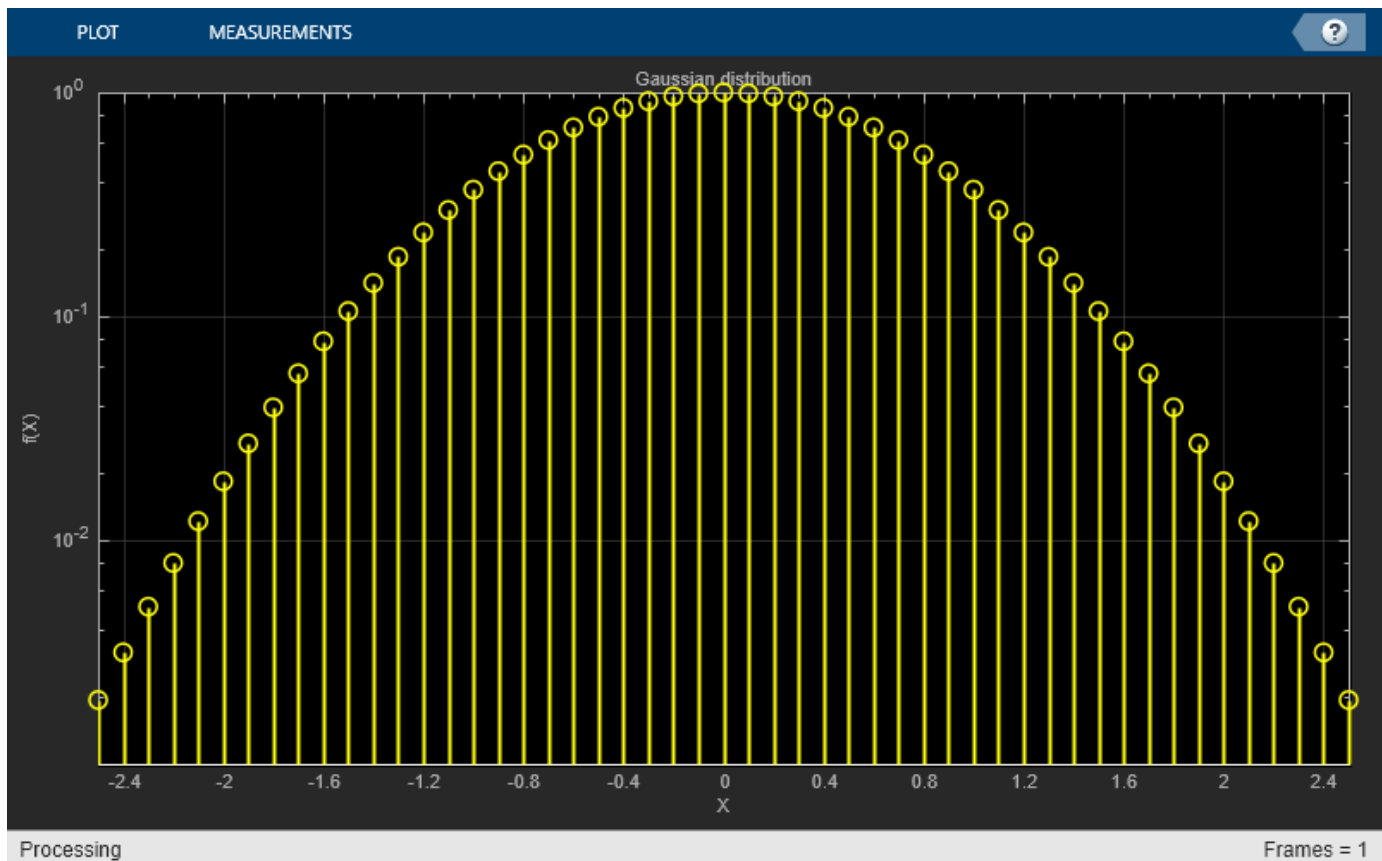
Call the Array Plot object to plot a Gaussian distribution.

```
scope(exp(-(-2.5:.1:2.5).*(-2.5:.1:2.5)))'
```



You can scale the y-axis to show the plot on a log scale.

```
release(scope)
scope.YScale = 'Log';
scope(exp(-(-2.5:.1:2.5).^2))
```



### Plot Changing Filter Weights

View least mean squares (LMS) adaptive filter weights on the Array Plot figure. Watch the filter weights change as they adapt to filter a noisy input signal.

Create an LMS adaptive filter System object™.

```
lmsFilter = dsp.LMSFilter(40,Method="Normalized LMS",...
    StepSize=0.002);
```

Create and configure a `dsp.AudioFileReader` System object to read the input signal from the specified audio file.

```
signalSource = dsp.AudioFileReader("dspafx_8000.wav",...
    SamplesPerFrame=40, ...
    PlayCount=Inf,...
    OutputDataType="double");
```

Create and configure a `dsp.FIRFilter` System object to filter random white noise, creating colored noise.

```
firFilter = dsp.FIRFilter(Numerator=fir1(39,0.25));
```

Create and configure a `dsp.ArrayPlot` object to display the adaptive filter weights.

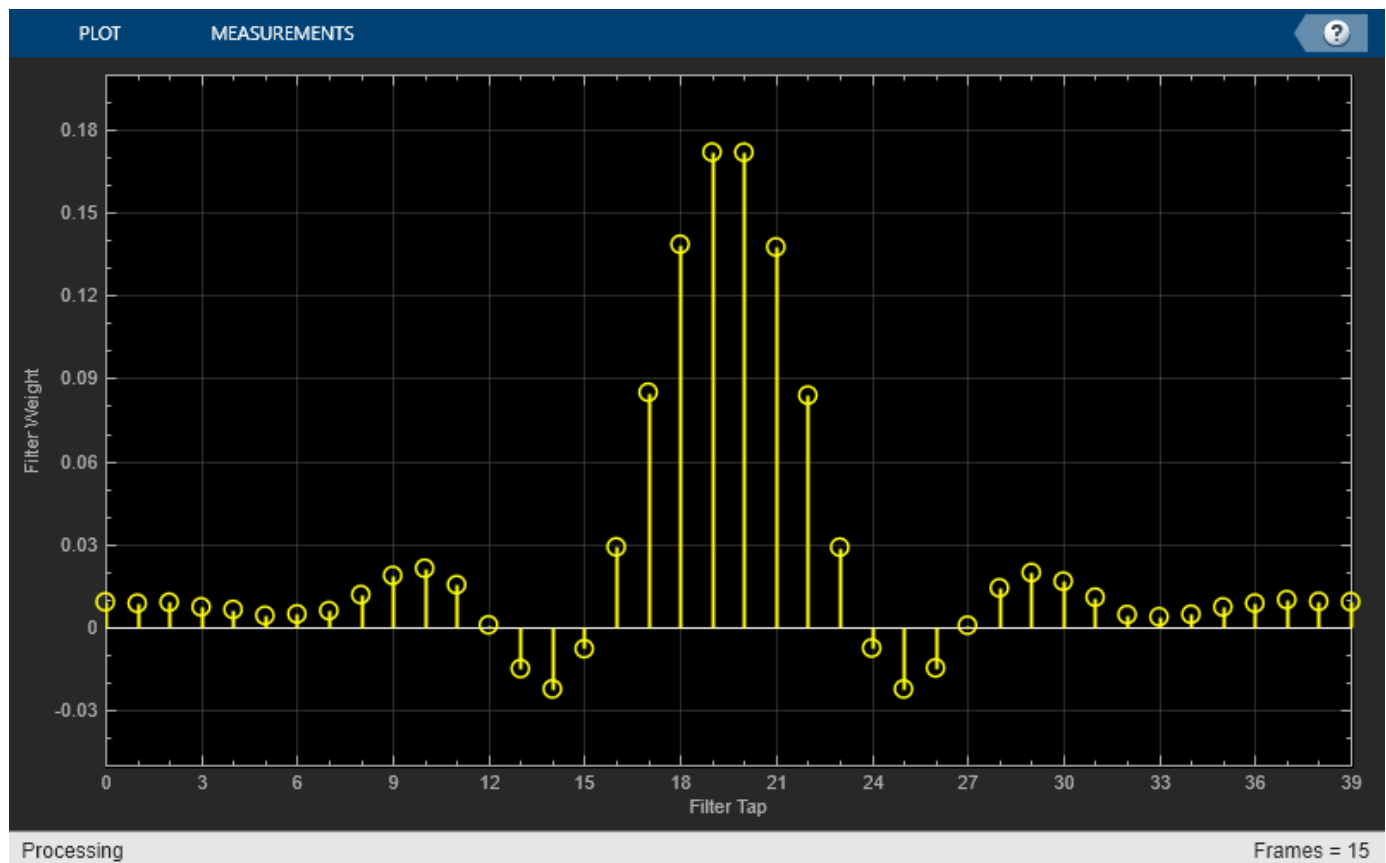
```
scope = dsp.ArrayPlot(XLabel="Filter Tap", ...
    YLabel="Filter Weight", ...
    YLimits=[-0.05 0.2]');

```

Plot the LMS filter weights as they adapt to a desired signal. Read from the audio file, produce random data, and filter the random data. Update the filter weights and plot the filter weights.

```
numplays = 0;
while numplays < 3
    [y, eof] = signalSource();
    noise = rand(40,1);
    noisefilt = firFilter(noise);
    desired = y + noisefilt;
    [~, ~, wts] = lmsFilter(noise,desired);
    scope(wts);
    numplays = numplays + eof;
end

```



### Power Spectrum of Multichannel Sinusoidal Signal

Compute the power spectrum of a multichannel sinusoidal signal using the `dsp.SpectrumEstimator` System object™. You can get the vector of frequencies at which the spectrum is estimated using the `getFrequencyVector` function. To compute the resolution bandwidth of the estimate (RBW), use the `getRBW` function.

Generate a three-channel sinusoid sampled at 1 kHz. Specify sinusoidal frequencies of 100, 200, and 300 Hz. The second and third channels have their phases offset from the first by  $\pi/2$  and  $\pi/4$ , respectively.

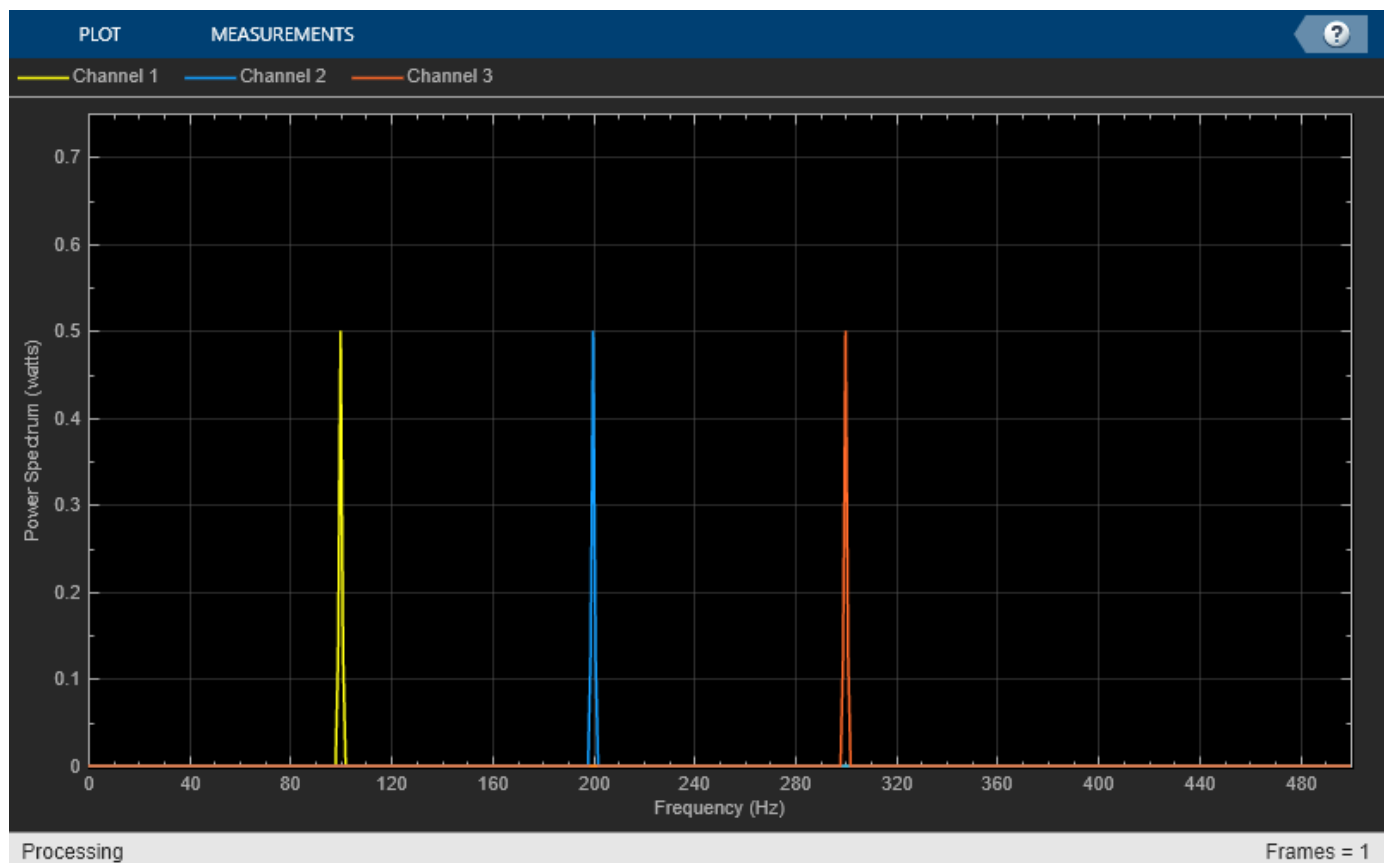
```
sineSignal = dsp.SineWave('SamplesPerFrame',1000,'SampleRate',1000, ...
    'Frequency',[100 200 300],'PhaseOffset',[0 pi/2 pi/4]);
```

Estimate and plot the one-sided spectrum of the signal. Use the `dsp.SpectrumEstimator` object for the computation and the `dsp.ArrayPlot` for the plotting.

```
estimator = dsp.SpectrumEstimator('FrequencyRange','onesided');
plotter = dsp.ArrayPlot('PlotType','Line','YLimits',[0 0.75], ...
    'YLabel','Power Spectrum (watts)','XLabel','Frequency (Hz)');
```

Step through to obtain the data streams and display the spectra of the three channels.

```
y = sineSignal();
pxx = estimator(y);
plotter(pxx)
```



Get the vector of frequencies at which the spectrum is estimated in Hz, using the `getFrequencyVector` function.

```
f = getFrequencyVector(estimator);
```

Compute the resolution bandwidth (RBW) of the estimate using the `getRBW` function.

```
rbw = getRBW(estimator)
```

```
rbw =  
    0.0015
```

The resolution bandwidth of the signal power spectrum is 0.0015 Hz. This frequency is the smallest frequency that can be resolved on the spectrum.

### Power and Max-Hold Spectra of Noisy Sine Wave

Generate a sine wave.

```
sineWave = dsp.SineWave('Frequency',100,...  
    'SampleRate',1000, ...  
    'SamplesPerFrame',1000);
```

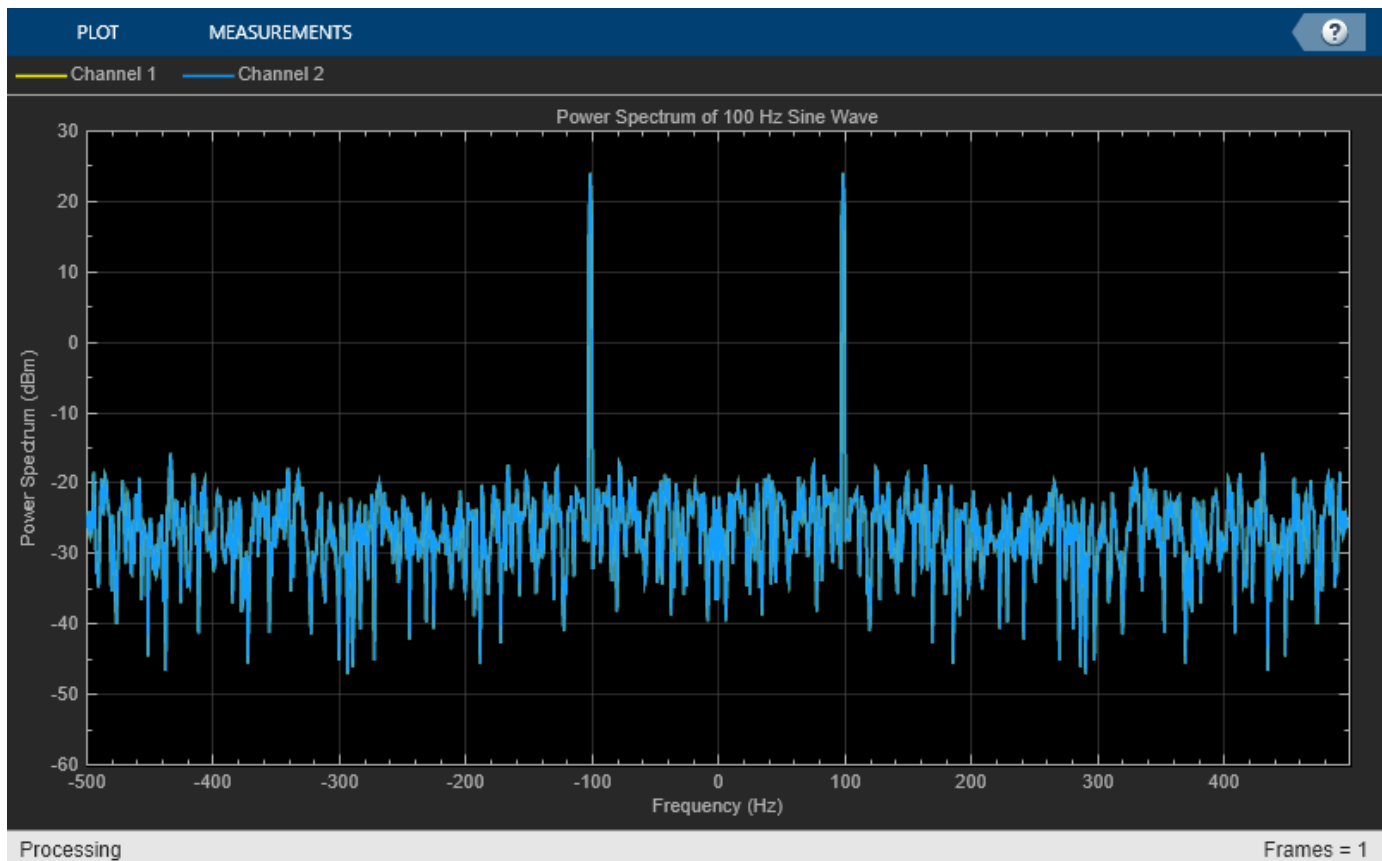
Use the spectrum estimator to compute the power spectrum and the max-hold spectrum of the sine wave. Use the Array Plot to display the spectra.

```
SE = dsp.SpectrumEstimator(...  
    'SampleRate',sineWave.SampleRate,...  
    'SpectrumType','Power','PowerUnits','dBm', ...  
    'FrequencyRange','centered',...  
    'OutputMaxHoldSpectrum',true);  
plotter = dsp.ArrayPlot('PlotType','Line',...  
    'XOffset',-500, ...  
    'YLimits',[-60 30],...  
    'Title','Power Spectrum of 100 Hz Sine Wave', ...  
    'YLabel','Power Spectrum (dBm)',...  
    'XLabel','Frequency (Hz)');
```

Add random noise to the sine wave. Stream in the data, and plot the power spectrum of the signal.

```
for ii = 1:10  
    x = sineWave() + 0.05*randn(1000,1);  
    [Pxx,Pmax] = SE(x);  
    plotter([Pxx Pmax])  
end
```





## Tips

- To close the Array Plot window and clear its associated data, use the MATLAB `clear` function.
- To hide or show the Array Plot window, use the `hide` and `show` functions.

## Version History

Introduced in R2013a

### R2022b: Channel names support array of strings

Starting in R2022b, you can specify the `ChannelNames` property of the `dsp.ArrayPlot` object as an array of strings.

```
ap = dsp.ArrayPlot(ChannelNames=["Input", "Noisy Output"]);
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Supports MEX code generation by treating the calls to the object as extrinsic. Does not support code generation for standalone applications.
- See “System Objects in MATLAB Code Generation” (MATLAB Coder).

### See Also

#### Objects

`dsp.MatrixViewer` | `timescope` | `spectrumAnalyzer` | `dsp.LogicAnalyzer` | `dsp.DynamicFilterVisualizer`

#### Blocks

Array Plot

#### Topics

“Configure Array Plot”

“Visualize Central Limit Theorem in Array Plot”

# dsp.ArrayVectorAdder

**Package:** dsp

(Removed) Add array to vector along specified dimension

---

**Note** `dsp.ArrayVectorAdder` has been removed. Use the `+` operator instead. For more information, see “Compatibility Considerations”.

---

## Description

The `ArrayVectorAdder` object adds an N-D array to a vector along a specified dimension. The length of the vector must equal the size of the N-D array along the specified dimension.

To add an N-D array to a vector along a specified dimension:

- 1 Define and set up your array-vector addition object. See “Construction” on page 4-69.
- 2 Call `step` to add the N-D array according to the properties of `dsp.ArrayVectorAdder`. The behavior of `step` is specific to each object in the toolbox.

---

**Note** Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj, x)` and `y = obj(x)` perform equivalent operations.

---

## Construction

`ava = dsp.ArrayVectorAdder` returns an array-vector addition object, `ava`, that adds a vector to an N-D array along the first dimension.

`ava = dsp.ArrayVectorAdder('PropertyName', PropertyValue, ...)` returns an array-vector addition object, `ava`, with each property set to the specified value.

## Properties

### Dimension

Dimension along which to add vector elements to input

Specify the dimension along which to add the input array to the elements of the vector as a positive integer. The length of the vector must match the size of the N-D array along the specified dimension. The default is 1.

### VectorSource

Source of vector

Specify the source of the vector values as `|Input port | Property |`. The default is `Input port`.

**Vector**

Vector values

Specify the vector values. This property applies only when you set the `VectorSource` property to `Property`. The default is `[0.5 0.25]`. This property is tunable.

**Fixed-Point Properties****FullPrecisionOverride**

Full precision override for fixed-point arithmetic

Specify whether to use full precision rules. If you set `FullPrecisionOverride` to `true`, which is the default, the object computes all internal arithmetic and output data types using full precision rules. These rules provide the most accurate fixed-point numerics. It also turns off the display of other fixed-point properties because they do not apply individually. These rules guarantee that no quantization occurs within the object. Bits are added, as needed, to ensure that no roundoff or overflow occurs. If you set `FullPrecisionOverride` to `false`, fixed-point data types are controlled through individual fixed-point property settings. For more information, see “Full Precision for Fixed-Point System Objects”.

**RoundingMethod**

Rounding method for fixed-point operations

Specify the rounding method as `| Ceiling | Convergent | Floor | Nearest | Round | Simplest | Zero |`. The default is `Floor`. This property applies only if the object is not in full precision mode.

**OverflowAction**

Overflow action for fixed-point operations

Specify the overflow action as `| Wrap | Saturate |`. The default is `Wrap`. This property applies only if the object is not in full precision mode.

**VectorDataType**

Vector word and fraction lengths

Specify the vector fixed-point data type as `| Same word length as input | Custom |`. This property applies when you set the `VectorSource` property to `Property`. The default is `Same word length as input`.

**CustomVectorDataType**

Vector word and fraction lengths

Specify the vector fixed-point type as a `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `VectorSource` property to `Property` and the `VectorDataType` property to `Custom`. The default is `numericType([], 16, 15)`.

### AccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as `| Full precision | Same as first input | Custom |`. The default is `Full precision`.

### CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies only when the `AccumulatorDataType` property is `Custom`. The default is `numericType([], 32, 30)`.

### OutputDataType

Output word and fraction lengths

Specify the output fixed-point data type as `| Same as accumulator | Same as first input | Custom |`.

### CustomOutputDataType

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies only when the “`OutputDataType`” on page 4-0 property is `Custom`. The default is `numericType([], 16, 15)`.

## Methods

`step`                    Add vector to N-D array

| Common to All System Objects |  |
|------------------------------|--|
| <code>release</code>         | Allow System object property value changes |

## Examples

### Add a Vector To a Matrix

**Note** If you are using R2016a or an earlier release, replace each call to the object with the equivalent `step` syntax. For example, `obj(x)` becomes `step(obj, x)`.

Add a 2-by-1 vector to a 2-by-2 matrix along the first dimension of the array.

```
ava = dsp.ArrayVectorAdder;
a = ones(2);
x = [1 2]';
y = ava(a, x);
```

## Algorithms

This object implements the algorithm, inputs, and outputs described on the Array-Vector Add block reference page. The object properties correspond to the block parameters, except:

The array-vector addition object does not have **Minimum** or **Maximum** options for data output.

## Version History

### Introduced in R2012a

#### **R2021b: dsp.ArrayVectorAdder System object will be removed**

*Warns starting in R2021b*

The dsp.ArrayVectorAdder System object will be removed in a future release. Use the + operator instead.

#### **R2023a: dsp.ArrayVectorAdder System object has been removed**

*Errors starting in R2023a*

The dsp.ArrayVectorAdder System object has been removed. Use the + operator instead.

### Update Code

This table shows how to update existing code to use the + operator.

| Discouraged Usage  | Recommended Replacement  |
|--|--|
| <b>Add along first dimension (column-wise)</b><br><pre>ava = dsp.ArrayVectorAdder(Dimension=1); a = ones(2); x = [1 2]'; y = ava(a,x)  y = 2x2     2    2     3    3</pre> | <b>Add along first dimension (column-wise)</b><br><pre>y = a + x y = 2x2     2    2     3    3</pre> |

| Discouraged Usage  | Recommended Replacement   |
|--|---|
| <p><b>Add along second dimension (row-wise)</b></p> <pre>ava = dsp.ArrayVectorAdder(Dimension=2); a = ones(2); x = [1 2]; y = ava(a,x)</pre> <p>y = 2×2</p> <pre>  2   3   2   3</pre>   | <p><b>Add along second dimension (row-wise)</b></p> <pre>y = a + x</pre> <p>y = 2×2</p> <pre>  2   3   2   3</pre>  |
| <p><b>Add along <math>N^{\text{th}}</math> dimension</b></p> <pre>d = [3 10 2 3 4]; a = randn(d);</pre> <p>Sum the array with a vector along the fourth dimension.</p> <pre>N = 4; ava = dsp.ArrayVectorAdder(Dimension=N); x = randn(1,d(N)); y = ava(a,x);</pre> | <p><b>Add along <math>N^{\text{th}}</math> dimension</b></p> <pre>C = repmat({1},1,length(d)); C{N} = ':'; xN(C{:}) = x; yNew = a + xN</pre> <pre>all(y == yNew,'all')</pre> <pre>ans =</pre> <pre>logical</pre> <pre>1</pre> |

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

## step

**System object:** `dsp.ArrayVectorAdder`

**Package:** `dsp`

Add vector to N-D array

### Syntax

`Y = step(ava,A)`

`Y = step(ava,A,V)`

### Description

---

**Note** Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

---

`Y = step(ava,A)` returns `Y`, the result of adding the input array `A` to the elements of the vector specified in the `Vector` property along the specified dimension when the `VectorSource` property is `Property`. The length of the vector specified in the `Vector` property must equal the length of the specified dimension of `A`.

`Y = step(ava,A,V)` returns `Y`, the result of adding the input array `A` to the elements of the input vector `V` along the specified dimension when the `VectorSource` property is `Input port`. The length of the input `V` must equal the length of the specified dimension of `A`.

---

**Note** `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---



# dsp.ArrayVectorDivider

**Package:** dsp

(Removed) Divide array by vector along specified dimension

---

**Note** `dsp.ArrayVectorDivider` has been removed. Use the `./` operator instead. For more information, see “Compatibility Considerations”.

---

## Description

The `ArrayVectorDivider` object divides an array by a vector along a specified dimension.

To divide an array by a vector along a specified dimension:

- 1 Define and set up your array-vector division object. See “Construction” on page 4-75.
- 2 Call `step` to divide the array according to the properties of `dsp.ArrayVectorDivider`. The behavior of `step` is specific to each object in the toolbox.

---

**Note** Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

---

## Construction

`avd = dsp.ArrayVectorDivider` returns an array-vector division object, `avd`, that divides an input array by the elements of a vector along the first dimension of the array.

`avd = dsp.ArrayVectorDivider('PropertyName',PropertyValue,...)` returns an array-vector division object, `avd`, with each property set to the specified value.

## Properties

### Dimension

Dimension along which to divide input by vector elements

Specify the dimension along which to divide the input array by the elements of a vector as a positive integer. The default is 1.

### VectorSource

Source of vector

Specify the source of the vector values as | Input port | Property |. The default is Input port.

### **Vector**

Vector values

Specify the vector values. This property applies when you set the `VectorSource` property to `Property`. The default is `[0.5 0.25]`. This property is tunable.

### **Fixed-Point Properties**

#### **RoundingMethod**

Rounding method for fixed-point operations

Specify the rounding method as `| Ceiling | Convergent | Floor | Nearest | Round | Simplest | Zero |`. The default is `Floor`.

#### **OverflowAction**

Overflow action for fixed-point operations

Specify the overflow action as `| Wrap | Saturate |`. The default is `Wrap`.

#### **VectorDataType**

Vector word and fraction lengths

Specify the vector fixed-point mode as `| Same word length as input | Custom |`. This property applies when you set the `VectorSource` property to `Property`. The default is `Same word length as input`.

#### **CustomVectorDataType**

Vector word and fraction lengths

Specify the vector fixed-point data type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `VectorSource` property to `Property` and the `VectorDataType` property to `Custom`. The default is `numericType([], 16, 15)`.

#### **OutputDataType**

Output word and fraction lengths

Specify the output fixed-point data type as `| Same as first input | Custom |`. The default is `Same as first input`.

#### **CustomOutputDataType**

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the “`OutputDataType`” on page 4-0 property to `Custom`. The default is `numericType([],16,15)`.

## Methods

`step` Divide array by vector

| Common to All System Objects |  |
|------------------------------|--|
| <code>release</code>         | Allow System object property value changes |

## Examples

### Divide a Matrix By a Vector

**Note** If you are using R2016a or an earlier release, replace each call to the object with the equivalent `step` syntax. For example, `obj(x)` becomes `step(obj,x)`.

```
avd = dsp.ArrayVectorDivider;
a = ones(2);
x = [2 3]';
y = avd(a, x)
```

`y = 2×2`

```
    0.5000    0.5000
    0.3333    0.3333
```

## Algorithms

This object implements the algorithm, inputs, and outputs described on the Array-Vector Divide block reference page. The object properties correspond to the block parameters, except:

The array-vector division object does not have **Minimum** or **Maximum** options for data output.

## Version History

### Introduced in R2012a

#### **R2021b: dsp.ArrayVectorDivider System object will be removed**

*Warns starting in R2021b*

`dsp.ArrayVectorDivider` System object will be removed in a future release. Use the `./` operator instead.

#### **R2023a: dsp.ArrayVectorDivider System object has been removed**

*Errors starting in R2023a*

The `dsp.ArrayVectorDivider` System object has been removed. Use the `./` operator instead.

### Update Code

This table shows how to update existing code to use the ./ operator.

| Discouraged Usage  | Recommended Replacement   |
|--|---|
| <p><b>Divide along first dimension (column-wise)</b></p> <pre> avd = dsp.ArrayVectorDivider(Dimension=1); a = ones(2); x = [2;3]; y = avd(a,x)  y = 2x2      0.5    0.5     0.333 0.333                     </pre>   | <p><b>Divide along first dimension (column-wise)</b></p> <pre> y = a ./ x  y = 2x2      0.5    0.5     0.333 0.333                     </pre>   |
| <p><b>Divide along second dimension (row-wise)</b></p> <pre> avd = dsp.ArrayVectorDivider(Dimension=2); a = ones(2); x = [2;3]; y = avd(a,x)  y = 2x2      0.5    0.333     0.5    0.333                     </pre>  | <p><b>Divide along second dimension (row-wise)</b></p> <pre> y = a ./ x'  y = 2x2      0.5    0.333     0.5    0.333                     </pre>   |
| <p><b>Divide along <math>N^{\text{th}}</math> dimension</b></p> <pre> d = [3 10 2 3 4]; a = randn(d);  Divide the array with a vector along the fourth dimension.  N = 4; avd = dsp.ArrayVectorDivider(Dimension=N); x = randn(1,d(N)); y = avd(a,x);                     </pre> | <p><b>Divide along <math>N^{\text{th}}</math> dimension</b></p> <pre> C = repmat({1},1,length(d)); C{N} = ':'; xN(C{:}) = x; yNew = a ./ xN  all(y == yNew,'all')  ans =      logical      1                     </pre> |

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

## step

**System object:** dsp.ArrayVectorDivider

**Package:** dsp

Divide array by vector

### Syntax

$Y = \text{step}(\text{avd}, A, V)$

$Y = \text{step}(\text{avd}, A)$

### Description

---

**Note** Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example,  $y = \text{step}(\text{obj}, x)$  and  $y = \text{obj}(x)$  perform equivalent operations.

---

$Y = \text{step}(\text{avd}, A, V)$  returns  $Y$ , the result of dividing the input array  $A$  by the elements of input vector  $V$  along the specified dimension when the `VectorSource` property is `Input port`. The length of the input  $V$  must equal the length of the specified dimension of  $A$ .

$Y = \text{step}(\text{avd}, A)$  returns  $Y$ , the result of dividing the input array  $A$  by the elements of the vector specified in the `Vector` property along the specified dimension when the `VectorSource` property is `Property`. The length of the vector specified in the `Vector` property must equal the length of the specified dimension of  $A$ .

---

**Note** `obj` specifies the System object on which to run this `step` method.

---

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

# **dsp.ArrayVectorMultiplier**

**Package:** dsp

(Removed) Multiply array by vector along specified dimension

---

**Note** `dsp.ArrayVectorMultiplier` has been removed. Use the `.*` operator instead. For more information, see “Compatibility Considerations”.

---

## **Description**

The `ArrayVectorMultiplier` object multiplies an array by a vector along a specified dimension.

To multiply an array by a vector along a specified:

- 1 Define and set up your array-vector multiplication object. See “Construction” on page 4-80.
- 2 Call `step` to multiply the array according to the properties of `dsp.ArrayVectorMultiplier`. The behavior of `step` is specific to each object in the toolbox.

---

**Note** Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj, x)` and `y = obj(x)` perform equivalent operations.

---

## **Construction**

`avm = dsp.ArrayVectorMultiplier` returns an array-vector multiplication object, `avm`, that multiplies an input N-D array by the elements of a vector along the second dimension.

`avm = dsp.ArrayVectorMultiplier('PropertyName', PropertyValue, ...)` returns an array-vector multiplication object, `avm`, with each property set to the specified value.

## **Properties**

### **Dimension**

Dimension along which to multiply input by vector elements

Specify the dimension along which to multiply the input array by the elements of vector as a positive integer. The default is 2.

### **VectorSource**

Source of vector

Specify the source of the vector values as one of `Input port` or `Property`. The default is `Input port`.

## Vector

Vector to multiply array

Specify the vector by which to multiply the array. This property applies when you set the VectorSource property to Property. The default is [0.5 0.25]. This property is tunable.

## Fixed-Point Properties

### RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as Ceiling, Convergent, Floor, Nearest, Round, Simplest, Zero. The default is floor.

### OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as Wrap or Saturate. The default is Wrap.

### VectorDataType

Vector word and fraction lengths

Specify the vector fixed-point data type as Same word length as input, Custom. This property applies when you set the VectorSource property to Property. The default is Same word length as input.

### CustomVectorDataType

Vector word and fraction lengths

Specify the vector fixed-point type as a numeric type object with a Signedness of Auto. This property applies when you set the VectorSource property to Property and the VectorDataType property to Custom. The default is numeric type ([], 16, 15).

### ProductDataType

Product word and fraction lengths

Specify the product fixed-point data type as Full precision, Same as first input, or Custom. The default is Full precision.

### CustomProductDataType

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `ProductDataType` property to `Custom`. The default is `numericType([], 32, 30)`.

### **AccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as `Full precision`, `Same as product`, `Same as first input`, or `Custom`. The default is `Full precision`.

### **CustomAccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `AccumulatorDataType` property to `Custom`. The default is `numericType([], 32, 30)`.

### **OutputDataType**

Output word and fraction lengths

Specify the output fixed-point data type as `Same as product`, `Same as first input`, or `Custom`. The default is `Same as product`.

### **CustomOutputDataType**

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `OutputDataType` property to `Custom`. The default is `numericType([], 16, 15)`.

## **Methods**

`step`                  Multiply array by vector

| <b>Common to All System Objects</b> |  |
|-------------------------------------|--|
| <code>release</code>                | Allow System object property value changes |

## **Examples**

### **Multiply a Matrix by a Vector**

---

**Note** If you are using R2016a or an earlier release, replace each call to the object with the equivalent `step` syntax. For example, `obj(x)` becomes `step(obj, x)`.

---



```
avm = dsp.ArrayVectorMultiplier;
a = ones(2);
x = [2 3]';
y = avm(a, x)
```

y = 2×2

```
 2   3
 2   3
```

## Algorithms

This object implements the algorithm, inputs, and outputs described on the Array-Vector Multiply block reference page. The object properties correspond to the block parameters, except:

- The array-vector multiplication object does not have **Minimum** or **Maximum** options for data output.

## Version History

### Introduced in R2012a

#### **R2021b: dsp.ArrayVectorMultiplier System object will be removed**

*Warns starting in R2021b*

dsp.ArrayVectorMultiplier System object will be removed in a future release. Use the .\* operator instead.

#### **R2023a: dsp.ArrayVectorMultiplier System object has been removed**

*Errors starting in R2023a*

The dsp.ArrayVectorMultiplier System object has been removed. Use the .\* operator instead.

### Update Code

This table shows how to update existing code to use the .\* operator.

| Discouraged Usage   | Recommended Replacement                             |
|---|---|
| <b>Multiply along first dimension (column-wise)</b>   | <b>Multiply along first dimension (column-wise)</b> |
| <pre>avm = dsp.ArrayVectorMultiplier(Dimension=1); a = ones(2); x = [2;3]; y = avm(a,x)</pre> | <pre>y = a .* x</pre>                               |
| <p>y = 2×2</p> <pre> 2   2  3   3</pre>   | <p>y = 2×2</p> <pre> 2   2  3   3</pre>             |

| Discouraged Usage   | Recommended Replacement  |
|---|--|
| <p><b>Multiply along second dimension (row-wise)</b></p> <pre>avm = dsp.ArrayVectorMultiplier(Dimension=2); a = ones(2); x = [2;3]; y = avm(a,x)</pre> <p>y = 2×2</p> <pre>    2    3     2    3</pre>  | <p><b>Multiply along second dimension (row-wise)</b></p> <pre>y = a .* x'</pre> <p>y = 2×2</p> <pre>    2    3     2    3</pre>  |
| <p><b>Multiply along <math>N^{\text{th}}</math> dimension</b></p> <pre>d = [3 10 2 3 4]; a = randn(d);</pre> <p>Multiply the array with a vector along the fourth dimension.</p> <pre>N = 4; avm = dsp.ArrayVectorMultiplier(Dimension=N); x = randn(1,d(N)); y = avm(a,x);</pre> | <p><b>Multiply along <math>N^{\text{th}}</math> dimension</b></p> <pre>C = repmat({1},1,length(d)); C{N} = ':'; xN(C{:}) = x; yNew = a .* xN</pre> <pre>all(y == yNew,'all')</pre> <pre>ans =     logical      1</pre> |

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

## step

**System object:** dsp.ArrayVectorMultiplier

**Package:** dsp

Multiply array by vector

### Syntax

$Y = \text{step}(\text{avm}, A, V)$

$Y = \text{step}(\text{avm}, A)$

### Description

---

**Note** Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example,  $y = \text{step}(\text{obj}, x)$  and  $y = \text{obj}(x)$  perform equivalent operations.

---

$Y = \text{step}(\text{avm}, A, V)$  returns  $Y$ , the result of multiplying the input array  $A$  by the elements of input vector  $V$  along the specified dimension when the `VectorSource` property is `Input port`. The length of the input  $V$  must equal the length of the specified dimension of  $A$ .

$Y = \text{step}(\text{avm}, A)$  returns  $Y$ , the result of multiplying the input array  $A$  by the elements of vector specified in `Vector` property along the specified dimension when the `VectorSource` property is set to `Property`. The length of the vector specified in `Vector` property must equal the length of the specified dimension of  $A$ .

---

**Note** `obj` specifies the System object on which to run this `step` method.

---

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

# **dsp.ArrayVectorSubtractor**

**Package:** dsp

(Removed) Subtract vector from array along specified dimension

---

**Note** `dsp.ArrayVectorSubtractor` has been removed. Use the `-` operator instead. For more information, see “Compatibility Considerations”.

---

## **Description**

The `ArrayVectorSubtractor` object subtracts a vector from an N-D array along a specified dimension.

To subtract a vector from an N-D array along a specified dimension:

- 1 Define and set up your array-vector subtraction object. See “Construction” on page 4-86.
- 2 Call `step` to subtract the vector according to the properties of `dsp.ArrayVectorSubtractor`. The behavior of `step` is specified to each object in the toolbox.

---

**Note** Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj, x)` and `y = obj(x)` perform equivalent operations.

---

## **Construction**

`avs = dsp.ArrayVectorSubtractor` returns an array-vector subtraction object, `avs`, that subtracts the elements of a vector from an N-D input array along the first dimension.

`avs = dsp.ArrayVectorSubtractor('PropertyName', PropertyValue, ...)` returns an array-vector subtraction object, `avs`, with each property set to the specified value.

## **Properties**

### **Dimension**

Dimension along which to subtract vector elements from input

Specify the dimension along which to subtract the elements of the vector from the input array as an integer-valued scalar greater than 0. The default is 1.

### **VectorSource**

Source of vector

Specify the source of the vector values as one of `Input port` or `Property`. The default is `Input port`.

## Vector

Vector values

Specify the vector values. This property applies when you set the `VectorSource` property to `Property`. The default is `[0.5 0.25]`. This property is tunable.

## Fixed-Point Properties

### FullPrecisionOverride

Full precision override for fixed-point arithmetic

Specify whether to use full precision rules. If you set `FullPrecisionOverride` to `true`, which is the default, the object computes all internal arithmetic and output data types using full precision rules. These rules provide the most accurate fixed-point numerics. It also turns off the display of other fixed-point properties because they do not apply individually. These rules guarantee that no quantization occurs within the object. Bits are added, as needed, to ensure that no roundoff or overflow occurs. If you set `FullPrecisionOverride` to `false`, fixed-point data types are controlled through individual fixed-point property settings. For more information, see “Full Precision for Fixed-Point System Objects”.

### RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as `Ceiling`, `Convergent`, `Floor`, `Nearest`, `Round`, `Simplest`, or `Zero`. This property applies only if the object is not in full precision mode.

### OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as one of `Wrap` or `Saturate`. The default is `Wrap`. This property applies only if the object is not in full precision mode.

### VectorDataType

Vector word and fraction lengths

Specify the vector fixed-point data type as `Same word length as input` or `Custom`. This property applies when you set the `VectorSource` property to `Property`. The default is `Same word length as input`.

### CustomVectorDataType

Vector word and fraction lengths

Specify the vector fixed-point type as a `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `VectorSource` property to `Property` and the `VectorDataType` property to `Custom`. The default is `numericType([],16,15)`.

**AccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as one of `Full precision`, `Same as first input`, or `Custom`. The default is `Full precision`.

**CustomAccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `AccumulatorDataType` property to `Custom`. The default is `numericType([],32,30)`.

**OutputDataType**

Output word and fraction lengths

Specify the output fixed-point data type as `Same as accumulator`, `Same as first input`, or `Custom`. The default is `Same as accumulator`.

**CustomOutputDataType**

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the “`OutputDataType`” on page 4-0 property to `Custom`. The default is `numericType([],16,15)`.

**Methods**

`step` Subtract vector from array along specified dimension

| Common to All System Objects |  |
|------------------------------|--|
| <code>release</code>         | Allow System object property value changes |

**Examples**

**Subtract Vector From Matrix**

---

**Note** If you are using R2016a or an earlier release, replace each call to the object with the equivalent `step` syntax. For example, `obj(x)` becomes `step(obj,x)`.

---

```

avs = dsp.ArrayVectorSubtractor;
a = ones(2);
x = [1 2]';
y = avs(a, x)

```

y = 2×2

```

    0    0
   -1   -1

```

## Algorithms

This object implements the algorithm, inputs, and outputs described on the Array-Vector Subtract block reference page. The object properties correspond to the block parameters, except:

The array-vector subtraction object does not have **Minimum** or **Maximum** options for data output.

## Version History

**Introduced in R2012a**

### R2021b: dsp.ArrayVectorSubtractor System object will be removed

*Warns starting in R2021b*

dsp.ArrayVectorSubtractor System object will be removed in a future release. Use the – operator instead.

### R2023a: dsp.ArrayVectorSubtractor System object has been removed

*Errors starting in R2023a*

The dsp.ArrayVectorSubtractor System object has been removed. Use the – operator instead.

## Update Code

This table shows how to update existing code to use the – operator.

| Discouraged Usage   | Recommended Replacement   |
|---|---|
| <b>Subtract along first dimension (column-wise)</b><br><pre> avs = dsp.ArrayVectorSubtractor(Dimension=1); a = ones(2); x = [1 2]'; y = avs(a,x) </pre> <p>y = 2×2</p> <pre>     0    0    -1   -1 </pre> | <b>Subtract along first dimension (column-wise)</b><br><pre> y = a - x </pre> <p>y = 2×2</p> <pre>     0    0    -1   -1 </pre> |

| Discouraged Usage   | Recommended Replacement  |
|---|--|
| <p><b>Subtract along second dimension (row-wise)</b></p> <pre> avs = dsp.ArrayVectorSubtractor(Dimension=2); a = ones(2); x = [1 2]; y = avs(a,x)  y = 2x2      0    -1     0    -1                     </pre>  | <p><b>Subtract along second dimension (row-wise)</b></p> <pre> y = a - x  y = 2x2      0    -1     0    -1                     </pre>  |
| <p><b>Subtract along <math>N^{\text{th}}</math> dimension</b></p> <pre> d = [3 10 2 3 4]; a = randn(d);  Subtract the array with a vector along the fourth dimension.  N = 4; avs = dsp.ArrayVectorSubtractor(Dimension=N); x = randn(1,d(N)); y = avs(a,x);                     </pre> | <p><b>Subtract along <math>N^{\text{th}}</math> dimension</b></p> <pre> C = repmat({1},1,length(d)); C{N} = ':'; xN(C{:}) = x; yNew = a - xN  all(y == yNew,'all')  ans =      logical      1                     </pre> |

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).



## step

**System object:** dsp.ArrayVectorSubtractor

**Package:** dsp

Subtract vector from array along specified dimension

### Syntax

$Y = \text{step}(\text{avs}, A, V)$

$Y = \text{step}(\text{avs}, A)$

### Description

---

**Note** Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example,  $y = \text{step}(\text{obj}, x)$  and  $y = \text{obj}(x)$  perform equivalent operations.

---

$Y = \text{step}(\text{avs}, A, V)$  returns  $Y$ . The value of  $Y$  results from subtracting the elements of input vector  $V$  from the input array  $A$  along the specified dimension when the `VectorSource` property is `Input port`. The length of the input  $V$  must equal the length of the specified dimension of  $A$ .

$Y = \text{step}(\text{avs}, A)$  returns  $Y$ . The value of  $Y$  results from subtracting the elements of the vector specified in the `Vector` property from the input array  $A$  along the specified dimension when the `VectorSource` property is `Property`. The length of the vector specified in the `Vector` property must equal the length of the specified dimension of  $A$ .

---

**Note** `obj` specifies the System object on which to run this `step` method.

---

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

# dsp.AsyncBuffer

**Package:** dsp

FIFO buffer

## Description

The `dsp.AsyncBuffer` System object writes samples to and reads samples from a first-in, first-out (FIFO) buffer. The `write` method writes data to the buffer, and the `read` method reads data from the buffer. When creating the object, you can set the number of samples (rows) of the buffer using the `Capacity` property. The number of channels (columns) is set during the first call to `write`. Initialize the buffer by calling `write` or `setup` before the first call to `read`.

The data that you write occupies the next available space in the buffer. If the buffer is full and all the data within it is unread (`asyncBuff.NumUnreadSamples == asyncBuff.Capacity`), the object overwrites the oldest data with any new data that comes in. The buffer removes data only when the data is overwritten, so you can reread data from the past. The `dsp.AsyncBuffer` object supports writing and reading variable frame size signals. For examples, see “Read Variable Frame Sizes from Buffer” on page 4-94 and “Write Variable Frame Sizes to Buffer” on page 4-96.

To write and read samples from a FIFO buffer:

- Create a `dsp.AsyncBuffer` object and set the properties of the object.
- Call `write` to write samples to the buffer.
- Call `read` to read samples from the buffer.
- Call `peek` to read samples without changing the number of unread samples in the buffer.

## Creation

### Syntax

```
asyncBuff = dsp.AsyncBuffer  
asyncBuff = dsp.AsyncBuffer(cap)
```

### Description

`asyncBuff = dsp.AsyncBuffer` returns an async buffer System object, `asyncBuff`, using the default properties.

`asyncBuff = dsp.AsyncBuffer(cap)` sets the `Capacity` property to `cap`.

Example: `asyncBuff = dsp.AsyncBuffer(200000);`

### Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

### Capacity — Number of writable/readable rows in buffer

192000 (default) | positive integer

Number of writable/readable rows in the buffer, specified as a positive integer greater than or equal to 2. The number of rows during each write to the buffer must not exceed the capacity of the buffer. If the buffer is full and all the data within is unread, the object overwrites the oldest data with any new data that comes in. The `CumulativeOverrun` property returned by `info` gives the number of samples overrun per channel since the last call to `reset`. The number of samples overrun is the number of unread samples overwritten.

By default, this property has data type `int32`.

Example: `asyncBuff = dsp.AsyncBuffer(200000);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### NumUnreadSamples — Number of unread samples in each channel

0 (default) | integer

This property is read-only.

Number of unread samples in each channel (column) of the buffer, specified as an integer greater than or equal to 0. The total number of unread samples in the buffer is `NumUnreadSamples × numChann`. The variable `numChann` is the number of channels in the buffer. The number of channels in the buffer is the number of data columns in the first call to `write`.

The `CumulativeUnderrun` property returned by the `info` method gives the number of samples underrun per channel since the last call to `reset`. Underrun occurs if you attempt to read more samples than available.

Example: `asyncBuff = dsp.AsyncBuffer; input = randn(512,1); numUnreadSamples = write(asyncBuff,input)`

Data Types: `int32`

## Usage

To write and read samples from the async buffer:

- Create a `dsp.AsyncBuffer` object and set the properties of the object.
- Call `write` to write samples to the buffer.
- Call `read` to read samples from the buffer.
- Call `peek` to read samples without changing the number of unread samples in the buffer.

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

`release(obj)`

### Specific to `dsp.AsyncBuffer`

`info` Get cumulative overrun and underrun  
`read` Read data from buffer  
`write` Write data to buffer  
`peek` Read data from buffer without changing number of unread samples

### Common to All System Objects

`step` Run System object algorithm  
`release` Release resources and allow changes to System object property values and input characteristics  
`reset` Reset internal states of System object

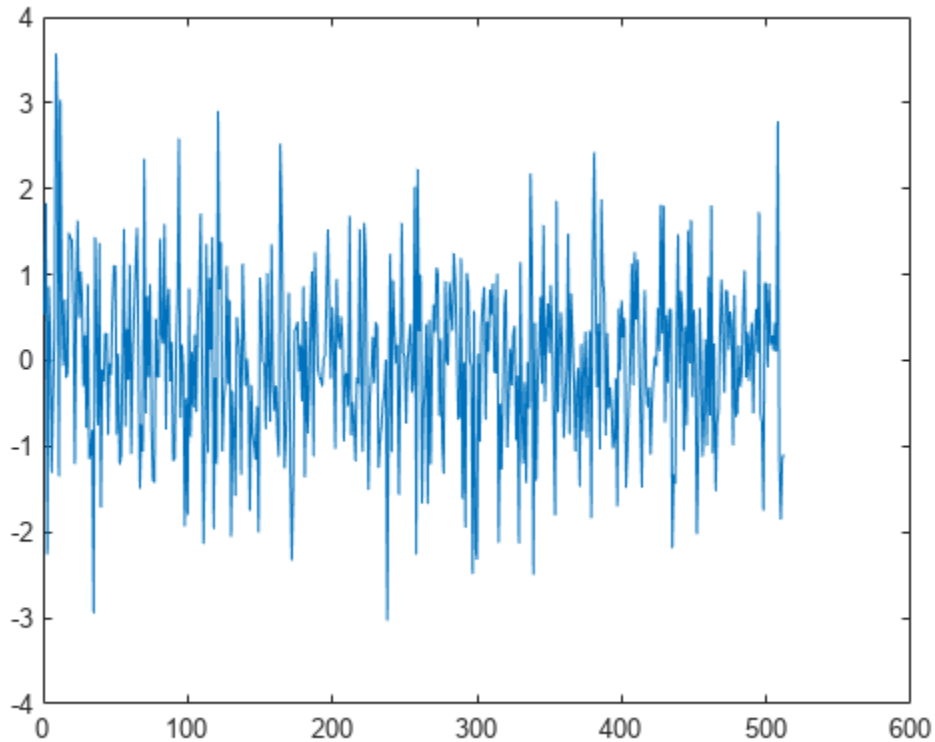
## Examples

### Read Variable Frame Sizes from Buffer

The `dsp.AsyncBuffer` System object™ supports reading variable frame sizes from the buffer.

Create a `dsp.AsyncBuffer` System object. The input is white Gaussian noise with a mean of 0, a standard deviation of 1, and a frame size of 512 samples. Write the input to the buffer using the `write` method.

```
asyncBuff = dsp.AsyncBuffer;  
input = randn(512,1);  
write(asyncBuff,input);  
plot(input)  
hold on
```



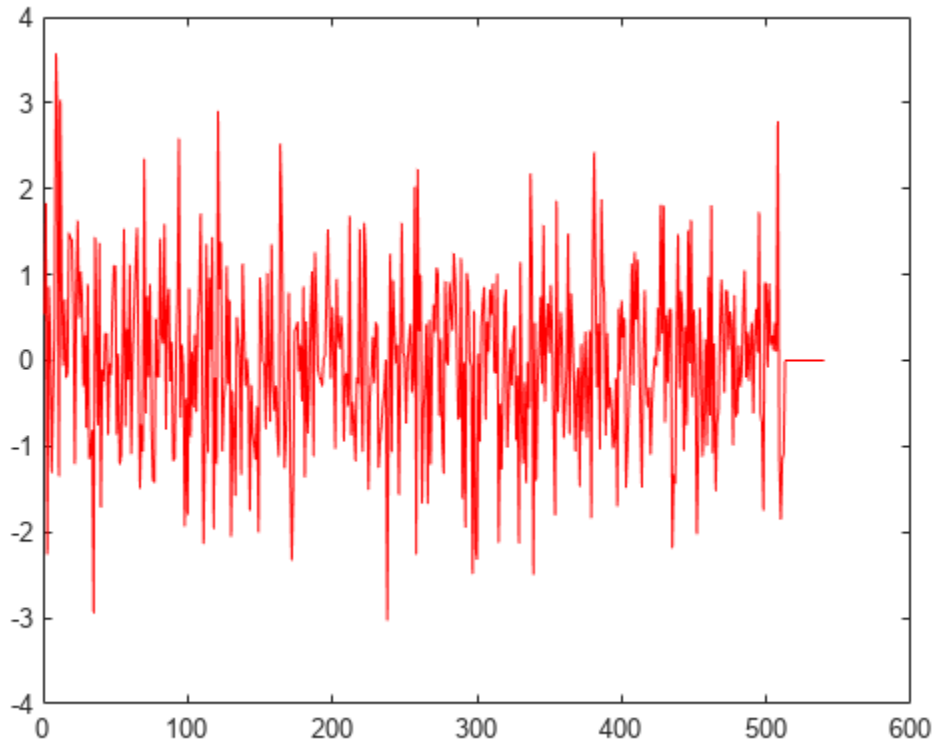
Store the data that is read from the buffer in `outTotal`.

Plot the input signal and data that is read from the buffer in the same plot. Read data from the buffer until all samples are read. In each iteration of the loop, `randi` determines the number of samples to read. Therefore, the signal is read in as a variable-size signal. The `prevIndex` variable keeps track of the previous index value that contains the data.

```

outTotal = zeros(size(input));
prevIndex = 0;
while asyncBuff.NumUnreadSamples ~= 0
    numToRead = randi([1,64]);
    out = read(asyncBuff,numToRead);
    outTotal(prevIndex+1:prevIndex+numToRead) = out;
    prevIndex = prevIndex+numToRead;
end
plot(outTotal,'r')
hold off

```



Verify that the input data and the data read from the buffer (excluding the underrun samples, if any) are the same. The cumulative number of overrun and underrun samples in the buffer is determined by the `info` function.

```
S = info(asyncBuff)
```

```
S = struct with fields:
    CumulativeOverrun: 0
    CumulativeUnderrun: 28
```

The `CumulativeUnderrun` field shows the number of samples underrun per channel. Underrun occurs if you attempt to read more samples than available.

### Write Variable Frame Sizes to Buffer

Write a sine wave of variable frame size to the buffer. Compute the FFT of the sine wave and visualize the result on an array plot.

Initialize the `dsp.AsyncBuffer`, `dsp.ArrayPlot`, and `dsp.FFT` System objects.

```
asynBuff = dsp.AsyncBuffer;
plotter = dsp.ArrayPlot;
fftObj = dsp.FFT('FFTLengthSource','Property','FFTLength',256);
```

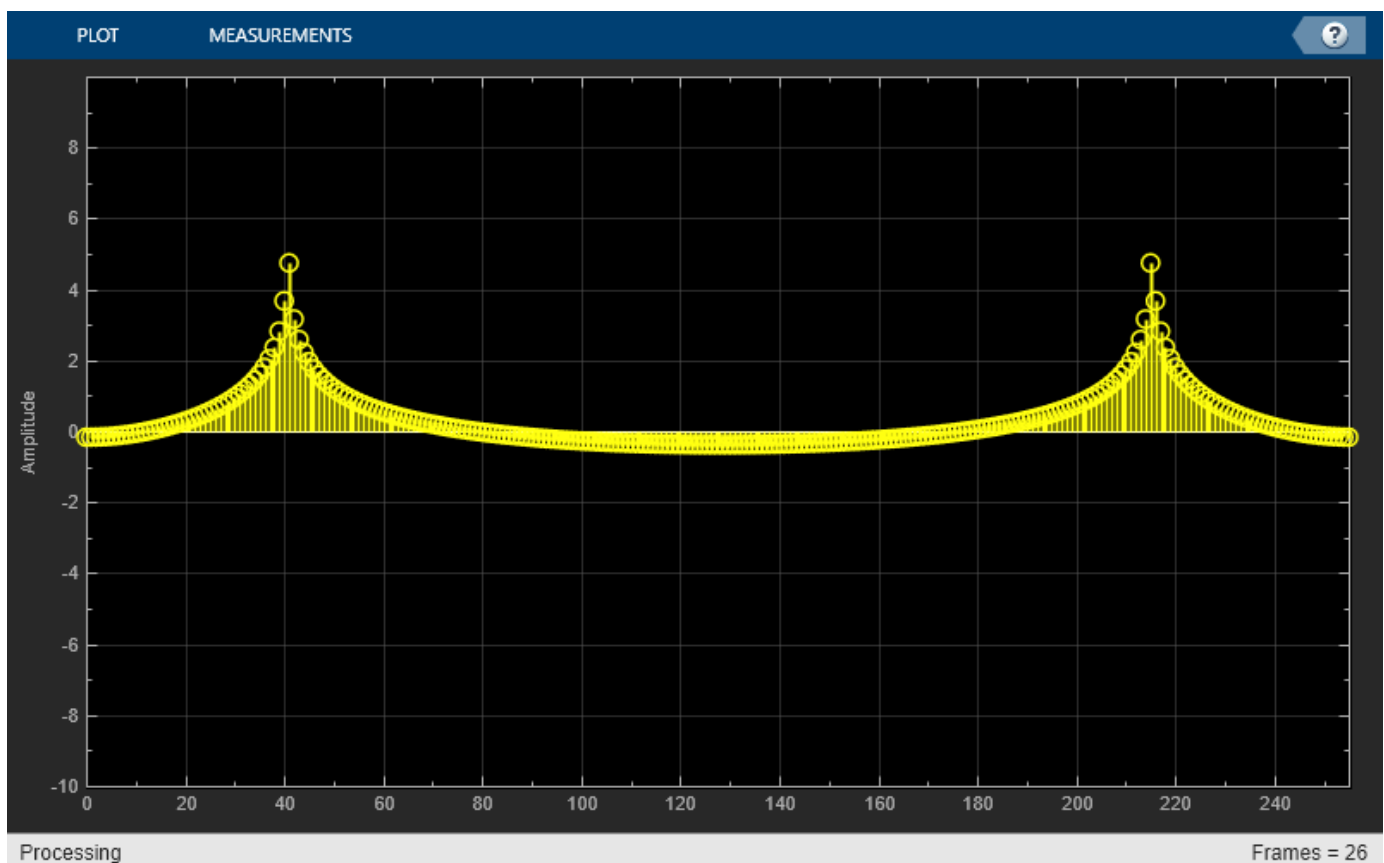
The sine wave is generated using the `sin` function in MATLAB. The `start` and `finish` variables mark the start and finish indices of each frame. If enough data is cached, read from the buffer and perform the FFT. View the FFT on an array plot.

```
start = 1;

for Iter = 1 : 2000
    numToWrite = randi([200,800]);
    finish = start + numToWrite;

    inputData = sin(start:finish)';
    start = finish + 1;

    write(asynBuff,inputData);
    while asynBuff.NumUnreadSamples >= 256
        x = read(asynBuff,256);
        X = abs(fft0bj(x));
        plotter(log(X));
    end
end
```



### Peek Data from Async Buffer

Read data from the async buffer without changing the number of unread samples using the `peek` function.

Create a `dsp.AsyncBuffer` System object™. The input is a column vector of 100 samples, 1 to 100. Write the data to the buffer.

```
asyncBuff = dsp.AsyncBuffer
asyncBuff =
  AsyncBuffer with properties:
      Capacity: 192000
  NumUnreadSamples: 0
```

```
input = (1:100)';
write(asyncBuff,input);
```

Peek at the first three samples. The output is [1 2 3]′.

```
out1 = peek(asyncBuff,3)
out1 = 3×1
     1
     2
     3
```

The `NumUnreadSamples` is 100, indicating that the peek function has not changed the number of unread samples in the buffer.

```
asyncBuff.NumUnreadSamples
ans = int32
    100
```

After peeking, read 50 samples using the `read` function. The output is [1:50]′.

```
out2 = read(asyncBuff,50)
out2 = 50×1
     1
     2
     3
     4
     5
     6
     7
     8
     9
    10
     :
```

The `NumUnreadSamples` is 50, indicating that the `read` function has changed the number of unread samples in the buffer.

```
asyncBuff.NumUnreadSamples
ans = int32
    50
```



Now peek again at the first three samples. The output is [51 52 53]'. Verify that the `NumUnreadSamples` is still 50.

```
out3 = peek(asyncBuff,3)
```

```
out3 = 3×1
```

```
51
52
53
```

```
asyncBuff.NumUnreadSamples
```

```
ans = int32
50
```

Read 50 samples again. The output now contains the sequence [51:100]'. Verify that `NumUnreadSamples` is 0.

```
out4 = read(asyncBuff)
```

```
out4 = 50×1
```

```
51
52
53
54
55
56
57
58
59
60
⋮
```

```
asyncBuff.NumUnreadSamples
```

```
ans = int32
0
```

## Limitations

Before calling the `read` method, you must initialize the buffer by calling either the `write` or `setup` method. For an example, see “Why Does the `dsp.AsyncBuffer` Object Error When You Call `read` Before `write`?”

## Version History

Introduced in R2017a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

### See Also

#### Functions

info | read | write | peek

#### Objects

dsp.Delay

#### Blocks

Buffer | Queue | Delay Line

#### Topics

“Why Does the dsp.AsyncBuffer Object Error When You Call read Before write?”

“Why Does Reading Data from the dsp.AsyncBuffer Object Give a Dimension Mismatch Error in MATLAB Function Block?”

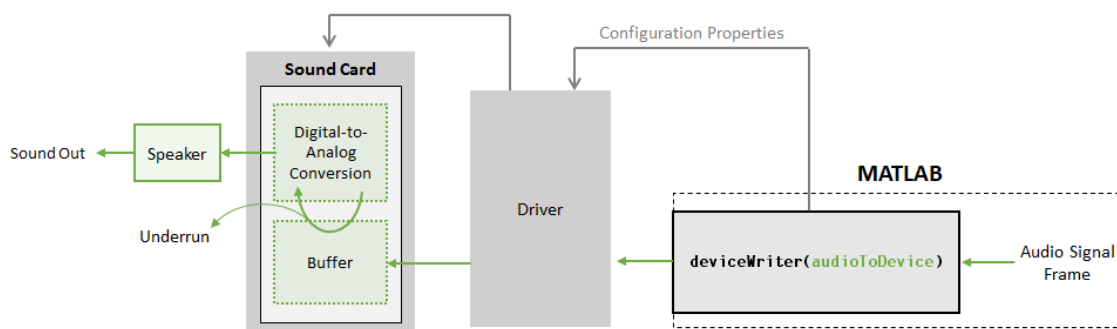
“High Resolution Spectral Analysis in MATLAB”

# audioDeviceWriter

Play to sound card

## Description

The `audioDeviceWriter` System object writes audio samples to an audio output device. Properties of the audio device writer specify the driver, the device, and device attributes such as sample rate, bit depth, and buffer size.



## Data Flow of Audio Device Writer

- Call the object to input an audio signal frame to the `audioDeviceWriter`.
- The `audioDeviceWriter` uses the specified driver to pass the frame (device input) to the buffer of your specified audio device.
- The audio device performs digital-to-analog conversion at the specified sample rate and bit depth.
- The audio device outputs an analog chunk to your speaker.

To stream data to an audio device:

- 1 Create the `audioDeviceWriter` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```

deviceWriter = audioDeviceWriter
deviceWriter = audioDeviceWriter(sampleRateValue)
deviceWriter = audioDeviceWriter( ___,Name,Value)
  
```

### Description

`deviceWriter = audioDeviceWriter` returns a System object, `deviceWriter`, that writes audio samples to an audio output device in real time.

`deviceWriter = audioDeviceWriter(sampleRateValue)` sets the `SampleRate` property to `sampleRateValue`.

`deviceWriter = audioDeviceWriter( ____, Name, Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

Example: `deviceWriter = audioDeviceWriter(48000, 'BitDepth', '8-bit integer')` creates a System object, `deviceWriter`, that operates at a 48 kHz sample rate and an 8-bit integer bit depth.

### Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see *System Design in MATLAB Using System Objects*.

#### Driver — Driver used to access audio device (Windows only)

'DirectSound' (default) | 'ASIO' | 'WASAPI'

Driver used to access your audio device, specified as 'DirectSound', 'ASIO', or 'WASAPI'.

- ASIO drivers do not come pre-installed on Windows machines. To use the 'ASIO' driver option, install an ASIO driver outside of MATLAB.

---

**Note** If `Driver` is specified as 'ASIO', use `asiosettings` to set the sound card buffer size to the buffer size of your `audioDeviceWriter` System object.

---

- WASAPI drivers are supported for exclusive-mode only.

ASIO and WASAPI drivers do not provide sample rate conversion. For ASIO and WASAPI drivers, set `SampleRate` to a sample rate supported by your audio device.

This property applies only on Windows machines. Linux machines always use the ALSA driver. Mac machines always use the CoreAudio driver.

To specify nondefault `Driver` values, you must have an Audio Toolbox license. If the toolbox is not installed, specifying nondefault `Driver` values returns an error.

Data Types: `char` | `string`

#### Device — Device used to play audio samples

default audio device (default) | character vector | string scalar

Device used to play audio samples, specified as a character vector or string scalar. Use `getAudioDevices` to list available devices for the selected driver.

Data Types: `char` | `string`

**SampleRate — Sample rate of signal sent to audio device (Hz)**

44100 (default) | positive integer

Sample rate of signal sent to audio device, in Hz, specified as a positive integer. The range of `SampleRate` depends on your audio hardware.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`**BitDepth — Data type used by the device**

'16-bit integer' (default) | '8-bit integer' | '24-bit integer' | '32-bit float'

Data type used by the device, specified as a character vector or string scalar. Before performing digital-to-analog conversion, the input data is cast to a data type specified by `BitDepth`.

To specify a nondefault `BitDepth`, you must have an Audio Toolbox license. If the toolbox is not installed, specifying a nondefault `BitDepth` returns an error.

Data Types: `char` | `string`**SupportVariableSizeInput — Support variable frame size**`false` (default) | `true`

Option to support variable frame size, specified as `true` or `false`.

- `false` -- If the `audioDeviceWriter` object is locked, the input must have the same frame size at each call. The buffer size of your audio device is the same as the input frame size.
- `true` -- If the `audioDeviceWriter` object is locked, the input frame size can change at each call. The buffer size of your audio device is specified through the `BufferSize` property.

Data Types: `char`**BufferSize — Buffer size of audio device**

4096 (default) | positive integer

Buffer size of audio device, specified as a positive integer.

---

**Note** If `Driver` is specified as 'ASIO', open the ASIO UI to set the sound card buffer size to the `BufferSize` value of your `audioDeviceWriter` System object.

---

**Dependencies**

To enable this property, set `SupportVariableSizeInput` to `true`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`**ChannelMappingSource — Source of mapping between input matrix and device channels**

'Auto' (default) | 'Property'

Source of mapping between columns of input matrix and channels of audio output device, specified as 'Auto' or 'Property'.

- 'Auto' -- Default settings determine the mapping between columns of input matrix and channels of audio output device. For example, suppose that your input is a matrix with four columns, and your audio device has four channels available. Column 1 of your input data writes to channel 1 of your device, column 2 of your input data writes to channel 2 of your device, and so on.

- 'Property' -- The ChannelMapping property determines the mapping between columns of input matrix and channels of audio output device.

Data Types: char | string

### ChannelMapping — Nondefault mapping between input matrix and device channels

[1:MaximumOutputChannels] (default) | scalar | vector

Nondefault mapping between columns of input matrix and channels of output device, specified as a scalar or vector of valid channel indices.

To selectively map between columns of the input matrix and your sound card's output channels, you must have an Audio Toolbox license. If the toolbox is not installed, specifying a nondefault ChannelMapping returns an error.

---

**Note** To ensure mono output on only one channel of a stereo device, use the default ChannelMapping setting and provide a stereo signal where one channel is all zeros.

**Example:** outputLeftOnly = [x(:,1) zeros(size(x,1),1)];

**Example:** outputRightOnly = [zeros(size(x,1),1) x(:,1)];

---

### Dependencies

To enable this property, set ChannelMappingSource to 'Property'.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

## Usage

### Syntax

```
numUnderrun = deviceWriter(audioToDevice)
```

### Description

numUnderrun = deviceWriter(audioToDevice) writes one frame of audio samples, audioToDevice, to the selected audio device and returns the number of audio samples underrun since the last call to deviceWriter.

**Note:** When you call the audioDeviceWriter System object, the audio device specified by the Device property is locked. An audio device can be locked by only one audioDeviceWriter at a time. To release the audio device, call release on your audioDeviceWriter System object.

### Input Arguments

#### audioToDevice — Audio to device

matrix

Audio signal to write to device, specified as a matrix. The columns of the matrix are treated as independent audio channels.

If `audioToDevice` is of data type `'double'` or `'single'`, the audio device writer clips values outside the range `[-1, 1]`. For other data types, the allowed input range is `[min, max]` of the specified data type.

Data Types: `single` | `double` | `int16` | `int32` | `uint8`

## Output Arguments

### **numUnderrun — Number of samples underrun**

scalar

Number of samples by which the audio device writer queue was underrun since the last call to `deviceWriter`.

Data Types: `uint32`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Specific to audioDeviceWriter

|                              |  |
|------------------------------|--|
| <code>getAudioDevices</code> | List available audio devices                         |
| <code>info</code>            | Characteristic information about audio device writer |

## Common to All System Objects

|                       |  |
|-----------------------|--|
| <code>clone</code>    | Create duplicate System object   |
| <code>isLocked</code> | Determine if System object is in use   |
| <code>release</code>  | Release resources and allow changes to System object property values and input characteristics |
| <code>reset</code>    | Reset internal states of System object   |
| <code>step</code>     | Run System object algorithm  |
| <code>setup</code>    | One-time set up tasks for System objects   |

## Examples

### Read from File and Write to Audio Device

Read an MP3 audio file and play it through your default audio output device.

Create a `dsp.AudioFileReader` object with default settings. Use the `audioinfo` function to return a structure containing information about the audio file.

```
fileReader = dsp.AudioFileReader('speech_dft.mp3');
fileInfo = audioinfo('speech_dft.mp3')

fileInfo = struct with fields:
    Filename: 'B:\matlab\toolbox\dsp\samples\speech_dft.mp3'
    CompressionMethod: 'MP3'
    NumChannels: 1
```

```
SampleRate: 22050
TotalSamples: 112893
Duration: 5.1199
  Title: []
  Comment: []
  Artist: []
BitRate: 64
```

Create an `audioDeviceWriter` object and specify the sample rate.

```
deviceWriter = audioDeviceWriter('SampleRate', fileInfo.SampleRate);
```

Call `setup` to reduce the computational load of initialization in an audio stream loop.

```
setup(deviceWriter, zeros(fileReader.SamplesPerFrame, ...
    fileInfo.NumChannels))
```

Use the `info` function to obtain the characteristic information about the device writer.

```
info(deviceWriter)
ans = struct with fields:
    Driver: 'DirectSound'
    DeviceName: 'Primary Sound Driver'
    MaximumOutputChannels: 2
```

In an audio stream loop, read an audio signal frame from the file, and write the frame to your device.

```
while ~isDone(fileReader)
    audioData = fileReader();
    deviceWriter(audioData);
end
```

Close the input file and release the device.

```
release(fileReader)
release(deviceWriter)
```

### Reduce Latency due to Output Device Buffer

*Latency* due to the output device buffer is the time delay of writing one frame of data. Modify default properties of your `audioDeviceWriter` System object™ to reduce latency due to device buffer size.

Create a `dsp.AudioFileReader` System object to read an audio file with default settings.

```
fileReader = dsp.AudioFileReader('speech_dft.mp3');
```

Create an `audioDeviceWriter` System object and specify the sample rate to match that of the audio file reader.

```
deviceWriter = audioDeviceWriter(...
    'SampleRate', fileReader.SampleRate);
```

Calculate the latency due to your device buffer, in seconds.



```
bufferLatency = fileReader.SamplesPerFrame/deviceWriter.SampleRate %#ok
bufferLatency = 0.0464
```

Set the `SamplesPerFrame` property of your `dsp.AudioFileReader` System object to 256. Calculate the buffer latency in seconds.

```
fileReader.SamplesPerFrame = 256;
bufferLatency = fileReader.SamplesPerFrame/deviceWriter.SampleRate
bufferLatency = 0.0116
```

### Determine and Decrease Underrun

*Underrun* refers to output signal silence, which occurs when the audio stream loop does not keep pace with the output device. Determine the underrun of an audio stream loop, add artificial computational load to the audio stream loop, and then modify properties of your `audioDeviceWriter` object to decrease underrun. Your results depend on your computer.

Create a `dsp.AudioFileReader` object, and specify the file to read. Use the `audioinfo` function to return a structure containing information about the audio file.

```
fileReader = dsp.AudioFileReader('speech_dft.mp3');
fileInfo = audioinfo('speech_dft.mp3');
```

Create an `audioDeviceWriter` object. Use the `SampleRate` of the file reader as the `SampleRate` of the device writer. Call `setup` to reduce the computational load of initialization in an audio stream loop.

```
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);
setup(deviceWriter,zeros(fileReader.SamplesPerFrame,...
    fileInfo.NumChannels))
```

Run your audio stream loop with input from file and output to device. Print the total samples underrun and the underrun in seconds.

```
totalUnderrun = 0;
while ~isDone(fileReader)
    input = fileReader();
    numUnderrun = deviceWriter(input);
    totalUnderrun = totalUnderrun + numUnderrun;
end
fprintf('Total samples underrun: %d.\n',totalUnderrun)
```

```
Total samples underrun: 0.
```

```
fprintf('Total seconds underrun: %d.\n',double(totalUnderrun)/double(deviceWriter.SampleRate))
```

```
Total seconds underrun: 0.
```

Release your `dsp.AudioFileReader` and `audioDeviceWriter` objects and set your counter variable to zero.

```
release(fileReader)
release(deviceWriter)
totalUnderrun = 0;
```

Use `pause` to mimic an algorithm that takes 0.075 seconds to process. The pause causes the audio stream loop to go slower than the device, which results in periods of silence in the output audio signal.

```
while ~isDone(fileReader)
    input = fileReader();
    numUnderrun = deviceWriter(input);
    totalUnderrun = totalUnderrun + numUnderrun;
    pause(0.075)
end
fprintf('Total samples underrun: %d.\n',totalUnderrun)
```

Total samples underrun: 68608.

```
fprintf('Total seconds underrun: %d.\n',double(totalUnderrun)/double(deviceWriter.SampleRate))
```

Total seconds underrun: 3.111474e+00.

Release your `audioDeviceReader` and `dsp.AudioFileWriter` and set the counter variable to zero.

```
release(fileReader)
release(deviceWriter)
totalUnderrun = 0;
```

Set the frame size of your audio stream loop to 2048. Because the `SupportVariableSizeInput` property of your `audioDeviceWriter` System object is set to `false`, the buffer size of your audio device is the same size as the input frame size. Increasing your device buffer size decreases underrun.

```
fileReader = dsp.AudioFileReader('speech_dft.mp3');
fileReader.SamplesPerFrame = 2048;
fileInfo = audioinfo('speech_dft.mp3');
```

```
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);
setup(deviceWriter,zeros(fileReader.SamplesPerFrame,fileInfo.NumChannels))
```

Calculate the total underrun.

```
while ~isDone(fileReader)
    input = fileReader();
    numUnderrun = deviceWriter(input);
    totalUnderrun = totalUnderrun + numUnderrun;
    pause(0.075)
end
fprintf('Total samples underrun: %d.\n',totalUnderrun)
```

Total samples underrun: 0.

```
fprintf('Total seconds underrun: %d.\n',double(totalUnderrun)/double(deviceWriter.SampleRate))
```

Total seconds underrun: 0.

The increased frame size reduces the total underrun of your audio stream loop. However, increasing the frame size also increases latency. Other approaches to reduce underrun include:

- Increasing the buffer size independent of input frame size. To increase buffer size independent of input frame size, you must first set `SupportVariableSizeInput` to `true`. This approach also increases latency.

- Decreasing the sample rate. Decreasing the sample rate reduces both latency and underrun at the cost of signal resolution.
- Choosing an optimal driver and device for your system.

## Version History

Introduced in R2016a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- “System Objects in MATLAB Code Generation” (MATLAB Coder)
- The executable generated from this System object relies on prebuilt dynamic library files (.dll files) included with MATLAB. Use the `packNGo` function to package the code generated from this object and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see “How To Run a Generated Executable Outside MATLAB”.

## See Also

`asiosettings` | `getAudioDevices` | Audio Device Writer | `dsp.AudioFileWriter` | `dsp.AudioFileReader`

### Topics

“How To Run a Generated Executable Outside MATLAB”

“Measure Audio Latency” (Audio Toolbox)

## getAudioDevices

List available audio devices

### Syntax

```
devices = getAudioDevices(deviceWriter)
```

### Description

`devices = getAudioDevices(deviceWriter)` returns a list of audio devices that are available and compatible with your I/O audio object, `deviceWriter`.

### Input Arguments

**deviceWriter** — Audio I/O object

`audioDeviceWriter` object

Audio I/O object, specified as an `audioDeviceWriter` object.

Data Types: `object`

### Output Arguments

**devices** — List of available and compatible devices

`array`

List of available and compatible devices. The list of audio devices depends on the specified `Driver` property of your object.

Data Types: `cell`

## Version History

Introduced in R2016a

### See Also

**Objects**

`audioDeviceWriter`

# dsp.AudioFileReader

**Package:** dsp

Stream from audio file

## Description

The `dsp.AudioFileReader` System object reads audio samples from an audio file.

To read audio samples from an audio file:

- 1 Create the `dsp.AudioFileReader` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
afr = dsp.AudioFileReader
afr = dsp.AudioFileReader(Filename)
afr = dsp.AudioFileReader(Name,Value)
```

### Description

`afr = dsp.AudioFileReader` returns an audio file reader System object, `afr` that reads audio from an audio file.

`afr = dsp.AudioFileReader(Filename)` returns an audio file reader object, `afr`, with `Filename` property set to `Filename`.

`afr = dsp.AudioFileReader(Name,Value)` returns an audio file reader System object, `afr`, with each specified property set to the specified value. Enclose each property name in single quotes. Unspecified properties have default values.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

### Filename — Name of audio file from which to read

'speech\_dft.mp3' (default) | character vector | string scalar

Specify the name of an audio file as a character vector or a string scalar. If `FilenameIsTunableInCodegen` is set to `false` and the file is on MATLAB path, then you do not need to specify the full name of the file. If `FilenameIsTunableInCodegen` is set to `true`, then the file name must either exist in the current directory, or you must specify the full file path. The file name can be an http web address like `'http://audio.wgbh.org:8004/'`. For an example, see “Read and Play Back Audio File from http Web Address” on page 4-115.

The `Filename` property is tunable in generated code. That is, you can pass the name of the audio file as an input while running the code generated from this object. File attributes such as the audio format, the number of audio channels, the sample rate, and the bit rate are not tunable and must match the attributes of the prototype audio file you specify through the `CodegenPrototypeFile` property. The specified prototype audio file determines the attributes and the type of audio files that can be read by the generated code. For an example, see “Tunable Audio File Name in Generated Code” on page 4-116.

The following table lists the supported audio file formats.

| Platforms   | File Name Extensions  |
|-------------|---|
| Windows     | .wav, .wma, .avi, .aif, .aifc, .aiff, .mp3, .opus, .au, .snd, .mp4, .m4a, .flac, .ogg, .mov |
| Non-Windows | .avi, .mp3, .mp4, .m4a, .wav, .flac, .ogg, .aif, .aifc, .aiff, .opus, .au, .snd, .mov       |

#### **PlayCount — Number of times to play file**

1 (default) | positive integer

Specify a positive integer as the number of times to play the file.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

#### **SamplesPerFrame — Number of samples in audio frame**

1024 (default) | positive integer

Specify the number of samples in an audio frame as a positive, scalar integer value.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

#### **OutputDataType — Data type of output**

'double' (default) | 'single' | 'int16' | 'uint8'

Set the data type of the audio data output from the audio file reader object. Specify the data type as 'double', 'single', 'int16', or 'uint8'.

#### **FilenameIsTunableInCodegen — Enable file name tunability in generated code**

false (default) | true

Set this property to true to enable tunability of `Filename` in generated C/C++ code. The file with the specified `Filename` must either exist in the current directory, or you must specify the full file path.

If `FilenameIsTunableInCodegen` is set to `true`, file attributes such as the audio format, the number of audio channels, the sample rate, and the bit rate are not tunable and must match the attributes of the prototype audio file you specify through the `CodegenPrototypeFile` property. The specified prototype audio file determines the attributes and the type of audio files that can be read by the generated code. For example, if the specified prototype file is a `.wav` file, then the generated

code can only read .wav files. If the specified prototype file sample rate is 44100 Hz, then the generated code can read files with a sample rate of 44100 Hz.

Data Types: `logical`

### CodegenPrototypeFile — Prototype audio file for code generation

'speech\_dft.mp3' (default) | character vector | string scalar

Specify the name of the prototype audio file used in code generation as a character vector or a string scalar. Specify the full path for the file only if the file is not on the MATLAB path. The file name can be an http web address, like 'http://audio.wgbh.org:8004/'.

The attributes of the audio file specified in `Filename` (such as the audio format, the number of audio channels, the sample rate, and the bite rate) must match the attributes of the audio file specified in `CodegenPrototypeFile`. The specified prototype audio file determines the attributes and the type of audio files that can be read by the generated code. For example, if the specified prototype file is a .wav file, then the generated code can only read .wav files. If the specified prototype file sample rate is 44100 Hz, then the generated code can read files with a sample rate of 44100 Hz.

The following table lists the supported audio file formats:

| Platforms   | File Name Extensions  |
|-------------|---|
| Windows     | .wav, .wma, .avi, .aif, .aifc, .aiff, .mp3, .opus, .au, .snd, .mp4, .m4a, .flac, .ogg, .mov |
| Non-Windows | .avi, .mp3, .mp4, .m4a, .wav, .flac, .ogg, .aif, .aifc, .aiff, .opus, .au, .snd, .mov       |

### Dependencies

This property applies only when the `FilenameIsTunableInCodegen` property is set to `true`.

### SampleRate — Sampling rate of audio file

positive scalar

This property is read-only.

This property displays the sample rate, in Hz, of the audio file.

Data Types: `double`

### ReadRange — Range of samples to be read

[1 inf] (default) | two-element row vector

Specify the sample range from which to read, as a vector in the form of [`StartSample` `EndSample`], where `StartSample` is the sample at which file reading starts, and `EndSample` is the sample at which file reading stops.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

## Usage

### Syntax

```
audio = afr()
```

```
[audio,eof] = afr()
```

**Description**

`audio = afr()` outputs one frame of audio samples, `audio`. You can specify the number of times to play the file using the `PlayCount` property. After playing the file for the number of times you specify, `audio` contains silence.

`[audio,eof] = afr()` returns an end-of-file indicator, `eof`. `eof` is true each time the output `audio` contains the last audio sample in the file.

**Output Arguments****audio — Audio samples**

column vector

One frame of audio samples, returned as a column vector of length equal to the value you specify in the `SamplesPerFrame` property. The data type of the audio output is specified in the `OutputDataType` property.

Data Types: `single` | `double` | `int16` | `uint8`

**eof — End-of-file indicator**

1 | 0

End-of-file indicator, returned as either a 1 or a 0. A value of 1 is output when `audio` contains the last audio sample in the file.

Data Types: `logical`

**Object Functions**

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

**Specific to `dsp.AudioFileReader`**

`info` Information about specific audio file

`isDone` End-of-file status (logical)

**Common to All System Objects**

`step` Run System object algorithm

`release` Release resources and allow changes to System object property values and input characteristics

`reset` Reset internal states of System object

**Examples****Read and Play Back Audio File**

Read and play back an audio file using the standard audio output device.



You can choose to read the entire data or specify a range of data to read from using the `ReadRange` property. By default, `ReadRange` is set to `[1 inf]`, indicating the file reader to read the entire data from the source. In this example, set `ReadRange` to `3Fs`, indicating the file reader to read the first 3 seconds of the data.

```
afr = dsp.AudioFileReader('speech_dft.mp3','ReadRange',[1 3*22050]);
adw = audioDeviceWriter('SampleRate', afr.SampleRate);

while ~isDone(afr)
    audio = afr();
    adw(audio);
end
release(afr);
release(adw);
```

### Read and Play Back Audio File from http Web Address

Read audio data from an http web address using the `dsp.AudioFileReader` System object™. Play back the data using the `audioDeviceWriter` System object.

#### Initialization

Create an audio file reader which reads data from `http://audio.wgbh.org:8004/`. Set the sample rate of the audio device writer to be the same as that of the audio file reader.

```
afr = dsp.AudioFileReader('http://audio.wgbh.org:8004/')
```

```
afr =
    dsp.AudioFileReader with properties:

        Filename: 'http://audio.wgbh.org:8004/'
        PlayCount: 1
        SamplesPerFrame: 1024
        OutputDataType: 'double'
        SampleRate: 44100
        ReadRange: [1 Inf]
```

```
adw = audioDeviceWriter(afr.SampleRate)
```

```
adw =
    audioDeviceWriter with properties:

        Driver: 'DirectSound'
        Device: 'Default'
        SampleRate: 44100

    Show all properties
```

### Read and Play Back

Read a specific amount of data from the web address directly and play the data back using the audio device writer.

```
for i = 1:1000
    audio = afr();
    adw(audio);
end
```

Close the input file and the audio output device.

```
release(afr)
release(adw)
```

### **Tunable Audio File Name in Generated Code**

Generate a MEX file from a function named `writeAudio`. This function reads an audio signal from the `funky-stereo.wav` file, decimates the signal by a factor of 2, and writes the decimated signal to a specified output file.

The `dsp.AudioFileReader` object reads the audio signal from `funky-stereo.wav` file. The `funky-stereo.wav` file has two channels, a sample rate of 44100 Hz, and a bit rate of 1411 kbps. The `CodegenPrototypeFile` property of the object is set to `rock-stereo.wav` file. The `rock-stereo.wav` file has the same file attributes, such as the number of audio channels, sample rate, and bit rate, as the `funky-stereo.wav` file. The `dsp.FIRDecimator` object decimates the input audio signal by a factor of 2. The `dsp.AudioFileWriter` object writes the decimated signal to the output file `myoutput.wav`. Due to the decimation process, the output file has a sample rate of 22050 Hz and a bit rate of 2822 kbps.

```
type writeAudio.m

function writeAudio(readfile,writefile)

afr = dsp.AudioFileReader('FilenameIsTunableInCodegen',true,...
    'CodegenPrototypeFile','rock-stereo.wav');
afr.Filename = readfile;
% Filename is funky-stereo.wav and CodegenPrototypeFile is
% rock-stereo.wav.

firdec = dsp.FIRDecimator(2,'auto'); % decimate by 2

afw = dsp.AudioFileWriter('SampleRate',22050);
afw.Filename = writefile;
while ~isDone(afr)
    audio = afr();
    audiod = firdec(audio);
    afw(audiod);
end

release(afr);
release(afw);

end
```

For generating code, specify file names to be variable-length character vectors of maximum length 500.

```
readfilename = coder.typeof('a',[1 500],[0 1]);
writefilename = coder.typeof('b',[1 500],[0 1]);
```

Generate a MEX file using the `codegen` function.

```
codegen writeAudio -args {readfilename,writefilename}
```

Code generation successful.

```
writeAudio_mex('funky-stereo.wav','myoutput.wav');
```

## Limitations

- For MP3, MPEG-4 AAC, and AVI audio files on Windows 7 or later and Linux platforms, `dsp.AudioFileReader` object can read fewer samples than expected. On Windows platforms, this is due to a limitation in the underlying Media Foundation framework. On Linux platforms, this is due to a limitation in the underlying GStreamer framework. If you require sample-accurate reading, work with WAV or FLAC files.
- To read OPUS files on a Linux machine, ensure that you have installed `libsndfile 1.0.29` or a later version. On Windows and Mac, this file is shipped with MATLAB and is available under `$MATLABROOT/bin/<platform>`.

## Algorithms

This object implements the algorithm, inputs, and outputs described on the From Multimedia File block reference page. The object properties correspond to the block parameters, except:

- The object has no corresponding property for the **Inherit sample time from file** block parameter. The object always inherits the sample time from the file.
- The object has no corresponding property for the **Output end-of-file indicator** parameter. The object always outputs EOF as the last output.
- The object has no corresponding property for the **Multimedia Outputs** parameter because audio is the only supported output.
- The object has no corresponding property for the **Image signal** block parameter.
- The object has no corresponding property for the **Output color format** parameter.
- The object has no corresponding property for the **Video output data type** parameter.

## Version History

**Introduced in R2012a**

### **R2022b: Support for OPUS audio file format**

The `dsp.AudioFileReader` object supports audio files in the OPUS file format (`.opus`).

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- See “System Objects in MATLAB Code Generation” (MATLAB Coder).
- The executable generated from this System object relies on prebuilt dynamic library files (.dll files) included with MATLAB. Use the `packNGo` function to package the code generated from this object and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see “How To Run a Generated Executable Outside MATLAB”.

### See Also

#### Functions

`info` | `isDone`

#### Objects

`dsp.AudioFileWriter`

#### Topics

“How To Run a Generated Executable Outside MATLAB”

# dsp.AudioFileWriter

**Package:** dsp

Stream to audio file

## Description

The `dsp.AudioFileWriter` System object writes audio samples to an audio file.

To write audio samples to an audio file:

- 1 Create the `dsp.AudioFileWriter` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
afw = dsp.AudioFileWriter
afw = dsp.AudioFileWriter(Filename)
afw = dsp.AudioFileWriter(Name,Value)
```

### Description

`afw = dsp.AudioFileWriter` returns an audio file writer System object, `afw`. This object writes audio samples to an audio file.

`afw = dsp.AudioFileWriter(Filename)` returns an audio file writer System object, `afw`. This object has the `Filename` property set to `Filename`.

`afw = dsp.AudioFileWriter(Name,Value)` returns an audio file writer object with each specified property set to the specified value. Enclose each property name in single quotes. Unspecified properties have default values.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

### **Filename — Name of audio file to which to write**

'output.wav' (default) | character vector | string scalar

Specify the name of the audio file as a character vector or a string scalar.

The `Filename` property is tunable in generated code. That is, you can pass the name of the audio file as an input while running the code generated from this object. For an example, see “Tunable Audio File Name in Generated Code” on page 4-116.

**FileFormat — Audio file format**

'WAV' (default) | 'AVI' | 'FLAC' | 'OGG' | 'MPEG4' | 'OPUS' | 'WMA'

Specify which audio file format the object writes. On Microsoft platforms, select one of 'AVI', 'WAV', 'FLAC', 'OGG', 'MPEG4', 'OPUS', 'WMA'. On Linux platforms, select one of 'AVI', 'WAV', 'FLAC', 'OPUS', or 'OGG'. On macOS platforms, select one of 'AVI', 'WAV', 'FLAC', 'OGG', 'OPUS', or 'MPEG4'. These abbreviations correspond to the following file formats:

- 'AVI' -- Audio-Video Interleave
- 'WAV' -- Microsoft WAVE Files
- 'WMA' -- Windows Media Audio
- 'FLAC' -- Free lossless audio Codec
- 'OGG' -- Ogg/Vorbis compressed audio file
- 'OPUS' -- Opus compressed audio file
- 'MPEG4' -- MPEG-4 AAC File — You can use both .m4a and .mp4 extensions

The default is 'WAV'.

**SampleRate — Sampling rate of audio data stream**

44100 (default) | positive scalar

Specify the sample rate of the input audio data as a positive, numeric scalar value.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**Compressor — Algorithm that compresses audio data**

'None (uncompressed)' (default) | 'CCITT A-Law' | 'CCITT u-Law' | 'GSM 6.10' | 'IMA ADPCM' | 'Microsoft ADPCM' | 'PCM'

Specify the type of compression algorithm the audio file writer uses to compress the audio data. Compression reduces the size of the audio file. Select 'None (uncompressed)' to save uncompressed audio data to the file. The other options available reflect the audio compression algorithms installed on your system. You can use tab completion to query valid Compressor options for your computer by typing `H.Compressor = '` and then pressing the tab key.

**Dependencies**

This property applies when writing WAV or AVI files on Windows platforms.

**Data Type — Data type of the uncompressed audio**

'int16' (default) | 'double' | 'single' | 'inherit' | 'int24' | 'int32' | 'uint8'

Specify the type of uncompressed audio data written to the file as 'int16', 'double', 'single', 'inherit', 'int24', 'int32', or 'uint8'.

**Dependencies**

This property only applies when writing uncompressed WAV files.

## Usage

## Syntax

```
afw(audio)
```

## Description

`afw(audio)` writes one frame of audio samples, `audio`, to the output file specified by `Filename`. `audio` is either a vector for mono audio input or an  $M$ -by- $N$  matrix for  $N$ -channel audio input respectively.

## Input Arguments

### **audio** — Audio samples

column vector | matrix

One frame of audio samples, returned as a column vector or a matrix. A column vector input indicates a mono audio input. An  $M$ -by- $N$  matrix indicates an  $N$ -channel audio input.

If the input is fixed-point, the input must be a signed fixed-point input with power-of-two slope and zero bias.

Data Types: `single` | `double` | `int16` | `int32` | `uint8` | `fi`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Common to All System Objects

|                      |  |
|----------------------|--|
| <code>step</code>    | Run System object algorithm  |
| <code>release</code> | Release resources and allow changes to System object property values and input characteristics |
| <code>reset</code>   | Reset internal states of System object   |

## Examples

### Write an Audio Signal to WAV File

Decimate an audio signal, and write it to disk as a WAV file.

```
afr = dsp.AudioFileReader('OutputDataType',...
    'double');
firdec = dsp.FIRDecimator; % decimate by 2
afw = dsp.AudioFileWriter...
    ('speech_dft.wav', ...
    'SampleRate', afr.SampleRate/2);
while ~isDone(afw)
```

```
        audio = afr();
        audiod = firdec(audio);
        afw(audiod);
end

release(afr);
release(afw);
```

### Tunable Audio File Name in Generated Code

Generate a MEX file from a function named `writeAudio`. This function reads an audio signal from the `funky-stereo.wav` file, decimates the signal by a factor of 2, and writes the decimated signal to a specified output file.

The `dsp.AudioFileReader` object reads the audio signal from `funky-stereo.wav` file. The `funky-stereo.wav` file has two channels, a sample rate of 44100 Hz, and a bit rate of 1411 kbps. The `CodegenPrototypeFile` property of the object is set to `rock-stereo.wav` file. The `rock-stereo.wav` file has the same file attributes, such as the number of audio channels, sample rate, and bit rate, as the `funky-stereo.wav` file. The `dsp.FIRDecimator` object decimates the input audio signal by a factor of 2. The `dsp.AudioFileWriter` object writes the decimated signal to the output file `myoutput.wav`. Due to the decimation process, the output file has a sample rate of 22050 Hz and a bit rate of 2822 kbps.

```
type writeAudio.m

function writeAudio(readfile,writefile)

afr = dsp.AudioFileReader('FilenameIsTunableInCodegen',true,...
    'CodegenPrototypeFile','rock-stereo.wav');
afr.Filename = readfile;
% Filename is funky-stereo.wav and CodegenPrototypeFile is
% rock-stereo.wav.

firdec = dsp.FIRDecimator(2,'auto'); % decimate by 2

afw = dsp.AudioFileWriter('SampleRate',22050);
afw.Filename = writefile;
while ~isDone(afr)
    audio = afr();
    audiod = firdec(audio);
    afw(audiod);
end

release(afr);
release(afw);

end
```

For generating code, specify file names to be variable-length character vectors of maximum length 500.

```
readfilename = coder.typeof('a',[1 500],[0 1]);
writefilename = coder.typeof('b',[1 500],[0 1]);
```

Generate a MEX file using the `codegen` function.



```
codegen writeAudio -args {readfilename,writefilename}
```

Code generation successful.

```
writeAudio_mex('funky-stereo.wav','myoutput.wav');
```

## Limitations

The following platform-specific restrictions apply when writing these files:

### Windows 7

- Only sample rates of 44100 Hz and 48000 Hz are supported for the MPEG-4 AAC file format. For other file formats, there is no restriction on the sample rate.
- Only mono or stereo outputs are allowed for the MPEG-4 AAC file format. For all other formats, more than two audio output channels are allowed.
- The output data is padded on both the front and back of the signal, with extra samples of silence.

Windows AAC encoder places sharp fade-in and fade-out on audio signals, causing the signals to be slightly longer in samples when written to disk.

- A minimum of 1025 samples per channel must be written to the MPEG-4 AAC file.

### macOS

- Only mono or stereo outputs are allowed for MPEG-4 AAC file format. For all other formats, more than two audio output channels are allowed.
- Not all sampling rates are supported, although the Mac Audio Toolbox API does not explicitly specify a restriction.

### Linux

- To support OPUS file format on a Linux machine, you must have `libsndfile 1.0.29` or a later version installed. On Windows and Mac, this file is shipped with MATLAB and is available under `$MATLABROOT/bin/<platform>`.

## Algorithms

This object implements the algorithm, inputs, and outputs described on the To Multimedia File block reference page. The object properties correspond to the block parameters, except:

- The object `FileFormat` property does not support video-only file formats.
- The object has no corresponding property for the **Write** parameter. The object writes only audio content to files.
- The object has no corresponding property for the **Video compressor** parameter.
- The object has no corresponding property for the **File color format** parameter.
- The object has no corresponding property for the **Image signal** parameter.

## Version History

**Introduced in R2012a**

### **R2022b: Support for OPUS audio file format**

The `dsp.AudioFileWriter` object supports OPUS file format (.opus).

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- See “System Objects in MATLAB Code Generation” (MATLAB Coder).
- The executable generated from this System object relies on prebuilt dynamic library files (.dll files) included with MATLAB. Use the `packNGo` function to package the code generated from this object and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see “How To Run a Generated Executable Outside MATLAB”.

## **See Also**

### **Objects**

`dsp.AudioFileReader`

### **Topics**

“How To Run a Generated Executable Outside MATLAB”

# dsp.Autocorrelator

**Package:** dsp

(Removed) Autocorrelation sequence

---

**Note** `dsp.Autocorrelator` has been removed. Use `xcorr` instead. For more information, see “Compatibility Considerations”.

---

## Description

The `Autocorrelator` object returns the autocorrelation sequence for a discrete-time, deterministic input, or the autocorrelation sequence estimate for a discrete-time, wide-sense stationary (WSS) random process at positive lags.

To obtain the autocorrelation sequence:

- 1 Create the `dsp.Autocorrelator` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
ac = dsp.Autocorrelator
ac = dsp.Autocorrelator(Name,Value)
```

### Description

`ac = dsp.Autocorrelator` returns an autocorrelator, `ac`, that computes the autocorrelation along the first dimension of an  $N$ -D array. By default, the autocorrelator computes the autocorrelation at lags from zero to  $N - 1$ , where  $N$  is the length of the input vector or the row dimension of the input matrix. Inputting a row vector results in a row of zero-lag autocorrelation sequence values, one for each column of the row vector. The default autocorrelator returns the unscaled autocorrelation and performs the computation in the time domain.

`ac = dsp.Autocorrelator(Name,Value)` returns an autocorrelator, `ac`, with each property set to the specified value.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

### **MaximumLagSource — Source of maximum lag**

Auto (default) | Property

Specify how to determine the range of lags for the autocorrelation as `Auto` or `Property`. If the value of `MaximumLagSource` is `Auto`, the autocorrelator computes the autocorrelation over all nonnegative lags in the interval  $[0, N-1]$ , where  $N$  is the length of the first dimension of the input. Otherwise, the object computes the autocorrelation using lags in the range  $[0, \text{MaximumLag}]$ .

### **MaximumLag — Maximum positive lag**

1 (default) | 0 | positive integer

Specify the maximum lag as an integer greater than or equal to 0. The `MaximumLag` must be less than the length of the input data.

### **Dependencies**

This property applies only when the `MaximumLagSource` property is `Property`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **Scaling — Autocorrelation function scaling**

None (default) | Biased | Unbiased | Unity at zero-lag

Specify the scaling to apply to the output as `None`, `Biased`, `Unbiased`, or `Unity at zero-lag`. Set this property to `None` to generate the autocorrelation function without scaling. This option is appropriate if you are computing the autocorrelation of a nonrandom (deterministic) input.

The `Biased` option scales the autocorrelation by  $1/N$ , where  $N$  is the length of the input data. Scaling by  $1/N$  yields a biased, finite-sample approximation to the theoretical autocorrelation of a WSS random process. In spite of the bias, scaling by  $1/N$  has the desirable property that the sample autocorrelation matrix is nonnegative definite, a property possessed by the theoretical autocorrelation matrices of all wide-sense stationary random processes. The Fourier transform of the biased autocorrelation estimate is the *periodogram*, a widely used estimate of the power spectral density of a WSS process.

The `Unbiased` option scales the estimate of the autocorrelation by  $1/N - 1$ . Scaling by  $N - 1$  produces an unbiased estimate of the theoretical autocorrelation. However, using the unbiased option, you can obtain an estimate of the autocorrelation function that fails to have the nonnegative definite property.

Use the `Unity at zero-lag` option to normalize the autocorrelation estimate as identically one at lag zero. The default is `None`.

### **Method — Domain for computing autocorrelations**

Time Domain (default) | Frequency Domain

Specify the domain for computing autocorrelations as `Time Domain` or `Frequency Domain`. You must set this property to `Time Domain` for fixed-point signals.

### **Fixed-Point Properties**

#### **FullPrecisionOverride — Full precision override for fixed-point arithmetic**

true (default) | false

Specify whether to use full precision rules. If you set `FullPrecisionOverride` to `true`, which is the default, the object computes all internal arithmetic and output data types using full precision rules. These rules provide the most accurate fixed-point numerics. It also turns off the display of other fixed-point properties because they do not apply individually. These rules guarantee that no quantization occurs within the object. Bits are added, as needed, to ensure that no roundoff or overflow occurs. If you set `FullPrecisionOverride` to `false`, fixed-point data types are controlled through individual fixed-point property settings. For more information, see “Full Precision for Fixed-Point System Objects”.

### **RoundingMethod — Rounding method for fixed-point operations**

Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Specify the rounding method as `Ceiling`, `Convergent`, `Floor`, `Nearest`, `Round`, `Simplest`, or `Zero`.

#### **Dependencies**

This property applies only when you set the `Method` property to `Time Domain` and the object is not in full precision mode.

### **OverflowAction — Overflow action for fixed-point operations**

Wrap (default) | Saturate

Specify the overflow action as `Wrap` or `Saturate`.

#### **Dependencies**

This property applies only when you set the `Method` property to `Time Domain` and the object is not in full precision mode.

### **ProductDataType — Product word and fraction lengths**

Full precision (default) | Same as input | Custom

Specify the product fixed-point data type as one of `Full precision`, `Same as input`, or `Custom`.

#### **Dependencies**

This property applies only when you set the `Method` property to `Time Domain`.

### **CustomProductDataType — Product word and fraction lengths**

`numerictype([], 32, 30)` (default) | `numerictype`

Specify the product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`.

#### **Dependencies**

This property applies only when you set the `Method` property to `Time Domain` and the `ProductDataType` property to `Custom`.

### **AccumulatorDataType — Accumulator word and fraction lengths**

Full precision (default) | Same as product | Same as input | Custom

Specify the accumulator fixed-point data type as one of `Full precision`, `Same as product`, `Same as input`, or `Custom`.

#### **Dependencies**

This property applies only when the `Method` property is `Time Domain`.

**CustomAccumulatorDataType — Accumulator word and fraction lengths**`numerictype([],32,30) (default) | numerictype`

Specify the accumulator fixed-point type as a scaled `numerictype` object with a Signedness of `Auto`.

**Dependencies**

This property applies only when you set the `Method` property to `Time Domain` and the `AccumulatorDataType` property to `Custom`.

**OutputDataType — Output word and fraction lengths**`Same as accumulator (default) | Same as product | Same as input | Custom`

Specify the output fixed-point data type as `Same as accumulator`, `Same as product`, `Same as input`, or `Custom`.

**Dependencies**

This property applies only when the `Method` property is `Time Domain`.

**CustomOutputDataType — Output word and fraction lengths**`numerictype([],16,15) (default) | numerictype`

Specify the output fixed-point type as a scaled `numerictype` object with a Signedness of `Auto`.

**Dependencies**

This property applies only when you set the `Method` property to `Time Domain` and the `OutputDataType` property to `Custom`.

**Usage****Syntax**`y = ac(x)`**Description**

`y = ac(x)` computes the autocorrelation sequence `y` for the columns of the input `x`.

**Input Arguments****x — Data input**`vector | matrix | N-D array`

Data input, specified as a vector, matrix, or an  $N$ -D array. The object accepts real-valued or complex-valued multichannel and multidimensional inputs. The input can be a fixed-point signal when you set the `Method` property to `'Time Domain'`. When the input signal is complex, the output signal is complex.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

## Output Arguments

### **y** — Autocorrelated output

vector | matrix | *N*-D array

Autocorrelated output of the two input signals. The size, data type, and complexity of the output matches that of the input signal.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fi

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Common to All System Objects

|         |  |
|---------|--|
| step    | Run System object algorithm  |
| release | Release resources and allow changes to System object property values and input characteristics |
| reset   | Reset internal states of System object   |

## Examples

### Autocorrelation of Noisy Sine Wave

Compute the autocorrelation of a sine wave in white Gaussian noise with approximate 95% upper and lower confidence limits.

```
S = rng('default');
% Sine wave with period N=4
x = 1.4*cos(pi/2*(1:100))+randn(100,1);
MaxLag = 20;
ac = dsp.Autocorrelator('MaximumLagSource',...
'Property','MaximumLag',MaxLag,'Scaling','Unity at zero-lag');
SigAutocorr = ac(x);
stem(SigAutocorr,'b','markerfacecolor',[0 0 1]);
line(1:MaxLag+1,1.96/sqrt(100)*ones(MaxLag+1,1),...
'linestyle','-','linewidth',2);
line(1:MaxLag+1,-1.96/sqrt(100)*ones(MaxLag+1,1),...
'linestyle','-','linewidth',2);
axis([1 20 -1 1]);
title('Sine Wave + Noise Autocorrelation'); xlabel('Lag');
```

As the generated figure shows, the autocorrelation estimate demonstrates the four sample periodic sine wave with excursions outside the 95% white Gaussian noise confidence limits every two samples.

## More About

### Autocorrelation

Autocorrelation is the correlation of a signal with itself at different points in time.

For a deterministic discrete-time sequence,  $x(n)$ , the autocorrelation is computed using the following relationship:

$$r_x(h) = \sum_{n=0}^{N-h-1} x^*(n)x(n+h) \quad h = 0, 1, \dots, N-1$$

where  $h$  is the lag and  $*$  denotes the complex conjugate. If the input is a length  $N$  realization of a WSS stationary random process,  $r_x(h)$  is an estimate of the theoretical autocorrelation:

$$\rho_x(h) = E\{x^*(n)x(n+h)\}$$

where  $E\{ \}$  is the expectation operator. The Unity at zero-lag normalization divides each sequence value by the autocorrelation or autocorrelation estimate at zero lag.

$$\frac{\rho_x(h)}{\rho_x(0)} = \frac{E\{x^*(n)x(n+h)\}}{E\{|x(0)|^2\}}$$

The most commonly used estimate of the theoretical autocorrelation of a WSS random process is the biased estimate:

$$\hat{\rho}_x(h) = \frac{1}{N} \sum_{k=0}^{N-h-1} x^*(k)x(k+h)$$

## Algorithms

### Time-Domain Computation

When you set the computation domain to time, the algorithm computes the autocorrelation of the input signal in the time domain. The input signal can be a fixed-point signal in this domain.

The autocorrelation sequence,  $y$ , is computed using this equation:

$$y_{i,j} = \sum_{k=0}^{M-l-1} u_{k,j}^* u_{(k+i),j} \quad 0 \leq i \leq l$$

- $y_{0,j}$  is the zero-lag element in the  $j$ th column of the input.
- $i$  is the index of the lag.
- $j$  is the index of the input data column.
- $*$  denotes the complex conjugate.
- $M$  is the number of elements in each column.



- $l$  is the maximum positive lag for autocorrelation. When you choose to compute the autocorrelation with all nonnegative lags,  $l=M-1$ . Otherwise,  $l$  is the maximum nonnegative integer lag value specified.
- $u$  is an  $M$ -by- $N$  input matrix.

### Frequency-Domain Computation

When you set the computation domain to frequency, the algorithm computes the autocorrelation in the frequency domain.

In this domain, the algorithm computes the autocorrelation sequence by taking the Fourier transform of the input signal, multiplying the Fourier transform with its complex conjugate, and taking the inverse Fourier transform of the product. In this domain, depending on the input length, the algorithm can require fewer computations.

## Version History

### Introduced in R2012a

#### **R2021b: dsp.Autocorrelator System object will be removed**

*Warns starting in R2021b*

`dsp.Autocorrelator` System object will be removed in a future release. Use the `xcorr` function instead.

#### **R2023a: dsp.Autocorrelator System object has been removed**

*Errors starting in R2023a*

The `dsp.Autocorrelator` System object has been removed. Use the `xcorr` function instead.

### Update Code

This table shows how to update existing code to use the `xcorr` function.

| Discouraged Usage   | Recommended Replacement   |
|---|---|
| <p>This function computes correlation over nonnegative lags.</p> <p><b>No maximum lag</b></p> <p>When you do not specify maximum lag, the object computes autocorrelation using nonnegative lags <math>[0 (M - 1)]</math>, where <math>M</math> is the length of the first dimension of the input.</p> <pre>acObj = dsp.Autocorrelator; % x is a column vector x = (1:100)'; yObj = acObj(x); figure; stem(yObj)</pre> <p><b>Specify maximum lag</b></p> <p>When you specify the maximum lag <math>MaxLag</math>, the object computes autocorrelation using lags in the range <math>[0 MaxLag]</math>.</p> <pre>MaxLag = 20; acObjLag = dsp.Autocorrelator(MaximumLagSource='Property', ... MaximumLag=MaxLag); % x is a column vector x = (1:100)'; yObjLag = acObjLag(x); figure; stem(yObjLag)</pre> | <p>This function computes correlation over positive and negative lags.</p> <p><b>No maximum lag</b></p> <p>When you do not specify maximum lag, the function computes autocorrelation using lags in the range <math>[-(M - 1) (M - 1)]</math>, where <math>M</math> is the length of the first dimension of the input.</p> <pre>[yfn,lags] = xcorr(x); figure; stem(lags,yfn)</pre> <p><b>Specify maximum lag</b></p> <p>When you specify the maximum lag <math>MaxLag</math>, the function computes autocorrelation using lags in the range <math>[-MaxLag MaxLag]</math>.</p> <pre>[yfnLag,lags] = xcorr(x,MaxLag); figure; stem(lags,yfnLag)</pre> |

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

## See Also

### Functions

xcorr

# dsp.BinaryFileReader

**Package:** dsp

Read data from binary file

## Description

The `dsp.BinaryFileReader` System object reads multichannel signal data from a binary file. If the header is not empty, then the header precedes the signal data. The System object specifies the prototype of the header, and the type, size, and complexity of the data. The first time you read the file, the reader reads the header, followed by the data. On subsequent calls, the reader reads the remaining data. Once the end of file is reached, the reader returns zeros of the specified data type, size, and complexity. The reader can read signal data from a binary file that is not created by the `dsp.BinaryFileWriter` System object.

The object accepts floating-point data or integer data. To read character data and fixed-point data, see the “Write and Read Character Data” on page 4-144 and “Write and Read Fixed-Point Data” on page 4-143 examples. The input data can be real or complex. When the data is complex, the object reads the data as interleaved real and imaginary components. For an example, see “Read Complex Data” on page 4-142. The reader assumes the default endianness of the host machine. To change the endianness, you can use the `swapbytes` function. For an example, see “Change Endianness of Data” on page 4-145.

To read data from a binary file:

- 1 Create the `dsp.BinaryFileReader` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
reader = dsp.BinaryFileReader
reader = dsp.BinaryFileReader(fname)
reader = dsp.BinaryFileReader(fname,Name,Value)
```

### Description

`reader = dsp.BinaryFileReader` creates a binary file reader object, `reader`, using the default properties.

`reader = dsp.BinaryFileReader(fname)` sets the `Filename` property to `fname`.

`reader = dsp.BinaryFileReader(fname,Name,Value)` with `Filename` set to `fname`, and each specified property set to the specified value. Unspecified properties have default values.

```
Example: reader =  
dsp.BinaryFileReader('myFilename.bin','SamplesPerFrame',1000,'NumChannels',2)  
;
```

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see *System Design in MATLAB Using System Objects*.

### Filename — Name of file

'Untitled.bin' (default) | character vector | string scalar

Name of the file from which the object reads the data, specified as a character vector. If the file is not on the MATLAB path, then specify the full path for the file.

### HeaderStructure — Size of header

struct('Field1',[]) (default) | structure

The structure specifies the prototype of the file header, that is, the size of the header and the data type of the field values. The structure can have an arbitrary number of fields. Each field of the structure must be a real matrix of a built-in type. For example, if `HeaderStructure` is set to `struct('field1',1:10,'field2',single(1))`, the object assumes that the header is formed by 10 real double-precision values followed by 1 single-precision value. If the file contains no header, you can set this property to an empty structure, `struct([])`. To retrieve the file header, call the `readHeader` method on the reader object.

### SamplesPerFrame — Number of samples per output frame

1024 (default) | positive integer

Number of samples per output frame, specified as a positive integer. `SamplesPerFrame` specifies the number of rows of the output matrix that the object returns. The size of the data is `SamplesPerFrame-by-NumChannels`. Once the end of file is reached, the reader returns zeros of the specified data type, size, and complexity.

### NumChannels — Number of channels

1 (default) | positive integer

Number of channels, specified as a positive integer. `NumChannels` specifies the number of columns of the output matrix that the object returns. This property defines the number of consecutive interleaved data samples stored in the file for each time instant. The size of the data is `SamplesPerFrame-by-NumChannels`. Once the end of file is reached, if the output matrix is not full, the object fills the matrix with zeros to make it a full-sized matrix.

### DataType — Type of data in file

'double' (default) | 'single' | 'int8' | 'int16' | 'int32' | 'int64' | 'uint8' | 'uint16' | 'uint32' | 'uint64'

Type of data in file, specified as a character vector. This property defines the data type of the matrix returned by the object algorithm.

**IsDataComplex — Specify data complexity**`false` (default) | `true`

Option to specify data complexity, specified as `false` or `true`. When this property is set to `true`, the reader treats the data as complex. The object reads the data as interleaved real and imaginary components. Consider a reader object configured to read the data as a 2-by-2 matrix. The object reads `[1 5 2 6 3 7 4 8]` as `[1 2; 3 4]+1j*[5 6; 7 8]`. If this property is set to `false`, the same object reads the data as `[1 5; 2 6]`.

**Usage****Syntax**`data = reader()`**Description**

`data = reader()` reads data from the binary file in a row-major format. The data type, size, and complexity of the data are determined by the properties of the reader object. Once the end of file is reached, the output contains zeros of the specified data type, size, and complexity.

**Output Arguments****data — Data output**`vector` | `matrix`

Data read by the binary file reader, returned as a vector or a matrix. The size of the data is given by `SamplesPerFrame-by-NumChannels`, where `SamplesPerFrame` and `NumChannels` are the properties of the `dsp.BinaryFileReader` object.

**Object Functions**

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

`release(obj)`**Specific to dsp.BinaryFileReader**`isDone`      End-of-data status`readHeader`    Read file header**Common to All System Objects**`step`      Run System object algorithm`release`    Release resources and allow changes to System object property values and input characteristics`reset`      Reset internal states of System object**Examples**

### Write and Read Binary Files

Create a binary file with a custom header using the `dsp.BinaryFileWriter` System object. Write data to this file. Read the header and data using the `dsp.BinaryFileReader` System object.

#### Write the Data

Specify the file header as a structure with the following fields:

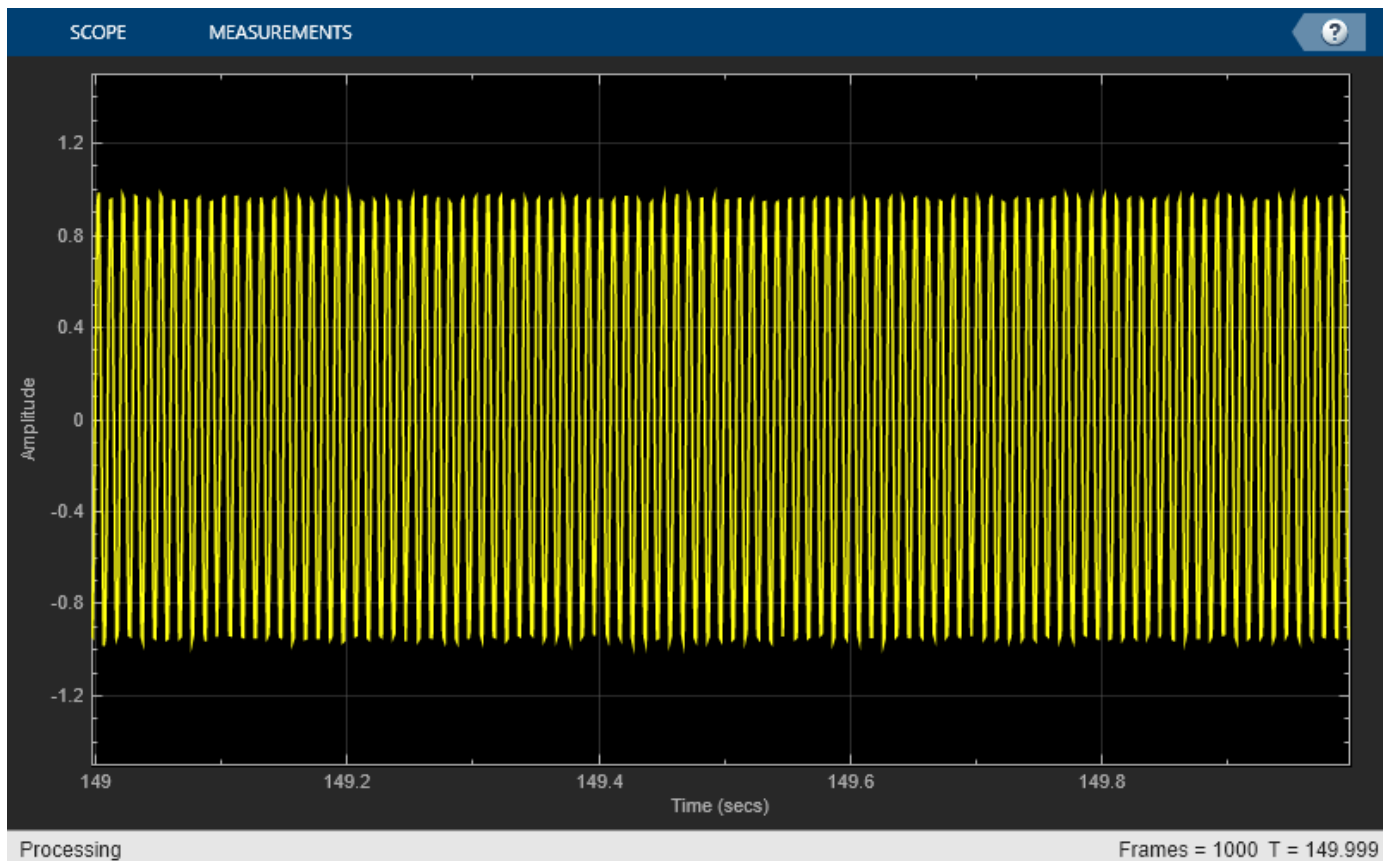
- `DataType` set to `double`.
- `Complexity` set to `false`.
- `FrameSize` (number of rows in the data matrix) set to 150.
- `NumChannels` (number of columns in the data matrix) set to 1.

Create a `dsp.BinaryFileWriter` object using this header. The object writes the header first, followed by the data, to `ex_file.bin`. The data is a noisy sine wave signal. View the data in a time scope.

```
L = 150;
header = struct('DataType','double',...
    'Complexity',false,...
    'FrameSize',L,...
    'NumChannels',1);
writer = dsp.BinaryFileWriter('ex_file.bin',...
    'HeaderStructure',header);

sine = dsp.SineWave('SamplesPerFrame',L);
scopewriter = timescope('YLimits',[-1.5 1.5],...
    'SampleRate',sine.SampleRate,...
    'TimeSpanSource','Property',...
    'TimeSpan',1);

for i = 1:1000
    data = sine() + 0.01*randn(L,1);
    writer(data);
    scopewriter(data)
end
```



Release the writer so that the reader can access the data from this file.

```
release(writer);
```

### Read the Data

Read the data from the binary file, `ex_file.bin`, using the `dsp.BinaryFileReader` object. The file contains the header data followed by the actual data. The object reads the binary data until the end of file is reached. Specify the header to the reader using the `HeaderStructure` property of the reader object.

If the exact header is not known on the reader side, you must at least specify the prototype of the header. That is, the number of fields, and the data type, size, and complexity of each field in the prototype must match with the header data written to the binary file. When the `readHeader` function reads the data from the binary file, the function extracts the header information based on how the fields are specified in the header prototype. For example, a header field set to `'double'` on the writer side can be specified as any string of 6 characters on the reader side. The `readHeader` function reads this field as a string of 6 characters from the binary file, which matches with `'double'`.

```
headerPrototype = struct('DataType','datatype',...
    'Complexity',false,...
    'FrameSize',1,...
    'NumChannels',10);
reader = dsp.BinaryFileReader(...
    'ex_file.bin',...
```

```
'HeaderStructure',headerPrototype);  
headerReader = readHeader(reader)
```

```
headerReader =  
  
struct with fields:  
  
    DataType: 'double'  
    Complexity: 0  
    FrameSize: 150  
    NumChannels: 1
```

The header data extracted by the `readHeader` function is assigned to the corresponding properties of the reader object.

```
reader.IsDataComplex = headerReader.Complexity;  
reader.DataType = headerReader.DataType;  
reader.NumChannels = headerReader.NumChannels;  
reader.SamplesPerFrame = headerReader.FrameSize;
```

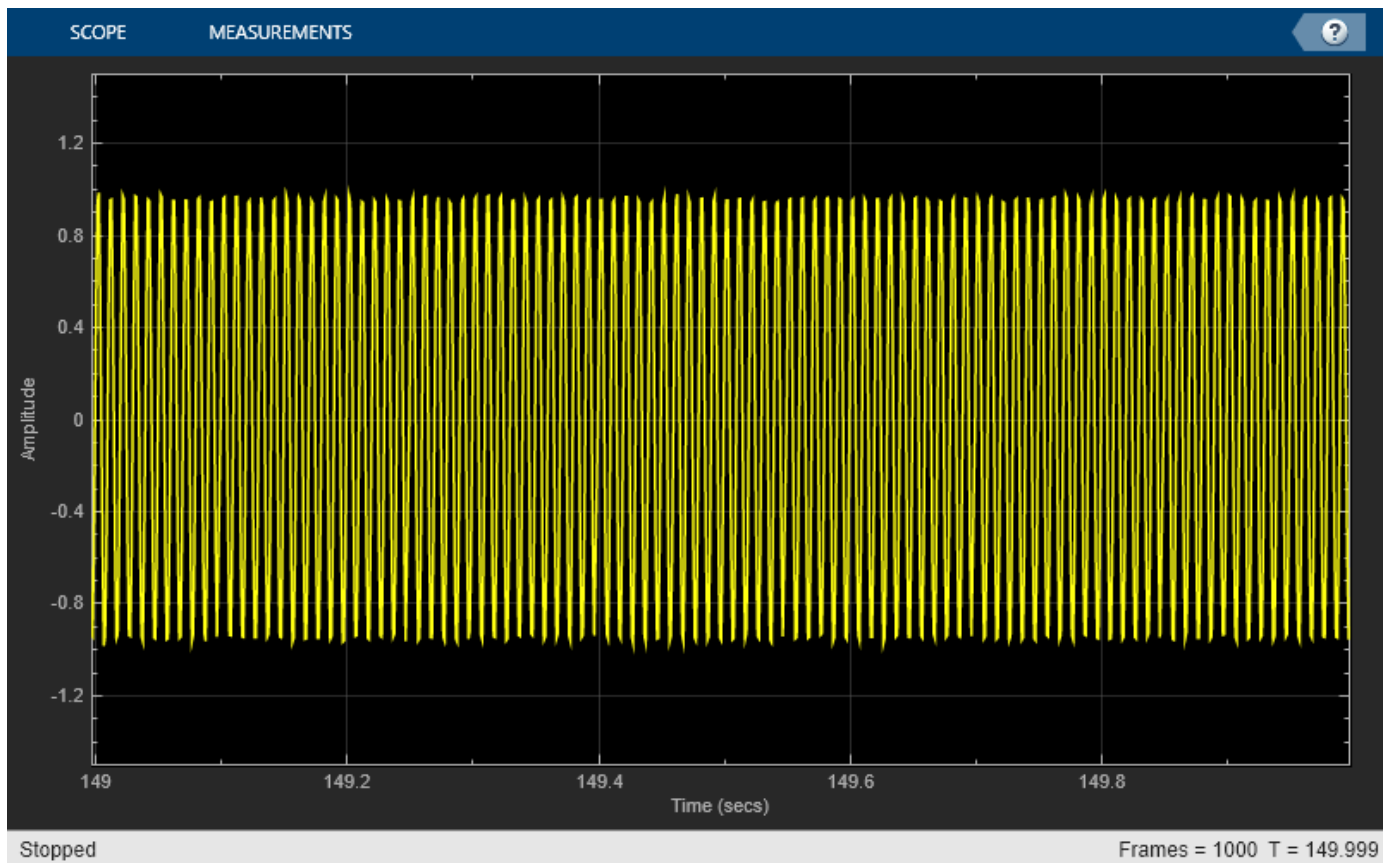
Initialize a scope on the reader side to view the extracted binary file data.

```
scopereader = timescope('YLimits',[-1.5 1.5],...  
    'SampleRate',sine.SampleRate,...  
    'TimeSpanSource','Property',...  
    'TimeSpan',1);
```

The data is read into a single channel (column) containing multiple frames, where each frame has 150 samples. View the data in a time scope.

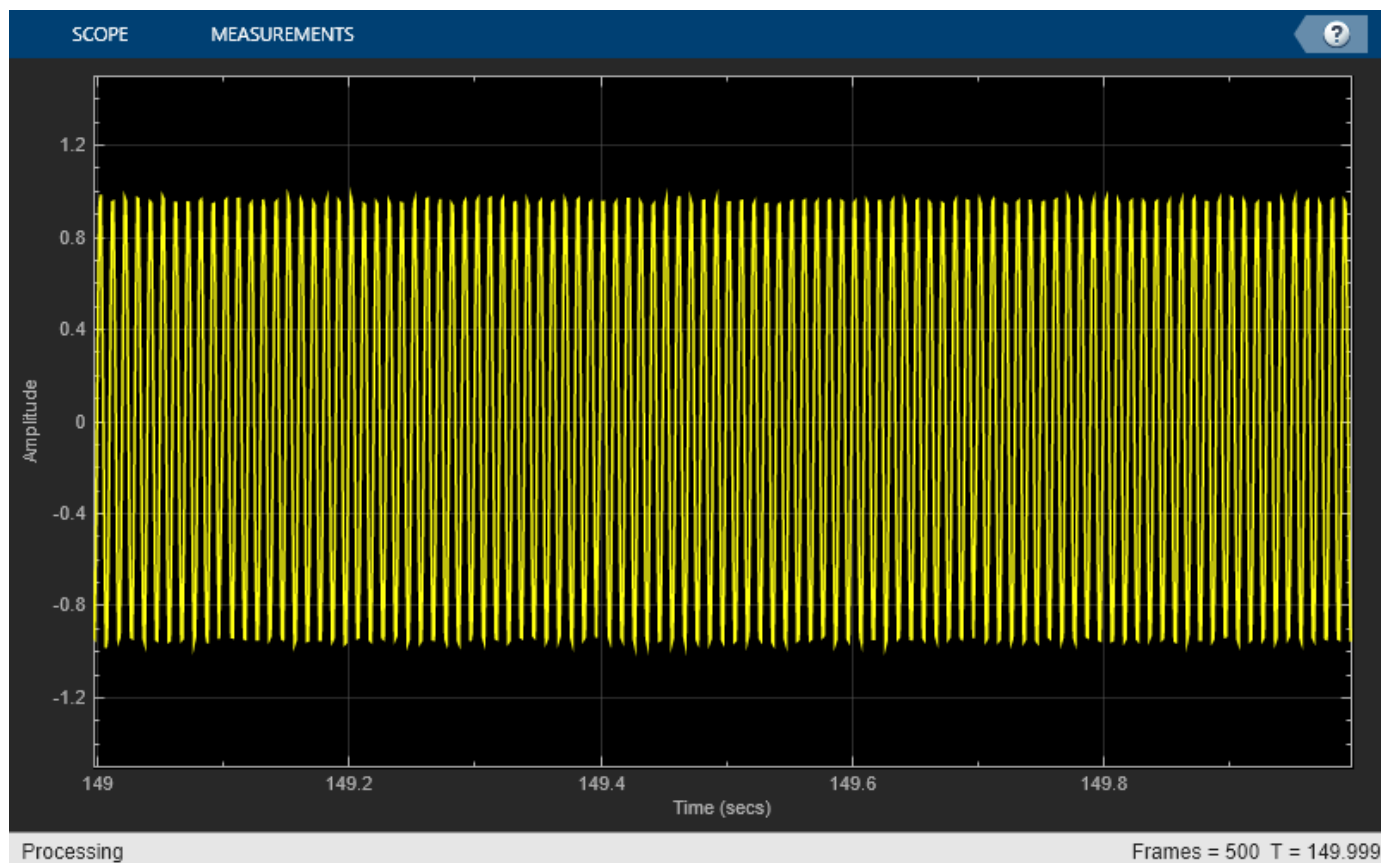
```
while ~isDone(reader)  
    out = reader();  
    scopereader(out)  
end  
release(reader);  
release(scopereader);
```





Set the reader to read data in frames of size 300. Verify that the data read matches the data written to the file.

```
reader.SamplesPerFrame = 300;  
while ~isDone(reader)  
    out = reader();  
    scopereader(out)  
end  
release(reader);
```



Even when the reader reads data with a different frame size, the output in both time scopes matches exactly.

### Write and Read Matrix Data

Use a `dsp.BinaryFileReader` System object™ to read data from a binary file in a row-major format.

#### Write the Data

Write the matrix `A` to the binary file `Matdata.bin` using a `dsp.BinaryFileWriter` object. The object writes the specified header followed by the data.

The header has the following format:

- `DataType` set to `double`.
- `Complexity` set to `false`.
- `FrameSize` (number of rows in the data matrix) set to 3.
- `NumChannels` (number of columns in the data matrix) set to 4.

```
A = [1 2 3 8; 4 5 6 10; 7 8 9 11];
header = struct('DataType','double',...
               'Complexity',false,...
```

```

        'FrameSize',3,...
        'NumChannels',4);
writer = dsp.BinaryFileWriter('Matdata.bin',...
    'HeaderStructure',header);
writer(A);

```

Release the writer so that the reader can access the data.

```
release(writer);
```

### Read the Data

Specify the header using the `HeaderStructure` property of the reader object. If the exact header is not known, you must at least specify the prototype of the header. That is, the number of fields, and the data type, size, and complexity of each field in the prototype must match with the header data written to the binary file. The `dsp.BinaryFileReader` object reads the binary file `Matdata.bin` until the end of file is reached. Configure the System object to read the data into 4 channels, with each channel containing 5 samples. Each loop of the iteration reads a channel (or frame) of data.

```

headerPrototype = struct('DataType','double',...
    'Complexity',false,...
    'FrameSize',5,...
    'NumChannels',4);
reader = dsp.BinaryFileReader('Matdata.bin',...
    'HeaderStructure',headerPrototype,...
    'NumChannels',4,...
    'SamplesPerFrame',5);
while ~isDone(reader)
    out = reader();
    display(out)
end

```

```
out = 5×4
```

|   |   |   |    |
|---|---|---|----|
| 1 | 2 | 3 | 8  |
| 4 | 5 | 6 | 10 |
| 7 | 8 | 9 | 11 |
| 0 | 0 | 0 | 0  |
| 0 | 0 | 0 | 0  |

Each frame of `out` contains frames of the matrix `A`, followed by zeros to complete the frame. The original matrix `A` contains 4 channels with 3 samples in each channel. The reader is configured to read data into 4 channels, with each channel containing 5 samples. Because there are not enough samples to complete the frame, the reader object appends zeros at the end of each frame.

### Read Header Data

Read the header data from a binary file using the `readHeader` function.

Write a header, followed by the data to a binary file named `myfile.dat`. The header is a 1-by-4 matrix of double precision values, followed by a 5-by-1 vector of single-precision values. The data is a sequence of 1000 double-precision values.

```

fid = fopen('myfile.dat','w');
fwrite(fid,[1 2 3 4],'double');

```

```
fwrite(fid, single((1:5).'), 'single');  
fwrite(fid, (1:1000).', 'double');  
fclose(fid);
```

Read the header using a `dsp.BinaryFileReader` object. Specify the expected header structure. This structure specifies only the format of the expected binary file header and does not contain the exact values.

```
reader = dsp.BinaryFileReader('myfile.dat');  
s = struct('A', zeros(1,4), 'B', ones(5,1, 'single'));  
reader.HeaderStructure = s;
```

Read the header using the `readHeader` function.

```
H = readHeader(reader);  
fprintf('H.A: ')  
  
H.A:  
  
fprintf('%d ', H.A);  
  
1 2 3 4  
  
fprintf('\nH.A datatype: %s\n', class(H.A))  
  
H.A datatype: double  
  
fprintf('H.B: ')  
  
H.B:  
  
fprintf('%d ', H.B);  
  
1 2 3 4 5  
  
fprintf('\nH.B datatype: %s\n', class(H.B))  
  
H.B datatype: single
```

### Read Complex Data

Read complex data from a binary file using the `dsp.BinaryFileReader` object.

Write a sequence of numbers to a binary file named `myfile.dat`. There is no header. The data is a 2-by-4 matrix of double-precision values. `fwrite` writes the data in a column-major format. That is, the 2-by-4 matrix `[1 2 3 4; 9 10 11 12]` is written as `[1 9 2 10 3 11 4 12]` in the binary file.

```
fid = fopen('myfile.dat', 'w');  
fwrite(fid, [1 2 3 4; 9 10 11 12], 'double');  
fclose(fid);
```

Specify the data to be complex using the `IsDataComplex` property. The object reads the data as interleaved real and imaginary components. The `SamplesPerFrame` and `NumChannels` properties specify the number of rows and columns of the output data. The header structure is specified as empty.

```
reader = dsp.BinaryFileReader('myfile.dat', 'SamplesPerFrame', 2, ...  
    'NumChannels', 2, 'IsDataComplex', true);
```

```

s = struct([]);
reader.HeaderStructure = s;
data = reader();
display(data);

data = 2x2 complex

    1.0000 + 9.0000i    2.0000 +10.0000i
    3.0000 +11.0000i    4.0000 +12.0000i

release(reader);

```

Alternatively, if you do not specify the data as complex, the reader reads the data as a SamplesPerFrame-by- NumChannel matrix of real values.

```

reader.IsDataComplex = false;
data = reader();
display(data);

data = 2x2

     1     9
     2    10

release(reader);

```

### Write and Read Fixed-Point Data

The `dsp.BinaryFileWriter` and `dsp.BinaryFileReader` System objects do not support writing and reading fixed-point data. As a workaround, you can write the stored integer portion of the `fi` data, read the data, and use this value to reconstruct the `fi` data.

#### Write the Fixed-Point Data

Create an `fi` object to represent 100 signed random numbers with a word length of 14 and a fraction length of 12. Write the stored integer portion of the `fi` object to the data file `myFile.dat`. The built-in data type is `int16`, which can be computed using `class(storeIntData)`.

```

data = randn(100,1);
fiDataWriter = fi(data,1,14,12);
storeIntData = storedInteger(fiDataWriter);

writer = dsp.BinaryFileWriter('myFile.dat');
writer(storeIntData);

```

Release the writer so that the reader can access the data.

```
release(writer);
```

#### Read the Fixed-Point Data

Specify the reader to read the stored integer data as `int16` data with 100 samples per data frame. The real-world value of the fixed-point number can be represented using

$2^{(-fractionLength)(storedInteger)}$ . If you know the signedness, word length, and fraction length of the fixed-point data, you can reconstruct the `fi` data using `fi(realValue, signedness, wordLength, fractionLength)`. In this example, the data is signed with a word length of 14 and a fraction length of 12.

```
reader = dsp.BinaryFileReader('Filename','myFile.dat',...
    'SamplesPerFrame',100,...
    'DataType','int16');
data = reader();
fractionLength = 12;
wordLength = 14;
realValue = 2^(-fractionLength)*double(data);

fiDataReader = fi(realValue,1,...
    wordLength,fractionLength);
```

Verify that the writer data is the same as the reader data.

```
isequal(fiDataWriter,fiDataReader)

ans = logical
     1
```

### Write and Read Character Data

The `dsp.BinaryFileWriter` and `dsp.BinaryFileReader` System objects do not support writing and reading characters. As a workaround, cast character data to one of the built-in data types and write the integer data. After the reader reads the data, convert the data to a character using the `char` function.

#### Write the Character Data

Cast a character into `uint8` using the `cast` function. Write the cast data to the data file `myFile.dat`.

```
data = 'binary_file';
castData = cast(data,'uint8');
writer = dsp.BinaryFileWriter('myFile.dat');
writer(castData);
```

Release the writer so that the reader can access the data.

```
release(writer);
```

#### Read the uint8 Data

Configure the reader to read the cast data as `uint8` data.

```
reader = dsp.BinaryFileReader('myFile.dat',...
    'DataType','uint8',...
    'SamplesPerFrame',11);
readerData = reader();
charData = char(readerData);
```

Verify that the writer data is the same as the reader data. By default, the reader returns the data in a column-major format.

```
strcmp(data,charData.')
```

```
ans = logical
     1
```

### Change Endianness of Data

By default, the dsp.BinaryFileReader System object™ uses the endianness of the host machine. To change the endianness, such as when the host machine that writes the data does not have the same endianness as the host machine that reads the data, use the `swapbytes` function.

Write a numeric array into `myfile.dat` in big endian format. Read the data using the dsp.BinaryFileReader object. The reader object reads the data in little endian format.

```
fid = fopen('myfile.dat','w','b');
fwrite(fid,[1 2 3 4 5 6 7 8],'double');
fclose(fid);
reader = dsp.BinaryFileReader('myfile.dat','SamplesPerFrame',8);
x = reader();
display(x);
```

```
x = 8×1
10-318 ×
```

```
0.3039
0.0003
0.0104
0.0206
0.0256
0.0307
0.0357
0.0408
```

`x` does not match the original data. Change the endianness of `x` using the `swapbytes` function.

```
y = swapbytes(x);
display(y);
```

```
y = 8×1
```

```
1
2
3
4
5
6
7
8
```

`y` matches the original data.

## **Version History**

**Introduced in R2016b**

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

## **See Also**

### **Objects**

`dsp.BinaryFileWriter`

### **Blocks**

Binary File Reader | Binary File Writer

### **Topics**

“Generate C Code from MATLAB Code”



# dsp.BinaryFileWriter

**Package:** dsp

Write data to binary files

## Description

The `dsp.BinaryFileWriter` System object writes multichannel signal data to a binary file. If the header is not empty, then the header precedes the signal data. The object specifies the file name and the structure of the header. The first time you write to the file, the object writes the header, followed by the data. On subsequent calls, the object writes the remaining data. If the header is empty, then no header is written.

The object can write floating-point data and integer data. To write character data and fixed-point data, see “Write and Read Character Data” on page 4-156 and “Write and Read Fixed-Point Data” on page 4-155. The input data can be real or complex. When the data is complex, the object writes the data as interleaved real and imaginary components. For an example, see “Write and Read Fixed-Point Data” on page 4-155. By default, the writer uses the endianness of the host machine. To change the endianness, you can use the `swapbytes` function. For an example, see “Change Endianness of Data Before Writing” on page 4-156 .

To write data to a binary file:

- 1 Create the `dsp.BinaryFileWriter` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
writer = dsp.BinaryFileWriter
writer = dsp.BinaryFileWriter(fname)
writer = dsp.BinaryFileWriter(fname,Name,Value)
```

### Description

`writer = dsp.BinaryFileWriter` creates a binary file writer object, `writer`, using the default properties.

`writer = dsp.BinaryFileWriter(fname)` sets the `Filename` property to `fname`.

`writer = dsp.BinaryFileWriter(fname,Name,Value)` with `Filename` set to `fname` and each property `Name` set to the specified `Value`. Unspecified properties have default values.

Example: `writer = dsp.BinaryFileWriter('myFilename.bin','HeaderStructure',struct('field1',1:10,'field2',single(1)));`

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

### Filename — Name of file

'untitled.bin' (default) | character vector | string scalar

Name of the file to which the object writes the data, specified as a character vector or a string scalar. You must specify the full path for the file.

### HeaderStructure — Header to write at beginning of file

struct([]) (default) | structure

Header to write at the beginning of the file, specified as a structure. The structure can have an arbitrary number of fields. Each field of the structure must be a real matrix of a built-in type. For example, if `HeaderStructure` is set to `struct('field1',1:10,'field2',single(1))`, the object writes a header formed by 10 double-precision values, `(1:10)`, followed by one single precision value, `single(1)`. If you do not specify a header, the object sets this property to an empty structure, `struct([])`.

## Usage

### Syntax

```
writer(data)
```

### Description

`writer(data)` writes data to the binary file in a row-major format. Each call to the algorithm writes the elements of `data` at the end of the file. At the first call to the algorithm, the object writes the header first, followed by the data. If the header is empty, then no header is written.

The input data can be real or complex. For complex data, real and imaginary parts are interleaved. For example, if the data equals `[1 2; 3 4]+1j*[5 6; 7 8]`, then the object writes the elements as `1 5 2 6 3 7 4 8`.

### Input Arguments

#### data — Data to be written to file

vector | matrix

Data to be written to the binary file in a row-major format, specified as a vector or a matrix. The object writes the data in row-major format. For example, if the input array is `[1 2 4 5; 8 7 9 2]`, the object writes the data as `[1 2 4 5 8 7 9 2]`.

The input data can be real or complex. For complex data, real and imaginary parts are interleaved. For example, if the data equals  $[1 \ 2; \ 3 \ 4]+1j*[5 \ 6; \ 7 \ 8]$ , then the object writes the elements as  $[1 \ 5 \ 2 \ 6 \ 3 \ 7 \ 4 \ 8]$ .

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Common to All System Objects

|                      |  |
|----------------------|--|
| <code>step</code>    | Run System object algorithm  |
| <code>release</code> | Release resources and allow changes to System object property values and input characteristics |
| <code>reset</code>   | Reset internal states of System object   |

## Examples

### Write and Read Binary Files

Create a binary file with a custom header using the `dsp.BinaryFileWriter` System object. Write data to this file. Read the header and data using the `dsp.BinaryFileReader` System object.

#### Write the Data

Specify the file header as a structure with the following fields:

- `DataType` set to `double`.
- `Complexity` set to `false`.
- `FrameSize` (number of rows in the data matrix) set to 150.
- `NumChannels` (number of columns in the data matrix) set to 1.

Create a `dsp.BinaryFileWriter` object using this header. The object writes the header first, followed by the data, to `ex_file.bin`. The data is a noisy sine wave signal. View the data in a time scope.

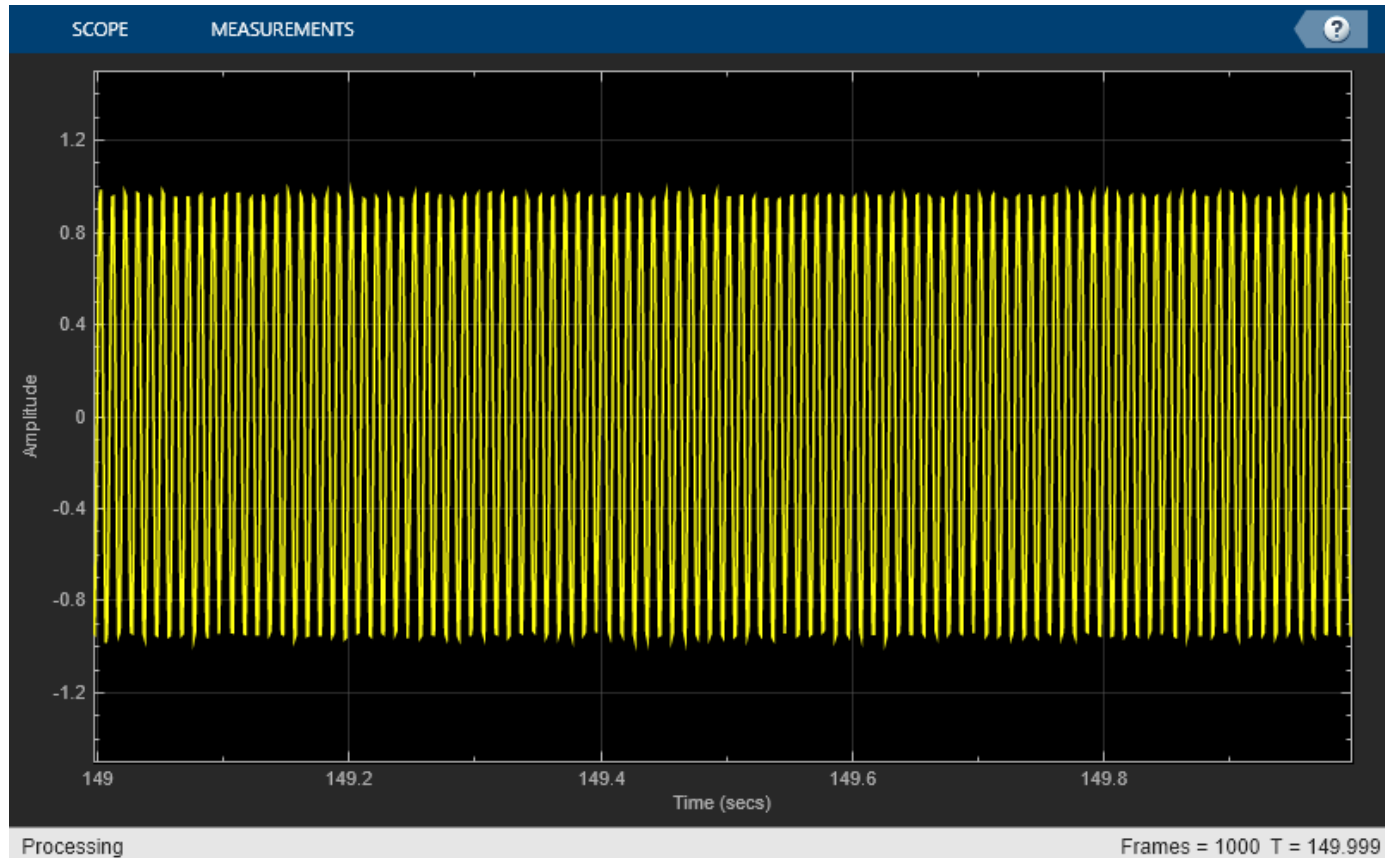
```
L = 150;
header = struct('DataType','double',...
    'Complexity',false,...
    'FrameSize',L,...
    'NumChannels',1);
writer = dsp.BinaryFileWriter('ex_file.bin',...
    'HeaderStructure',header);

sine = dsp.SinWave('SamplesPerFrame',L);
scopewriter = timescope('YLimits',[-1.5 1.5],...
    'SampleRate',sine.SampleRate,...
    'TimeSpanSource','Property',...
    'TimeSpan',1);
```

```

for i = 1:1000
    data = sine() + 0.01*randn(L,1);
    writer(data);
    scopewriter(data)
end

```



Release the writer so that the reader can access the data from this file.

```
release(writer);
```

### Read the Data

Read the data from the binary file, `ex_file.bin`, using the `dsp.BinaryFileReader` object. The file contains the header data followed by the actual data. The object reads the binary data until the end of file is reached. Specify the header to the reader using the `HeaderStructure` property of the reader object.

If the exact header is not known on the reader side, you must at least specify the prototype of the header. That is, the number of fields, and the data type, size, and complexity of each field in the prototype must match with the header data written to the binary file. When the `readHeader` function reads the data from the binary file, the function extracts the header information based on how the fields are specified in the header prototype. For example, a header field set to `'double'` on the writer side can be specified as any string of 6 characters on the reader side. The `readHeader` function reads this field as a string of 6 characters from the binary file, which matches with `'double'`.

```
headerPrototype = struct('DataType','datatype',...
    'Complexity',false,...
    'FrameSize',1,...
    'NumChannels',10);
reader = dsp.BinaryFileReader(...
    'ex_file.bin',...
    'HeaderStructure',headerPrototype);
headerReader = readHeader(reader)
```

```
headerReader =
```

```
struct with fields:
```

```
    DataType: 'double'
    Complexity: 0
    FrameSize: 150
    NumChannels: 1
```

The header data extracted by the `readHeader` function is assigned to the corresponding properties of the reader object.

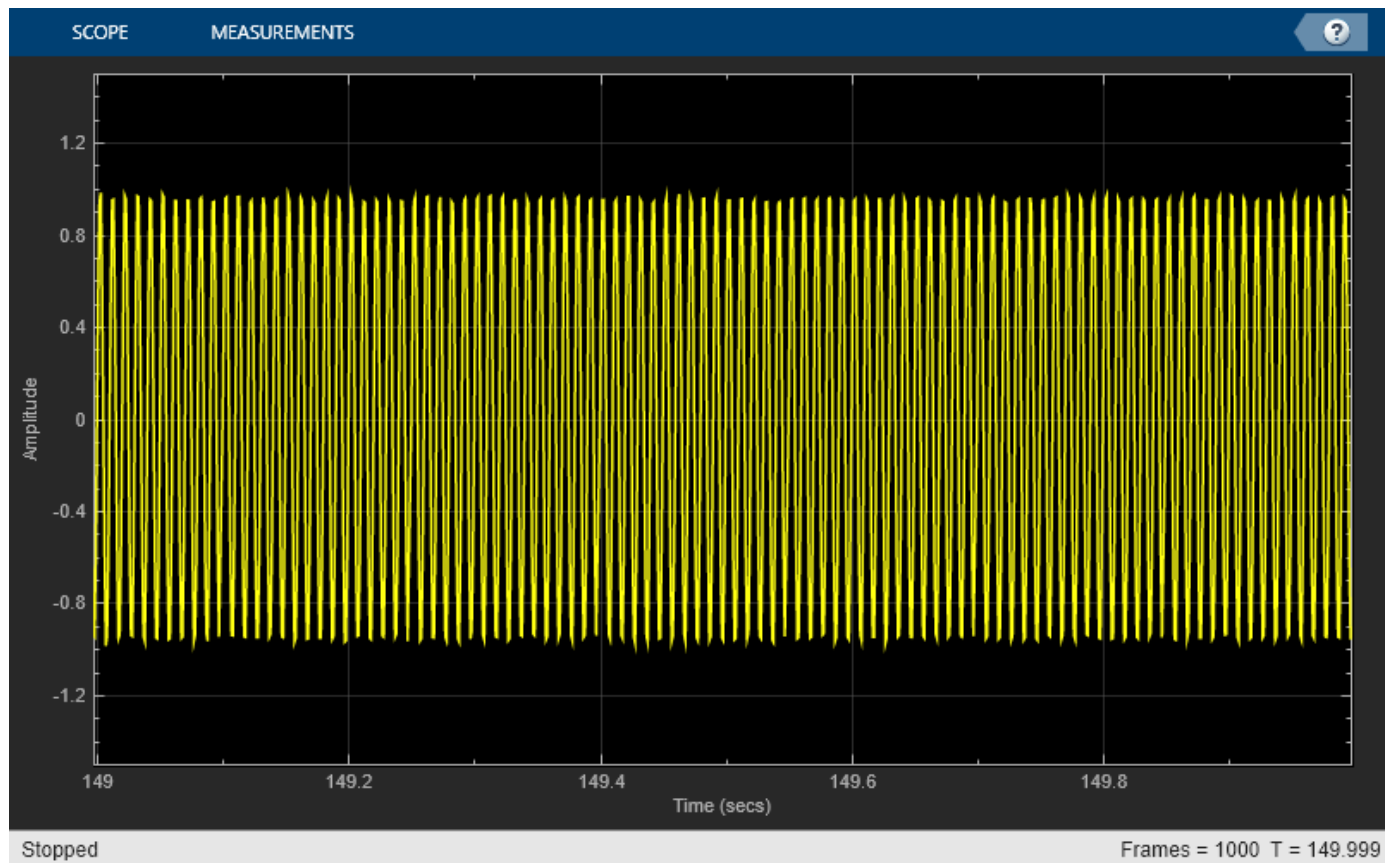
```
reader.IsDataComplex = headerReader.Complexity;
reader.DataType = headerReader.DataType;
reader.NumChannels = headerReader.NumChannels;
reader.SamplesPerFrame = headerReader.FrameSize;
```

Initialize a scope on the reader side to view the extracted binary file data.

```
scopereader = timescope('YLimits',[-1.5 1.5],...
    'SampleRate',sine.SampleRate,...
    'TimeSpanSource','Property',...
    'TimeSpan',1);
```

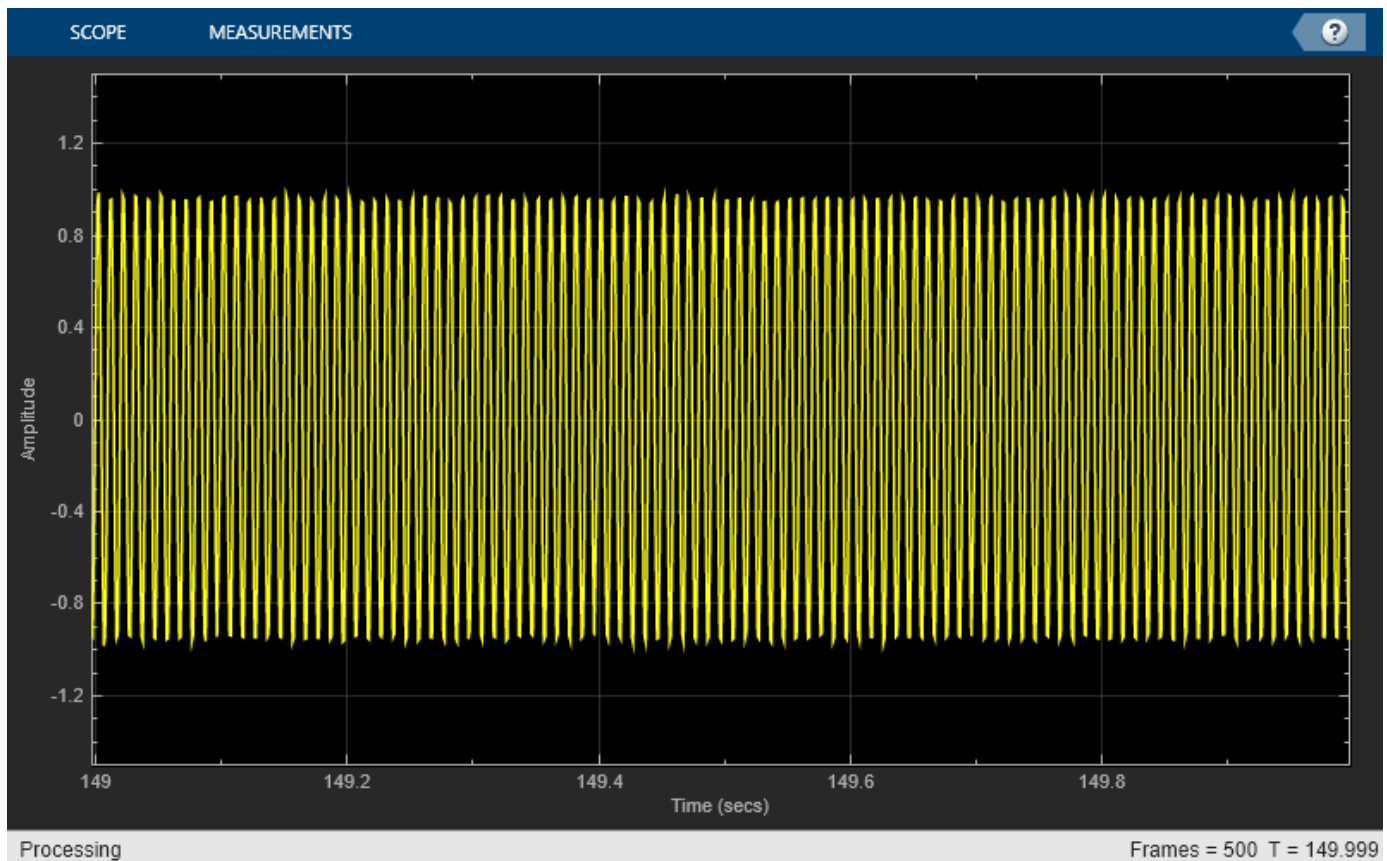
The data is read into a single channel (column) containing multiple frames, where each frame has 150 samples. View the data in a time scope.

```
while ~isDone(reader)
    out = reader();
    scopereader(out)
end
release(reader);
release(scopereader);
```



Set the reader to read data in frames of size 300. Verify that the data read matches the data written to the file.

```
reader.SamplesPerFrame = 300;  
while ~isDone(reader)  
    out = reader();  
    scopereader(out)  
end  
release(reader);
```



Even when the reader reads data with a different frame size, the output in both time scopes matches exactly.

### Write and Read Matrix Data

Use a `dsp.BinaryFileReader` System object™ to read data from a binary file in a row-major format.

#### Write the Data

Write the matrix `A` to the binary file `Matdata.bin` using a `dsp.BinaryFileWriter` object. The object writes the specified header followed by the data.

The header has the following format:

- `DataType` set to `double`.
- `Complexity` set to `false`.
- `FrameSize` (number of rows in the data matrix) set to 3.
- `NumChannels` (number of columns in the data matrix) set to 4.

```
A = [1 2 3 8; 4 5 6 10; 7 8 9 11];
header = struct('DataType','double',...
    'Complexity',false,...
```

```
        'FrameSize',3,...
        'NumChannels',4);
writer = dsp.BinaryFileWriter('Matdata.bin',...
    'HeaderStructure',header);
writer(A);
```

Release the writer so that the reader can access the data.

```
release(writer);
```

### Read the Data

Specify the header using the `HeaderStructure` property of the reader object. If the exact header is not known, you must at least specify the prototype of the header. That is, the number of fields, and the data type, size, and complexity of each field in the prototype must match with the header data written to the binary file. The `dsp.BinaryFileReader` object reads the binary file `Matdata.bin` until the end of file is reached. Configure the System object to read the data into 4 channels, with each channel containing 5 samples. Each loop of the iteration reads a channel (or frame) of data.

```
headerPrototype = struct('DataType','double',...
    'Complexity',false,...
    'FrameSize',5,...
    'NumChannels',4);
reader = dsp.BinaryFileReader('Matdata.bin',...
    'HeaderStructure',headerPrototype,...
    'NumChannels',4,...
    'SamplesPerFrame',5);
while ~isDone(reader)
    out = reader();
    display(out)
end
```

```
out = 5×4
```

```
     1     2     3     8
     4     5     6    10
     7     8     9    11
     0     0     0     0
     0     0     0     0
```

Each frame of `out` contains frames of the matrix `A`, followed by zeros to complete the frame. The original matrix `A` contains 4 channels with 3 samples in each channel. The reader is configured to read data into 4 channels, with each channel containing 5 samples. Because there are not enough samples to complete the frame, the reader object appends zeros at the end of each frame.

### Write Complex Data

Create a `dsp.BinaryFileWriter` object which writes to a file named `myfile.dat`. There is no header. The data is complex.

```
writer = dsp.BinaryFileWriter('myfile.dat');
data = [1 2 3 4]+1i*[5 6 7 8];
writer(data);
release(writer);
```



Read the data using the `dsp.BinaryFileReader` System object™. To view data in the format it is written to the file, set the `IsDataComplex` property to `false`. The reader object reads the data as a sequence of numbers in a row major format. Set `SamplesPerFrame` to 1 and `NumChannels` to 8.

```
reader = dsp.BinaryFileReader('myfile.dat','SamplesPerFrame',1,...
    'NumChannels',8);
s = struct([]);
reader.HeaderStructure = s;
dataRead = reader();
```

You can see that the real and imaginary components of the original data are sample interleaved.

```
display(dataRead);
```

```
dataRead = 1×8
```

```
    1    5    2    6    3    7    4    8
```

### Write and Read Fixed-Point Data

The `dsp.BinaryFileWriter` and `dsp.BinaryFileReader` System objects do not support writing and reading fixed-point data. As a workaround, you can write the stored integer portion of the `fi` data, read the data, and use this value to reconstruct the `fi` data.

#### Write the Fixed-Point Data

Create an `fi` object to represent 100 signed random numbers with a word length of 14 and a fraction length of 12. Write the stored integer portion of the `fi` object to the data file `myFile.dat`. The built-in data type is `int16`, which can be computed using `class(storeIntData)`.

```
data = randn(100,1);
fiDataWriter = fi(data,1,14,12);
storeIntData = storedInteger(fiDataWriter);
```

```
writer = dsp.BinaryFileWriter('myFile.dat');
writer(storeIntData);
```

Release the writer so that the reader can access the data.

```
release(writer);
```

#### Read the Fixed-Point Data

Specify the reader to read the stored integer data as `int16` data with 100 samples per data frame. The real-world value of the fixed-point number can be represented using  $2^{(-fractionLength)(storedInteger)}$ . If you know the signedness, word length, and fraction length of the fixed-point data, you can reconstruct the `fi` data using `fi(realValue,signedness,wordLength,fractionLength)`. In this example, the data is signed with a word length of 14 and a fraction length of 12.

```
reader = dsp.BinaryFileReader('Filename','myFile.dat',...
    'SamplesPerFrame',100,...
    'DataType','int16');
```

```
data = reader();
fractionLength = 12;
wordLength = 14;
realValue = 2^(-fractionLength)*double(data);
```

```
fiDataReader = fi(realValue,1,...
    wordLength,fractionLength);
```

Verify that the writer data is the same as the reader data.

```
isequal(fiDataWriter,fiDataReader)
```

```
ans = logical
     1
```

### Write and Read Character Data

The `dsp.BinaryFileWriter` and `dsp.BinaryFileReader` System objects do not support writing and reading characters. As a workaround, cast character data to one of the built-in data types and write the integer data. After the reader reads the data, convert the data to a character using the `char` function.

#### Write the Character Data

Cast a character into `uint8` using the `cast` function. Write the cast data to the data file `myFile.dat`.

```
data = 'binary_file';
castData = cast(data,'uint8');
writer = dsp.BinaryFileWriter('myFile.dat');
writer(castData);
```

Release the writer so that the reader can access the data.

```
release(writer);
```

#### Read the uint8 Data

Configure the reader to read the cast data as `uint8` data.

```
reader = dsp.BinaryFileReader('myFile.dat',...
    'DataType','uint8',...
    'SamplesPerFrame',11);
readerData = reader();
charData = char(readerData);
```

Verify that the writer data is the same as the reader data. By default, the reader returns the data in a column-major format.

```
strcmp(data,charData.')
```

```
ans = logical
     1
```

## Change Endianness of Data Before Writing

By default, the `dsp.BinaryFileWriter` System object™ uses the endianness of the host machine. To change the endianness, use the `swapbytes` function.

Write a numeric array into `myfile.dat` using the `dsp.BinaryFileWriter` object. Before writing the data, change the endianness of the data using the `swapbytes` function.

```
data = [1 2 3 4 2 2];  
swapData = swapbytes(data);  
writer = dsp.BinaryFileWriter('myfile.dat');  
writer(swapData);
```

## Version History

Introduced in R2016b

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

## See Also

### Objects

`dsp.BinaryFileReader`

### Blocks

Binary File Reader | Binary File Writer

### Topics

“Generate C Code from MATLAB Code”

## dsp.BiquadFilter

**Package:** dsp

IIR filter using biquadratic structures

### Description

The `dsp.BiquadFilter` object implements a cascade of biquadratic sections, where the coefficients for each section are supplied by a separate row of an  $N$ -by-6 second-order sections (SOS) matrix. Each row of the SOS matrix contains the numerator and denominator coefficients of the corresponding section of the filter. The resulting filter can be applied to a vector or matrix input, where each column represents a channel of data that is processed independently.

To implement an IIR filter structure using biquadratic or SOS:

- 1 Create the `dsp.BiquadFilter` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

### Creation

#### Syntax

```
biquad = dsp.BiquadFilter
biquad = dsp.BiquadFilter(sosmatrix,scalevalues)
biquad = dsp.BiquadFilter(Name,Value)
```

#### Description

`biquad = dsp.BiquadFilter` returns a biquadratic IIR (SOS) filter System object, `biquad`, which independently filters each channel (column) of the input over time using the SOS section `[1 0.3 0.4 1 0.1 0.2]` with a direct-form II transposed structure.

`biquad = dsp.BiquadFilter(sosmatrix,scalevalues)` returns a biquadratic filter object, with the `SOSMatrix` property set to `sosmatrix` and the `ScaleValues` property set to `scalevalues`.

`biquad = dsp.BiquadFilter(Name,Value)` returns a biquadratic filter object, `biquad`, with each property set to the specified value.

#### Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

### Structure — Filter structure

'Direct form II transposed' (default) | 'Direct form I' | 'Direct form I transposed' | 'Direct form II'

Specify the filter structure as 'Direct form I', 'Direct form I transposed', 'Direct form II', 'Direct form II transposed'.

### SOSMatrixSource — SOS matrix source

'Property' (default) | 'Input port'

Specify the source of the SOS matrix as 'Property' or 'Input port'.

### SOSMatrix — SOS matrix

[1 0.3 0.4 1 0.1 0.2] (default) | N-by-6 matrix

Specify the second-order section (SOS) matrix as an N-by-6 matrix, where N is the number of sections in the filter. Each row of the SOS matrix contains the numerator and denominator coefficients of the corresponding section of the filter. The system function,  $H(z)$ , of a biquad filter is:

$$H(z) = \frac{\sum_{k=0}^2 b_k z^{-k}}{1 - \sum_{l=1}^2 a_l z^{-l}}$$

The coefficients are ordered in the rows of the SOS matrix as  $(b_0, b_1, b_2, 1, -a_1, -a_2)$ . You can use coefficients of real or complex values. This property applies only when you set the `SOSMatrixSource` property to `Property`. The leading denominator coefficient of the biquad filter,  $a_0$ , equals 1 for each filter section, regardless of the specified value.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

### ScaleValues — Scale values for each biquad section

1 (default) | scalar | vector

Specify the scale values to apply before and after each section of a biquad filter. `ScaleValues` must be either a scalar or a vector of length N+1, where N is the number of sections. If you set this property to a scalar, the scalar value is used as the gain value only before the first filter section. The remaining gain values are set to 1. If you set this property to a vector of N+1 values, each value is used for a separate section of the filter.

### Dependencies

This property applies only when you set the `SOSMatrixSource` property to `Property`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### InitialConditions — Initial conditions for direct form II structures

0 (default) | scalar | vector | matrix

Specify the initial conditions of the filter states when the `Structure` property is one of `Direct form II` | `Direct form II transposed`. The number of states or delay elements (zeros and

poles) in a direct-form II biquad filter equals twice the number of filter sections. You can specify the initial conditions as a scalar, vector, or matrix.

When you specify a scalar value, the biquad filter initializes all delay elements in the filter to that value. When you specify a vector of length equal to the number of delay elements in the filter, each vector element specifies a unique initial condition for the corresponding delay element.

The biquad filter applies the same vector of initial conditions to each channel of the input signal. When you specify a vector of length equal to the product of the number of input channels and the number of delay elements in the filter, each element specifies a unique initial condition for the corresponding delay element in the corresponding channel. When you specify a matrix with the same number of rows as the number of delay elements in the filter, and one column for each channel of the input signal, each element specifies a unique initial condition for the corresponding delay element in the corresponding channel.

#### **Dependencies**

This property applies only when you set the `Structure` property to one of `Direct form II` or `Direct form II transposed`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

#### **NumeratorInitialConditions — Initial conditions on zeros side**

0 (default) | `scalar` | `vector` | `matrix`

Specify the initial conditions of the filter states on the side of the filter structure with the zeros. The number of states or delay elements in the numerator of a direct-form I biquad filter equals twice the number of filter sections. You can specify the initial conditions as a scalar, vector, or matrix. When you specify a scalar, the biquad filter initializes all delay elements on the zeros side in the filter to that value. When you specify a vector of length equal to the number of delay elements on the zeros side in the filter, each vector element specifies a unique initial condition for the corresponding delay element on the zeros side.

The biquad filter applies the same vector of initial conditions to each channel of the input signal. When you specify a vector of length equal to the product of the number of input channels and the number of delay elements on the zeros side in the filter, each element specifies a unique initial condition for the corresponding delay element on the zeros side in the corresponding channel. When you specify a matrix with the same number of rows as the number of delay elements on the zeros side in the filter, and one column for each channel of the input signal, each element specifies a unique initial condition for the corresponding delay element on the zeros side in the corresponding channel.

#### **Dependencies**

This property applies only when you set the `Structure` property to one of `Direct form I` or `Direct form I transposed`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

#### **DenominatorInitialConditions — Initial conditions on poles side**

0 (default) | `scalar` | `vector` | `matrix`

Specify the initial conditions of the filter states on the side of the filter structure with the poles. The number of denominator states, or delay elements, in a direct-form I (noncanonic) biquad filter equals twice the number of filter sections. You can specify the initial conditions as a scalar, vector, or matrix. When you specify a scalar, the biquad filter initializes all delay elements on the poles side of the filter to that value. When you specify a vector of length equal to the number of delay elements on the poles

side in the filter, each vector element specifies a unique initial condition for the corresponding delay element on the poles side.

The object applies the same vector of initial conditions to each channel of the input signal. When you specify a vector of length equal to the product of the number of input channels and the number of delay elements on the poles side in the filter, each element specifies a unique initial condition for the corresponding delay element on the poles side in the corresponding channel. When you specify a matrix with the same number of rows as the number of delay elements on the poles side in the filter, and one column for each channel of the input signal, each element specifies a unique initial condition for the corresponding delay element on the poles side in the corresponding channel.

#### Dependencies

This property only applies when you set the `Structure` property to one of `Direct form I` or `Direct form I transposed`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

#### OptimizeUnityScaleValues — Optimize unity scale values

`true` (default) | `false`

When this Boolean property is set to `true`, the biquad filter removes all unity scale gain computations. This reduces the number of computations and increases the fixed-point accuracy.

#### Dependencies

This property applies only when you set the `SOSMatrixSource` property to `Property`.

#### ScaleValuesInputPort — How to specify scale values

`true` (default) | `false`

Select how to specify scale values. By default, this property is `true`, and the scale values are specified via the input port. When this property is `false`, all scale values are 1.

#### Dependencies

This property applies only when the `SOSMatrixSource` property is `Input port`.

#### Fixed-Point Properties

##### RoundingMethod — Rounding method for fixed-point operations

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

Specify the rounding method.

##### OverflowAction — Overflow action for fixed-point operations

`Wrap` (default) | `Saturate`

Specify the overflow action as one of `Wrap` or `Saturate`.

##### MultiplicandDataType — Multiplicand word and fraction lengths

`Same as output` (default) | `Custom`

Specify the multiplicand fixed-point data type as one of `Same as output` or `Custom`.

#### Dependencies

This property applies only when you set the `Structure` property to `Direct form I transposed`.

**CustomMultiplicandDataType — Custom multiplicand word and fraction lengths**`numerictype([],32,30)` (default) | `numerictype`

Specify the multiplicand fixed-point type as a scaled `numerictype` object with a Signedness of Auto.

**Dependencies**

This property applies only when you set the `MultiplicandDataType` property to Custom.

**SectionInputDataType — Section input word and fraction lengths**`Same as input` (default) | `Custom`

Specify the section input fixed-point data type as either `Same as input` or `Custom`.

**CustomSectionInputDataType — Custom section input word and fraction lengths**`numerictype([],16,15)` (default) | `numerictype`

Specify the section input fixed-point type as a scaled `numerictype` object with a Signedness of Auto.

**Dependencies**

This property applies only when you set the `SectionInputDataType` property to Custom.

**SectionOutputDataType — Section output word and fraction lengths**`Same as section input` (default) | `Custom`

Specify the section output fixed-point data type as either `Same as section input` or `Custom`.

**CustomSectionOutputDataType — Custom section output word and fraction lengths**`numerictype([],16,15)` (default) | `numerictype`

Specify the section output fixed-point type as a signed, scaled `numerictype` object with a Signedness of Auto.

**Dependencies**

This property applies only when you set the `SectionOutputDataType` property to Custom.

**NumeratorCoefficientsDataType — Numerator coefficients word and fraction lengths**`Same word length as input` (default) | `Custom`

Specify the numerator coefficients fixed-point data type as `Same word length as input` or `Custom`. Setting this property also sets the `DenominatorCoefficientsDataType` and `ScaleValuesDataType` properties to the same value.

**Dependencies**

This property applies only when you set the `SOSMatrixSource` property to Property.

**CustomNumeratorCoefficientsDataType — Custom numerator coefficients word and fraction lengths**`numerictype([],16,15)` (default) | `numerictype`

Specify the numerator coefficients fixed-point type as a `numerictype` object with a Signedness of Auto. The word length of the `CustomNumeratorCoefficientsDataType`,



CustomDenominatorCoefficientsDataType, and CustomScaleValuesDataType properties must be the same.

#### Dependencies

This property applies only when you set the SOSMatrixSource property to Property and the NumeratorCoefficientsDataType property to Custom.

#### DenominatorCoefficientsDataType — Denominator coefficients word and fraction lengths

Same word length as input (default) | Custom

Specify the denominator coefficients fixed-point data type as Same word length as input or Custom. Setting this property also sets the NumeratorCoefficientsDataType and ScaleValuesDataType properties to the same value.

#### Dependencies

This property applies only when you set the SOSMatrixSource property to Property.

#### CustomDenominatorCoefficientsDataType — Custom denominator coefficients word and fraction lengths

numerictype([],16,15) (default) | numerictype

Specify the denominator coefficients fixed-point type as a numerictype object with a Signedness of Auto. The CustomNumeratorCoefficientsDataType, CustomDenominatorCoefficientsDataType, and CustomScaleValuesDataType properties must have the same word lengths.

#### Dependencies

This property applies only when you set the SOSMatrixSource property to Property and the DenominatorCoefficientsDataType property to Custom.

#### ScaleValuesDataType — Scale values word and fraction lengths

Same word length as input (default) | Custom

Specify the scale values fixed-point data type as Same word length as input or Custom. Setting this property also sets the NumeratorCoefficientsDataType and DenominatorCoefficientsDataType properties to the same value.

#### Dependencies

This property applies only when you set the SOSMatrixSource property to Property.

#### CustomScaleValuesDataType — Custom scale values word and fraction lengths

numerictype([],16,15) (default) | numerictype

Specify the scale values fixed-point type as a numerictype object with a Signedness of Auto. The CustomNumeratorCoefficientsDataType, CustomDenominatorCoefficientsDataType, and CustomScaleValuesDataType properties must have the same word lengths.

#### Dependencies

This property applies only when you set the SOSMatrixSource property to Property and the ScaleValuesDataType property to Custom.

#### NumeratorProductDataType — Numerator product word and fraction lengths

Same as input (default) | Custom | Full precision

Specify the mode to determine the numerator product fixed-point data type as:

- `Same as input` (default) — The numerator product word and fraction lengths are same as that of the input.
- `Custom` — Enables the `CustomNumeratorProductDataType` property, which you can use to specify the custom numerator product data type. Specify the data type as a `numericType` object.
- `Full precision` — Use full-precision rules to specify the data type. These rules provide the most accurate fixed-point numerics. The rules prevent quantization from occurring within the object. Bits are added, as needed, so that no roundoff or overflow occurs. For more information, see “Full Precision for Fixed-Point System Objects”.

Setting this property also sets the `DenominatorProductDataType` property to the same value.

#### **CustomNumeratorProductDataType — Custom numerator product word and fraction lengths**

`numericType([], 32, 30)` (default) | `numericType`

Specify the product fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. The `CustomNumeratorProductDataType` and `CustomDenominatorProductDataType` properties must have the same word lengths.

#### **Dependencies**

This property applies only when you set the `NumeratorProductDataType` property to `Custom`.

#### **DenominatorProductDataType — Denominator product word and fraction lengths**

`Same as input` (default) | `Custom` | `Full precision`

Specify the mode to determine the denominator product fixed-point data type as:

- `Same as input` (default) — The denominator product word and fraction lengths are same as that of the input.
- `Custom` — Enables the `CustomDenominatorProductDataType` property, which you can use to specify the custom denominator product data type. Specify the data type as a `numericType` object.
- `Full precision` — Use full-precision rules to specify the data type. These rules provide the most accurate fixed-point numerics. The rules prevent quantization from occurring within the object. Bits are added, as needed, so that no roundoff or overflow occurs. For more information, see “Full Precision for Fixed-Point System Objects”.

Setting this property also sets the `NumeratorProductDataType` property to the same value.

#### **CustomDenominatorProductDataType — Custom denominator product word and fraction lengths**

`numericType([], 32, 30)` (default) | `numericType`

Specify the product fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. The `CustomNumeratorProductDataType` and `CustomDenominatorProductDataType` properties must have the same word lengths.

#### **Dependencies**

This property applies only when you set the `DenominatorProductDataType` to `Custom`.

**NumeratorAccumulatorDataType — Numerator accumulator word and fraction lengths**

Same as product (default) | Same as input | Custom

Specify the numerator accumulator fixed-point data type as Same as input, Same as product, or Custom. Setting this property also sets the DenominatorAccumulatorDataType property to the same value.

**CustomNumeratorAccumulatorDataType — Custom numerator accumulator word and fraction lengths**

numericitytype([],32,30) (default) | numericitytype

Specify the numerator accumulator fixed-point type as a scaled numericitytype object with a Signedness of Auto. The CustomNumeratorAccumulatorDataType and CustomDenominatorAccumulatorDataType properties must have the same word lengths.

**Dependencies**

This property applies only when you set the NumeratorAccumulatorDataType property to Custom.

**DenominatorAccumulatorDataType — Denominator accumulator word and fraction lengths**

Same as product (default) | Same as input | Custom

Specify the denominator accumulator fixed-point data type as Same as input, Same as product, or Custom. Setting this property also sets the NumeratorAccumulatorDataType property to the same value.

**CustomDenominatorAccumulatorDataType — Custom denominator accumulator word and fraction lengths**

numericitytype([],32,30) (default) | numericitytype

Specify the denominator accumulator fixed-point type as a scaled numericitytype object with a Signedness of Auto. The CustomNumeratorAccumulatorDataType and CustomDenominatorAccumulatorDataType properties must have the same word lengths.

**Dependencies**

This property applies only when you set the DenominatorAccumulatorDataType property to Custom.

**StateDataType — State word and fraction lengths**

Same as accumulator (default) | Same as input | Custom

Specify the state fixed-point data type as Same as input, Same as accumulator, or Custom.

**Dependencies**

This property applies when you set the “Structure” on page 4-0 property to Direct form II or Direct form II transposed.

**CustomStateDataType — Custom state word and fraction lengths**

numericitytype([],16,15) (default) | numericitytype

Specify the state fixed-point type as a scaled numericitytype object with a Signedness of Auto.

**Dependencies**

This property applies only when you set the StateDataType property to Custom.

**NumeratorStateDataType — Numerator state word and fraction lengths**

Same as accumulator (default) | Same as input | Custom

Specify the numerator state fixed-point data type as Same as input, Same as accumulator, or Custom. Setting this property also sets the DenominatorStateDataType property to the same value.

**Dependencies**

This property applies only when you set the “Structure” on page 4-0 property to Direct form I transposed.

**CustomNumeratorStateDataType — Custom numerator state word and fraction lengths**

numerictype([],16,15) (default) | numerictype

Specify the numerator state fixed-point type as a scaled numerictype object with a Signedness of Auto. The CustomNumeratorProductDataType and CustomDenominatorProductDataType properties must have the same word lengths.

**Dependencies**

This property applies only when you set the StateDataType property to Custom.

**DenominatorStateDataType — Denominator state word and fraction lengths**

Same as accumulator (default) | Same as input | Custom

Specify the denominator state fixed-point data type as Same as input, Same as accumulator, or Custom. Setting this property also sets the NumeratorStateDataType property to the same value.

**Dependencies**

This property applies only when you set the “Structure” on page 4-0 property to Direct form I transposed.

**CustomDenominatorStateDataType — Custom denominator state word and fraction lengths**

numerictype([],16,15) (default) | numerictype

Specify the denominator state fixed-point type as a scaled numerictype object with a Signedness of Auto. The CustomNumeratorStateDataType and CustomDenominatorStateDataType properties must have the same word lengths.

**Dependencies**

This property applies only when you set the StateDataType property to Custom.

**OutputDataType — Output word and fraction lengths**

Same as accumulator (default) | Same as input | Custom

Specify the output fixed-point data type as Same as input, Same as accumulator, or Custom.

**CustomOutputDataType — Custom output word and fraction lengths**

numerictype([],16,15) (default) | numerictype

Specify the output fixed-point type as a scaled numerictype object with a Signedness of Auto.

**Dependencies**

This property applies only when you set the “OutputDataType” on page 4-0 property to Custom.

## Usage

### Syntax

```
y = biquad(x)
y = biquad(x,num,den)
y = biquad(x,num,den,g)
```

### Description

`y = biquad(x)` filters the input signal `x`, and outputs the filtered values, `y`. The biquad filter object filters each channel of the input signal over successive calls to the algorithm.

`y = biquad(x,num,den)` filters the input using `num` as the numerator coefficients, and `den` as the denominator coefficients of the biquad filter. This configuration applies when the `SOSMatrixSource` property is `Input port` and the `ScaleValuesInputPort` property is `false`.

`y = biquad(x,num,den,g)` specifies the scale values, `g`, of the biquad filter. This configuration applies when the `SOSMatrixSource` property is `Input Port` and the `ScaleValuesInputPort` property is `true`.

### Input Arguments

#### **x** — Data input

vector | matrix

Data input, specified as a vector or a matrix. This object also accepts variable-size inputs. Once the object is locked, you can change the size of each input channel, but you cannot change the number of channels.

The data type of all the inputs must be the same. If the input is fixed-point, it must be signed fixed point with power-of-two slope and zero bias.

The complexity of `x`, `num`, and `den` must be the same.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `fi`

#### **num** — Numerator coefficients

3-by-*N* matrix

Numerator coefficients, specified as a 3-by-*N* numeric matrix, where *N* is the number of biquad filter sections. The complexity of `x`, `num`, and `den` must be the same.

The data type of all the inputs must be the same. If `num` is fixed point, it must be signed fixed point with power-of-two slope and zero bias.

### Dependencies

This input applies only when you set `SOSMatrixSource` property is `Input port`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `fi`

Complex Number Support: Yes

#### **den** — Denominator coefficients

2-by-*N* matrix

Denominator coefficients, specified as a 2-by- $N$  numeric matrix, where  $N$  is the number of biquad filter sections. The object assumes that the first denominator coefficient of each section is 1.

The data type of all the inputs must be the same. If `den` is fixed point, it must be signed fixed point with power-of-two slope and zero bias.

The complexity of `x`, `num`, and `den` must be the same.

### Dependencies

This input applies only when you set `SOSMatrixSource` property is `InputPort`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `fi`

Complex Number Support: Yes

### **g** — Scale values

1-by- $(N + 1)$  vector

Scale values of the biquad filter, specified as a 1-by- $(N + 1)$  numeric vector, where  $N$  is the number of biquad filter sections.

The data type of all the inputs must be the same. If `g` is fixed point, it must be signed fixed point with power-of-two slope and zero bias.

### Dependencies

This input applies when the `SOSMatrixSource` property is `InputPort` and the `ScaleValuesInputPort` property is `true`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `fi`

### Output Arguments

#### **y** — Filtered output

vector | matrix

Filtered output, returned as a vector or a matrix. The size, data type, and complexity of the output signal matches that of the input signal.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `fi`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

### Specific to `dsp.BiquadFilter`

|                     |  |
|---------------------|--|
| <code>freqz</code>  | Frequency response of discrete-time filter System object     |
| <code>fvtool</code> | Visualize frequency response of DSP filters                  |
| <code>impz</code>   | Impulse response of discrete-time filter System object       |
| <code>info</code>   | Information about filter System object                       |
| <code>coeffs</code> | Returns the filter System object coefficients in a structure |
| <code>cost</code>   | Estimate cost of implementing filter System object           |

|             |   |
|-------------|---|
| scale       | Scale second-order sections   |
| scaleopts   | Create an options object for second-order section scaling                     |
| scalecheck  | Check scaling of biquadratic filter   |
| cumsec      | Cumulative second-order section of the biquadratic filter                     |
| generatehdl | Generate HDL code for quantized DSP filter (requires Filter Design HDL Coder) |
| tf          | Convert discrete-time filter System object to transfer function               |
| reorder     | Reorder second-order sections of biquadratic filter System object             |

## Common to All System Objects

|         |  |
|---------|--|
| step    | Run System object algorithm  |
| release | Release resources and allow changes to System object property values and input characteristics |
| reset   | Reset internal states of System object   |

## Examples

### Filter a Signal Using Biquadratic Filter

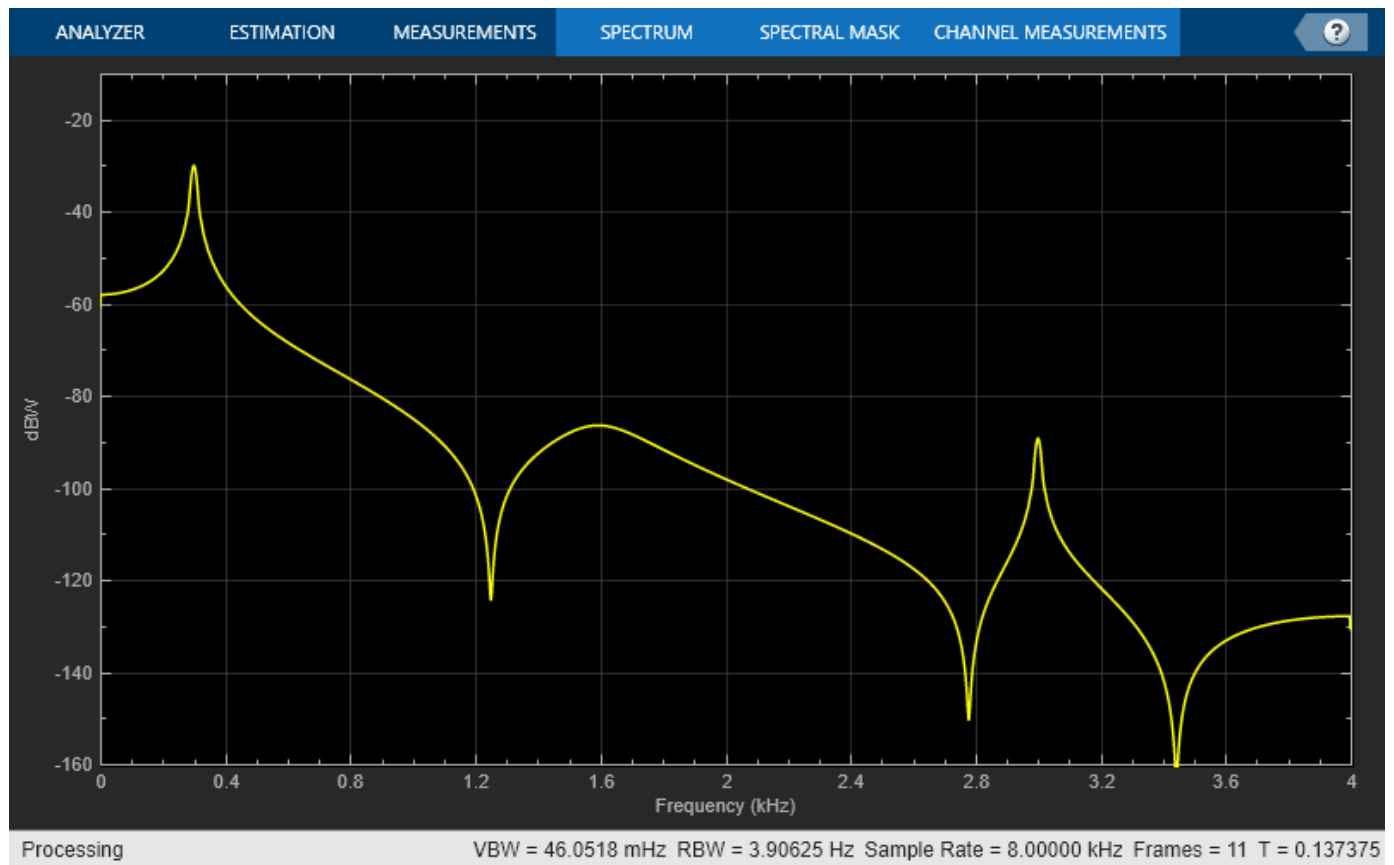
Use a fourth order, lowpass biquadratic filter object with a normalized cutoff frequency of 0.4 to filter high frequencies from an input signal. Display the result as a power spectrum using the Spectrum Analyzer.

```
t = (0:1000)'/8e3;
xin = sin(2*pi*0.3e3*t)+sin(2*pi*3e3*t); % Input is 0.3 & amp;
                                     % 3kHz sinusoids
src = dsp.SignalSource(xin, 100);
sink = dsp.SignalSink;

[z,p,k] = ellip(4,1,60,.4); % Set up the filter
[sosMatrix,scaleValues] = zp2sos(z,p,k);
biquad = dsp.BiquadFilter(sosMatrix,scaleValues,Structure="Direct form I");

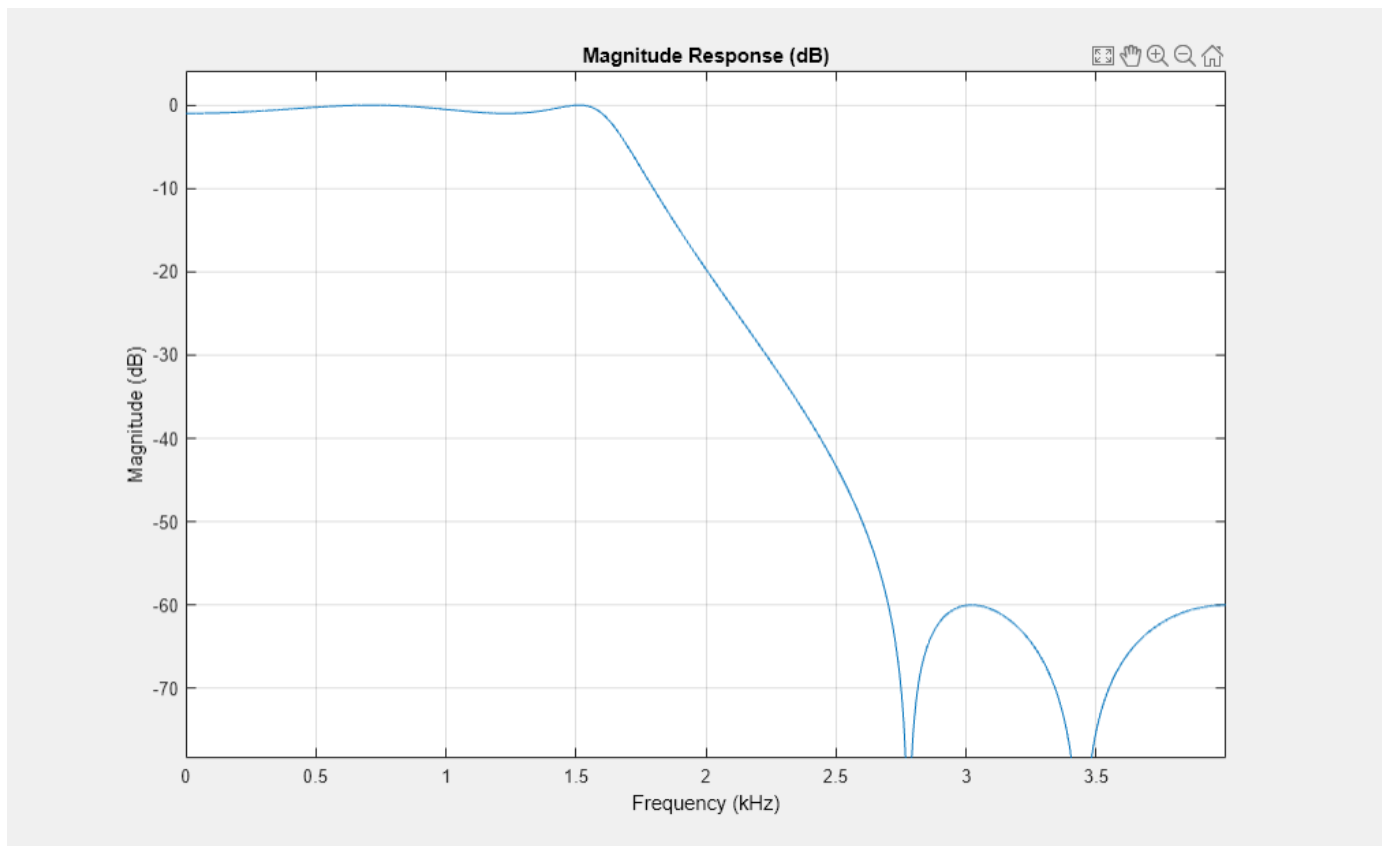
sa = spectrumAnalyzer(SampleRate=8e3,...
    Method="welch",...
    PlotAsTwoSidedSpectrum=false,...
    OverlapPercent=80,SpectrumUnits="dBW",...
    YLimits=[-160 -10]);

while ~isDone(src)
    input = src();
    filteredOutput = biquad(input);
    sink(filteredOutput);
    sa(filteredOutput)
end
```



```
filteredResult = sink.Buffer;  
fvtool(biquad,Fs=8000)
```





### Linf-norm Scaling of a Biquadratic SOS Filter

Demonstrate the Linf-norm scaling of a biquadratic SOS filter using the `scale` function.

```
Fs = 8000;
Fcutoff = 2000;
[z,p,k] = butter(10,Fcutoff/(Fs/2));
[sosMatrix,scaleValues] = zp2sos(z,p,k);
sosFilt = dsp.SOSFilter(Structure='Direct form I', ...
    Numerator=sosMatrix(:,1:3),Denominator=sosMatrix(:,4:6), ...
    HasScaleValues=true,ScaleValues=scaleValues)
```

```
sosFilt =
    dsp.SOSFilter with properties:
        Structure: 'Direct form I'
        CoefficientSource: 'Property'
        Numerator: [5x3 double]
        Denominator: [5x3 double]
        HasScaleValues: true
        ScaleValues: [0.0029 1 1 1 1 1]
```

Show all properties

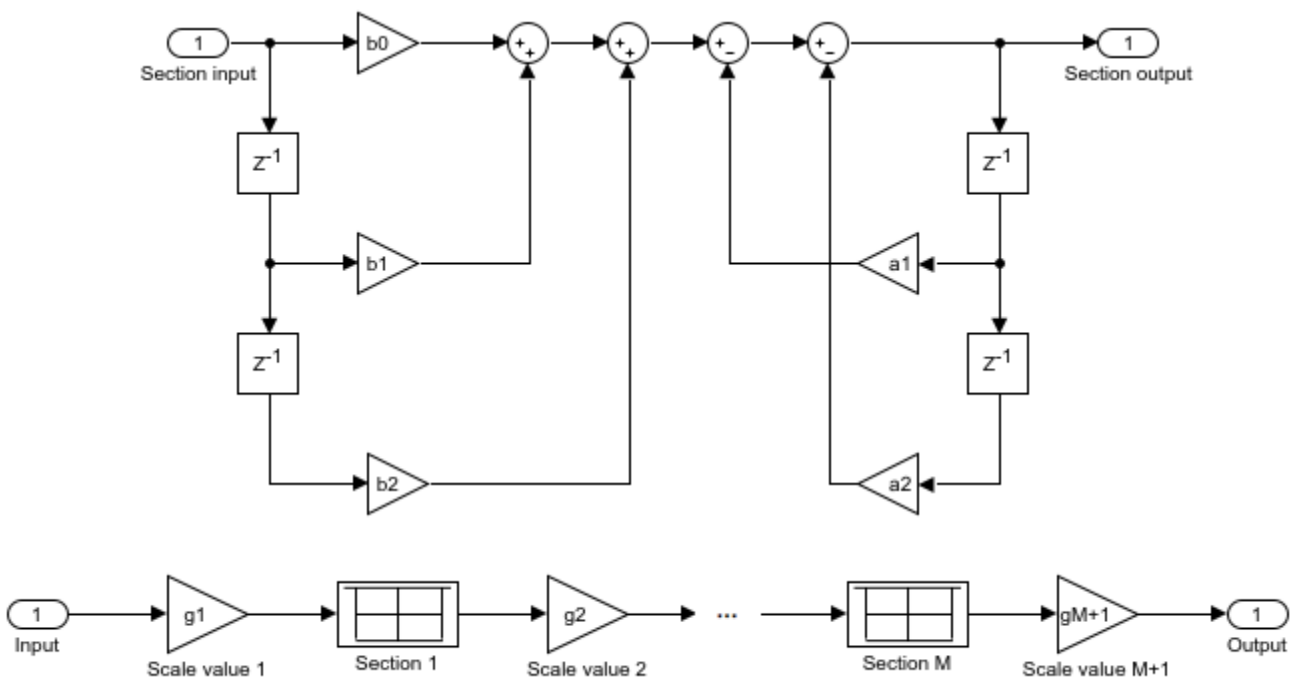
```
scale(sosFilt, 'linf', scalevalueconstraint='none', maxscalevalue=2)
```

## More About

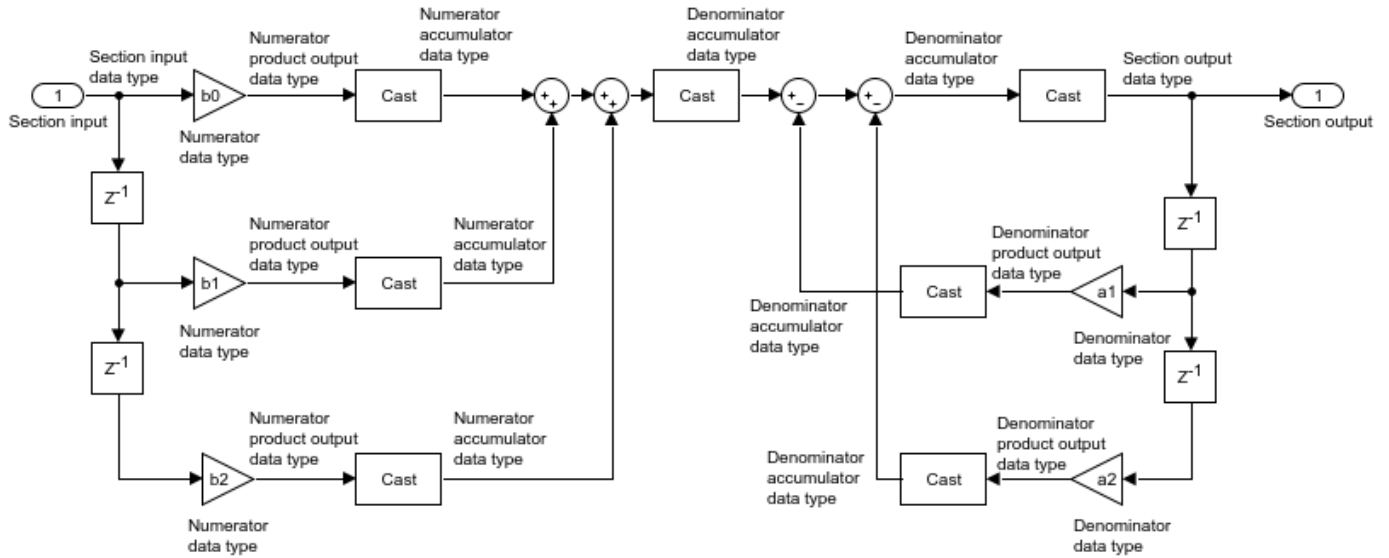
### Fixed Point

The following diagrams show the data types used in the `dsp.BiquadFilter` object when the input is fixed-point. For each filter structure the object supports, the data types shown in the diagrams can be set through the respective fixed-point properties of the object.

#### Direct Form I

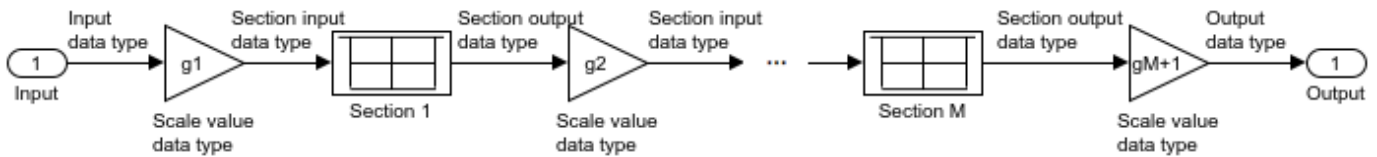


The following diagram shows the data types for one section of the filter for fixed-point signals.

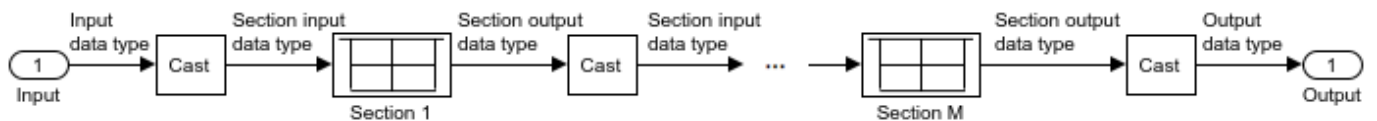


The following diagrams show the fixed-point data types between filter sections.

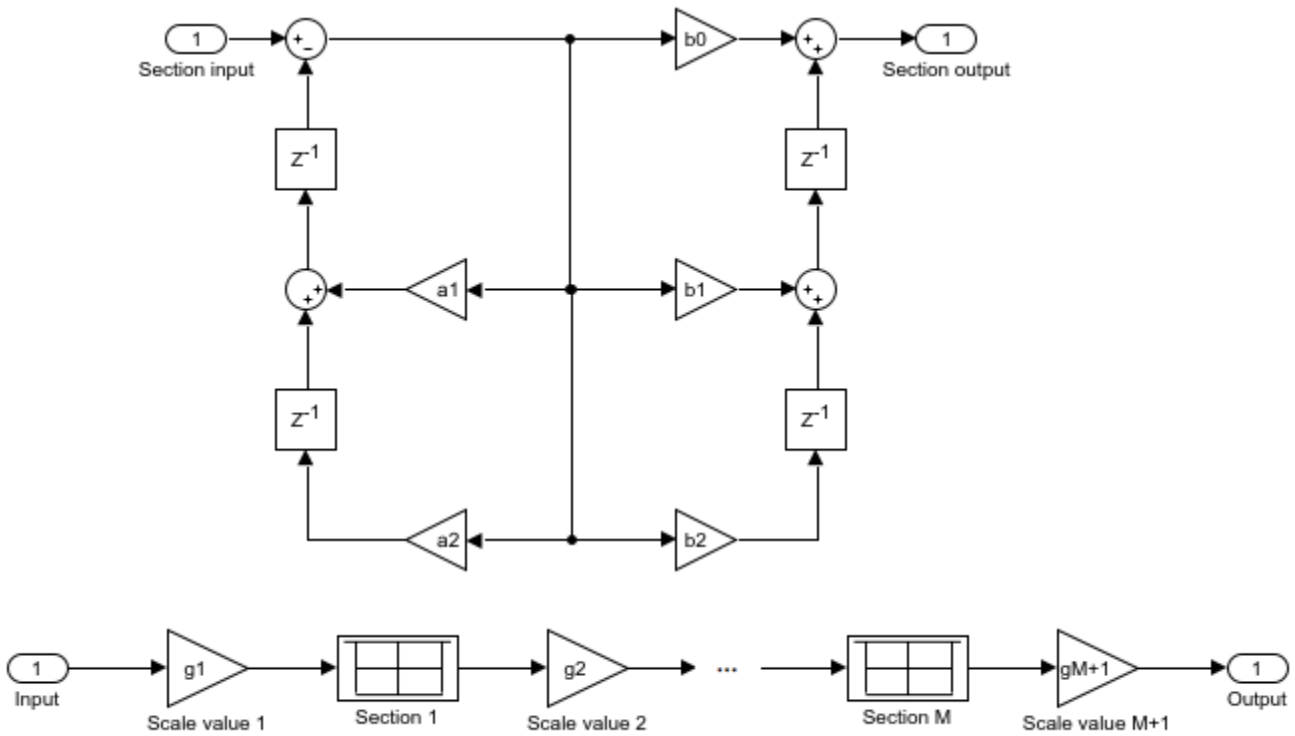
When the data is not optimized:



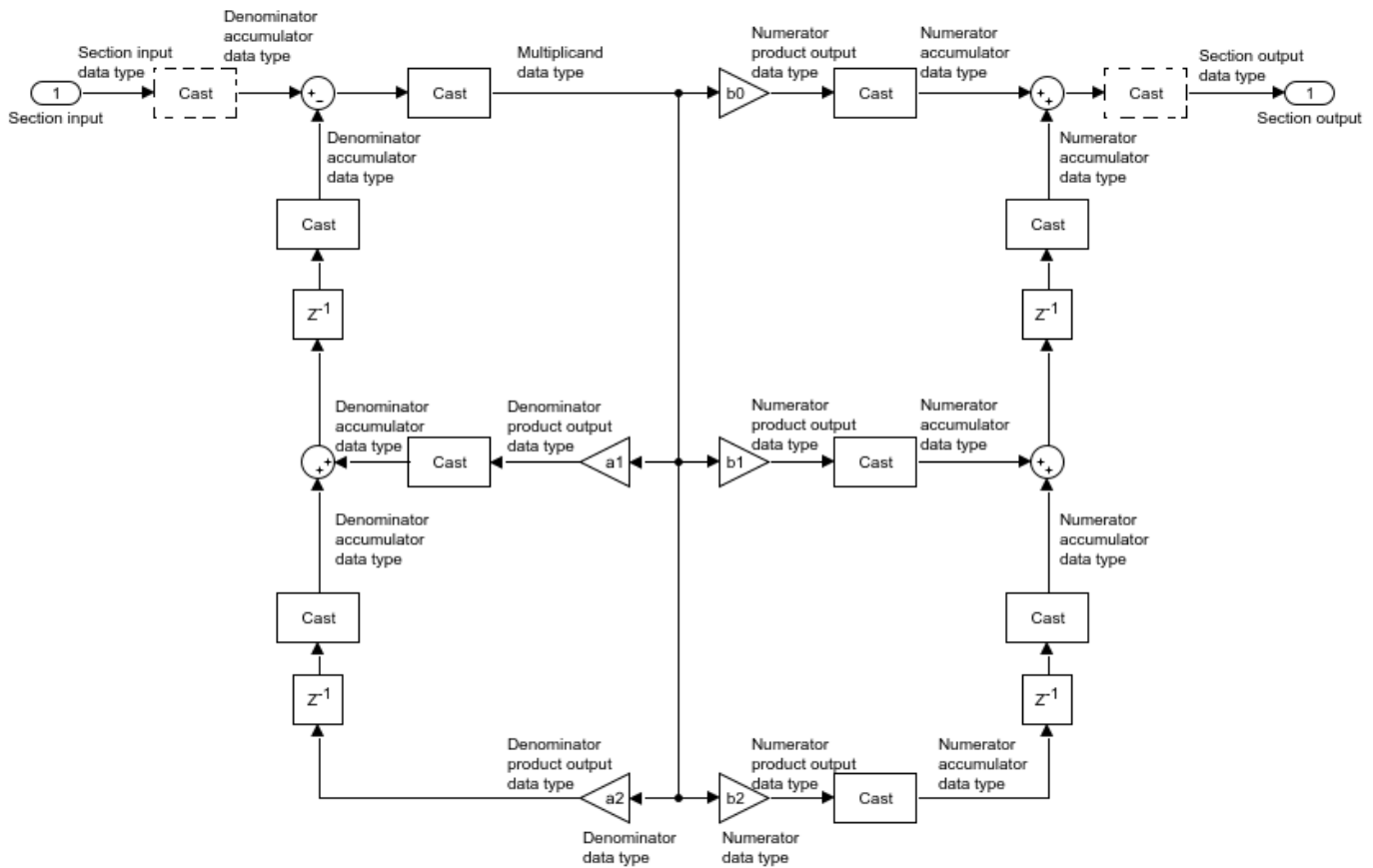
When you specify OptimizeUnityScaleValues to true, and scale values to 1:



### Direct Form I Transposed



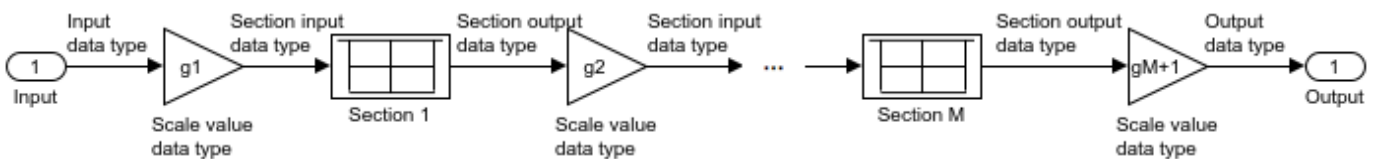
The following diagram shows the data types for one section of the filter for fixed-point signals.



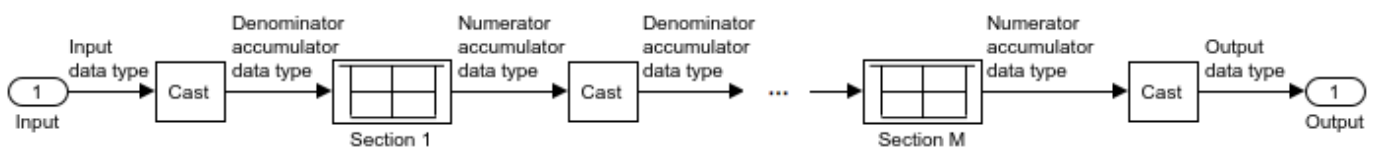
The dashed casts are omitted when you specify `OptimizeUnityScaleValues` to `true`, and scale values to 1.

The following diagrams show the fixed-point data types between filter sections.

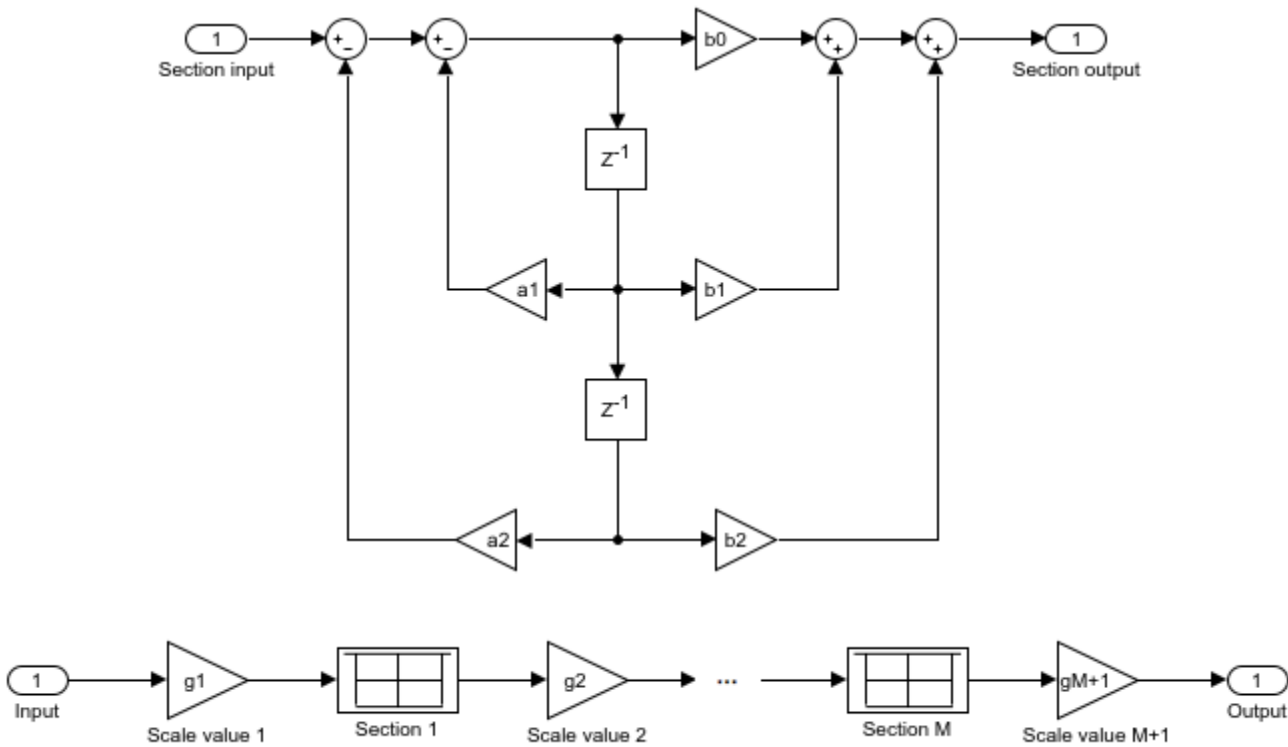
When the data is not optimized:



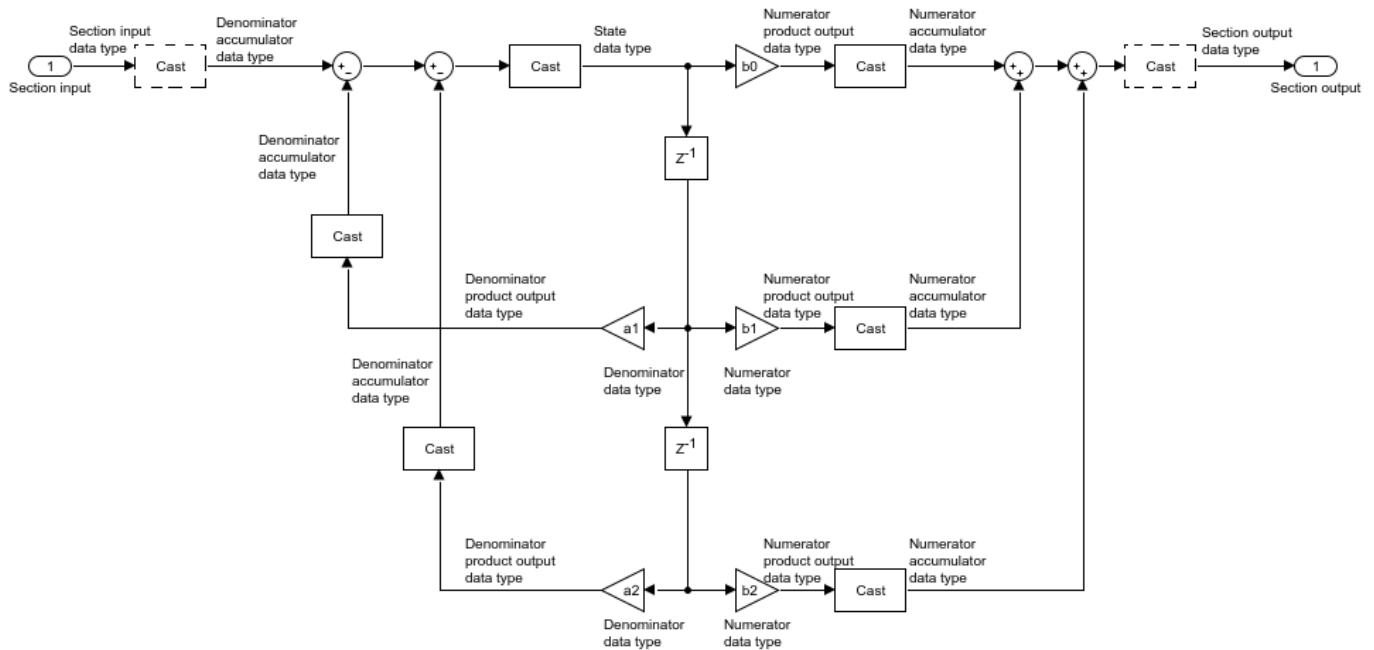
When you specify `OptimizeUnityScaleValues` to `true`, and scale values to 1:



### Direct Form II



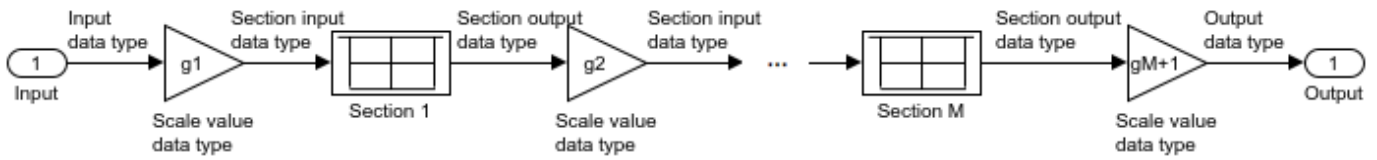
The following diagram shows the data types for one section of the filter for fixed-point signals.



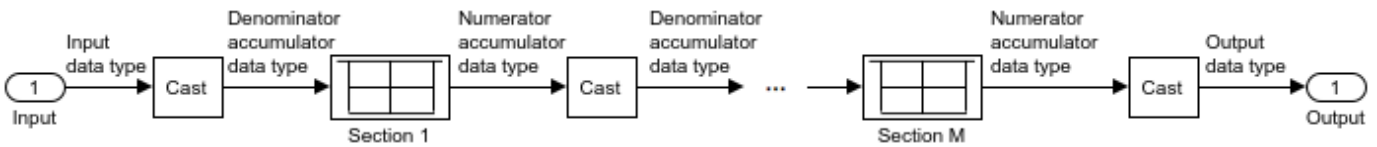
The dashed casts are omitted when you specify `OptimizeUnityScaleValues` to `true`, and scale values to 1.

The following diagrams show the fixed-point data types between filter sections.

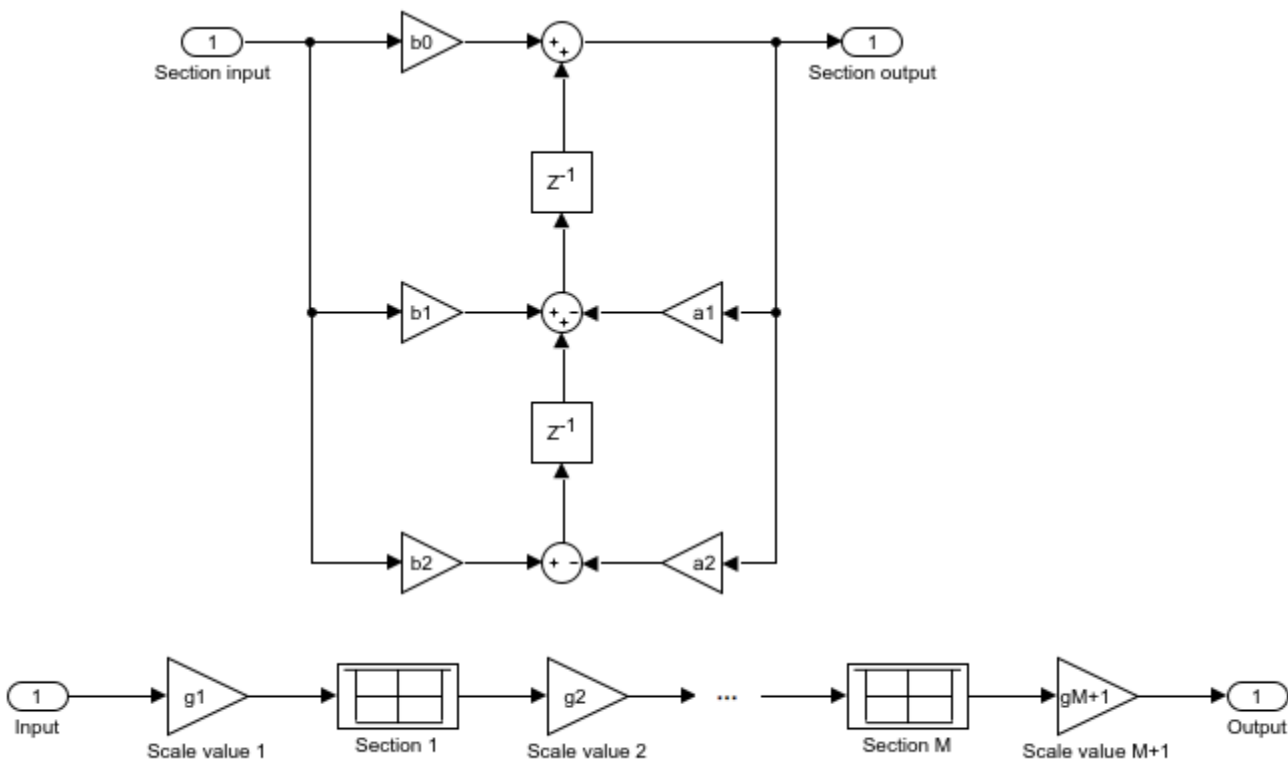
When the data is not optimized:



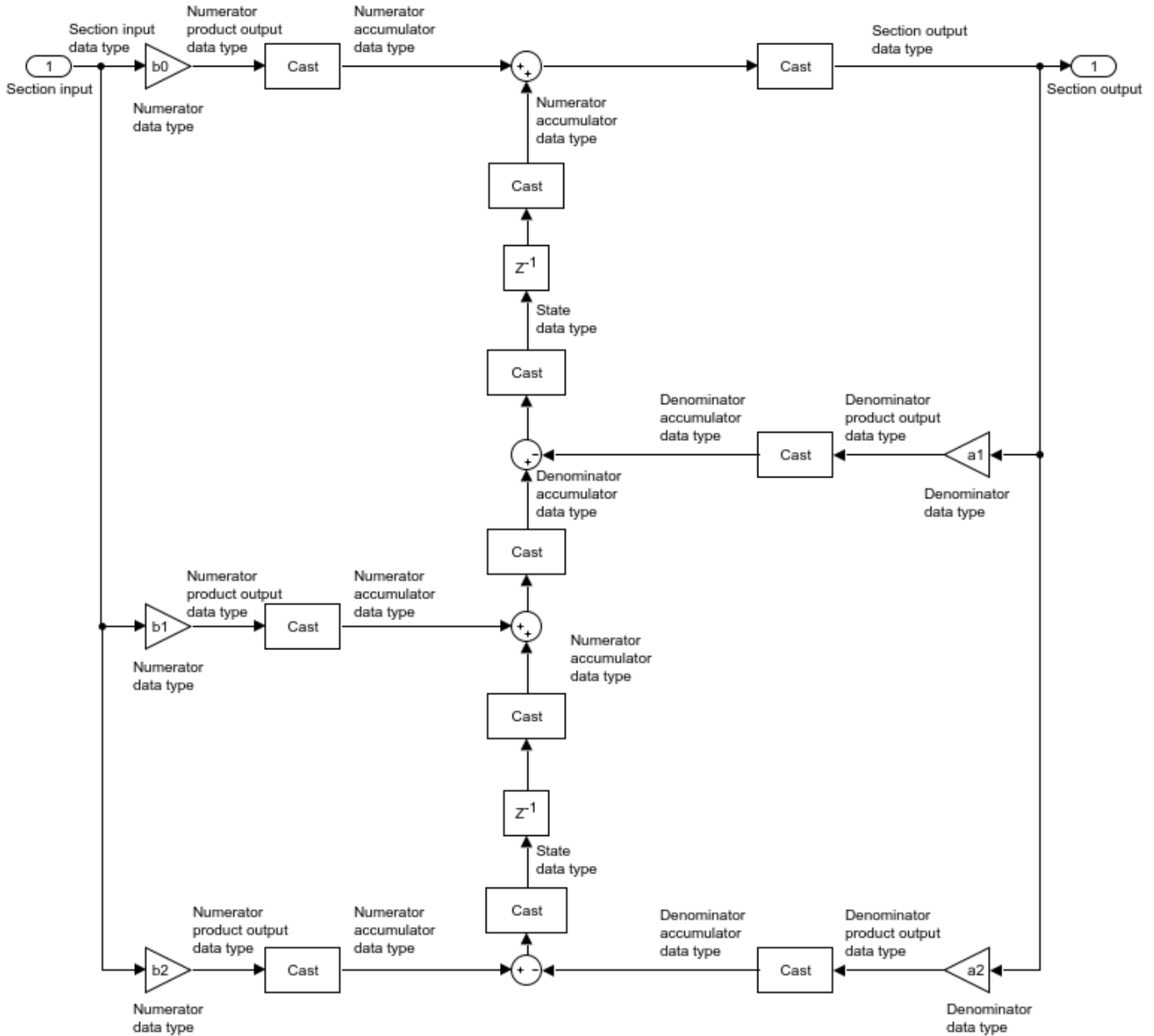
When you specify `OptimizeUnityScaleValues` to `true`, and scale values to 1:



**Direct Form II Transposed**

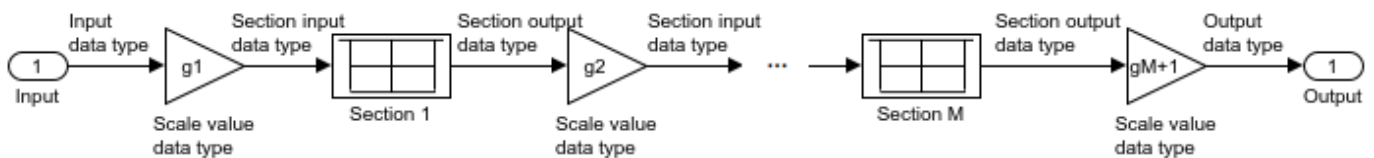


The following diagram shows the data types for one section of the filter for fixed-point signals.



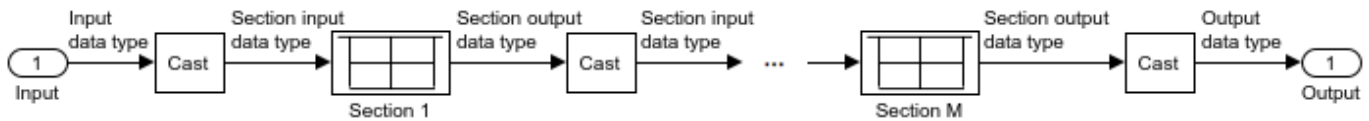
The following diagrams show the fixed-point data types between filter sections.

When the data is not optimized:



When you specify `OptimizeUnityScaleValues` to `true`, and scale values to 1:





## Algorithms

This object implements the algorithm, inputs, and outputs described on the Biquad Filter block reference page. The object properties correspond to the block parameters, except:

- **Coefficient source**
- **Action when the  $a_0$  values of the SOS matrix are not one** - the biquad filter object assumes the zero-th-order denominator coefficient equals 1 regardless of the specified value. The biquad filter object does not support the Error or Warn options found in the corresponding block.

Both this object and its corresponding block support variable-size input. When you call the object, it can handle an input argument which is changing in size.

## Version History

Introduced in R2012a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

This object supports HDL code generation with the HDL Coder or Filter Design HDL Coder™ products. For HDL Coder workflows and limitations, see “HDL Code Generation for System Objects” (HDL Coder). For Filter Design HDL Coder workflows and limitations, see “Generate HDL Code for Filter System Objects” (Filter Design HDL Coder).

## See Also

### Functions

freqz | fvtool | impz | info | coeffs | cost | scale | scaleopts | scalecheck | cumsec | generatehdl | tf

### Objects

dsp.FIRFilter | dsp.IIRFilter | dsp.SOSFilter

### Blocks

Biquad Filter

**Topics**

“Analysis Methods for Filter System Objects” on page 3-2

# dsp.BlockLMSFilter

**Package:** dsp

Compute output, error, and weights using block LMS adaptive algorithm

## Description

The `dsp.BlockLMSFilter` System object computes output, error, and weights using the block LMS adaptive algorithm.

To compute the output, error, and weights:

- 1 Create the `dsp.BlockLMSFilter` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
blms = dsp.BlockLMSFilter
blms = dsp.BlockLMSFilter(length,blocksize)
blms = dsp.BlockLMSFilter(Name,Value)
```

### Description

`blms = dsp.BlockLMSFilter` returns an adaptive FIR filter, `blms`, that filters the input signal and computes filter weights based on the block least mean squares (LMS) algorithm.

`blms = dsp.BlockLMSFilter(length,blocksize)` returns an adaptive FIR filter, `blms`, with the `Length` property set to `length` and the `BlockSize` property set to `blocksize`.

`blms = dsp.BlockLMSFilter(Name,Value)` returns an adaptive FIR filter, `blms`, with each specified property set to the specified value. Enclose each property name in single quotes. Unspecified properties have default values.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

### Length — Length of FIR filter weights vector

32 (default) | positive integer

Specify the length of the FIR filter weights vector as a positive integer scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**BlockSize — Number of samples acquired before weight adaptation**

32 (default) | positive integer

Specify the number of samples of the input signal to acquire before the object updates the filter weights. The input frame length must be an integer multiple of the block size.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**StepSizeSource — Source of adaptation step size**

Property (default) | Input port

Specify the source of the adaptation step size factor as Property or Input port.

**StepSize — Adaptation step size**

0.1 (default) | nonnegative scalar

Specify the adaptation step size factor as a scalar, nonnegative numeric value.

**Tunable:** Yes

**Dependencies**

This property applies only when you set the “StepSizeSource” on page 4-0 property to 'Property'.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**LeakageFactor — Leakage factor used in leaky LMS algorithm**

1 (default) | scalar

Specify the leakage factor used in leaky LMS algorithm as a scalar numeric value between 0 and 1, both inclusive. When the value is less than 1, the System object implements a leaky LMS algorithm. The default is 1, providing no leakage in the adapting algorithm.

**Tunable:** Yes

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**InitialWeights — Initial values of filter weights**

0 (default) | scalar | vector

Specify the initial values of the filter weights as a scalar or a vector of length equal to the “Length” on page 4-0 property value.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**AdaptInputPort — Additional input to enable adaptation of filter weights**

false (default) | true

Specify when the object should adapt the filter weights. By default, the value of this property is `false`, and the filter continuously updates the filter weights. When this property is set to `true`, an adaptation control input is provided to the object. If the value of this input is nonzero, the filter continuously updates the filter weights. If the input is zero, the filter weights remain at their current value.

**WeightsResetInputPort — Additional input to enable weights reset**`false (default) | true`

Specify whether the FIR filter can reset the filter weights. By default, the value of this property is `false`, and the object does not reset the weights. When this property is set to `true`, you must provide a reset control input to the object, and the `WeightsResetCondition` property applies. The object resets the filter weights based on the values of the `WeightsResetCondition` property and the reset input to the object algorithm.

**WeightsResetCondition — Condition that triggers the resetting of filter weights**`Non-zero (default) | Rising edge | Falling edge | Either edge`

Specify the event to reset the filter weights as one of `Rising edge`, `Falling edge`, `Either edge`, or `Non-zero`. The object resets the filter weights based on the values of this property and the reset signal input to the object algorithm.

**Dependencies**

This property applies only when you set the `WeightsResetInputPort` property to `true`.

**WeightsOutputPort — Output filter weights**`true (default) | false`

Set this property to `true` to output the adapted filter weights. The default is `true`.

**Usage****Syntax**

```
[y,err,wts] = blms(x,d)
[y,err] = blms(x,d)
[ ___ ] = blms(x,d,mu)
[ ___ ] = blms(x,d,a)
[ ___ ] = blms(x,d,r)
[y,err,wts] = blms(x,d,mu,a,r)
```

**Description**

`[y,err,wts] = blms(x,d)` filters the input `x`, using `d` as the desired signal, and returns the filtered output in `y`. The filter error is `err`, and the estimated filter weights is `wts`. The filter weights update once for every block of data that the object processes.

`[y,err] = blms(x,d)` returns only the filtered output `y` and the filter error `err` when the `WeightsOutputPort` property is `false`.

`[ ___ ] = blms(x,d,mu)` uses `mu` as the step size when you set the `StepSizeSource` property to `Input port`. These input arguments can be used with any of the previous sets of output arguments.

`[ ___ ] = blms(x,d,a)` uses `a` as the adaptation control when you set the `AdaptInputPort` property to `true`. When `a` is nonzero, the filter continuously updates the filter weights. When `a` is zero, the filter weights remain constant.

[ \_\_\_ ] = `blms(x,d,r)` uses `r` as a reset signal when you set the `WeightsResetInputPort` property to `true`. Use the `WeightsResetCondition` property to set the reset trigger condition. If a reset event occurs, the filter resets the filter weights to their initial values.

[`y,err,wts`] = `blms(x,d,mu,a,r)` filters input `x`, using `d` as the desired signal, `mu` as the step size, `a` as the adaptation control, and `r` as the reset signal. The object returns the filtered output `y`, the filter error `err`, and the adapted filter weights `wts`. Set the properties appropriately to provide all possible inputs.

### Input Arguments

#### **x** — Data input

scalar | column vector

The signal to be filtered by the block LMS filter. The input, `x`, and the desired signal, `d`, must have the same size and data type.

The input length must be an integer multiple of the `BlockSize` property value.

Data Types: `single` | `double`

#### **d** — Desired signal

scalar | column vector

The LMS filter adapts its filter weights, `wts`, to minimize the error, `err`, and converge the input signal `x` to the desired signal `d` as closely as possible.

The input, `x`, and the desired signal, `d`, must have the same size and data type.

Data Types: `single` | `double`

#### **mu** — Step size

nonnegative scalar

Adaptation step size factor, specified as a scalar, nonnegative numeric value. The data type of the step size input must match the data type of `x` and `d`.

A small step size ensures a small steady state error between the output `y` and the desired signal `d`. If the step size is small, the convergence speed of the filter decreases. To improve the convergence speed, increase the step size. Note that if the step size is large, the filter can become unstable. To compute the maximum step size the filter can accept without becoming unstable, use the `maxstep` function.

### Dependencies

This property applies only when you set the “StepSizeSource” on page 4-0 property to 'Input port'.

Data Types: `single` | `double`

#### **a** — Adaptation control

scalar

Adaptation control input that controls how the filter weights are updated. If the value of this input is nonzero, the object continuously updates the filter weights. If the value of this input is zero, the filter weights remain at their current value.

**Dependencies**

This input is required when the `AdaptInputPort` property is set to `true`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `logical`

**r — Reset signal**

scalar

Reset signal that resets the filter weights based on the values of the `WeightsResetInputPort` property.

**Dependencies**

This input is required when the `WeightsResetInputPort` property is set to `true`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `logical`

**Output Arguments****y — Filtered output**

scalar | column vector

Filtered output, returned as a scalar or a column vector. The object adapts its filter weights to converge the input signal `x` to match the desired signal `d`. The filter outputs the converged signal.

Data Types: `single` | `double`

**err — Difference between output and desired signal**

scalar | column vector

Difference between the output signal `y` and the desired signal `d`, returned as a scalar or a column vector. The data type of `err` matches the data type of `y`. The objective of the adaptive filter is to minimize this error. The object adapts its weights to converge towards optimal filter weights that produce an output signal that matches closely with the desired signal.

Data Types: `single` | `double`

**wts — Adaptive filter weights**

scalar | column vector

Adaptive filter weights, returned as a scalar or a column vector of length specified by the value in the `Length` property.

The data type of `wts` matches the data type of `y`.

Data Types: `single` | `double`

**Object Functions**

To use an object function, specify the `System` object as the first input argument. For example, to release system resources of a `System` object named `obj`, use this syntax:

```
release(obj)
```

### Specific to `dsp.BlockLMSFilter`

`mse` Estimated mean squared error for adaptive filters  
`maxstep` Maximum step size for LMS adaptive filter convergence

### Common to All System Objects

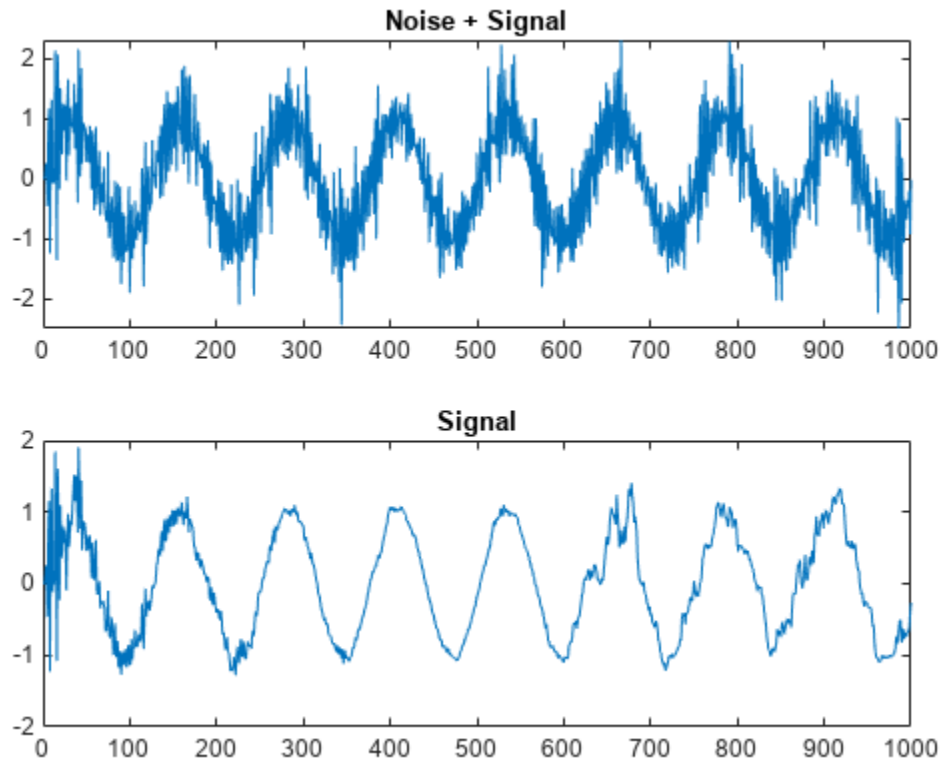
`step` Run System object algorithm  
`release` Release resources and allow changes to System object property values and input characteristics  
`reset` Reset internal states of System object

## Examples

### Remove Noise Using Block LMS Adaptive Algorithm

```
blms = dsp.BlockLMSFilter(10,5);  
blms.StepSize = 0.01;  
blms.WeightsOutputPort = false;  
filt = dsp.FIRFilter;  
filt.Numerator = fir1(10,[.5, .75]);  
x = randn(1000,1); % Noise  
d = filt(x) + sin(0:.05:49.95)'; % Noise + Signal  
[y, err] = blms(x, d);  
subplot(2,1,1);  
plot(d);  
title('Noise + Signal');  
subplot(2,1,2);  
plot(err);  
title('Signal');
```





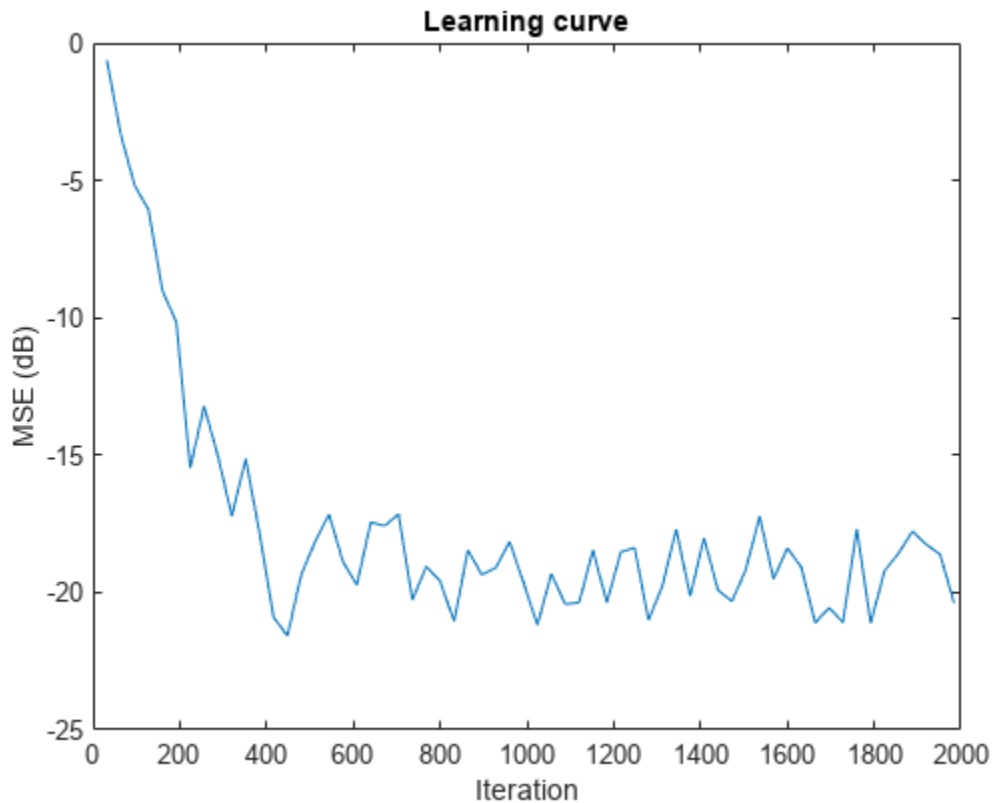
### System Identification of FIR Filter Using Block LMS Filter

```

fir = fir1(31,0.5);
% FIR system to be identified
firFilter = dsp.FIRFilter('Numerator',fir);
iirFilter = dsp.IIRFilter('Numerator',sqrt(0.75),...
    'Denominator',[1 -0.5]);
x = iirFilter(sign(randn(2000,25)));
% Observation noise signal
n = 0.1*randn(size(x));
% Desired signal
d = firFilter(x)+n;
% Filter length
l = 32;
% Block LMS Step size
mu = 0.008;
% Decimation factor for analysis
% and simulation results
m = 32;
fir = dsp.BlockLMSFilter(l,'StepSize',mu);
[simmse,meanWsim,Wsim,traceKsim] = msesim(fir,x,d,m);
plot(m*(1:length(simmse)),10*log10(simmse));
xlabel('Iteration'); ylabel('MSE (dB)');
% Plot the learning curve for

```

```
% block LMS filter used in system identification  
title('Learning curve')
```



## Algorithms

This object implements the algorithm, inputs, and outputs described on the Block LMS Filter block reference page. The object properties correspond to the block parameters.

## Version History

**Introduced in R2012a**

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

## See Also

### Objects

[dsp.LMSFilter](#) | [dsp.FIRFilter](#)

# dsp.Channelizer

**Package:** dsp

Polyphase FFT analysis filter bank

## Description

The `dsp.Channelizer` System object separates a broadband input signal into multiple narrow subbands using a fast Fourier transform (FFT)-based analysis filter bank. The filter bank uses a prototype lowpass filter and is implemented using a polyphase structure. You can specify the filter coefficients directly or through design parameters.

To separate a broadband signal into multiple narrow subbands:

- 1 Create the `dsp.Channelizer` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
channelizer = dsp.Channelizer
channelizer = dsp.Channelizer(M)
channelizer = dsp.Channelizer(M,D)
channelizer = dsp.Channelizer(Name,Value)
```

### Description

`channelizer = dsp.Channelizer` creates a polyphase FFT analysis filter bank System object that separates a broadband input signal into multiple narrowband output signals. This object implements the inverse operation of the `dsp.ChannelSynthesizer` System object.

`channelizer = dsp.Channelizer(M)` creates an  $M$ -band polyphase FFT analysis filter bank, with the `NumFrequencyBands` property set to  $M$ .

Example: `channelizer = dsp.Channelizer(16);`

`channelizer = dsp.Channelizer(M,D)` creates an  $M$ -band polyphase FFT analysis filter bank, with the `DecimationFactor` property set to  $D$ .

Example: `channelizer = dsp.Channelizer(16,8);`

`channelizer = dsp.Channelizer(Name,Value)` creates a polyphase FFT analysis filter bank with each specified property set to the specified value. Enclose each property name in single quotes. For example, you can set `NumTapsPerBand` to 20 and `StopbandAttenuation` to 140.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

### Main Properties

#### NumFrequencyBands — Number of frequency bands

8 (default) | positive integer greater than 1

Number of frequency bands  $M$  into which the object separates the input broadband signal, specified as a positive integer greater than 1. This property corresponds to the number of polyphase branches and the FFT length used in the filter bank.

Example: 16

Example: 64

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

#### DecimationFactor — Decimation factor

8 (default) | positive integer

Decimation factor  $D$  specified as a positive integer less than or equal to the number of frequency bands  $M$ . The default value of this property equals the number of frequency bands specified.

If the decimation factor  $D$  equals the number of frequency bands  $M$ , then the  $M/D$  ratio equals 1, and the channelizer is known as the maximally decimated channelizer.

If the  $M/D$  ratio is greater than 1, the output sample rate is different from the channel spacing, and the channelizer is known as the non-maximally decimated channelizer. If the ratio is an integer, the channelizer is known as the integer-oversampled channelizer. If the ratio is not an integer, say  $4/3$ , the channelizer is known as the rationally oversampled channelizer. For more details, see "Algorithm" on page 4-200.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

#### Specification — Filter design parameters or coefficients

"Number of taps per band and stopband attenuation" (default) | "Coefficients"

Filter design parameters or filter coefficients, specified as one of these options:

- "Number of taps per band and stopband attenuation" — Specify the filter design parameters through the `NumTapsPerBand` and `StopbandAttenuation` properties.
- "Coefficients" — Specify the filter coefficients directly using the `LowpassCoefficients` property.

#### NumTapsPerBand — Number of filter coefficients per frequency band

12 (default) | positive integer

Number of filter coefficients each polyphase branch uses, specified as a positive integer. The number of polyphase branches matches the number of frequency bands. The total number of filter coefficients

for the prototype lowpass filter is given by  $\text{NumFrequencyBands} \times \text{NumTapsPerBand}$ . For a given stopband attenuation, increasing the number of taps per band narrows the transition width of the filter. As a result, there is more usable bandwidth for each frequency band at the expense of increased computation.

Example: 8

Example: 16

**Dependencies**

This property applies when you set `Specification` to "Number of taps per band and stopband attenuation".

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**StopbandAttenuation — Stopband attenuation**

80 (default) | positive real scalar

Stopband attenuation of the lowpass filter, specified as a positive real scalar in dB. This value controls the maximum amount of aliasing from one frequency band to the next. When the stopband attenuation increases, the passband ripple decreases. For a given stopband attenuation, increasing the number of taps per band narrows the transition width of the filter. As a result, there is more usable bandwidth for each frequency band at the expense of increased computation.

Example: 80

**Dependencies**

This property applies when you set `Specification` to "Number of taps per band and stopband attenuation".

Data Types: `single` | `double`

**LowpassCoefficients — Coefficients of prototype lowpass filter**

[1×49 double] (default) | row vector

Coefficients of the prototype lowpass filter, specified as a row vector. The default vector of coefficients is obtained using `rcosdesign(0.25,6,8,"sqrt")`. There must be at least one coefficient per frequency band. If the length of the lowpass filter is less than the number of frequency bands, the object zero-pads the coefficients.

If you specify complex coefficients, the object designs a prototype filter that is centered at a nonzero frequency, also known as a bandpass filter. The modulated versions of the prototype bandpass filter appear with respect to the prototype filter and are wrapped around the frequency range  $[-F_s, F_s]$ . For an example, see "Channelizer with Complex Coefficients" on page 4-196.

**Tunable:** Yes

**Dependencies**

This property applies when you set `Specification` to "Coefficients".

Data Types: `single` | `double`

**Code Generation Properties****AllowArbitraryInputLength — Allow arbitrary input length in generated code**

false (default) | true

Allow arbitrary frame length for fixed-size input signals in the generated code, specified as `true` or `false`. When you specify:

- `true` -- The input frame length does not have to be a multiple of the decimation factor. The output of the object in the generated code is a variable-size array.
- `false` -- The input frame length must be a multiple of the decimation factor.

When you specify variable-size signals, the input frame length can be arbitrary and the object ignores this property in the generated code. When you run this object in MATLAB, the object supports arbitrary input frame lengths for fixed-size and variable-size signals and this property does not affect the object behavior.

Data Types: `logical`

## Usage

### Syntax

```
channOut = channelizer(input)
```

### Description

`channOut = channelizer(input)` separates the broadband input signal into a number of narrow band signals contained in the columns of the channelizer output.

### Input Arguments

#### **input** — Data input

`vector` | `matrix`

Data input, specified as an  $L$ -by-1 vector or an  $L$ -by- $N$  matrix, with  $N > 1$ .

Under most conditions, the number of inputs rows  $L$  can be arbitrary and does not have to be a multiple of the number of frequency bands. See this table for details.

| Input Signal  | When you Run Object in MATLAB                | When you Generate Code Using MATLAB Coder   |
|---------------|--|---|
| Fixed-size    | Object supports arbitrary input frame length | Object supports arbitrary input frame length when you set <code>AllowArbitraryInputLength</code> to <code>true</code> while generating code |
| Variable-size | Object supports arbitrary input frame length | Object supports arbitrary input frame length  |

Variable-size signals change in frame length once you lock the object while the fixed-size signals remain constant. When the object does not support arbitrary frame length, the input frame length must be a multiple of the number of frequency bands.

Data Types: `single` | `double`

## Output Arguments

### channOut — Channelizer output

matrix | 3-D array

Channelizer output, returned as a matrix or a 3-D array.

If the input is an  $L$ -by-1 vector, then the output signal has an upper bound size of  $\text{ceil}(L/M)$ -by- $M$ , where  $M$  is the number of frequency bands. Each narrowband signal forms a column in the output.

If the input has more than one channel, that is, it is of the form  $L$ -by- $N$  with  $N > 1$ , then the output signal has an upper bound size of  $\text{ceil}(L/M)$ -by- $M$ -by- $N$ .

Data Types: `single` | `double`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Specific to dsp.Channelizer

|                                  |  |
|----------------------------------|--|
| <code>coeffs</code>              | Coefficients of prototype lowpass filter                     |
| <code>tf</code>                  | Return transfer function of overall prototype lowpass filter |
| <code>polyphase</code>           | Return polyphase matrix                                      |
| <code>freqz</code>               | Frequency response of filters in channelizer                 |
| <code>fvtool</code>              | Visualize the filters in the channelizer                     |
| <code>bandedgeFrequencies</code> | Compute the bandedge frequencies                             |
| <code>centerFrequencies</code>   | Compute center frequencies                                   |
| <code>getFilters</code>          | Return matrix of channelizer FIR filters                     |

## Common to All System Objects

|                      |  |
|----------------------|--|
| <code>step</code>    | Run System object algorithm  |
| <code>release</code> | Release resources and allow changes to System object property values and input characteristics |
| <code>reset</code>   | Reset internal states of System object   |

## Examples

### Channelize and Synthesize Sine Wave in MATLAB

Channelize and synthesize a sine wave signal with multiple frequencies using an  $M$ -channel filter bank.

The  $M$ -channel filter bank contains an analysis filter bank section and a synthesis filter bank section. The `dsp.Channelizer` object implements the analysis filter bank section. The `dsp.ChannelSynthesizer` object implements the synthesis filter bank section. These objects use an efficient polyphase structure to implement the filter bank. For more details, see **Polyphase Implementation** under **Algorithms** on the object reference pages.



## Initialization

Initialize the `dsp.Channelizer` and `dsp.ChannelSynthesizer` System objects. Each object is set up with 8 frequency bands, 8 polyphase branches in each filter, 12 coefficients per polyphase branch, and a stopband attenuation of 140 dB. Use a sine wave with multiple frequencies as the input signal. View the input spectrum and the output spectrum using a spectrum analyzer.

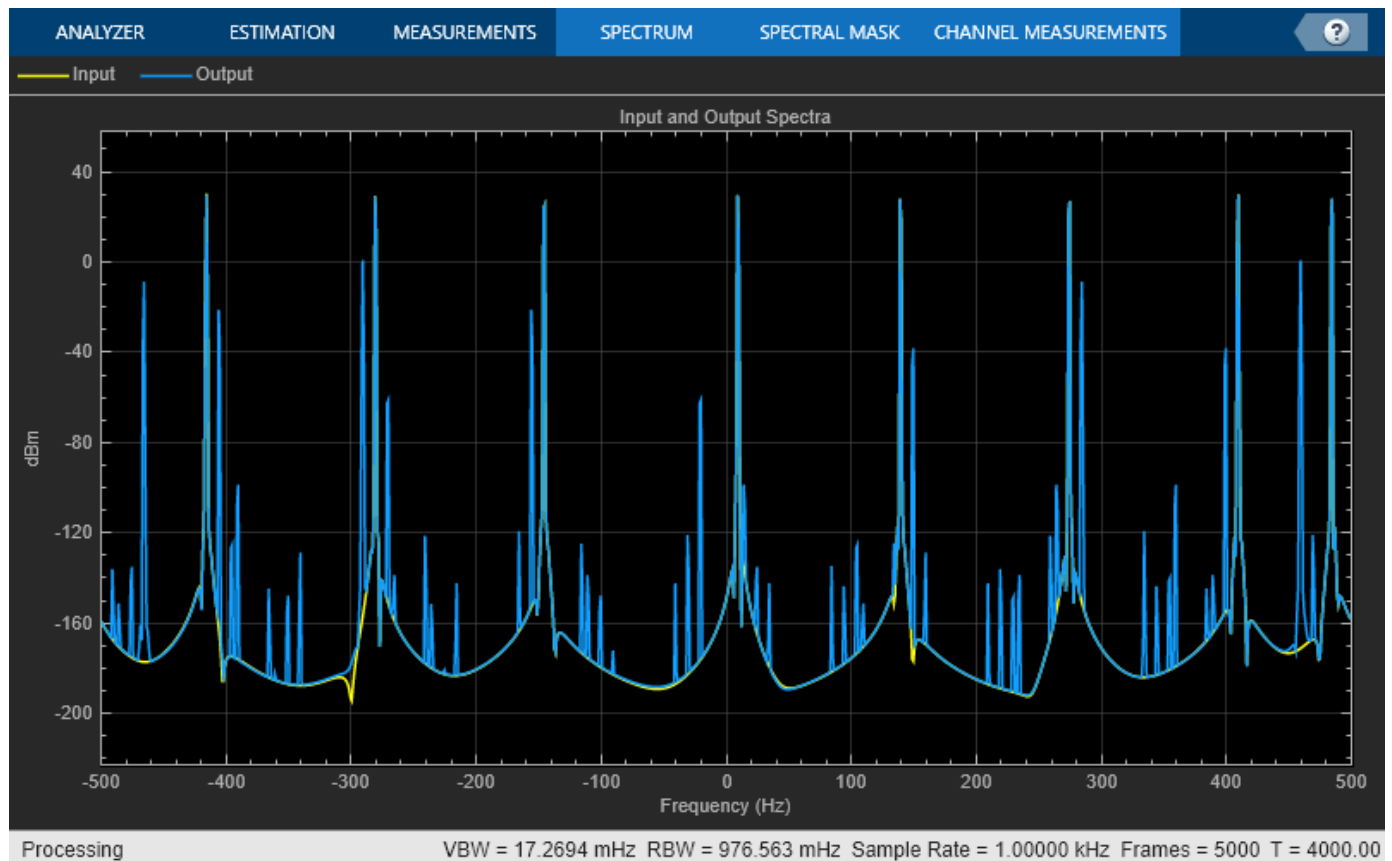
```
offsets = [-40, -30, -20, 10, 15, 25, 35, -15];
sinewave = dsp.SineWave('ComplexOutput', true, 'Frequency', ...
    offsets+(-375:125:500), 'SamplesPerFrame', 800);

channelizer = dsp.Channelizer('StopbandAttenuation', 140);
synthesizer = dsp.ChannelSynthesizer('StopbandAttenuation', 140);
spectrumAnalyzer = spectrumAnalyzer('ShowLegend', true, ...
    'SampleRate', sinewave.SampleRate, ...
    'ChannelNames', {'Input', 'Output'}, ...
    'Title', "Input and Output Spectra");
```

## Streaming

Use the channelizer to split the broadband input signal into multiple narrow bands. Then pass the multiple narrowband signals into the synthesizer, which merges these signals to form the broadband signal. Compare the spectra of the input and output signals. The input and output spectra match very closely.

```
for i = 1:5000
    x = sum(sinewave(), 2);
    y = channelizer(x);
    v = synthesizer(y);
    spectrumAnalyzer(x, v)
end
```



### Channelizer with Complex Coefficients

Create a `dsp.Channelizer` object and set the `LowpassCoefficients` property to a vector of complex coefficients.

#### Complex Coefficients

Using `firpm`, determine the coefficients of a Park-McClellan's optimal equiripple FIR filter of order 30, and frequency and amplitude characteristics described by  $F = [0 \ 0.2 \ 0.4 \ 1.0]$  and  $A = [1 \ 1 \ 0 \ 0]$  vectors, respectively.

Create a complex version of these coefficients by multiplying with a complex exponential. The resultant frequency response is that of a bandpass filter at the specified frequency, in this case 0.4.

```
blowpass = firpm(30,[0 .2 .4 1],[1 1 0 0]);
N = length(blowpass)-1;
Fc = 0.4;
j = complex(0,1);
bbandpass = blowpass.*exp(j*Fc*pi*(0:N));
```

#### Channelizer

Create a `dsp.Channelizer` object with 4 frequency bands and set the `Specification` property to `'Coefficients'`.

```
chann = dsp.Channelizer(4, 'Specification', "Coefficients");
```

Pass the complex coefficients to the channelizer. The prototype filter is a bandpass filter with a center frequency of 0.4. The modulated versions of this filter appear with respect to the prototype filter and are wrapped around the frequency range  $[-Fs Fs]$ .

```
chann.LowpassCoefficients = bbandpass
```

```
chann =
```

```
  dsp.Channelizer with properties:
```

```
    Main
```

```
      NumFrequencyBands: 4
```

```
      DecimationFactor: 4
```

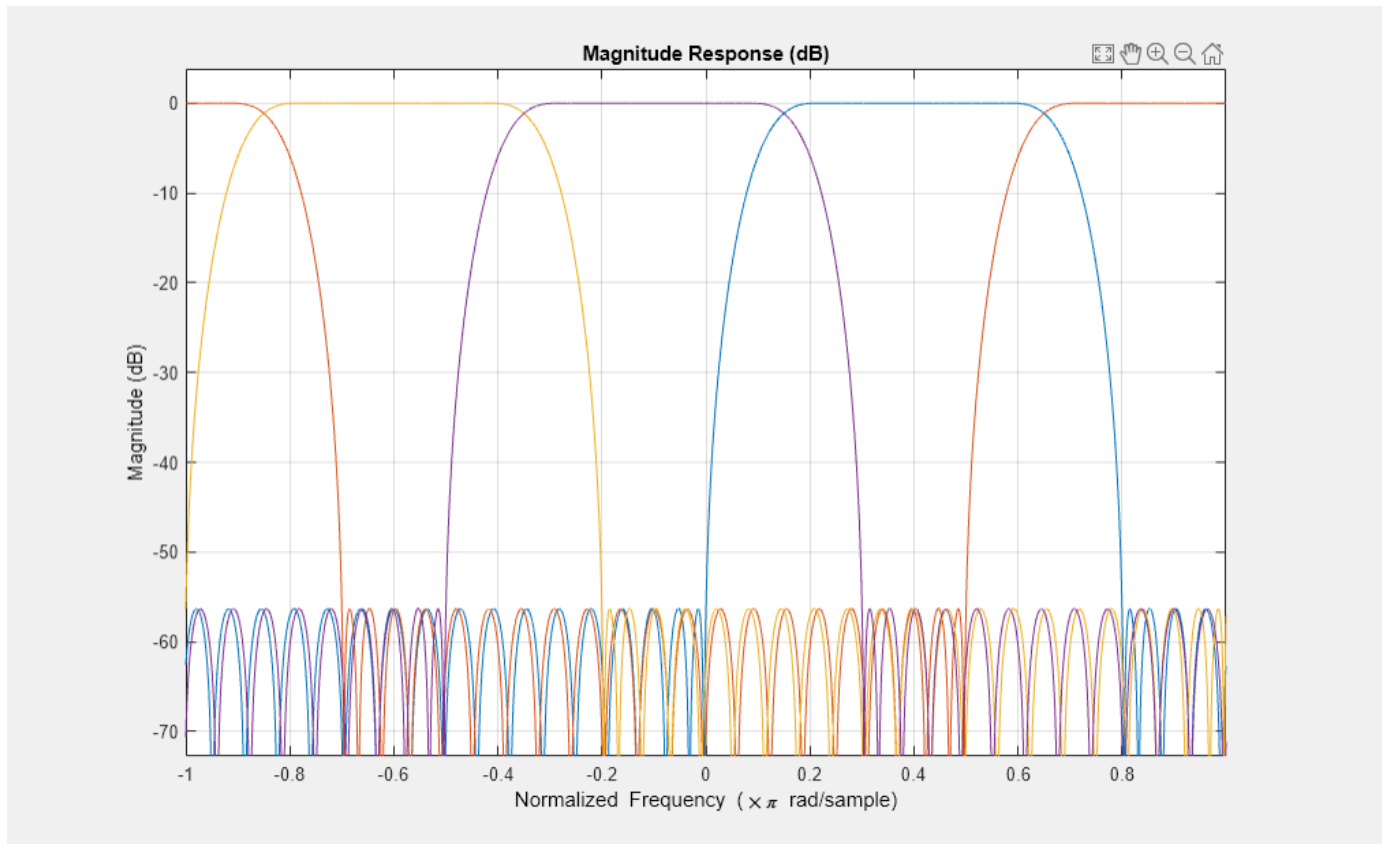
```
      Specification: 'Coefficients'
```

```
      LowpassCoefficients: [0.0019 + 0.0000i 0.0005 + 0.0016i 0.0011 - 0.0008i 0.0051 + 0.0037i -0
```

```
    Show all properties
```

Visualize the frequency response of the channelizer.

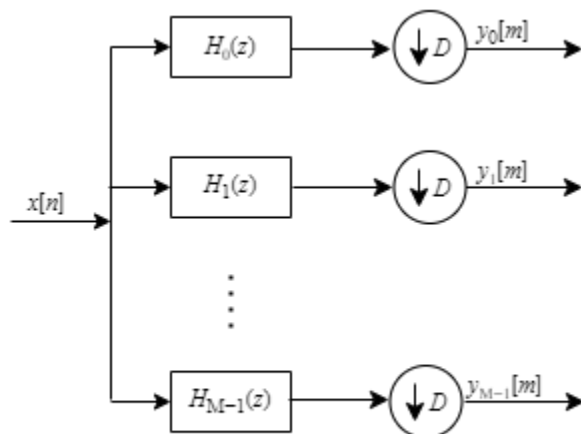
```
fvtool(chann)
```



## More About

### Analysis Filter Bank

The generic analysis filter bank consists of a series of parallel bandpass filters that split an input broadband signal,  $x[n]$ , into a series of narrow subbands. Each bandpass filter retains a different portion of the input signal. After the bandwidth is reduced by one of the bandpass filters, the signal is downsampled to a lower sampling rate commensurate with the new bandwidth.



Conceptual Diagram of Generic Analysis Filter Bank

## Prototype Lowpass Filter

To implement the analysis filter bank efficiently, the channelizer uses a prototype lowpass filter.

The prototype lowpass filter has an impulse response of  $h[n]$ , a normalized two-sided bandwidth of  $2\pi/M$ , and a cutoff frequency of  $\pi/M$ .  $M$  is the number of frequency bands, that is, the branches of the analysis filter bank. This value corresponds to the FFT length that the filter bank uses.  $M$  can be high on the order of 2048 or more. The stopband attenuation determines the minimum level of interference (aliasing) from one frequency band to another. The passband ripple must be small so that the input signal is not distorted in the passband.

The prototype lowpass filter corresponds to  $H_0(z)$  in the filter bank. The first branch of the filter bank contains  $H_0(z)$  followed by the decimator. The other  $M - 1$  branches contain filters that are modulated versions of the prototype filter. The modulation factor is given by the following equation:

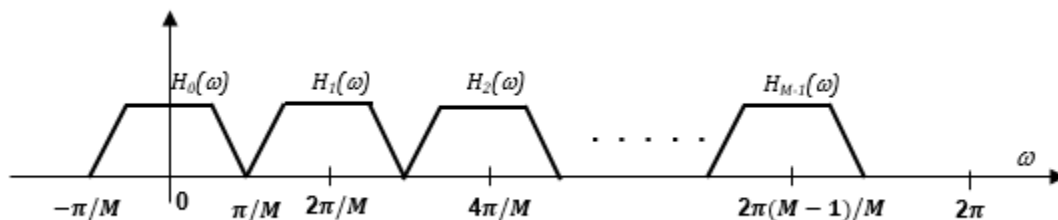
$$e^{-jw_k n}, \quad w_k = 2\pi k/M, \quad k = 0, 1, \dots, M - 1$$

## Using the Prototype Lowpass Filter

The transfer function of the modulated  $k$ th bandpass filter is given by:

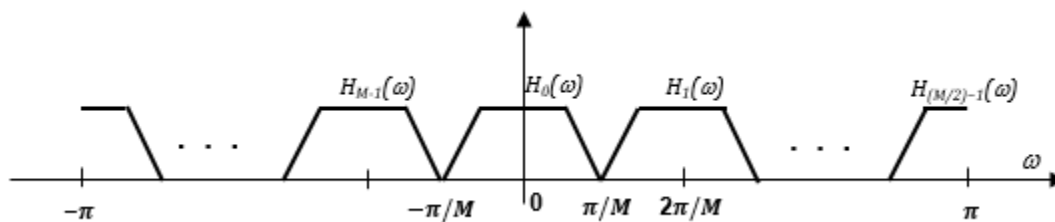
$$H_k(z) = H_0(z e^{-jw_k}), \quad w_k = 2\pi k/M, \quad k = 1, 2, \dots, M - 1$$

This figure shows the frequency response of  $M$  filters.



To obtain the frequency response characteristics of the filter  $H_k(z)$ , where  $k = 1, \dots, M-1$ , uniformly shift the frequency response of the prototype filter,  $H_0(z)$ , by multiples of  $2\pi/M$ . Each subband filter,  $H_k(z)$ ,  $\{k = 1, \dots, M - 1\}$ , is derived from the prototype filter.

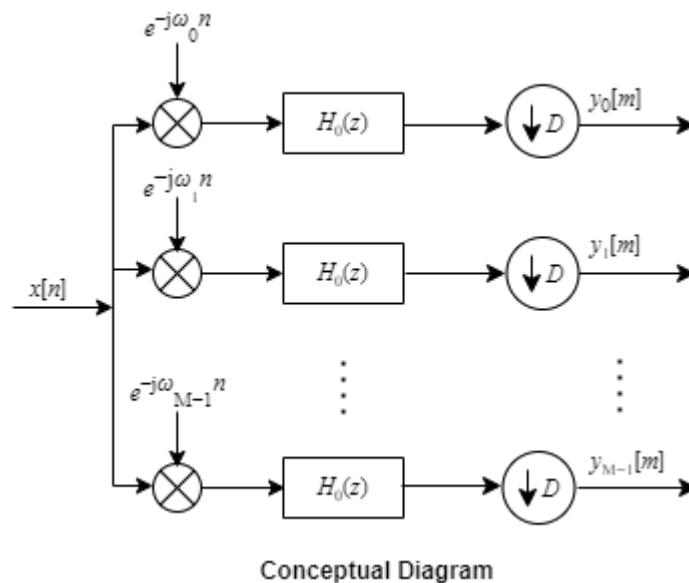
Following is an equivalent representation of the frequency response diagram with  $\omega$  ranging from  $[-\pi, \pi]$ .



### Shift Narrow Subbands to Baseband

The frequency components in the input signal,  $x[n]$ , are translated in frequency to baseband by multiplying  $x[n]$  with the complex exponentials,  $e^{-j\omega_k n}$ ,  $\omega_k = 2\pi k/M$ ,  $k = 1, 2, \dots, M-1$ , where  $\omega_k = 2\pi k/M$ , and  $k = 1, 2, \dots, M-1$ . The resulting product signals are passed through the lowpass filters,  $H_0(z)$ . The output of the lowpass filter is relatively narrow in bandwidth. Downsample the signal commensurate with the new bandwidth. Choose a decimation factor,  $D \leq M$ , where  $M$  is the number of branches of the analysis filter bank. When  $D < M$ , the channelizer is known as oversampled or non-maximally decimated channelizer.

The figure shows an analysis filter bank that uses the prototype lowpass filter.



$y_1[m], y_2[m], \dots, y_{M-1}[m]$  are narrow subband signals translated into baseband.

## Algorithms

### Polyphase Implementation

The analysis filter bank can be implemented efficiently using the polyphase structure. For more details on the analysis filter bank, see Analysis Filter Bank on page 4-198.

To derive the polyphase structure, start with the transfer function of the prototype lowpass filter:

$$H_0(z) = b_0 + b_1 z^{-1} + \dots + b_N z^{-N}$$

$N + 1$  is the length of the prototype filter.

You can rearrange this equation as follows:

$$H_0(z) = \begin{aligned} & (b_0 + b_M z^{-M} + b_{2M} z^{-2M} + \dots + b_{N-M+1} z^{-(N-M+1)}) + \\ & z^{-1} (b_1 + b_{M+1} z^{-M} + b_{2M+1} z^{-2M} + \dots + b_{N-M+2} z^{-(N-M+1)}) + \\ & \quad \vdots \\ & z^{-(M-1)} (b_{M-1} + b_{2M-1} z^{-M} + b_{3M-1} z^{-2M} + \dots + b_N z^{-(N-M+1)}) \end{aligned}$$

$M$  is the number of polyphase components.

You can write this equation as:

$$H_0(z) = E_0(z^M) + z^{-1} E_1(z^M) + \dots + z^{-(M-1)} E_{M-1}(z^M)$$

$E_0(z^M), E_1(z^M), \dots, E_{M-1}(z^M)$  are polyphase components of the prototype lowpass filter  $H_0(z)$ .

The other filters in the filter bank,  $H_k(z)$ , where  $k = 1, \dots, M-1$ , are modulated versions of this prototype filter.

You can write the transfer function of the  $k^{\text{th}}$  modulated bandpass filter as  $H_k(z) = H_0(z e^{-jw_k})$ .

Replacing  $z$  with  $z e^{jw_k}$ ,

$$H_k(z) = h_0 + h_1 e^{-jw_k} z^{-1} + h_2 e^{-j2w_k} z^{-2} \dots + h_N e^{-jNw_k} z^{-N}$$

$N + 1$  is the length of the  $k^{\text{th}}$  filter.

In polyphase form, the equation is as follows:

$$H_k(z) = \begin{bmatrix} 1 & e^{-jw_k} & e^{-j2w_k} & \dots & e^{-j(M-1)w_k} \end{bmatrix} \begin{bmatrix} E_0(z^M) \\ z^{-1} E_1(z^M) \\ \vdots \\ z^{-(M-1)} E_{M-1}(z^M) \end{bmatrix}$$

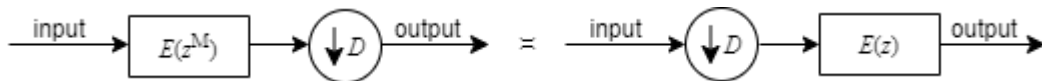
For all  $M$  channels in the filter bank, the transfer function  $H(z)$  is given by:

$$H(z) = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & e^{-jw_1} & e^{-j2w_1} & \dots & e^{-j(M-1)w_1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & e^{-jw_{M-1}} & e^{-j2w_{M-1}} & \dots & e^{-j(M-1)w_{M-1}} \end{bmatrix} \begin{bmatrix} E_0(z^M) \\ z^{-1} E_1(z^M) \\ \vdots \\ z^{-(M-1)} E_{M-1}(z^M) \end{bmatrix}$$

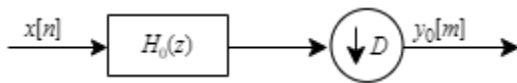
### Maximally decimated channelizer

When  $D = M$ , the channelizer is known as the maximally decimated channelizer or critically sampled channelizer.

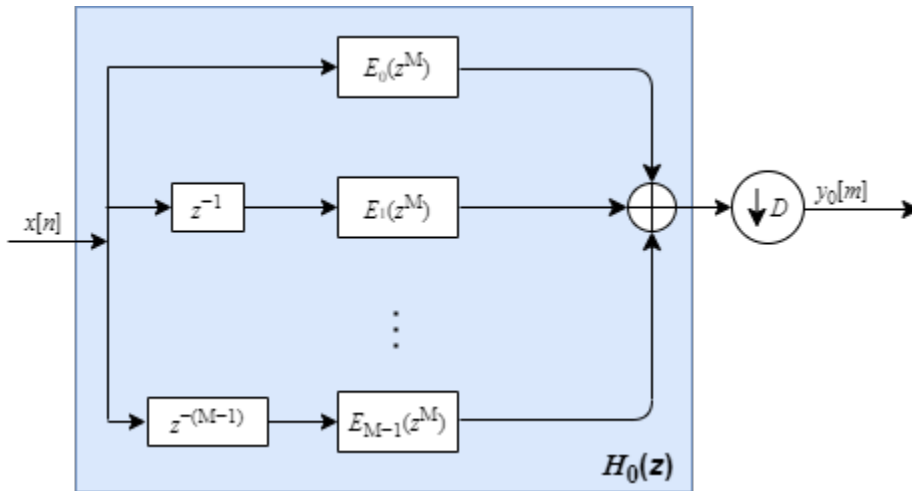
Here is the multirate noble identity for decimation, assuming that  $D = M$ .



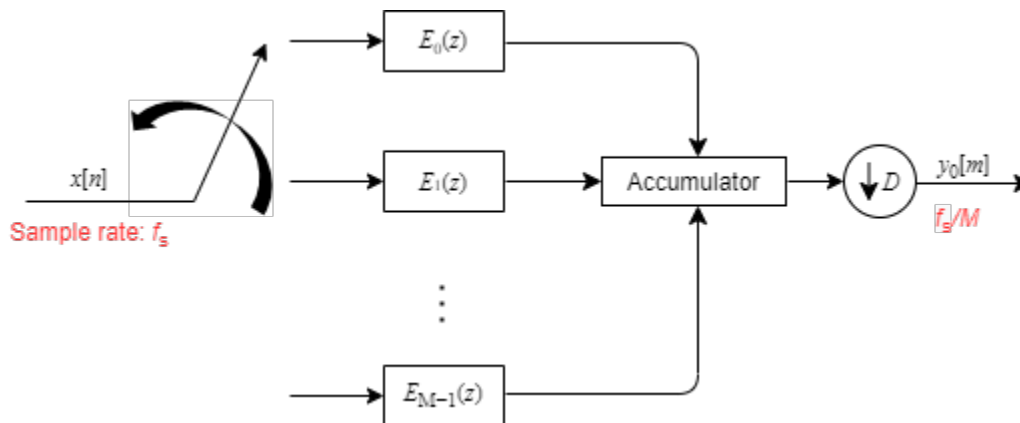
For example, consider the first branch of the filter bank that contains the lowpass filter.



Replace  $H_0(z)$  with its polyphase representation on page 4-200.



After applying the noble identity for decimation, you can replace the delays and the decimation factor with a commutator switch. The switch starts on the first branch 0 and moves in the counterclockwise direction as shown in the following diagram. The accumulator at the output receives the processed input samples from each branch of the polyphase structure and accumulates these processed samples until the switch goes to branch 0. When the switch goes to branch 0, the accumulator outputs the accumulated value.

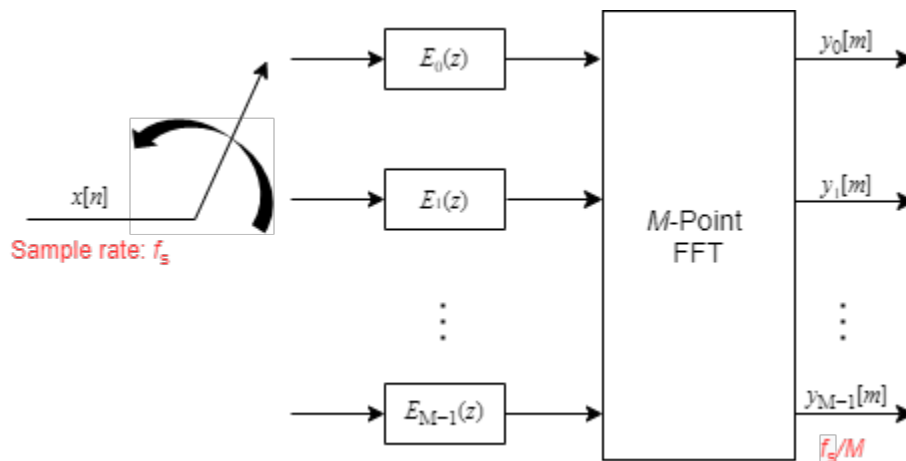


For all  $M$  channels in the filter bank, the transfer function  $H(z)$  is given by:

$$H(z) = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & e^{-jw_1} & e^{-j2w_1} & \dots & e^{-j(M-1)w_1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & e^{-jw_{M-1}} & e^{-j2w_{M-1}} & \dots & e^{-j(M-1)w_{M-1}} \end{bmatrix} \begin{bmatrix} E_0(z) \\ E_1(z) \\ \vdots \\ E_{M-1}(z) \end{bmatrix}$$

The matrix on the left is a discrete Fourier transform (DFT) matrix. With the DFT matrix, the efficient implementation of the lowpass prototype-based filter bank looks like this.





When the first input sample is delivered, the switch feeds this input to the branch 0 and the channelizer computes the first set of output values. As more input samples come in, the switch moves in the counterclockwise direction through branches  $M-1$ ,  $M-2$ , all the way up to branch 0, delivering one sample at a time to each branch. When the switch comes to branch 0, the channelizer outputs the next set of output values. This process continues as the data keeps coming in. Every time the switch comes to the first branch 0, the channelizer outputs  $y_0[m]$ ,  $y_1[m]$ , ...,  $y_{M-1}[m]$ . Each branch in the channelizer effectively outputs one sample for every  $M$  samples it receives. Hence, the sample rate at the output of the channelizer is  $f_s/M$ .

### Non-maximally decimated or oversampled channelizer

When  $D < M$ , the channelizer is known as the non-maximally decimated channelizer or oversampled channelizer. In this configuration, the output sample rate is different from the channel spacing. The non-maximally decimated channelizers offer increased design freedom, but at the expense of increasing computational cost.

If the ratio  $M/D$  equals an integer that is greater than 1 and is less than or equal to  $M-1$ , the channelizer is known as integer-oversampled channelizer. If the ratio  $M/D$  is not an integer, then the channelizer is known as rationally-oversampled channelizer.

In this configuration, when the first input sample is delivered, the switch feeds this input to branch 0 and the channelizer computes the first set of output values. As more input samples come in, the switch moves in the counterclockwise direction through branches  $D-1$ ,  $D-2$ , all the way up to branch 0, delivering one sample at a time to each branch. When the switch comes to branch 0, the channelizer outputs the next set of output values. This process continues as the data keeps coming in. Every time the switch comes to the first branch 0, the channelizer outputs  $y_0[m]$ ,  $y_1[m]$ , ...,  $y_{M-1}[m]$ .

As more data keeps coming in and the switch feeds these samples to the first  $D$  addresses, the formal contents of these addresses are shifted to the next set of  $D$  addresses, and this process of data shift continues every time there is a new set of  $D$  input samples.

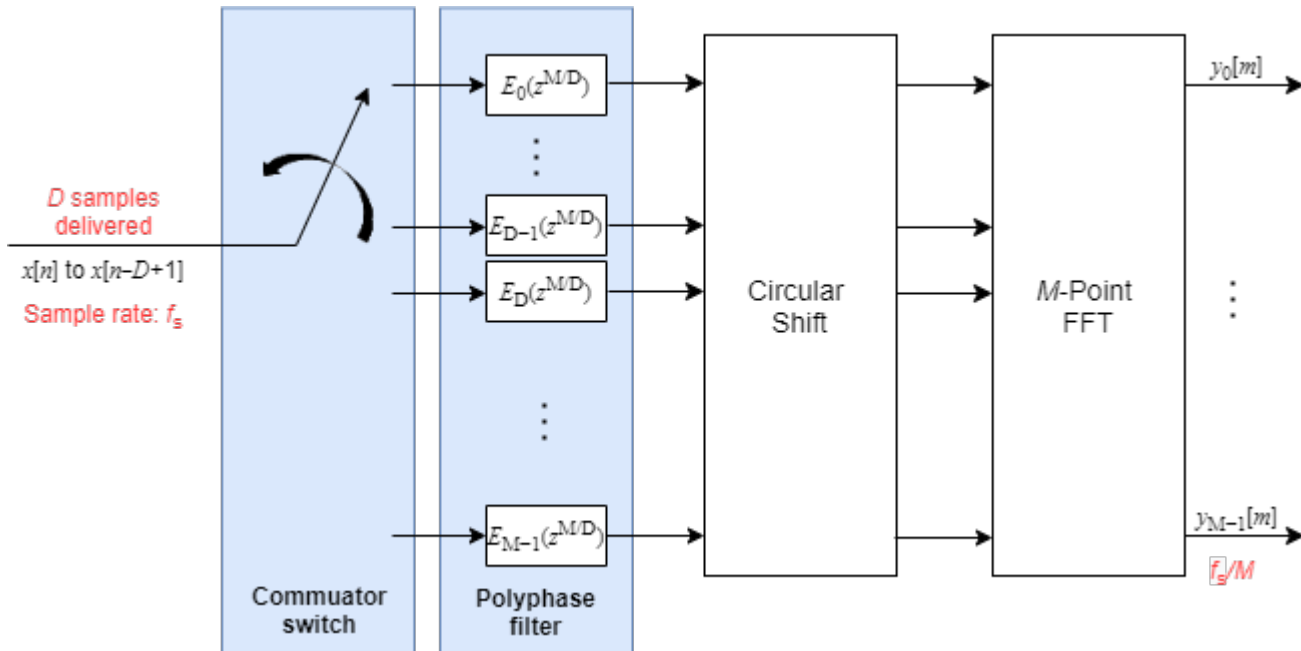
For every  $D$  input samples that are fed to the polyphase structure, the channelizer outputs  $M$  samples,  $y_0[m]$ ,  $y_1[m]$ , ...,  $y_{M-1}[m]$ . This process increases the output sample rate from  $f_s/M$  in the case of a maximally decimated channelizer, to  $f_s/D$  in the case of a non-maximally decimated channelizer.

For more details, see [2].

After each  $D$ -point data sequence is delivered to the partitioned  $M$ -stage polyphase filter, the outputs of the  $M$  stages are computed and conditioned for delivery to the  $M$ -point FFT. The data shifting

through the filter introduces frequency-dependent phase shift. To correct for this phase shift and alias all bands to DC, a circular shift buffer is inserted after the polyphase filters and before the  $M$ -point FFT.

With the commutator switch followed by  $M$ -stage polyphase filter, circular shift buffer, and a DFT matrix, the efficient implementation of the lowpass prototype-based filter bank looks like this.



## Version History

Introduced in R2016b

### R2022b: Support for arbitrary input frame length

Starting in R2022b, this object supports an input signal with an arbitrary frame length, so the input frame length does not have to be a multiple of the number of frequency bands.

When you generate code, to support arbitrary frame length for fixed-size signals, you must set the `AllowArbitraryInputLength` property to `true` while generating code.

## References

- [1] Harris, Fredric J, *Multirate Signal Processing for Communication Systems*, Prentice Hall PTR, 2004.
- [2] Harris, F.J., Chris Dick, and Michael Rice. "Digital Receivers and Transmitters Using Polyphase Filter Banks for Wireless Communications." *IEEE Transactions on Microwave Theory and Techniques*. 51, no. 4 (2003).

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

## See Also

### Functions

`coeffs` | `tf` | `polyphase` | `freqz` | `fvtool` | `bandedgeFrequencies` | `centerFrequencies` | `getFilters`

### Objects

`dsp.ChannelSynthesizer` | `dsp.FIRHalfbandDecimator` | `dsp.FIRHalfbandInterpolator` | `dsp.IIRHalfbandDecimator` | `dsp.DyadicAnalysisFilterBank`

### Blocks

Channelizer | Channel Synthesizer | Dyadic Analysis Filter Bank | Two-Channel Analysis Subband Filter

### Functions

`firpr2chfb`

# dsp.ChannelSynthesizer

**Package:** dsp

Polyphase FFT synthesis filter bank

## Description

The `dsp.ChannelSynthesizer` System object merges multiple narrowband signals into a broadband signal by using an FFT based synthesis filter bank. The filter bank uses a prototype lowpass filter and is implemented using a polyphase structure. You can specify the filter coefficients directly or through design parameters.

To merge multiple narrowband signals into a broadband signal:

- 1 Create the `dsp.ChannelSynthesizer` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
synthesizer = dsp.ChannelSynthesizer  
synthesizer = dsp.ChannelSynthesizer(Name,Value)
```

### Description

`synthesizer = dsp.ChannelSynthesizer` creates a synthesizer object, using the default properties.

`synthesizer = dsp.ChannelSynthesizer(Name,Value)` specifies additional properties using `Name,Value` pairs. Unspecified properties have default values.

Example: `synthesizer = dsp.ChannelSynthesizer('NumTapsPerBand',20,'StopbandAttenuation',140)`

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

### Specification — Filter design parameters or coefficients

"Number of taps per band and stopband attenuation" (default) | "Coefficients"

Filter design parameters or filter coefficients, specified as one of these options:

- "Number of taps per band and stopband attenuation" — Specify the filter design parameters through the `NumTapsPerBand` and `StopbandAttenuation` properties.
- "Coefficients" — Specify the filter coefficients directly using `LowpassCoefficients`.

### **NumTapsPerBand — Number of filter coefficients per frequency band**

12 (default) | positive integer

Number of filter coefficients each polyphase branch uses, specified as a positive integer. The number of polyphase branches matches the number of frequency bands. The total number of filter coefficients for the prototype lowpass filter is given by product of the number of frequency bands and `NumTapsPerBand`. For a given stopband attenuation, increasing the number of taps per band narrows the transition width of the filter. As a result, there is more usable bandwidth for each frequency band at the expense of increased computation.

#### **Dependencies**

This property applies when you set `Specification` to "Number of taps per band and stopband attenuation".

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **StopbandAttenuation — Stopband attenuation**

80 (default) | positive real scalar

Stopband attenuation of the lowpass filter, specified as a positive real scalar in dB. This value controls the maximum amount of aliasing from one frequency band to the next. Larger is the stopband attenuation, smaller is the passband ripple.

#### **Dependencies**

This property applies when you set `Specification` to "Number of taps per band and stopband attenuation".

Data Types: `single` | `double`

### **LowpassCoefficients — Coefficients of the prototype lowpass filter**

[1×49 double] (default) | row vector

Coefficients of the prototype lowpass filter, specified as a row vector. The default vector of coefficients is obtained using `rcosdesign(0.25,6,8,"sqrt")`. There must be at least one coefficient per frequency band. If the length of the lowpass filter is less than the number of frequency bands, the object zero-pads the coefficients.

If you specify complex coefficients, the object designs a prototype filter that is centered at a nonzero frequency, also known as a bandpass filter. The modulated versions of the prototype bandpass filter appear with respect to the prototype filter and are wrapped around the frequency range  $[-F_s/2, F_s/2]$ .

**Tunable:** Yes

### Dependencies

This property applies when you set `Specification` to "Coefficients".

Data Types: `single` | `double`

### Usage

### Syntax

```
synthOut = synthesizer(input)
```

### Description

`synthOut = synthesizer(input)` merges the narrowband input signals contained as columns in `input` into broadband signal, `synthOut`.

### Input Arguments

#### **input** — Narrowband signals

matrix | 3-D array

Narrowband signals, specified as a matrix or a 3-D array. Each narrowband signal is stored as a column in the input signal. The number of columns in `input` corresponds to the number of frequency bands of the filter bank. If `input` is three-dimensional, each matrix corresponds to a separate channel. If  $M$  is the number of frequency bands, and `input` is an  $L$ -by- $M$  matrix, then the output signal, `synthOut`, has dimensions  $L \times M$ -by-1. If `input` has more than one channel, that is, it has dimensions  $L$ -by- $M$ -by- $N$ , then `synthOut` has dimensions  $L \times M$ -by- $N$ .

This object also accepts variable-size inputs. That is, once the object is locked, you can change the size of each input channel. The number of channels cannot change.

Data Types: `single` | `double`

### Output Arguments

#### **synthOut** — Merged broadband signal

matrix | 3-D array

Merged broadband signal, returned as a matrix or a 3-D array. If  $M$  is the number of frequency bands, and `input` is an  $L$ -by- $M$  matrix, then the output signal, `synthOut`, has dimensions  $L \times M$ -by-1. If `input` has more than one channel, that is, it has dimensions  $L$ -by- $M$ -by- $N$ , then `synthOut` has dimensions  $L \times M$ -by- $N$ .

Data Types: `single` | `double`

### Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Specific to dsp.ChannelSynthesizer

coeffs      Coefficients of prototype lowpass filter  
 tf          Return transfer function of overall prototype lowpass filter  
 polyphase   Return polyphase matrix

## Common to All System Objects

step        Run System object algorithm  
 release    Release resources and allow changes to System object property values and input characteristics  
 reset      Reset internal states of System object

## Examples

### Channelize and Synthesize Sine Wave in MATLAB

Channelize and synthesize a sine wave signal with multiple frequencies using an  $M$ -channel filter bank.

The  $M$ -channel filter bank contains an analysis filter bank section and a synthesis filter bank section. The dsp.Channelizer object implements the analysis filter bank section. The dsp.ChannelSynthesizer object implements the synthesis filter bank section. These objects use an efficient polyphase structure to implement the filter bank. For more details, see **Polyphase Implementation** under **Algorithms** on the object reference pages.

### Initialization

Initialize the dsp.Channelizer and dsp.ChannelSynthesizer System objects. Each object is set up with 8 frequency bands, 8 polyphase branches in each filter, 12 coefficients per polyphase branch, and a stopband attenuation of 140 dB. Use a sine wave with multiple frequencies as the input signal. View the input spectrum and the output spectrum using a spectrum analyzer.

```
offsets = [-40, -30, -20, 10, 15, 25, 35, -15];
sinewave = dsp.SineWave('ComplexOutput', true, 'Frequency', ...
    offsets+(-375:125:500), 'SamplesPerFrame', 800);

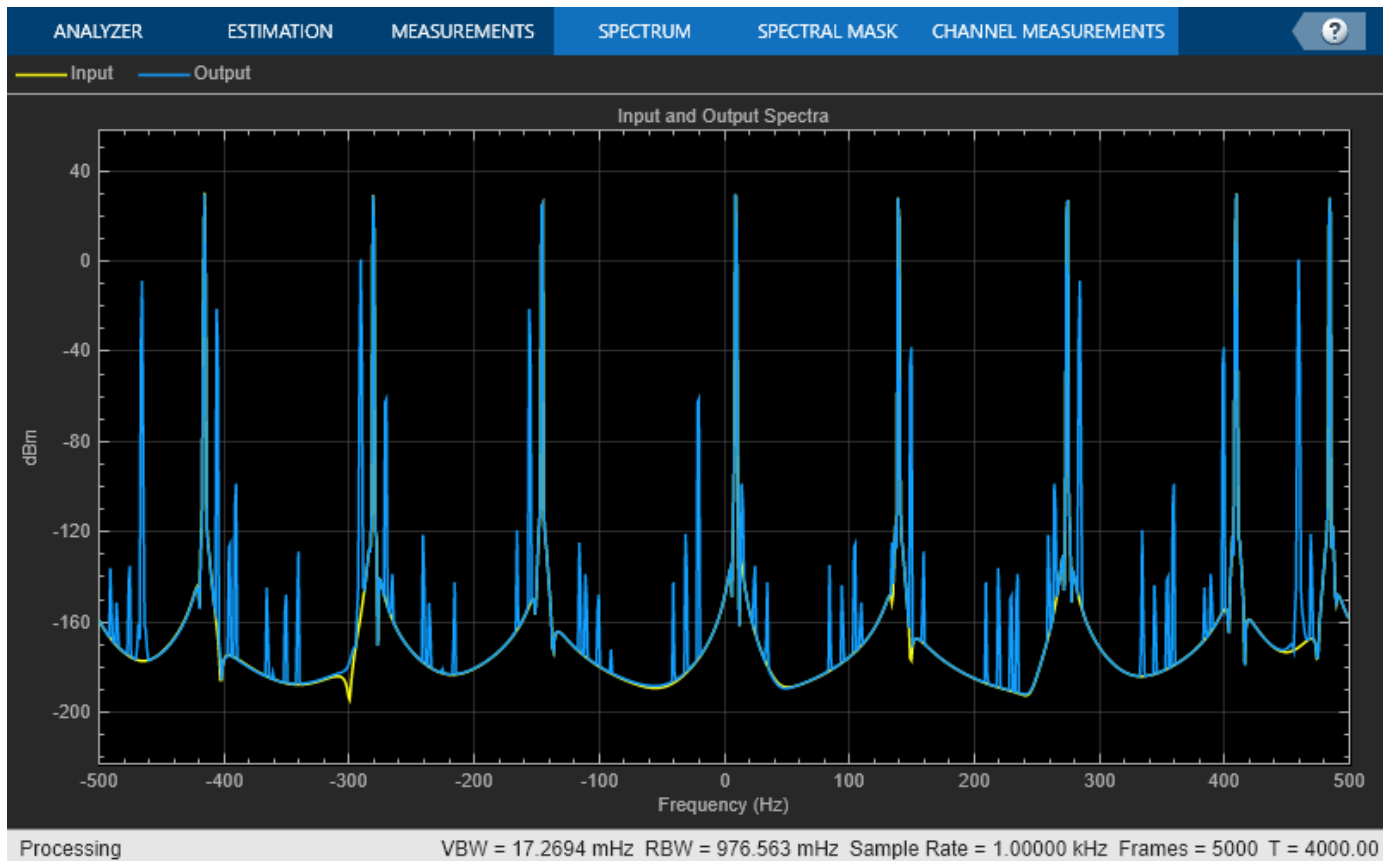
channelizer = dsp.Channelizer('StopbandAttenuation', 140);
synthesizer = dsp.ChannelSynthesizer('StopbandAttenuation', 140);
spectrumAnalyzer = spectrumAnalyzer('ShowLegend', true, ...
    'SampleRate', sinewave.SampleRate, ...
    'ChannelNames', {'Input', 'Output'}, ...
    'Title', "Input and Output Spectra");
```

### Streaming

Use the channelizer to split the broadband input signal into multiple narrow bands. Then pass the multiple narrowband signals into the synthesizer, which merges these signals to form the broadband signal. Compare the spectra of the input and output signals. The input and output spectra match very closely.

```
for i = 1:5000
    x = sum(sinewave(), 2);
    y = channelizer(x);
    v = synthesizer(y);
```

```
spectrumAnalyzer(x,v)
end
```

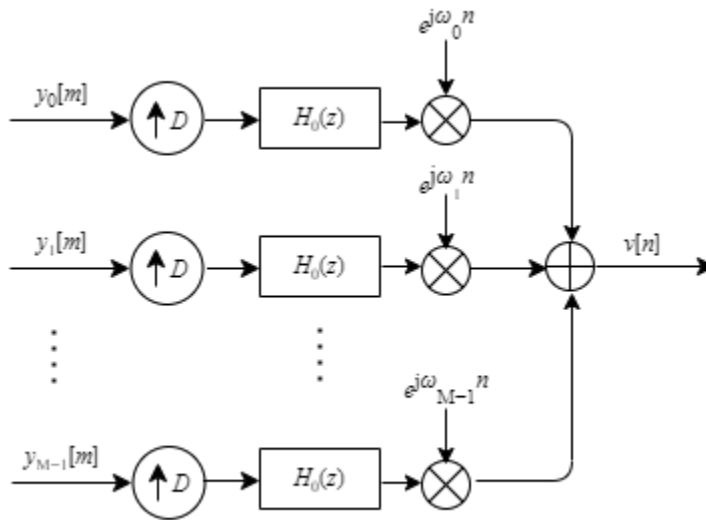


## More About

### Synthesis Filter Bank

The synthesis filter bank consists of a set of parallel bandpass filters that merge multiple input narrowband signals,  $y_0[m]$ ,  $y_1[m]$ , ...,  $y_{M-1}[m]$  into a single broadband signal,  $v[n]$ . The input narrowband signals are in the baseband. Each narrowband signal is interpolated to a higher sampling rate by using the upsampler, and then filtered by the lowpass filter. A complex exponential that follows the lowpass filter centers the baseband signal around  $\omega_k$ .





Conceptual Diagram

### Prototype Lowpass Filter

To implement the synthesis filter bank efficiently, the synthesizer uses a prototype lowpass filter. This filter has an impulse response of  $h[n]$ , a normalized two-sided bandwidth of  $2\pi/M$ , and a cutoff frequency of  $\pi/M$ .  $M$  is the number of frequency bands, that is, the branches of the synthesis filter bank. This value corresponds to the FFT length that the filter bank uses.  $M$  can be high, in the order of 2048 or more. The stopband attenuation determines the minimum level of interference (aliasing) from one frequency band to another. The passband ripple must be small so that the input signal is not distorted in the passband.

The prototype lowpass filter models the first branch of the filter bank. The other  $M - 1$  branches are modeled by filters that are modulated versions of the prototype filter. The modulation factor is given by  $e^{jw_k n}$ ,  $w_k = 2\pi k/M$ ,  $k = 0, 1, \dots, M - 1$ .

The output of each bandpass filter forms a specific portion of the broadband signal. The output of all the branches are added to form the broadband signal,  $v[n]$ .

## Algorithms

### Polyphase Implementation

The synthesis filter bank can be implemented efficiently using the polyphase structure.

To derive the polyphase structure, start with the transfer function of the prototype lowpass filter.

$$H_0(z) = b_0 + b_1 z^{-1} + \dots + b_N z^{-N}$$

$N + 1$  is the length of the prototype filter.

You can rearrange this equation as follows:

$$H_0(z) = \begin{aligned} & (b_0 + b_M z^{-M} + b_{2M} z^{-2M} + \dots + b_{N-M+1} z^{-(N-M+1)}) + \\ & z^{-1} (b_1 + b_{M+1} z^{-M} + b_{2M+1} z^{-2M} + \dots + b_{N-M+2} z^{-(N-M+1)}) + \\ & \quad \vdots \\ & z^{-(M-1)} (b_{M-1} + b_{2M-1} z^{-M} + b_{3M-1} z^{-2M} + \dots + b_N z^{-(N-M+1)}) \end{aligned}$$

$M$  is the number of polyphase components.

You can write this equation as:

$$H_0(z) = E_0(z^M) + z^{-1} E_1(z^M) + \dots + z^{-(M-1)} E_{M-1}(z^M)$$

$E_0(z^M), E_1(z^M), \dots, E_{M-1}(z^M)$  are polyphase components of the prototype lowpass filter,  $H_0(z)$ .

The other filters in the filter bank,  $H_k(z)$ , where  $k = 1, \dots, M-1$ , are modulated versions of this prototype filter.

You can write the transfer function of the  $k$ th modulated bandpass filter as  $H_k(z) = H_0(z e^{jw_k})$ . Replacing  $z$  with  $z e^{jw_k}$ ,

$$H_k(z) = h_0 + h_1 e^{jw_k} z^{-1} + h_2 e^{j2w_k} z^{-2} \dots + h_N e^{jNw_k} z^{-N}$$

$N + 1$  is the length of the  $k$ th filter.

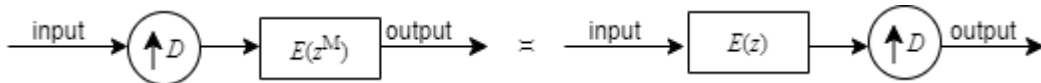
In polyphase form, the equation is as follows:

$$H_k(z) = \begin{bmatrix} 1 & e^{jw_k} & e^{j2w_k} & \dots & e^{j(M-1)w_k} \end{bmatrix} \begin{bmatrix} E_0(z^M) \\ z^{-1} E_1(z^M) \\ \vdots \\ z^{-(M-1)} E_{M-1}(z^M) \end{bmatrix}$$

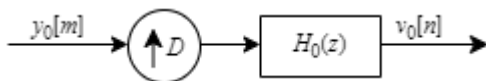
For all  $M$  channels in the filter bank, the MIMO transfer function,  $H(z)$ , is given by:

$$H(z) = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & e^{jw_1} & e^{j2w_1} & \dots & e^{j(M-1)w_1} \\ & & & \vdots & \\ 1 & e^{jw_{M-1}} & e^{j2w_{M-1}} & \dots & e^{j(M-1)w_{M-1}} \end{bmatrix} \begin{bmatrix} E_0(z^M) \\ z^{-1} E_1(z^M) \\ \vdots \\ z^{-(M-1)} E_{M-1}(z^M) \end{bmatrix}$$

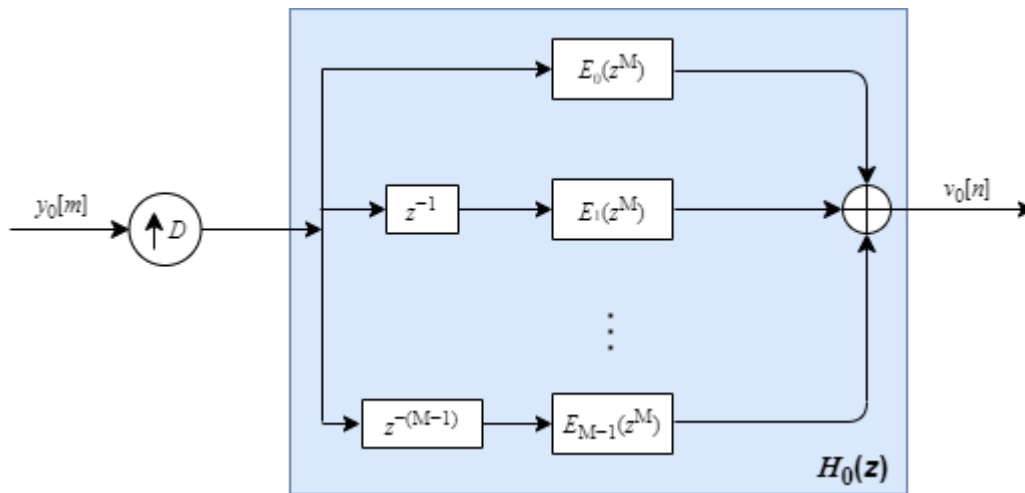
Here is the multirate noble identity for interpolation, assuming that  $D = M$ :



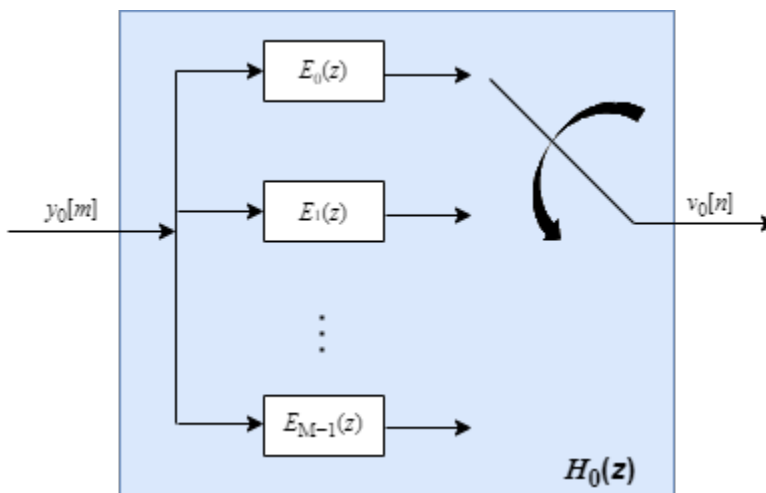
For illustration, consider the first branch of the filter bank that contains the lowpass filter.



Replace  $H_0(z)$  with its polyphase representation.



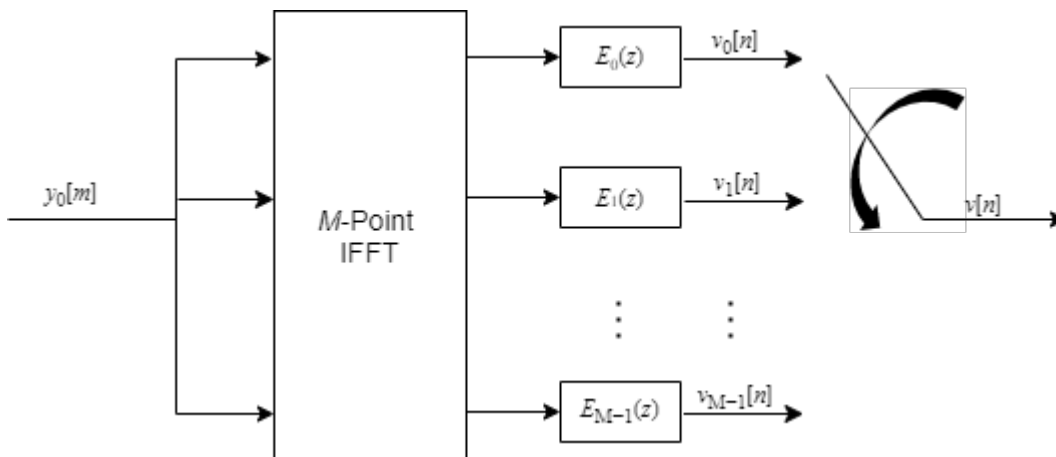
After applying the noble identity for interpolation, you can replace the delays, interpolation factor, and the adder with a commutator switch.



For all the  $M$  channels in the filter bank, the MIMO transfer function,  $H(z)$ , is given by:

$$H(z) = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & e^{jw_1} & e^{j2w_1} & \dots & e^{j(M-1)w_1} \\ & & & \vdots & \\ 1 & e^{jw_{M-1}} & e^{j2w_{M-1}} & \dots & e^{j(M-1)w_{M-1}} \end{bmatrix} \begin{bmatrix} E_0(z) \\ E_1(z) \\ \vdots \\ E_{M-1}(z) \end{bmatrix}$$

The matrix on the left is an IDFT matrix. With the IDFT matrix, the efficient implementation of the lowpass prototype based filter bank looks like the following.



## Version History

Introduced in R2016b

## References

- [1] Harris, Fredric J, *Multirate Signal Processing for Communication Systems*, Prentice Hall PTR, 2004.
- [2] Harris, F.J., Chris Dick, Michael Rice. "Digital Receivers and Transmitters Using Polyphase Filter Banks for Wireless Communications." *IEEE Transactions on microwave theory and techniques*. Vol. 51, Number 4, April 2003.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

## See Also

### Functions

`coeffs` | `tf` | `polyphase`

### Objects

`dsp.Channelizer` | `dsp.FIRHalfbandDecimator` | `dsp.FIRHalfbandInterpolator` | `dsp.IIRHalfbandInterpolator` | `dsp.DyadicSynthesisFilterBank`

### Blocks

Channel Synthesizer | Channelizer | Dyadic Analysis Filter Bank | Two-Channel Analysis Subband Filter

**Functions**

`firpr2chfb`

## **dsp.Chirp**

**Package:** dsp

Generate swept-frequency cosine (chirp) signal

### **Description**

The `Chirp` object generates a swept-frequency cosine (chirp) signal.

To generate the chirp signal:

- 1 Create the `dsp.Chirp` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

### **Creation**

#### **Syntax**

```
chirp = dsp.Chirp  
chirp = dsp.Chirp(Name,Value)
```

#### **Description**

`chirp = dsp.Chirp` returns a chirp signal, `chirp`, with unity amplitude.

`chirp = dsp.Chirp(Name,Value)` returns a chirp signal, `chirp`, with each specified property set to the specified value.

#### **Properties**

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

##### **Type — Frequency sweep type**

`Linear` (default) | `Swept cosine` | `Logarithmic` | `Quadratic`

Specify the frequency sweep type as `Swept cosine`, `Linear`, `Logarithmic`, or `Quadratic`. This property specifies how the output instantaneous frequency sweep varies over time.

##### **SweepDirection — Sweep direction**

`Unidirectional` (default) | `Bidirectional`

Specify the sweep direction as either `Unidirectional` or `Bidirectional`.

### **InitialFrequency — Initial frequency (hertz)**

1000 (default) | scalar greater than or equal to 0

When you set the `Type` property to `Linear`, `Quadratic`, or `Logarithmic`, this property specifies the initial instantaneous frequency in hertz of the output chirp signal. When you set the `Type` property to `Logarithmic`, the value of this property is one less than the actual initial frequency of the sweep. Also, when the sweep is logarithmic, the initial frequency must be less than the target frequency, specified by the `TargetFrequency` property.

**Tunable:** Yes

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **TargetFrequency — Target frequency (hertz)**

4000 (default) | scalar greater than or equal to 0

When you set the `Type` property to `Linear`, `Quadratic`, or `Logarithmic`, this property specifies the instantaneous frequency of the output signal in hertz at the target time. When you set the `Type` property to `Swept Cosine`, the target frequency is the instantaneous frequency of the output at half the target time. Also, when the sweep is logarithmic, the target frequency must be greater than the initial frequency, specified by the `InitialFrequency` property.

**Tunable:** Yes

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **TargetTime — Target time**

1 (default) | scalar greater than or equal to 0

When you set the `Type` property to `Linear`, `Quadratic`, or `Logarithmic`, this property specifies the target time in seconds at which the target frequency is reached. When you set the `Type` property to `Swept cosine`, this property specifies the time at which the sweep reaches  $2f_{tgt} - f_{init}$  Hz, where  $f_{tgt}$  is the `TargetFrequency` and  $f_{init}$  is the `InitialFrequency`. The target time should not be greater than the sweep time, specified by the `SweepTime` property.

**Tunable:** Yes

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **SweepTime — Sweep time**

1 (default) | scalar greater than or equal to 0

When you set the `SweepDirection` property to `Unidirectional`, the sweep time in seconds is the period of the output frequency sweep. When you set the `SweepDirection` property to `Bidirectional`, the sweep time is half the period of the output frequency sweep. The sweep time should be no less than the target time, specified by the `TargetTime`. This property must be a positive numeric scalar.

**Tunable:** Yes

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **InitialPhase — Initial phase**

0 (default) | scalar greater than or equal to 0

Specify initial phase of the output in radians at time  $t = 0$ .

**Tunable:** Yes

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**SampleRate — Sample rate**

8000 (default) | positive scalar

Specify the sampling rate of the output in hertz as a positive numeric scalar.

Data Types: `single` | `double` | `logical`

**SamplesPerFrame — Samples per output frame**

1 (default) | positive integer

Specify the number of samples to buffer into each output as a positive integer. The default value is 1.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**OutputDataType — Output data type**

`double` (default) | `single`

Specify the output data type as `double` or `single`. The default value is `double`.

## Usage

## Syntax

```
y = chirp()
```

### Description

`y = chirp()` returns a swept-frequency cosine output, `y`.

### Output Arguments

**y — Swept-frequency cosine output**

scalar | column vector

Swept-frequency cosine output, returned as a scalar or a column vector. The length of the output vector equals the value you specify in the `SamplesPerFrame` property.

Data Types: `double`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Common to All System Objects

`step`      Run System object algorithm

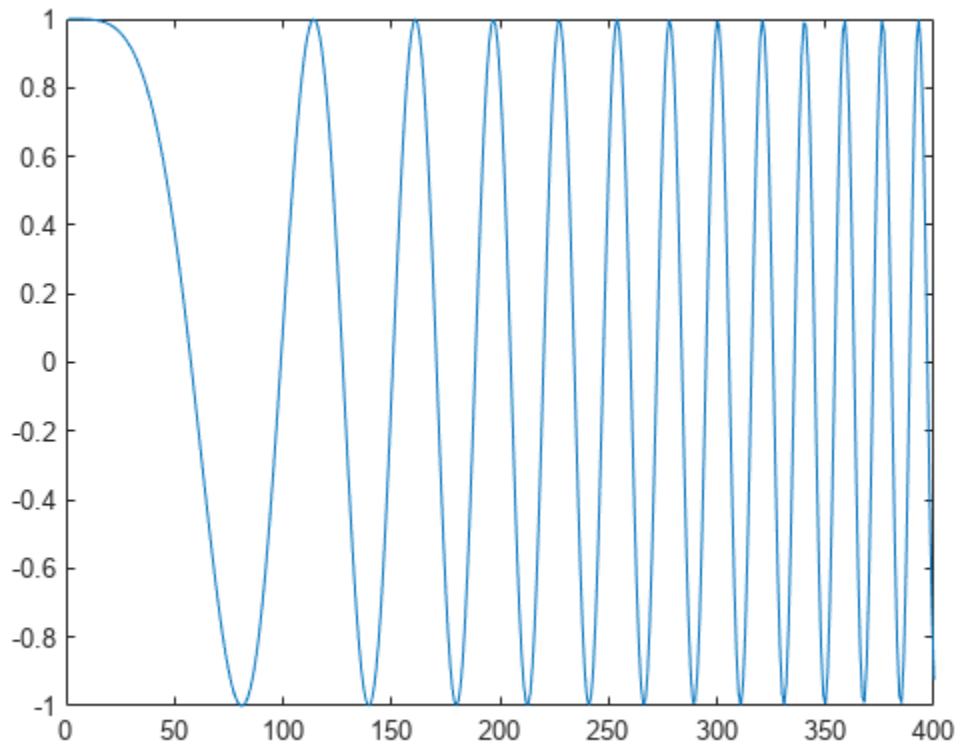


release Release resources and allow changes to System object property values and input characteristics  
reset Reset internal states of System object

## Examples

### Generate a Bidirectional Swept Chirp Signal

```
chirp = dsp.Chirp(...  
    'SweepDirection', 'Bidirectional', ...  
    'TargetFrequency', 25, ...  
    'InitialFrequency', 0, ...  
    'TargetTime', 1, ...  
    'SweepTime', 1, ...  
    'SamplesPerFrame', 400, ...  
    'SampleRate', 400);  
  
plot(chirp());
```



## Algorithms

This object implements the algorithm, inputs, and outputs described on the Chirp block reference page. The object properties correspond to the block parameters.

## **Version History**

**Introduced in R2012a**

### **See Also**

#### **Objects**

`dsp.SineWave`

# dsp.CICCompensationDecimator

**Package:** dsp

Compensate for CIC decimation filter using FIR decimator

## Description

You can compensate for the shortcomings of a CIC decimator, namely its passband droop and wide transition region, by following it with a compensation decimator. This System object lets you design and use such a filter.

To compensate for the shortcomings of a CIC filter using an FIR decimator:

- 1 Create the `dsp.CICCompensationDecimator` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```

ciccompdec = dsp.CICCompensationDecimator
ciccompdec = dsp.CICCompensationDecimator(decim)
ciccompdec = dsp.CICCompensationDecimator(cic)
ciccompdec = dsp.CICCompensationDecimator(cic,decim)
ciccompdec = dsp.CICCompensationDecimator( ____,Name,Value)

```

### Description

`ciccompdec = dsp.CICCompensationDecimator` returns a System object, `ciccompdec`, that applies an FIR decimator to each channel of an input signal. Using the properties of the object, the decimation filter can be designed to compensate for a preceding CIC filter.

`ciccompdec = dsp.CICCompensationDecimator(decim)` returns a CIC compensation decimator System object, with the `DecimationFactor` property set to `decim`.

`ciccompdec = dsp.CICCompensationDecimator(cic)` returns a CIC compensation decimator System object, with the `CICRateChangeFactor`, `CICNumSections`, and `CICDifferentialDelay` properties specified in the `dsp.CICDecimator` System object, `cic`.

`ciccompdec = dsp.CICCompensationDecimator(cic,decim)` returns a CIC compensation decimator System object, `ciccompdec`, with the `CICRateChangeFactor`, `CICNumSections`, and `CICDifferentialDelay` properties specified in the `dsp.CICDecimator` System object `cic`, and the `DecimationFactor` property set to `decim`.

`ciccompdec = dsp.CICCompensationDecimator( ____,Name,Value)` returns a CIC compensation decimator object with each specified property set to the specified value. Enclose each property name in quotes. You can use this syntax with any previous input argument combinations.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

### **CICDifferentialDelay** — Differential delay of the CIC filter being compensated

1 (default) | positive integer scalar

Specify the differential delay of the CIC filter being compensated as a positive integer scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **CICNumSections** — Number of sections of the CIC filter being compensated

2 (default) | positive integer scalar

Specify the number of sections of the CIC filter being compensated as a positive integer scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **CICRateChangeFactor** — Rate-change factor of the CIC filter being compensated

2 (default) | positive integer scalar

Specify the rate-change factor of the CIC filter being compensated as a positive integer scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **DecimationFactor** — Decimation factor of compensator

2 (default) | positive integer scalar

Specify the decimation factor of the compensator System object as a positive integer scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **DesignForMinimumOrder** — Design filter of minimum order or of specified order

`true` (default) | `false`

Specify whether to design a filter of minimum order or a filter of specified order as a logical scalar. The default is `true`, which corresponds to a filter of minimum order.

### **FilterOrder** — Order of decimation compensator filter

12 (default) | positive integer scalar

Specify the order of the decimation compensator filter as a positive integer scalar.

### **Dependencies**

This property applies only when you set the `DesignForMinimumOrder` property to `false`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **PassbandFrequency** — Passband edge frequency in hertz

100000 (default) | positive real scalar

Specify the passband edge frequency as a positive real scalar expressed in hertz. PassbandFrequency must be less than  $F_s/2$ , where  $F_s$  is the input sample rate.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

#### **PassbandRipple — Filter passband ripple in decibels**

0.1 (default) | positive real scalar

Specify the filter passband ripple as a positive real scalar expressed in decibels.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

#### **SampleRate — Input sample rate in hertz**

1200000 (default) | positive real scalar

Specify the input sample rate as a positive real scalar expressed in hertz.

Data Types: single | double

#### **StopbandAttenuation — Filter stopband attenuation in decibels**

60 (default) | positive real scalar

Specify the filter stopband attenuation as a positive real scalar expressed in decibels.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

#### **StopbandFrequency — Stopband edge frequency in hertz**

400000 (default) | positive real scalar

Specify the stopband edge frequency as a positive real scalar expressed in hertz. StopbandFrequency must be less than  $F_s/2$ , where  $F_s$  is the input sample rate.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### **Fixed-Point Properties**

#### **CoefficientsDataType — Word and fraction lengths of coefficients**

numericType(1,16) (default) | numericType object

Word and fraction lengths of coefficients, specified as a signed or unsigned numericType object. The default, numericType(1,16) corresponds to a signed numeric type object with 16-bit coefficients and a fraction length determined based on the coefficient values, to give the best possible precision.

This property is not tunable.

Word length of the output is same as the word length of the input. Fraction length of the output is computed such that the entire dynamic range of the output can be represented without overflow. For details on how the fraction length of the output is computed, see “Fixed-Point Precision Rules for Avoiding Overflow in FIR Filters”.

#### **RoundingMethod — Rounding method for output fixed-point operations**

'Floor' (default) | 'Ceiling' | 'Convergent' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'

Rounding method for output fixed-point operations, specified as a character vector. For more information on the rounding modes, see “Precision and Range”.

## Usage

### Syntax

```
y = ciccompdecim(x)
```

### Description

`y = ciccompdecim(x)` returns the filtered and downsampled values,  $y$ , of the input signal,  $x$ .

### Input Arguments

#### **x** — Data input

vector | matrix

Data input, specified as a vector or a matrix. The System object treats a  $K_i$ -by- $N$  input matrix as  $N$  independent channels, decimating each channel over the first dimension.

The number of input rows  $K_i$  can be arbitrary and does not have to be a multiple of the decimation factor.

This object supports variable-size input signals, that is, the frame length (number of rows) of the signal can change even when the object is locked. However, the number of channels (columns) must remain constant.

This object does not support complex unsigned fixed-point data.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fi

### Output Arguments

#### **y** — Filtered and downsampled signal

vector | matrix

Filtered and downsampled signal, returned as a vector or matrix.

When the input is of size  $K_i$ -by- $N$ , and  $K_i$  is not a multiple of the decimation factor  $M$ , the output signal has an upper bound size of  $\text{ceil}(K_i/M)$ -by- $N$ . If  $K_i$  is a multiple of the decimation factor, then the output is of size  $(K_i/M)$ -by- $N$ . The number of channels (columns) does not change.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fi

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

### Specific to `dsp.CICCompensationDecimator`

`freqz`            Frequency response of discrete-time filter System object

|             |   |
|-------------|---|
| fvtool      | Visualize frequency response of DSP filters                                   |
| info        | Information about filter System object  |
| cost        | Estimate cost of implementing filter System object                            |
| coeffs      | Returns the filter System object coefficients in a structure                  |
| polyphase   | Polyphase decomposition of multirate filter                                   |
| generatehdl | Generate HDL code for quantized DSP filter (requires Filter Design HDL Coder) |

## Common to All System Objects

|         |  |
|---------|--|
| step    | Run System object algorithm  |
| release | Release resources and allow changes to System object property values and input characteristics |
| reset   | Reset internal states of System object   |

## Examples

### Impulse and Frequency Response of CIC Compensation Decimator

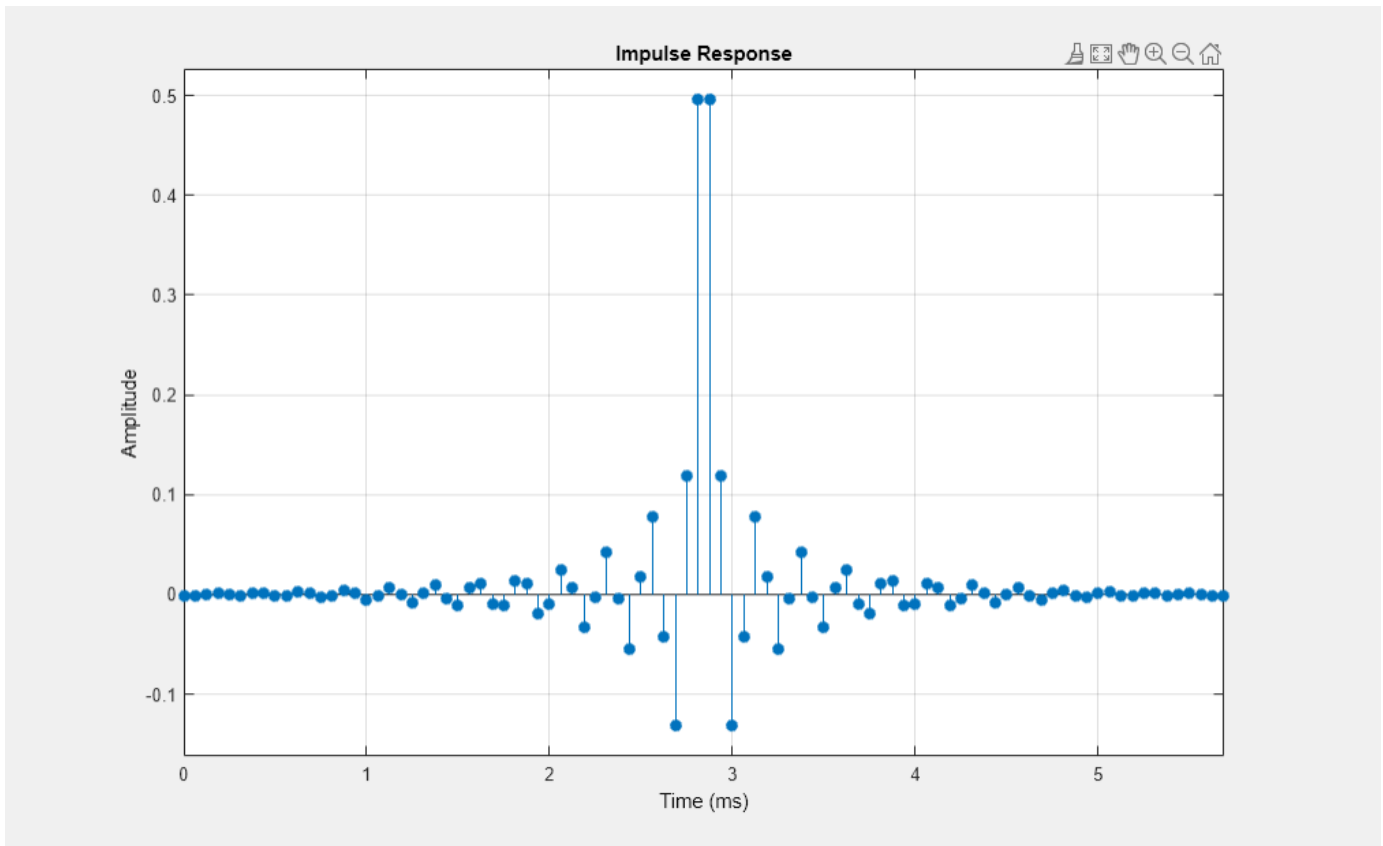
Design an CIC compensation decimator. Specify the decimation factor to be 2, passband frequency to be 4 kHz, stopband frequency to be 4.5 kHz, and the input sample rate to be 16 kHz.

```
fs = 16e3;
fPass = 4e3;
fStop = 4.5e3;
```

```
CICCompDecim = dsp.CICCompensationDecimator('DecimationFactor',2,...
    'PassbandFrequency',fPass, ...
    'StopbandFrequency',fStop,...
    'SampleRate',fs);
```

Plot the impulse response. The group delay of the filter is 45.5.

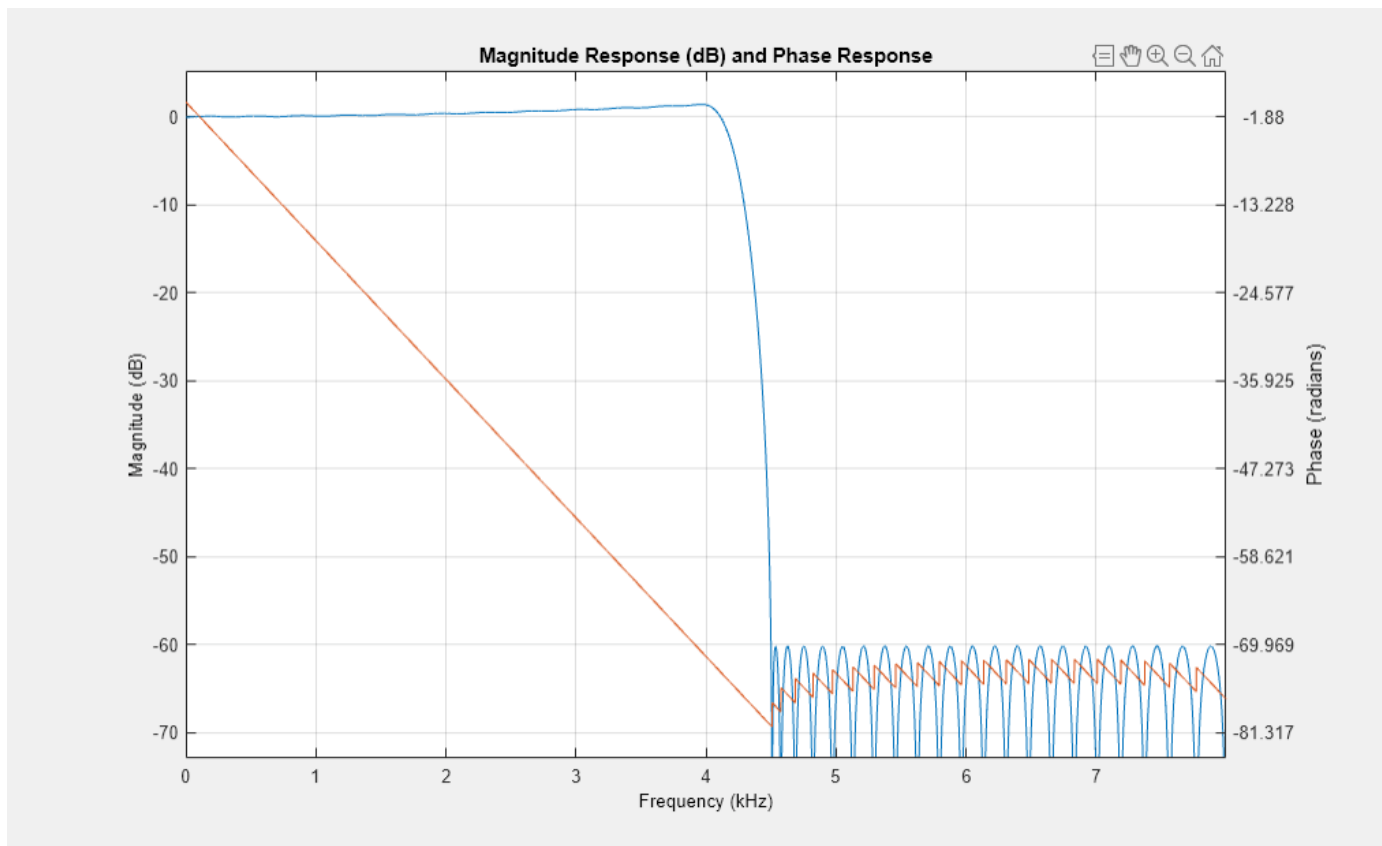
```
fvtool(CICCompDecim,'Analysis','impulse')
```



Plot the magnitude and Phase response.

```
fvtool(CICCompDecim,'Analysis','freq')
```





## Compensation Decimator Design

Design a compensation decimator for an existing CIC decimator having six sections and a decimation factor of 6.

```
CICDecim = dsp.CICDecimator('DecimationFactor',6, ...
    'NumSections',6);
```

Construct the compensation decimator. Specify a decimation factor of 2, an input sample rate of 16 kHz, a passband frequency of 4 kHz, and a stopband frequency of 4.5 kHz.

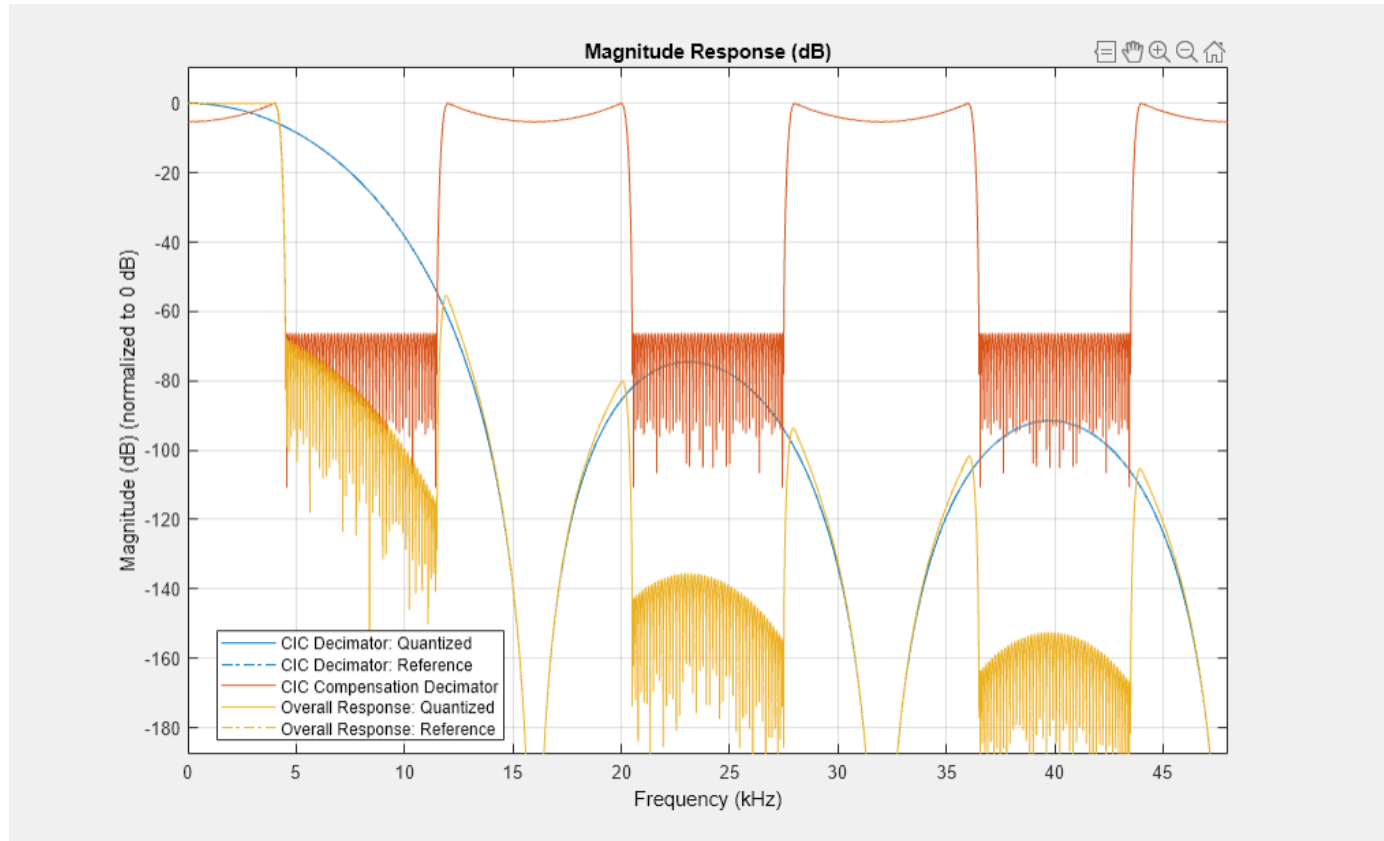
```
fs = 16e3;
fPass = 4e3;
fStop = 4.5e3;
```

```
CICCompDecim = dsp.CICCompensationDecimator(CICDecim, ...
    'DecimationFactor',2,'PassbandFrequency',fPass, ...
    'StopbandFrequency',fStop,'SampleRate',fs);
```

Visualize the frequency response of the cascade. Normalize all magnitude responses to 0 dB.

```
filtCasc = dsp.FilterCascade(CICDecim,CICCompDecim);
f = fvtool(CICDecim, CICCompDecim, filtCasc, ...
    'Fs', [fs*6 fs fs*6]);
```

```
f.NormalizeMagnitudeTo1 = 'on';
legend(f, 'CIC Decimator', 'CIC Compensation Decimator', ...
       'Overall Response');
```



Apply the design to a 1200-sample random input signal. Store the decimated output along the first dimension of the `y` array.

```
x = dsp.SignalSource(fi(rand(1200,1),1,16,15),'SamplesPerFrame',120);
```

```
y = fi(zeros(100,1),1,32,20);
```

```
for ind = 1:10
    x2 = CICDecim(x());
    y((ind-1)*10+1:ind*10,1) = CICCompDecim(x2);
end
```

## Algorithms

The response of a CIC filter is given by:

$$H_{cic}(\omega) = \left[ \frac{\sin\left(\frac{RD\omega}{2}\right)}{\sin\left(\frac{\omega}{2}\right)} \right]^N$$

$R$ ,  $D$ , and  $N$  are the rate change factor, the differential delay, and the number of sections in the CIC filter, respectively.

After decimation, the CIC response has the form:

$$H_{cic}(\omega) = \left[ \frac{\sin\left(\frac{D\omega}{2}\right)}{\sin\left(\frac{\omega}{2R}\right)} \right]^N$$

The normalized version of this last response is the one that the CIC compensator needs to compensate. Hence, the passband response of the CIC compensator should take the following form:

$$H_{ciccomp}(\omega) = \left[ RD \frac{\sin\left(\frac{\omega}{2R}\right)}{\sin\left(\frac{D\omega}{2}\right)} \right]^N \text{ for } \omega \leq \omega_p < \pi$$

where  $\omega_p$  is the passband frequency of the CIC compensation filter.

Notice that when  $\omega/2R \ll \pi$ , the previous equation for  $H_{ciccomp}(\omega)$  can be simplified using the fact that  $\sin(x) \cong x$ :

$$H_{ciccomp}(\omega) \approx \left[ \frac{\left(\frac{D\omega}{2}\right)}{\sin\left(\frac{D\omega}{2}\right)} \right]^N = \left[ \text{sinc}\left(\frac{D\omega}{2}\right) \right]^{-N} \text{ for } \omega \leq \omega_p < \pi$$

This previous equation is the inverse sinc approximation to the true inverse passband response of the CIC filter.

## Version History

### Introduced in R2014b

#### **R2023a: Support for arbitrary input frame length**

This object supports an input signal with an arbitrary frame length, so the input frame length does not have to be a multiple of the decimation factor.

#### **R2023a: Support for variable-size input signal**

The input signal to this object can be a variable-size signal, that is, the frame length (number of rows) of the signal can change even when the object is locked. However, the number of channels (columns) must remain constant.

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- See “System Objects in MATLAB Code Generation” (MATLAB Coder).
- This object supports code generation for ARM Cortex-M and ARM Cortex-A processors.
- In the code you generate from this object, the decimator output is a fixed-size signal if the input signal is fixed size and is a multiple of the decimation factor. If the input is not a multiple of the decimation factor or if the input is a variable-size signal, then the decimator output is of variable-size.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

This object supports HDL code generation with the Filter Design HDL Coder product. For workflows and limitations, see “Generate HDL Code for Filter System Objects” (Filter Design HDL Coder).

### **See Also**

#### **Functions**

`freqz` | `fvtool` | `info` | `cost` | `coeffs` | `polyphase` | `generatehdl`

#### **Objects**

`dsp.CICCompensationInterpolator` | `dsp.CICDecimator` | `dsp.CICInterpolator`

#### **Topics**

“Analysis Methods for Filter System Objects” on page 3-2

# dsp.CICCompensationInterpolator

**Package:** dsp

Compensate for CIC interpolation filter using FIR interpolator

## Description

You can compensate for the shortcomings of a CIC interpolator, namely its passband droop and wide transition region, by preceding it with a compensation interpolator. This System object lets you design and use such a filter.

To compensate for the shortcomings of a CIC filter using an FIR interpolator:

- 1 Create the `dsp.CICCompensationInterpolator` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
ciccompint = dsp.CICCompensationInterpolator
ciccompint = dsp.CICCompensationInterpolator(interp)
ciccompint = dsp.CICCompensationInterpolator(cic)
ciccompint = dsp.CICCompensationInterpolator(cic,interp)
ciccompint = dsp.CICCompensationInterpolator( ___,Name,Value)
```

### Description

`ciccompint = dsp.CICCompensationInterpolator` returns a System object, `ciccompint`, that applies an FIR interpolator to each channel of an input signal. Using the properties of the object, the interpolation filter can be designed to compensate for a subsequent CIC filter.

`ciccompint = dsp.CICCompensationInterpolator(interp)` returns a CIC compensation interpolator System object, `ciccompint`, with the `InterpolationFactor` property set to `interp`.

`ciccompint = dsp.CICCompensationInterpolator(cic)` returns a CIC compensation interpolator System object, `ciccompint`, with the `CICRateChangeFactor`, `CICNumSections`, and `CICDifferentialDelay` properties specified in the `dsp.CICInterpolator` System object `cic`.

`ciccompint = dsp.CICCompensationInterpolator(cic,interp)` returns a CIC compensation interpolator System object, `ciccompint`, with the `CICRateChangeFactor`, `CICNumSections`, and `CICDifferentialDelay` properties specified in the `dsp.CICInterpolator` System object `cic`, and the `InterpolationFactor` property set to `interp`.

`ciccompint = dsp.CICCompensationInterpolator( ___,Name,Value)` returns a CIC compensation interpolator object with each specified property set to the specified value. Enclose each property name in quotes. You can use this syntax with any previous input argument combinations.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see *System Design in MATLAB Using System Objects*.

### **CICDifferentialDelay** — Differential delay of the CIC filter being compensated

1 (default) | positive integer scalar

Specify the differential delay of the CIC filter being compensated as a positive integer scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **CICNumSections** — Number of sections of the CIC filter being compensated

2 (default) | positive integer scalar

Specify the number of sections of the CIC filter being compensated as a positive integer scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **CICRateChangeFactor** — Rate-change factor of the CIC filter being compensated

2 (default) | positive integer scalar

Specify the rate-change factor of the CIC filter being compensated as a positive integer scalar. The default is 2.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **DesignForMinimumOrder** — Design filter of minimum order or of specified order

`true` (default) | `false`

Specify whether to design a filter of minimum order or a filter of specified order as a logical scalar. The default is `true`, which corresponds to a filter of minimum order.

### **FilterOrder** — Order of interpolation compensator filter

12 (default) | positive integer scalar

Specify the order of the interpolation compensator filter as a positive integer scalar.

#### **Dependencies**

This property applies only when you set the `DesignForMinimumOrder` property to `false`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **InterpolationFactor** — Interpolation factor of compensator

2 (default) | positive integer scalar

Specify the interpolation factor of the compensator System object as a positive integer scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **PassbandFrequency — Passband edge frequency in hertz**

1000000 (default) | positive real scalar

Specify the passband edge frequency as a positive real scalar expressed in hertz. PassbandFrequency must be less than  $F_s/2$ , where  $F_s$  is the output sample rate.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **PassbandRipple — Filter passband ripple in decibels**

0.1 (default) | positive real scalar

Specify the filter passband ripple as a positive real scalar expressed in decibels.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **SampleRate — Input sample rate in hertz**

6000000 (default) | positive real scalar

Specify the input sample rate as a positive real scalar expressed in hertz.

Data Types: `single` | `double`

### **StopbandAttenuation — Filter stopband attenuation in decibels**

60 (default) | positive real scalar

Specify the filter stopband attenuation as a positive real scalar expressed in decibels.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **StopbandFrequency — Stopband edge frequency in hertz**

4000000 (default) | positive real scalar

Specify the stopband edge frequency as a positive real scalar expressed in hertz. StopbandFrequency must be less than  $F_s/2$ , where  $F_s$  is the output sample rate.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

## **Fixed-Point Properties**

### **CoefficientsDataType — Word and fraction lengths of coefficients**

`numericType(1,16)` (default) | `numericType` object

Word and fraction lengths of coefficients, specified as a signed or unsigned `numericType` object. The default, `numericType(1,16)` corresponds to a signed numeric type object with 16-bit coefficients and a fraction length determined based on the coefficient values, to give the best possible precision.

This property is not tunable.

Word length of the output is same as the word length of the input. Fraction length of the output is computed such that the entire dynamic range of the output can be represented without overflow. For details on how the fraction length of the output is computed, see “Fixed-Point Precision Rules for Avoiding Overflow in FIR Filters”.

### **RoundingMethod — Rounding method for output fixed-point operations**

'Floor' (default) | 'Ceiling' | 'Convergent' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'

Rounding method for output fixed-point operations, specified as a character vector. For more information on the rounding modes, see “Precision and Range”.

## Usage

### Syntax

```
y = ciccompint(x)
```

### Description

`y = ciccompint(x)` outputs the upsampled and filtered values,  $y$ , of the input signal,  $x$ .

### Input Arguments

#### **x** — Data input

vector | matrix

Data input, specified as a vector or a matrix. The System object treats a  $K_i \times N$  input matrix as  $N$  independent channels, interpolating each channel over the first dimension.

This object does not support complex unsigned fixed-point data.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

### Output Arguments

#### **y** — Upsampled and filtered signal

vector | matrix

Upsampled and filtered signal, returned as a vector or matrix. For a  $K_i \times N$  input matrix, the result is a  $K_o \times N$  output matrix, where  $K_o = K_i \times L$  and  $L$  is the interpolation factor.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

### Specific to `dsp.CICCompensationInterpolator`

|                          |   |
|--------------------------|---|
| <code>freqz</code>       | Frequency response of discrete-time filter System object                      |
| <code>fvtool</code>      | Visualize frequency response of DSP filters                                   |
| <code>info</code>        | Information about filter System object  |
| <code>cost</code>        | Estimate cost of implementing filter System object                            |
| <code>coeffs</code>      | Returns the filter System object coefficients in a structure                  |
| <code>polyphase</code>   | Polyphase decomposition of multirate filter                                   |
| <code>generatehdl</code> | Generate HDL code for quantized DSP filter (requires Filter Design HDL Coder) |



## Common to All System Objects

step     Run System object algorithm  
release   Release resources and allow changes to System object property values and input characteristics  
reset     Reset internal states of System object

## Examples

### Impulse and Frequency Response of CIC Compensation Interpolator

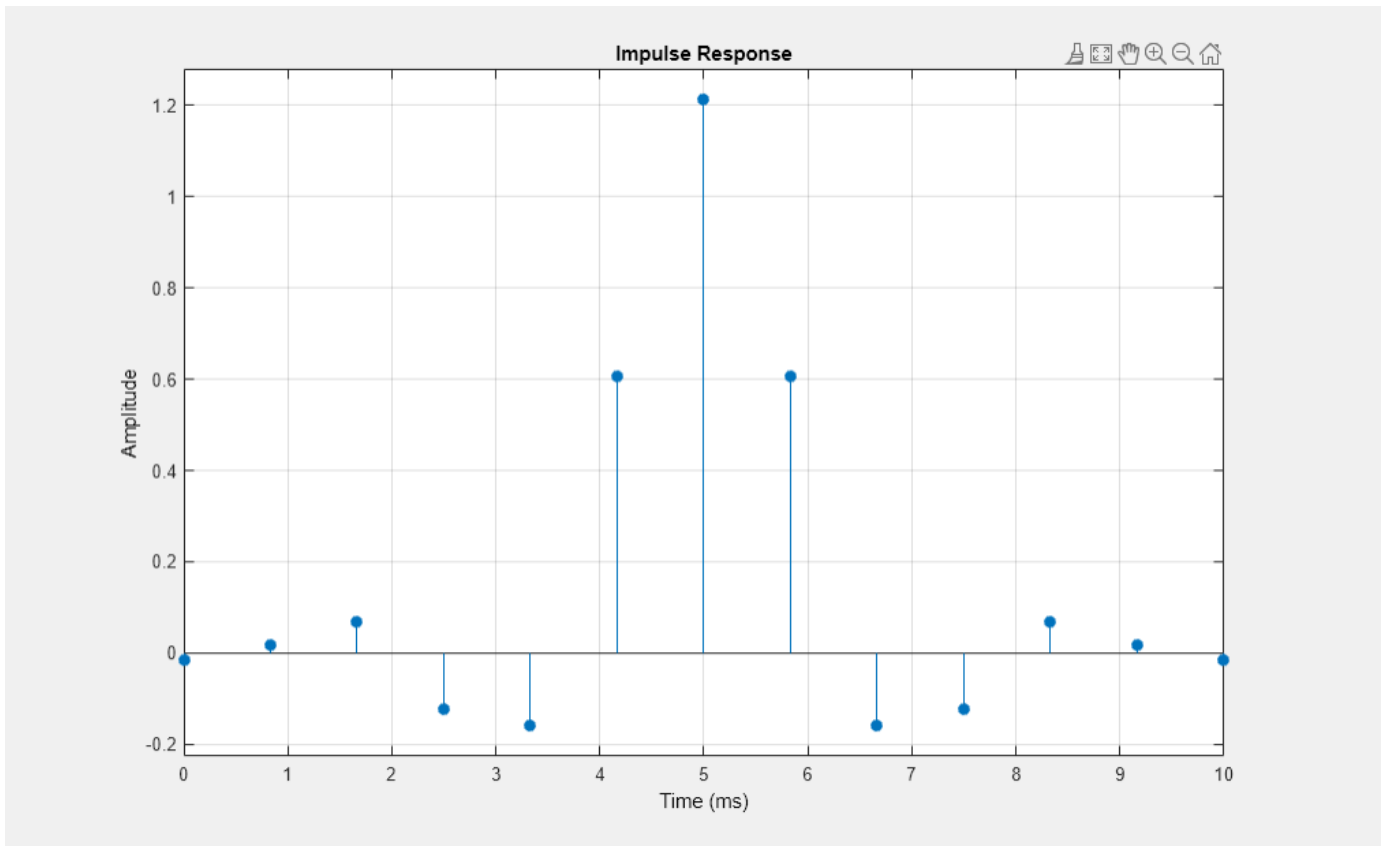
Design an CIC compensation interpolator. Specify the interpolation factor to be 2, passband frequency to be 200 Hz, stopband frequency to be 500 Hz, and the input sample rate to be 600 Hz.

```
fs = 600;  
fPass = 200;  
fStop = 500;
```

```
CICCompInterp = dsp.CICCompensationInterpolator(...  
    'InterpolationFactor',2,...  
    'PassbandFrequency',fPass,...  
    'StopbandFrequency',fStop,...  
    'SampleRate',fs);
```

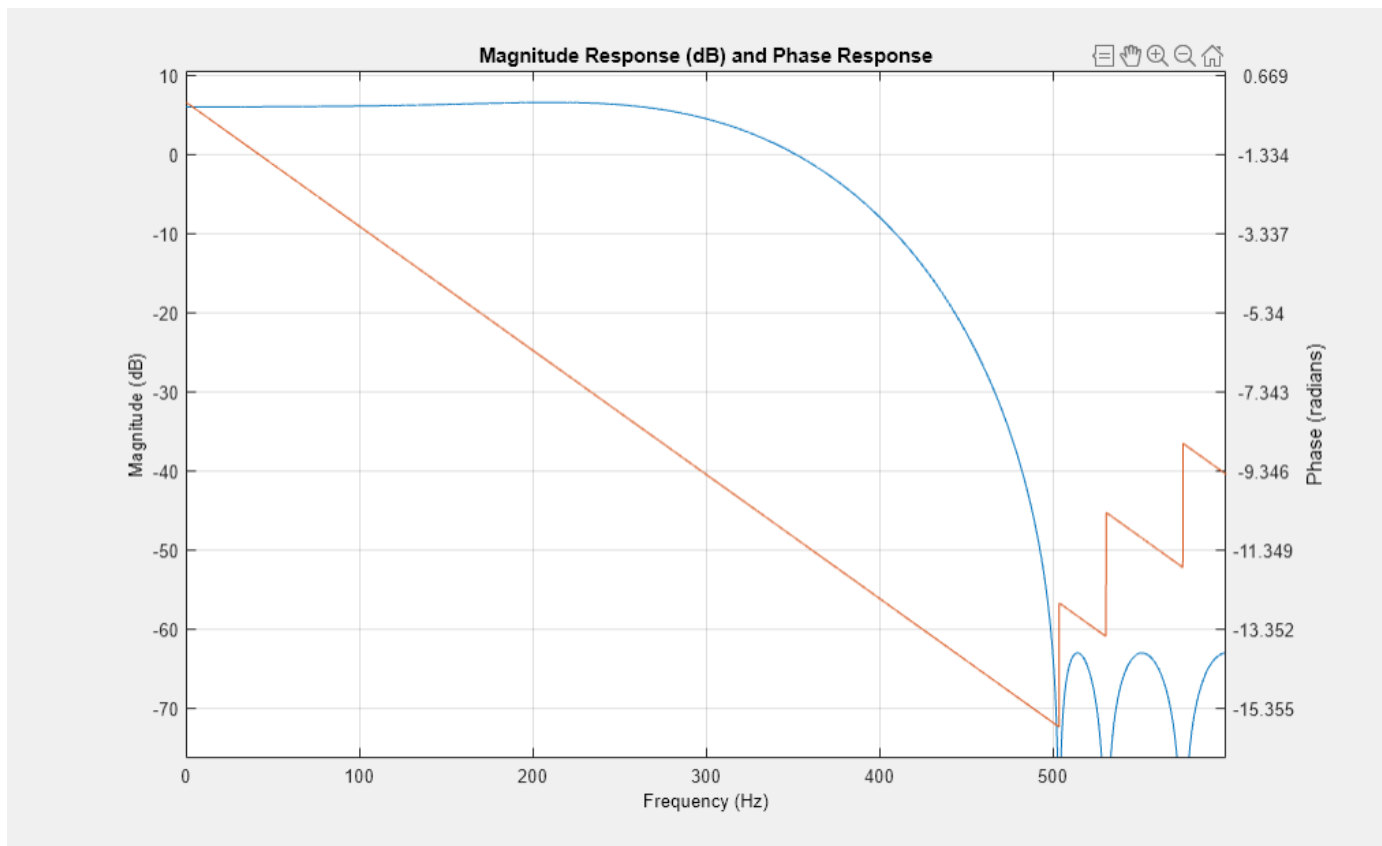
Plot the impulse response. The zeroth order coefficient is delayed 6 samples, which is equal to the group delay of the filter.

```
fvtool(CICCompInterp, 'Analysis', 'impulse')
```



Plot the magnitude and Phase response.

```
fvtool(CICCompInterp, 'Analysis', 'freq')
```



### Compensation Interpolator Design

Design a compensation interpolator for an existing CIC interpolator having six sections and an interpolation factor of 16.

```
CICInterp = dsp.CICInterpolator('InterpolationFactor',16, ...
    'NumSections',6);
```

Construct the compensation interpolator. Specify an interpolation factor of 2, an input sample rate of 600 Hz, a passband frequency of 100 Hz, and a stopband frequency of 250 Hz. Set the minimum attenuation of alias components in the stopband to be at least 80 dB.

```
fs = 600;
fPass = 100;
fStop = 250;
ast = 80;
```

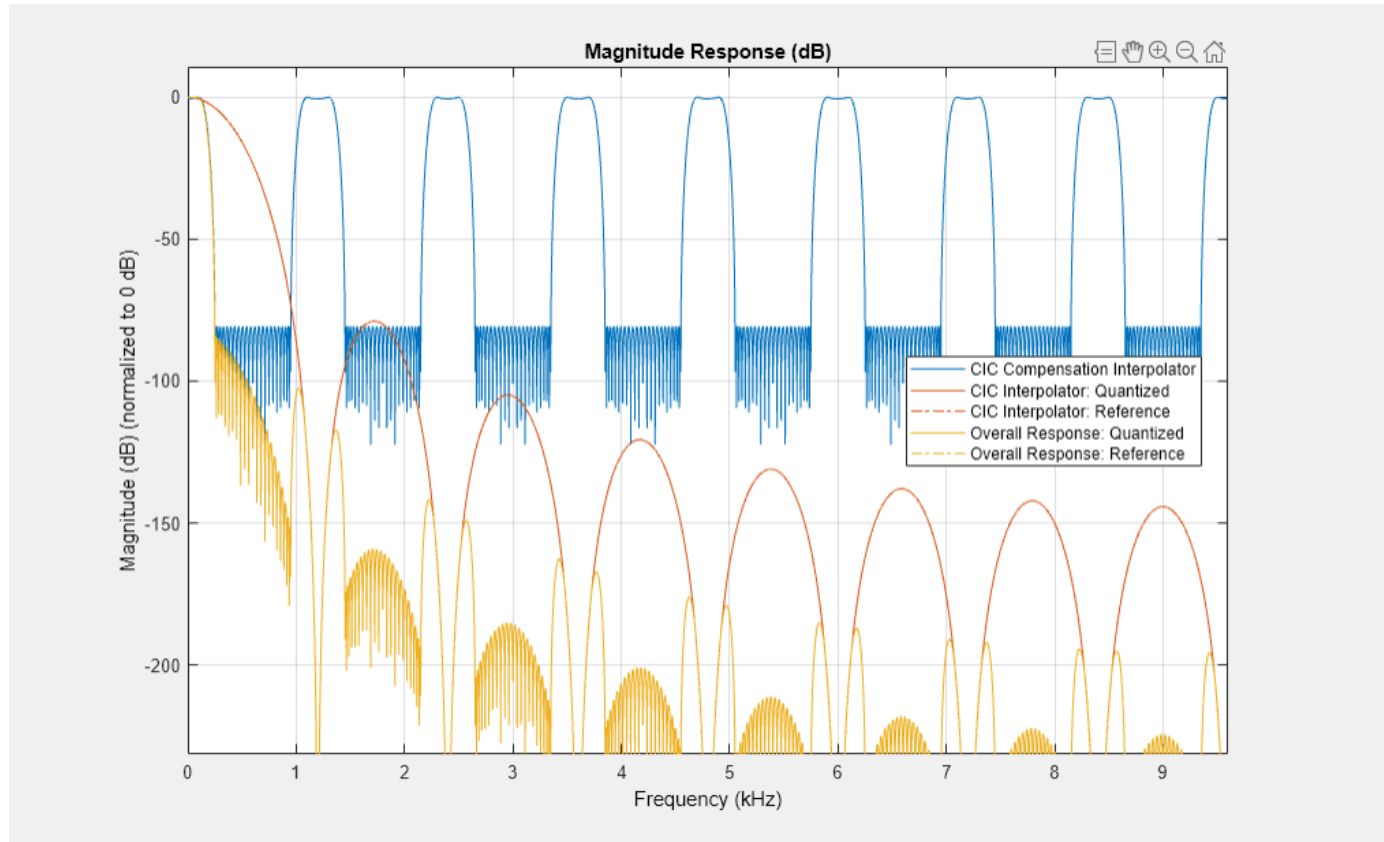
```
CICCompInterp = dsp.CICCompensationInterpolator(CICInterp, ...
    'InterpolationFactor',2,'PassbandFrequency',fPass, ...
    'StopbandFrequency',fStop,'StopbandAttenuation',ast, ...
    'SampleRate',fs);
```

Visualize the frequency response of the cascade. Normalize all magnitude responses to 0 dB.

```
FC = dsp.FilterCascade(CICCompInterp, CICInterp);
```

```
f = fvtool(CICCompInterp,CICInterp,FC, ...
    'Fs', [fs*2 fs*16*2 fs*16*2]);

f.NormalizeMagnitudeto1 = 'on';
legend(f,'CIC Compensation Interpolator','CIC Interpolator', ...
    'Overall Response');
```



Apply the design to a 1000-sample random input signal.

```
x = dsp.SignalSource(fi(rand(1000,1),1,16,15), 'SamplesPerFrame', 100);

y = fi(zeros(32000,1),1,32,20);
for ind = 1:10
    x2 = CICCompInterp(x());
    y(((ind-1)*3200)+1:ind*3200) = CICInterp(x2);
end
```

## Algorithms

The response of a CIC filter is given by:

$$H_{Cic}(\omega) = \left[ \frac{\sin\left(\frac{RD\omega}{2}\right)}{\sin\left(\frac{\omega}{2}\right)} \right]^N$$

$R$ ,  $D$ , and  $N$  are the rate change factor, the differential delay, and the number of sections in the CIC filter, respectively.

After decimation, the CIC response has the form:

$$H_{cic}(\omega) = \left[ \frac{\sin\left(\frac{D\omega}{2}\right)}{\sin\left(\frac{\omega}{2R}\right)} \right]^N$$

The normalized version of this last response is the one that the CIC compensator needs to compensate. Hence, the passband response of the CIC compensator should take the following form:

$$H_{ciccomp}(\omega) = \left[ RD \frac{\sin\left(\frac{\omega}{2R}\right)}{\sin\left(\frac{D\omega}{2}\right)} \right]^N \text{ for } \omega \leq \omega_p < \pi$$

where  $\omega_p$  is the passband frequency of the CIC compensation filter.

Notice that when  $\omega/2R \ll \pi$ , the previous equation for  $H_{ciccomp}(\omega)$  can be simplified using the fact that  $\sin(x) \cong x$ :

$$H_{ciccomp}(\omega) \approx \left[ \frac{\left(\frac{D\omega}{2}\right)}{\sin\left(\frac{D\omega}{2}\right)} \right]^N = \left[ \text{sinc}\left(\frac{D\omega}{2}\right) \right]^{-N} \text{ for } \omega \leq \omega_p < \pi$$

This previous equation is the inverse sinc approximation to the true inverse passband response of the CIC filter.

## Version History

Introduced in R2014b

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

This object supports code generation for ARM Cortex-M and ARM Cortex-A processors.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

This object supports HDL code generation with the Filter Design HDL Coder product. For workflows and limitations, see “Generate HDL Code for Filter System Objects” (Filter Design HDL Coder).

## See Also

### Functions

freqz | fvtool | info | cost | coeffs | polyphase | generatehdl

**Objects**

`dsp.CICCompensationDecimator` | `dsp.CICDecimator` | `dsp.CICInterpolator`

**Topics**

“Analysis Methods for Filter System Objects” on page 3-2

# dsp.CICDecimator

**Package:** dsp

Decimate signal using cascaded integrator-comb (CIC) filter

## Description

The `dsp.CICDecimator` System object decimates an input signal using a cascaded integrator-comb (CIC) decimation filter. The CIC decimation filter structure consists of  $N$  sections of cascaded integrators, followed by a rate change by a factor of  $R$ , followed by  $N$  sections of cascaded comb filters. For details, see “Algorithms” on page 4-253. The `NumSections` property specifies  $N$ , the number of sections in the CIC filter. The `DecimationFactor` property specifies  $R$ , the decimation factor. The `getFixedPointInfo` function returns the word lengths and fraction lengths of the fixed-point sections and the output for the `dsp.CICDecimator` System object. You can also generate HDL code for this System object using the `generatehdl` function.

---

**Note** This object requires a Fixed-Point Designer license.

---

To decimate a signal using a CIC filter:

- 1 Create the `dsp.CICDecimator` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
cicDecim = dsp.CICDecimator
cicDecim = dsp.CICDecimator(R,M,N)
cicDecim = dsp.CICDecimator(Name,Value)
```

### Description

`cicDecim = dsp.CICDecimator` creates a CIC decimation System object that applies a CIC decimation filter to the input signal.

`cicDecim = dsp.CICDecimator(R,M,N)` creates a CIC decimation object with the `DecimationFactor` property set to  $R$ , the `DifferentialDelay` property set to  $M$ , and the `NumSections` property set to  $N$ .

`cicDecim = dsp.CICDecimator(Name,Value)` creates a CIC decimation object with each specified property set to the specified value. Enclose each property name in single quotes. You can use this syntax with any previous input argument combination.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

### **DecimationFactor** — Decimation factor

2 (default) | positive integer

Factor by which the input signal is decimated, specified as a positive integer.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **DifferentialDelay** — Differential delay of filter comb sections

1 (default) | positive integer

Differential delay value used in each of the comb sections of the filter, specified as a positive integer. For details, see “Algorithms” on page 4-253. If the differential delay is a built-in integer data type, the decimation factor must be the same integer data type or `double`. For example, if the differential delay is an `int8`, then the decimation factor must be an `int8` or `double`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **NumSections** — Number of integrator and comb sections

2 (default) | positive integer

Number of integrator and comb sections of the CIC filter, specified as a positive integer. This number indicates the number of sections in either the comb part or the integrator part of the filter. The total number of sections in the CIC filter is twice the number of sections given by this property.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **FixedPointDataType** — Fixed-point property designations

Full precision (default) | Minimum section word lengths | Specify word lengths | Specify word and fraction lengths

Fixed-point property designations, specified as one of the following:

- **Full precision** - The word length and fraction length of the CIC filter sections and the object output operate in full precision.
- **Minimum section word lengths** - Specify the output word length through the `OutputWordLength` property. The object determines the filter section data type and the output fraction length that give the best possible precision. For details, see `getFixedPointInfo` and `cicDecimOut` argument.
- **Specify word lengths** - Specify the word lengths of the CIC filter sections and the object output through the `SectionWordLengths` and `OutputWordLength` properties. The object determines the corresponding fraction lengths to give the best possible precision. For details, see `getFixedPointInfo` and the `cicDecimOut` argument.
- **Specify word and fraction lengths** - Specify the word length and fraction length of the CIC filter sections and the object output through the `SectionWordLengths`, `SectionFractionLengths`, `OutputWordLength`, and `OutputFractionLength` properties.



**SectionWordLengths — Fixed-point word lengths for each filter section**

[16 16 16 16] (default) | scalar | vector

Fixed-point word lengths to use for each filter section, specified as a scalar or a row vector of integers. The word length must be greater than or equal to 2. If you specify a scalar, the value applies to all the sections of the filter. If you specify a vector, the vector must be of length  $2 \times \text{NumSections}$ .

Example: 32

Example: [32 32 32 32]

**Dependencies**

This property applies when you set the `FixedPointDataType` property to 'Specify word lengths' or 'Specify word and fraction lengths'.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**SectionFractionLengths — Fixed-point fraction lengths for each filter section**

0 (default) | scalar | vector

Fixed-point fraction lengths to use for each filter section, specified as a scalar or a row vector of integers. The fraction length can be negative, 0, or positive. If you specify a scalar, the value applies to all the sections of the filter. If you specify a vector, the vector must be of length  $2 \times \text{NumSections}$ .

Example: -2

Example: [-2 0 5 8]

**Dependencies**

This property applies when you set the `FixedPointDataType` property to 'Specify word and fraction lengths'.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**OutputWordLength — Fixed-point word length for filter output**

32 (default) | scalar integer

Fixed-point word length to use for the filter output, specified as a scalar integer greater than or equal to 2.

**Dependencies**

This property applies when you set the `FixedPointDataType` property to 'Minimum section word lengths', 'Specify word lengths', or 'Specify word and fraction lengths'.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**OutputFractionLength — Fixed-point fraction length for filter output**

0 (default) | scalar integer

Fixed-point fraction length to use for the filter output, specified as a scalar integer.

**Dependencies**

This property applies when you set `FixedPointDataType` property to 'Specify word and fraction lengths'.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

## Usage

### Syntax

```
cicDecimOut = cicDecim(input)
```

### Description

`cicDecimOut = cicDecim(input)` decimates the input using a CIC decimator.

### Input Arguments

#### **input** — Data input

vector | matrix

Data input, specified as a vector or matrix. The number of rows in the input must be a multiple of the “DecimationFactor” on page 4-0 . If the input is of `single` or `double` data type, property settings related to the fixed-point data types are ignored.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

### Output Arguments

#### **cicDecimOut** — CIC decimator output

vector | matrix

Decimated output, returned as a vector or a matrix. The output frame size equals  $(1 / \text{DecimationFactor}) \times \text{input frame size}$ . The complexity of the output data matches that of the input data. If the input is `single` or `double`, the output data type matches the input data type.

If the input is of built-in integer data type or of fixed-point data type, the output word length and fraction length depend on the fixed-point data type setting you choose through the “FixedPointDataType” on page 4-0 property.

### Full precision

When the `FixedPointDataType` is set to 'Full precision', the following relationship applies:

$$WL_{\text{output}} = WL_{\text{input}} + \text{NumSect}$$

$$FL_{\text{output}} = FL_{\text{input}}$$

where,

- $WL_{\text{output}}$  -- Word length of the output data.
- $FL_{\text{output}}$  -- Fraction length of the output data.
- $WL_{\text{input}}$  -- Word length of the input data.
- $FL_{\text{input}}$  -- Fraction length of the input data.
- $\text{NumSect}$  -- Number of sections in the CIC filter specified through the “NumSections” on page 4-0 property.

The  $WL_{\text{input}}$  and  $FL_{\text{input}}$  are inherited from the data input you pass to the object algorithm. For built-in integer inputs, the fraction length is 0.

### Minimum section word lengths

When the `FixedPointDataType` property is set to 'Minimum section word lengths', the output word length is the value you specify in the “OutputWordLength” on page 4-0 property. The output fraction length,  $FL_{\text{output}}$ , is given by:

$$FL_{\text{output}} = WL_{\text{output}} - (WL_{\text{input}} - FL_{\text{input}} + \text{NumSect})$$

### Specify word and fraction lengths

When the `FixedPointDataType` property is set to 'Specify word and fraction lengths', the output word length and fraction length are the values you specify in the “OutputWordLength” on page 4-0 and “OutputFractionLength” on page 4-0 properties.

### Specify word lengths

When the `FixedPointDataType` property is set to 'Specify word lengths', the output word length is the value you specify in the `OutputWordLength` property. The output fraction length,  $FL_{\text{output}}$ , is given by:

$$FL_{\text{output}} = WL_{\text{output}} - (WL_{\text{input}} - FL_{\text{input}} + \text{NumSect})$$

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

### Specific to dsp.CICDecimator

|                                |   |
|--------------------------------|---|
| <code>generatehdl</code>       | Generate HDL code for quantized DSP filter (requires Filter Design HDL Coder) |
| <code>impz</code>              | Impulse response of discrete-time filter System object                        |
| <code>freqz</code>             | Frequency response of discrete-time filter System object                      |
| <code>phasez</code>            | Phase response of discrete-time filter System object (unwrapped)              |
| <code>fvtool</code>            | Visualize frequency response of DSP filters                                   |
| <code>gain</code>              | Gain of CIC filter System object  |
| <code>getFixedPointInfo</code> | Get fixed-point word and fraction lengths                                     |
| <code>info</code>              | Information about filter System object  |

### Common to All System Objects

|                      |  |
|----------------------|--|
| <code>step</code>    | Run System object algorithm  |
| <code>release</code> | Release resources and allow changes to System object property values and input characteristics |
| <code>reset</code>   | Reset internal states of System object   |

For a list of filter analysis methods this object supports, type `dsp.CICDecimator.helpFilterAnalysis` in the MATLAB command prompt. For the corresponding function reference pages, see “Analysis Methods for Filter System Objects” on page 3-2.

## Examples

### Decimate a Signal Using a CICDecimator Object

Create a `dsp.CICDecimator System` object™ with `DecimationFactor` set to 4. Decimate a signal from 44.1 kHz to 11.025 kHz.

```
cicdec = dsp.CICDecimator(4);  
cicdec.FixedPointDataType = 'Minimum section word lengths';  
cicdec.OutputWordLength = 16;
```

Create a fixed-point sinusoidal input signal of 1024 samples, with a sampling frequency of 44.1e3 Hz.

```
Fs = 44.1e3;  
% 0.0232 sec signal  
n = (0:1023)';  
x = fi(sin(2*pi*1e3/Fs*n), true, 16, 15);
```

Create a `dsp.SignalSource` object.

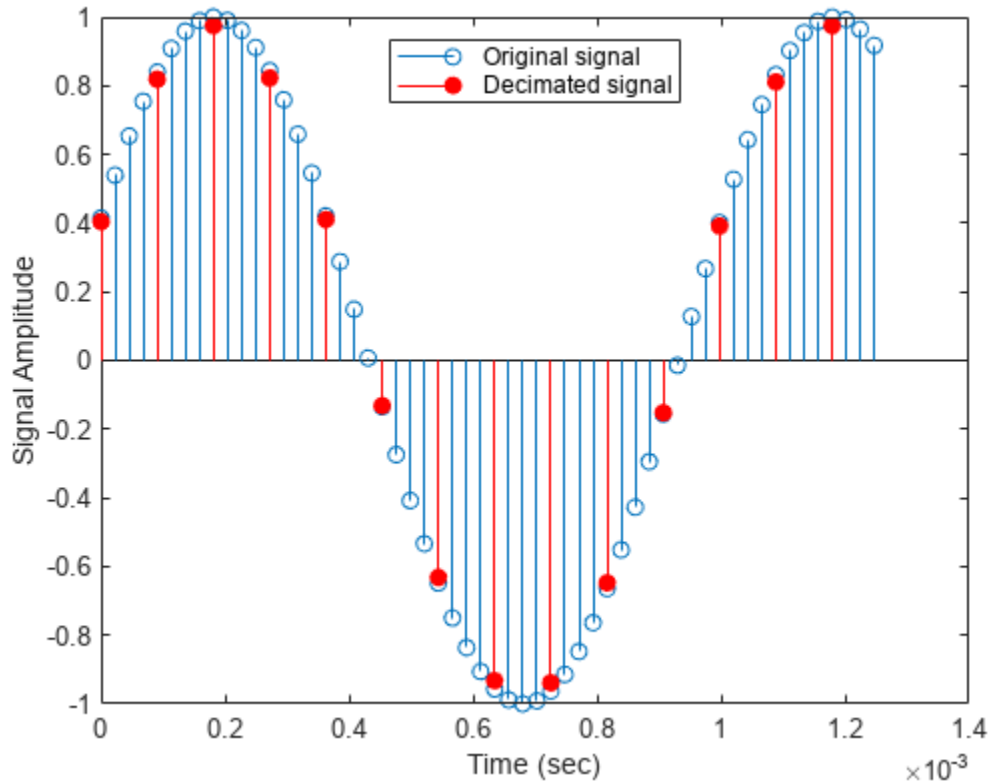
```
src = dsp.SignalSource(x, 64);
```

Decimate the output with 16 samples per frame.

```
y = zeros(16, 16);  
for ii = 1:16  
    y(ii, :) = cicdec(src());  
end
```

Plot the first frame of the original and decimated signals. Output latency is 2 samples.

```
D = cicdec.DecimationFactor;  
diffDelay = cicdec.DifferentialDelay;  
NumSect = cicdec.NumSections;  
gainCIC = ...  
    (D*diffDelay)^NumSect;  
stem(n(1:56)/Fs, double(x(4:59)))  
hold on;  
stem(n(1:14)/(Fs/D), double(y(1, 3:end))/gainCIC, ...  
    'r', 'filled')  
xlabel('Time (sec)')  
ylabel('Signal Amplitude')  
legend('Original signal', ...  
    'Decimated signal', ...  
    'Location', 'north')  
hold off;
```



Using the `info` method in 'long' format, obtain the word lengths and fraction lengths of the fixed-point filter sections and the filter output.

```
info(cicdec, 'long')
```

```
ans =
'Discrete-Time FIR Multirate Filter (real)
-----
Filter Structure      : Cascaded Integrator-Comb Decimator
Decimation Factor    : 4
Differential Delay    : 1
Number of Sections   : 2
Stable                : Yes
Linear Phase         : Yes (Type 1)

Implementation Cost
Number of Multipliers      : 0
Number of Adders          : 4
Number of States          : 4
Multiplications per Input Sample : 0
Additions per Input Sample : 2.5

Fixed-Point Info
Section word lengths      : 20 19 19 18
Section fraction lengths : 15 14 14 13
Output word length       : 16
```

```
Output fraction length : 11
```

### Determine the Section and Output Word Lengths and Fraction Lengths

Using the `getFixedPointInfo` function, you can determine the word lengths and fraction lengths of the fixed-point sections and the output of the `dsp.CICDecimator` and `dsp.CICInterpolator` System objects. The data types of the filter sections and the output depend on the `FixedPointDataType` property of the filter System object™.

#### Full precision

Create a `dsp.CICDecimator` object. The default value of the `NumSections` property is 2. This value indicates that there are two integrator and comb sections. The WLS and FLS vectors returned by the `getFixedPointInfo` function contain five elements each. The first two elements represent the two integrator sections. The third and fourth elements represent the two comb sections. The last element represents the filter output.

```
cicD = dsp.CICDecimator

cicD =
  dsp.CICDecimator with properties:

    DecimationFactor: 2
    DifferentialDelay: 1
    NumSections: 2
    FixedPointDataType: 'Full precision'
```

By default, the `FixedPointDataType` property of the object is set to 'Full precision'. Calling the `getFixedPointInfo` function on this object with the input numeric type, `nt`, yields the following word length and fraction length vectors.

```
nt = numericType(1,16,15)

nt =

    DataTypeMode: Fixed-point: binary point scaling
    Signedness: Signed
    WordLength: 16
    FractionLength: 15

[WLs,FLs] = getFixedPointInfo(cicD,nt) %#ok

WLs = 1x5

    18    18    18    18    18

FLs = 1x5

    15    15    15    15    15
```

For details on how the word lengths and fraction lengths are computed, see the description for *Output Arguments*.

If you lock the `cicD` object by passing an input to its algorithm, you do not need to pass the `nt` argument to the `getFixedPointInfo` function.

```
input = int64(randn(8,1))
input = 8x1 int64 column vector

     1
     2
    -2
     1
     0
    -1
     0
     0

output = cicD(input)
output =
     0
     1
     3
     0

        DataTypeMode: Fixed-point: binary point scaling
           Signedness: Signed
           WordLength: 66
        FractionLength: 0

[WLs,FLs] = getFixedPointInfo(cicD) %#ok
WLs = 1x5
     66     66     66     66     66

FLs = 1x5
     0     0     0     0     0
```

The output and section word lengths are the sum of input word length, 64 in this case, and the number of sections, 2. The output and section fraction lengths are 0 since the input is a built-in integer.

### Minimum section word lengths

Release the object and change the `FixedPointDataType` property to 'Minimum section word lengths'. Determine the section and output fixed-point information when the input is fixed-point data, `fi(randn(8,2),1,24,15)`.

```
release(cicD);
cicD.FixedPointDataType = 'Minimum section word lengths'

cicD =
    dsp.CICDecimator with properties:
```

```
DecimationFactor: 2
DifferentialDelay: 1
    NumSections: 2
FixedPointDataType: 'Minimum section word lengths'
OutputWordLength: 32
```

```
inputF = fi(randn(8,2),1,24,15)
```

```
inputF =
    3.5784   -0.1241
    2.7694    1.4897
   -1.3499    1.4090
    3.0349    1.4172
    0.7254    0.6715
   -0.0630   -1.2075
    0.7148    0.7172
   -0.2050    1.6302
```

```
    DataTypeMode: Fixed-point: binary point scaling
    Signedness: Signed
    WordLength: 24
    FractionLength: 15
```

```
[WLs, FLs] = getFixedPointInfo(cicD,numericType(inputF)) %#ok
```

```
WLs = 1x5
```

```
    26    26    26    26    32
```

```
FLs = 1x5
```

```
    15    15    15    15    21
```

### Specify word and fraction lengths

Change the `FixedPointDataType` property to 'Specify word and fraction lengths'. Determine the fixed-point information using the `getFixedPointInfo` function.

```
cicD.FixedPointDataType = 'Specify word and fraction lengths'
```

```
cicD =
    dsp.CICDecimator with properties:
```

```
    DecimationFactor: 2
    DifferentialDelay: 1
    NumSections: 2
    FixedPointDataType: 'Specify word and fraction lengths'
    SectionWordLengths: [16 16 16 16]
    SectionFractionLengths: 0
    OutputWordLength: 32
    OutputFractionLength: 0
```

```
[WLs, FLs] = getFixedPointInfo(cicD,numericType(inputF)) %#ok
```



```

WLs = 1x5
    16    16    16    16    32

```

```

FLs = 1x5
    0     0     0     0     0

```

The section and output word lengths and fraction lengths are assigned as per the respective fixed-point properties of the `cicD` object. These values are not determined by the input numeric type. To confirm, call the `getFixedPointInfo` function without passing the `numericType` input argument.

```
[WLs, FLs] = getFixedPointInfo(cicD) %#ok
```

```

WLs = 1x5
    16    16    16    16    32

```

```

FLs = 1x5
    0     0     0     0     0

```

### Specify word lengths

To specify the word lengths of the filter section and output, set the `FixedPointDataType` property to `'Specify word lengths'`.

```
cicD.FixedPointDataType = 'Specify word lengths'
```

```

cicD =
  dsp.CICDecimator with properties:
    DecimationFactor: 2
    DifferentialDelay: 1
    NumSections: 2
    FixedPointDataType: 'Specify word lengths'
    SectionWordLengths: [16 16 16 16]
    OutputWordLength: 32

```

The `getFixedPointInfo` function requires the input numeric type because that information is used to compute the section and word fraction lengths.

```
[WLs, FLs] = getFixedPointInfo(cicD,numericType(inputF))
```

```

WLs = 1x5
    16    16    16    16    32

```

```

FLs = 1x5
    5     5     5     5    21

```

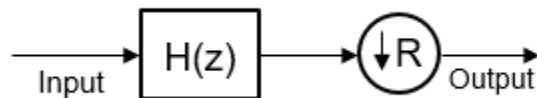
For more details on how the function computes the word and fraction lengths, see the description for *Output Arguments*.

## More About

### CIC Filter

CIC filters are an optimized class of linear phase FIR filters composed of a comb part and an integrator part.

The CIC decimation filter is conceptually given by a single rate CIC filter,  $H(z)$  which is a lowpass anti-imaging filter, followed by a downsampler. The CIC decimation filter decreases the sample rate of an input signal by an integer factor using a cascaded integrator-comb (CIC) filter.



In a more efficient implementation, the single rate CIC filter  $H(z)$  is factorized this way:

$$H(z) = \left[ \sum_{k=0}^{RM-1} z^{-k} \right]^N = \frac{(1 - z^{-RM})^N}{(1 - z^{-1})^N} = \frac{1}{(1 - z^{-1})^N} \cdot \frac{(1 - z^{-RM})^N}{1} = H_I^N(z) \cdot H_C^N(z)$$

where,

- $H_I$  is the transfer function of the integrator part of the filter containing  $N$  stages of integrators.
- $H_C$  is the transfer function of the  $N$  sections of the cascaded comb filters, each with a width of  $RM$ .
- $N$  is the number of sections. The number of sections in a CIC filter is defined as the number of sections in either the comb part *or* the integrator part of the filter. This value does not represent the total number of sections throughout the entire filter.
- $R$  is the decimation factor.
- $M$  is the differential delay.

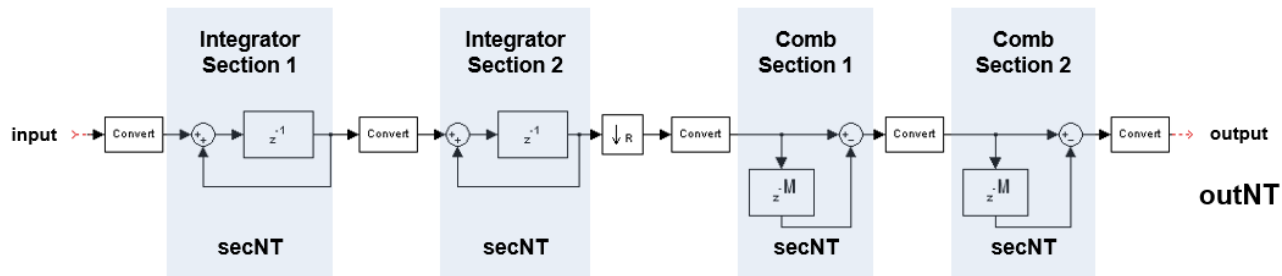
In the overall multirate realization, the algorithm applies the noble identity for decimation and moves the rate change factor,  $R$ , to follow after the  $N$  sections of the cascaded integrators. The transfer function of the resulting filter is given by the following equation:

$$H(z) = \frac{(1 - z^{-M})^N}{(1 - z^{-1})^N}$$

For a block diagram that shows the multirate implementation, see Algorithms on page 4-253.

### Fixed Point

The fixed-point signal diagram shows the data types that the `dsp.CICDecimator` object uses for fixed-point signals.



where,

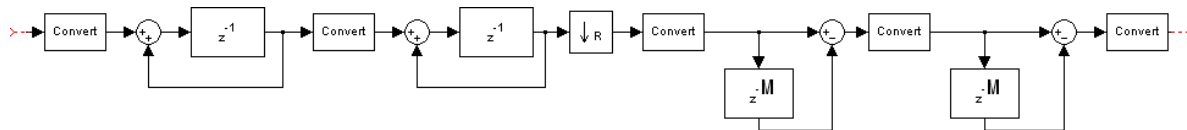
- `secNT = numerictype(1,secWL,secFL)`
- `outNT = numerictype(1,outWL,outFL)`
- `secWL` is the section word length you specify through the “SectionWordLengths” on page 4-0 property.
- `secFL` is the section fraction length you specify through the “SectionFractionLengths” on page 4-0 property.
- `outWL` is the output word length you specify through the “OutputWordLength” on page 4-0 property.
- `outFL` is the output fraction length you specify through the “OutputFractionLength” on page 4-0 property.

The value of `NumSections` in this diagram is 2.

## Algorithms

### CIC Decimation Filter

The CIC decimation filter in More About on page 4-252 is realized as a cascade of  $N$  sections of the integrators followed by a rate change factor of  $R$ , followed by  $N$  sections of comb filters.



This diagram shows two sections of cascaded integrators and two sections of cascaded comb filters. The unit delay in the integrator portion of the CIC filter can be located in either the feedforward or the feedback path. These two configurations yield identical filter frequency response. However, the numerical outputs from these two configurations are different due to the latency. This block puts the unit delay in the feedforward path of the integrator because it is a preferred configuration for HDL implementation.

## Version History

Introduced in R2012a

## References

- [1] Hogenauer, E.B. "An Economical Class of Digital Filters for Decimation and Interpolation." *IEEE Transactions on Acoustics, Speech and Signal Processing*. Volume 29, Number 2, 1981, 155-162.
- [2] Meyer-Baese, U. *Digital Signal Processing with Field Programmable Gate Arrays*. New York: Springer, 2001.
- [3] Harris, Fredric J. *Multirate Signal Processing for Communication Systems*. Indianapolis, IN: Prentice Hall PTR, 2004.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

This object supports HDL code generation with the Filter Design HDL Coder product. For workflows and limitations, see "Generate HDL Code for Filter System Objects" (Filter Design HDL Coder).

## See Also

### Functions

`generatehdl` | `impz` | `freqz` | `phasez` | `fvtool` | `gain` | `getFixedPointInfo` | `info`

### Objects

`dsp.CICInterpolator` | `dsp.CICCompensationDecimator` | `dsp.CICCompensationInterpolator`

### Blocks

CIC Decimation | CIC Interpolation | CIC Compensation Interpolator | CIC Compensation Decimator

### Topics

"GSM Digital Down Converter in MATLAB"

"Analysis Methods for Filter System Objects" on page 3-2

# dsp.CICInterpolator

**Package:** dsp

Interpolate signal using cascaded integrator-comb filter

## Description

The `dsp.CICInterpolator` System object interpolates an input signal using a cascaded integrator-comb (CIC) interpolation filter. The CIC interpolation filter structure consists of  $N$  sections of cascaded comb filters, followed by a rate change by a factor of  $R$ , followed by  $N$  sections of cascaded integrators. For details, see “Algorithms” on page 4-268. The `NumSections` property specifies  $N$ , the number of sections in the CIC filter. The `InterpolationFactor` property specifies  $R$ , the interpolation factor. The `getFixedPointInfo` function returns the word lengths and fraction lengths of the fixed-point sections and the output for the `dsp.CICInterpolator` System object. You can also generate HDL code for this System object using the `generatehdl` function.

---

**Note** This object requires a Fixed-Point Designer license.

---

To interpolate a signal using a CIC filter:

- 1 Create the `dsp.CICInterpolator` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
cicInterp = dsp.CICInterpolator
cicInterp = dsp.CICInterpolator(R,M,N)
cicInterp = dsp.CICInterpolator(Name,Value)
```

### Description

`cicInterp = dsp.CICInterpolator` creates a CIC interpolation System object that applies a CIC interpolation filter to the input signal.

`cicInterp = dsp.CICInterpolator(R,M,N)` creates a CIC interpolation object with the `InterpolationFactor` property set to  $R$ , the `DifferentialDelay` property set to  $M$ , and the `NumSections` property set to  $N$ .

`cicInterp = dsp.CICInterpolator(Name,Value)` creates a CIC interpolation object with each specified property set to the specified value. Enclose each property name in single quotes. You can use this syntax with any previous input argument combinations.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

### **InterpolationFactor** — Interpolation factor

2 (default) | positive integer

Factor by which the input signal is interpolated, specified as a positive integer.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **DifferentialDelay** — Differential delay of filter comb sections

1 (default) | positive integer

Differential delay value used in each of the comb sections of the filter, specified as a positive integer. For details, see “Algorithms” on page 4-268. If the differential delay is of built-in integer class data type, the interpolation factor must be the same integer data type or `double`. For example, if the differential delay is an `int8`, then the interpolation factor must be an `int8` or `double`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **NumSections** — Number of integrator and comb sections

2 (default) | positive integer

Number of integrator and comb sections of the CIC filter, specified as a positive integer. This number indicates the number of sections in either the comb part or the integrator part of the filter. The total number of sections in the CIC filter is twice the number of sections given by this property.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **FixedPointDataType** — Fixed-point property designations

Full precision (default) | Minimum section word lengths | Specify word lengths | Specify word and fraction lengths

Fixed-point property designations, specified as one of the following:

- `Full precision` - The word length and fraction length of the CIC filter sections and the object output operate in full precision.
- `Minimum section word lengths` - Specify the output word length through the `OutputWordLength` property. The object determines the filter section data type and the output fraction length that give the best possible precision. For details, see `getFixedPointInfo` and `cicInterpOut` argument.
- `Specify word lengths` - Specify the word lengths of the CIC filter sections and the object output through the `SectionWordLengths` and `OutputWordLength` properties. The object determines the corresponding fraction lengths to give the best possible precision. For details, see `getFixedPointInfo` and the `cicInterpOut` argument.
- `Specify word and fraction lengths` - Specify the word length and fraction length of the CIC filter sections and the object output through the `SectionWordLengths`, `SectionFractionLengths`, `OutputWordLength`, and `OutputFractionLength` properties.

**SectionWordLengths — Fixed-point word lengths for each filter section**

[16 16 16 16] (default) | scalar | vector

Fixed-point word lengths to use for each filter section, specified as a scalar or a row vector of integers. The word length must be greater than or equal to 2. If you specify a scalar, the value applies to all the sections of the filter. If you specify a vector, the vector must be of length  $2 \times \text{NumSections}$ .

Example: 32

Example: [32 32 32 32]

**Dependencies**

This property applies when you set the `FixedPointDataType` property to 'Specify word lengths' or 'Specify word and fraction lengths'.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**SectionFractionLengths — Fixed-point fraction lengths for each filter section**

0 (default) | scalar | vector

Fixed-point fraction lengths to use for each filter section, specified as a scalar or a row vector of integers. The fraction length can be negative, 0, or positive. If you specify a scalar, the value applies to all the sections of the filter. If you specify a vector, the vector must be of length  $2 \times \text{NumSections}$ .

Example: -2

Example: [-2 0 5 8]

**Dependencies**

This property applies when you set the `FixedPointDataType` property to 'Specify word and fraction lengths'.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**OutputWordLength — Fixed-point word length for filter output**

32 (default) | scalar integer

Fixed-point word length to use for the filter output, specified as a scalar integer greater than or equal to 2.

**Dependencies**

This property applies when you set the `FixedPointDataType` property to one of 'Minimum section word lengths', 'Specify word lengths', or 'Specify word and fraction lengths'.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**OutputFractionLength — Fixed-point fraction length for filter output**

0 (default) | scalar integer

Fixed-point fraction length to use for the filter output, specified as a scalar integer.

**Dependencies**

This property applies when you set the `FixedPointDataType` property to 'Specify word and fraction lengths'.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

## Usage

## Syntax

```
cicInterpOut = cicInterp(input)
```

## Description

`cicInterpOut = cicInterp(input)` interpolates the input using a CIC interpolator.

## Input Arguments

### **input** — Data input

vector | matrix

Data input, specified as a vector or matrix. If the input is of `single` or `double` data type, property settings related to the fixed-point data types are ignored.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

## Output Arguments

### **cicInterpOut** — CIC interpolator output

vector | matrix

Interpolated output, returned as a vector or a matrix. The output frame size equals (“InterpolationFactor” on page 4-0 )  $\times$  input frame size. The complexity of the output data matches that of the input data. If the input is `single` or `double`, the output data type matches the input data type.

If the input is of built-in integer data type or of fixed-point data type, the output word length and fraction length depend on the fixed-point data type setting you choose through the “FixedPointDataType” on page 4-0 property.

## Full precision

When the `FixedPointDataType` property is set to 'Full precision', the following relationship applies:

$$WL_{\text{output}} = WL_{\text{input}} + NumSect$$

$$FL_{\text{output}} = FL_{\text{input}}$$

where,

- $WL_{\text{output}}$  -- Word length of the output data.
- $FL_{\text{output}}$  -- Fraction length of the output data.
- $WL_{\text{input}}$  -- Word length of the input data.
- $FL_{\text{input}}$  -- Fraction length of the input data.
- $NumSect$  -- Number of sections in the CIC filter specified through the “NumSections” on page 4-0 property.



$WL_{input}$  and  $FL_{input}$  are inherited from the data input you pass to the object algorithm. For built-in integer inputs, the fraction length is 0.

### Minimum section word lengths

When the `FixedPointDataType` property is set to 'Minimum section word lengths', the output word length is the value you specify in "OutputWordLength" on page 4-0 property. The output fraction length,  $FL_{output}$  is given by the following equation:

$$FL_{output} = WL_{output} - (WL_{input} - FL_{input} + NumSect)$$

### Specify word and fraction lengths

When the `FixedPointDataType` is set to 'Specify word and fraction lengths', the output word length and fraction length are the values you specify in the "OutputWordLength" on page 4-0 and "OutputFractionLength" on page 4-0 properties.

### Specify word lengths

When the `FixedPointDataType` is set to 'Specify word lengths', the output word length is the value you specify in the `OutputWordLength` property. The output fraction length,  $FL_{output}$  is given by the following equation:

$$FL_{output} = WL_{output} - (WL_{input} - FL_{input} + NumSect)$$

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

## Object Functions

To use an object function, specify the `System` object as the first input argument. For example, to release system resources of a `System` object named `obj`, use this syntax:

```
release(obj)
```

### Specific to dsp.CICInterpolator

|                                |   |
|--------------------------------|---|
| <code>generatehdl</code>       | Generate HDL code for quantized DSP filter (requires Filter Design HDL Coder) |
| <code>impz</code>              | Impulse response of discrete-time filter System object                        |
| <code>freqz</code>             | Frequency response of discrete-time filter System object                      |
| <code>phasez</code>            | Phase response of discrete-time filter System object (unwrapped)              |
| <code>fvtool</code>            | Visualize frequency response of DSP filters                                   |
| <code>gain</code>              | Gain of CIC filter System object  |
| <code>getFixedPointInfo</code> | Get fixed-point word and fraction lengths                                     |
| <code>info</code>              | Information about filter System object  |

### Common to All System Objects

|                      |  |
|----------------------|--|
| <code>step</code>    | Run System object algorithm  |
| <code>release</code> | Release resources and allow changes to System object property values and input characteristics |
| <code>reset</code>   | Reset internal states of System object   |

For a list of filter analysis methods this object supports, type `dsp.CICInterpolator.helpFilterAnalysis` in the MATLAB command prompt. For the

corresponding function reference pages, see “Analysis Methods for Filter System Objects” on page 3-2.

## Examples

### Interpolate Signal Using CICInterpolator System object

Create a `dsp.CICInterpolator` System object™ with `InterpolationFactor` set to 2. Interpolate a fixed-point signal by a factor of 2 from 22.05 kHz to 44.1 kHz.

```
cicint = dsp.CICInterpolator(2)

cicint =
  dsp.CICInterpolator with properties:

    InterpolationFactor: 2
    DifferentialDelay: 1
    NumSections: 2
    FixedPointDataType: 'Full precision'
```

Create a `dsp.SineWave` object with `SampleRate` set to 22.05 kHz, `SamplesPerFrame` set to 32, and `OutputDataType` set to 'Custom'. To generate a fixed-point signal, set the `CustomOutputDataType` property to a `numericType` object. For the purpose of this example, set the value to `numericType([], 16)`. The fraction length is computed based on the values of the generated sinusoidal signal to give the best possible precision.

To generate a fixed-point signal, set the `Method` property of the `dsp.SineWave` object to 'Table lookup'. This method of generating the sinusoidal signal requires that the period of every sinusoid in the output be evenly divisible by the sample period. That is,  $1/f_i T_s = k_i$  must be an integer value for every channel  $i = 1, 2, \dots, N$ . The value of  $T_s$  equals  $1/F_s$ , the variable  $f_i$  is the frequency of the sinusoidal signal, and  $F_s$  is the sample rate of the signal. In other words, the ratio  $F_s/f_i$  must be an integer. For more details, see the “Algorithms” on page 4-1009 section on the `dsp.SineWave` object page.

In this example,  $F_s$  is set to 22050 Hz and  $f_i$  is set to 1050 Hz.

```
Fs = 22.05e3;
sine = dsp.SineWave('Frequency', 1050, ...
    'SampleRate', Fs, ...
    'SamplesPerFrame', 32, ...
    'Method', 'Table lookup', ...
    'OutputDataType', 'Custom')

sine =
  dsp.SineWave with properties:

    Amplitude: 1
    Frequency: 1050
    PhaseOffset: 0
    ComplexOutput: false
    Method: 'Table lookup'
    TableOptimization: 'Speed'
    SamplesPerFrame: 32
```

```

    SampleRate: 22050
    OutputDataType: 'Custom'

```

Show all properties

In each loop of the iteration, stream in a frame of the fixed-point sinusoidal signal sampled at 22.05 kHz. Interpolate the streamed signal by a factor of 2. The interpolated output has 64 samples per frame.

```

for i = 1:16
    x = sine();
    y = cicint(x);
end

```

The output of the CIC interpolation filter is amplified by a specific gain value. You can determine this value using the `gain` function. This gain equals the gain of the  $2N^{\text{th}}$  stage of the CIC interpolation filter and equals  $(I \times D)^N / I$ , where  $I$  is the interpolation factor,  $D$  is the differential delay, and  $N$  is the number of sections of the CIC interpolator.

```

gainCIC = gain(cicint)

gainCIC = 2

```

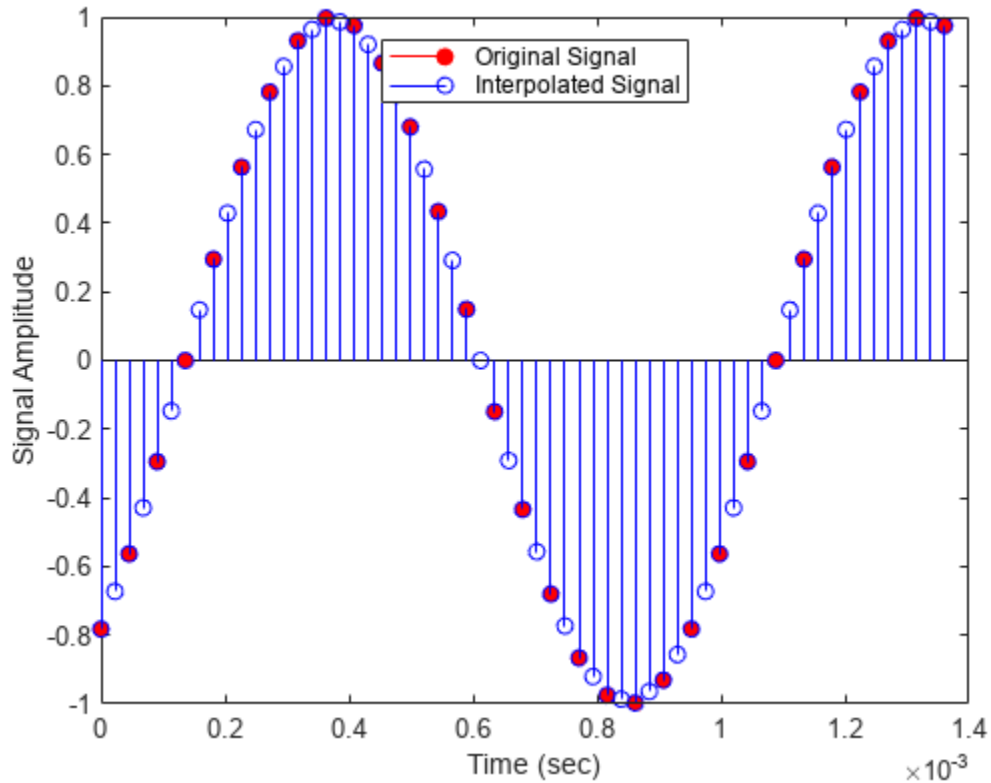
To adjust this amplified output and to match it to the amplitude of the original signal, divide the CIC interpolated signal with the computed gain value.

Compare the last frames of the original and the interpolated signals. While plotting, account for the output latency of 2 samples.

```

n = (0:63)';
stem(n(1:31)/Fs,double(x(1:31)),'r','filled')
hold on;
I = cicint.InterpolationFactor;
stem(n(1:61)/(Fs*I), ...
     double(y(4:end))/gainCIC,'b')
xlabel('Time (sec)')
ylabel('Signal Amplitude')
legend('Original Signal',...
       'Interpolated Signal',...
       'location','north')
hold off;

```



Using the `info` function in the 'long' format, obtain the word lengths and fraction lengths of the fixed-point filter sections and the filter output.

```
info(cicint, 'long')
```

```
ans =
'Discrete-Time FIR Multirate Filter (real)
-----
Filter Structure      : Cascaded Integrator-Comb Interpolator
Interpolation Factor : 2
Differential Delay   : 1
Number of Sections  : 2
Stable               : Yes
Linear Phase        : Yes (Type 1)

Implementation Cost
Number of Multipliers : 0
Number of Adders      : 4
Number of States      : 4
Multiplications per Input Sample : 0
Additions per Input Sample : 6

Fixed-Point Info
Section word lengths  : 17 17 17 17
Section fraction lengths : 14 14 14 14
Output word length    : 17
```

```
Output fraction length : 14
```

## Determine the Section and Output Word Lengths and Fraction Lengths

Using the `getFixedPointInfo` function, you can determine the word lengths and fraction lengths of the fixed-point sections and the output of the `dsp.CICDecimator` and `dsp.CICInterpolator` System objects. The data types of the filter sections and the output depend on the `FixedPointDataType` property of the filter System object™.

### Full precision

Create a `dsp.CICDecimator` object. The default value of the `NumSections` property is 2. This value indicates that there are two integrator and comb sections. The WLS and FLs vectors returned by the `getFixedPointInfo` function contain five elements each. The first two elements represent the two integrator sections. The third and fourth elements represent the two comb sections. The last element represents the filter output.

```
cicD = dsp.CICDecimator
cicD =
    dsp.CICDecimator with properties:
        DecimationFactor: 2
        DifferentialDelay: 1
        NumSections: 2
        FixedPointDataType: 'Full precision'
```

By default, the `FixedPointDataType` property of the object is set to 'Full precision'. Calling the `getFixedPointInfo` function on this object with the input numeric type, `nt`, yields the following word length and fraction length vectors.

```
nt = numerictype(1,16,15)
nt =
    DataTypeMode: Fixed-point: binary point scaling
    Signedness: Signed
    WordLength: 16
    FractionLength: 15

[WLs,FLs] = getFixedPointInfo(cicD,nt) %#ok
WLs = 1x5
    18    18    18    18    18
FLs = 1x5
    15    15    15    15    15
```

For details on how the word lengths and fraction lengths are computed, see the description for *Output Arguments*.

If you lock the `cicD` object by passing an input to its algorithm, you do not need to pass the `nt` argument to the `getFixedPointInfo` function.

```
input = int64(randn(8,1))
input = 8x1 int64 column vector

     1
     2
    -2
     1
     0
    -1
     0
     0

output = cicD(input)
output =
     0
     1
     3
     0

        DataTypeMode: Fixed-point: binary point scaling
          Signedness: Signed
         WordLength: 66
    FractionLength: 0

[WLs,FLs] = getFixedPointInfo(cicD) %#ok
WLs = 1x5
    66    66    66    66    66

FLs = 1x5
     0     0     0     0     0
```

The output and section word lengths are the sum of input word length, 64 in this case, and the number of sections, 2. The output and section fraction lengths are 0 since the input is a built-in integer.

### Minimum section word lengths

Release the object and change the `FixedPointDataType` property to 'Minimum section word lengths'. Determine the section and output fixed-point information when the input is fixed-point data, `fi(randn(8,2),1,24,15)`.

```
release(cicD);
cicD.FixedPointDataType = 'Minimum section word lengths'

cicD =
    dsp.CICDecimator with properties:
```

```

    DecimationFactor: 2
    DifferentialDelay: 1
        NumSections: 2
    FixedPointDataType: 'Minimum section word lengths'
    OutputWordLength: 32

```

```
inputF = fi(randn(8,2),1,24,15)
```

```
inputF =
    3.5784   -0.1241
    2.7694    1.4897
   -1.3499    1.4090
    3.0349    1.4172
    0.7254    0.6715
   -0.0630   -1.2075
    0.7148    0.7172
   -0.2050    1.6302

```

```

    DataTypeMode: Fixed-point: binary point scaling
    Signedness: Signed
    WordLength: 24
    FractionLength: 15

```

```
[WLs, FLs] = getFixedPointInfo(cicD,numericType(inputF)) %#ok
```

```
WLs = 1x5
```

```
    26    26    26    26    32
```

```
FLs = 1x5
```

```
    15    15    15    15    21
```

### Specify word and fraction lengths

Change the `FixedPointDataType` property to 'Specify word and fraction lengths'. Determine the fixed-point information using the `getFixedPointInfo` function.

```
cicD.FixedPointDataType = 'Specify word and fraction lengths'
```

```
cicD =
    dsp.CICDecimator with properties:
```

```

    DecimationFactor: 2
    DifferentialDelay: 1
        NumSections: 2
    FixedPointDataType: 'Specify word and fraction lengths'
    SectionWordLengths: [16 16 16 16]
    SectionFractionLengths: 0
    OutputWordLength: 32
    OutputFractionLength: 0

```

```
[WLs, FLs] = getFixedPointInfo(cicD,numericType(inputF)) %#ok
```

```
WLs = 1x5
    16    16    16    16    32
```

```
FLs = 1x5
     0     0     0     0     0
```

The section and output word lengths and fraction lengths are assigned as per the respective fixed-point properties of the `cicD` object. These values are not determined by the input numeric type. To confirm, call the `getFixedPointInfo` function without passing the `numericType` input argument.

```
[WLs, FLs] = getFixedPointInfo(cicD) %#ok
```

```
WLs = 1x5
    16    16    16    16    32
```

```
FLs = 1x5
     0     0     0     0     0
```

### Specify word lengths

To specify the word lengths of the filter section and output, set the `FixedPointDataType` property to `'Specify word lengths'`.

```
cicD.FixedPointDataType = 'Specify word lengths'
```

```
cicD =
  dsp.CICDecimator with properties:
    DecimationFactor: 2
    DifferentialDelay: 1
    NumSections: 2
    FixedPointDataType: 'Specify word lengths'
    SectionWordLengths: [16 16 16 16]
    OutputWordLength: 32
```

The `getFixedPointInfo` function requires the input numeric type because that information is used to compute the section and word fraction lengths.

```
[WLs, FLs] = getFixedPointInfo(cicD,numericType(inputF))
```

```
WLs = 1x5
    16    16    16    16    32
```

```
FLs = 1x5
     5     5     5     5    21
```



For more details on how the function computes the word and fraction lengths, see the description for *Output Arguments*.

## More About

### CIC Filter

CIC filters are an optimized class of linear phase FIR filters composed of a comb part and an integrator part.

The CIC interpolation filter is conceptually given by an upsampler followed by a single rate CIC filter,  $H(z)$ , which is a lowpass anti-imaging filter. The CIC interpolation filter increases the sample rate of an input signal by an integer factor using a cascaded integrator-comb (CIC) filter.



In a more efficient implementation, the single rate CIC filter  $H(z)$  is factorized this way:

$$H(z) = \left[ \sum_{k=0}^{RM-1} z^{-k} \right]^N = \frac{(1 - z^{-RM})^N}{(1 - z^{-1})^N} = \frac{(1 - z^{-RM})^N}{1} \cdot \frac{1}{(1 - z^{-1})^N} = H_C^N(z) \cdot H_I^N(z)$$

where,

- $H_C$  is the transfer function of the  $N$  sections of the cascaded comb filters, each with a width of  $RM$ .
- $H_I$  is the transfer function of the integrator part of the filter containing  $N$  stages of integrators.
- $N$  is the number of sections. The number of sections in a CIC filter is defined as the number of sections in either the comb part *or* the integrator part of the filter. This value does not represent the total number of sections throughout the entire filter.
- $R$  is the interpolation factor.
- $M$  is the differential delay.

In the overall multirate realization, the algorithm applies the noble identity for interpolation and moves the rate change factor,  $R$ , to follow after the  $N$  sections of the cascaded comb filters.

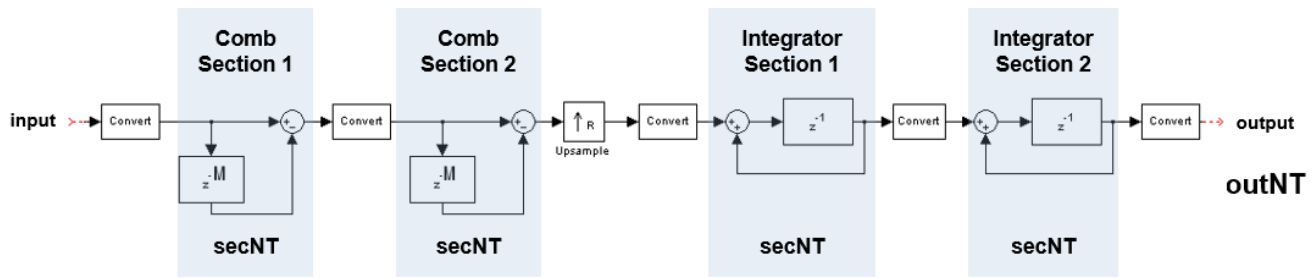
The transfer function of the resulting filter is given by the following equation:

$$H(z) = \frac{(1 - z^{-M})^N}{(1 - z^{-1})^N}$$

For a block diagram that shows the multirate implementation, see Algorithms on page 4-268.

### Fixed Point

The fixed-point signal diagram shows the data types that the `dsp.CICInterpolator` object uses for fixed-point signals.



where,

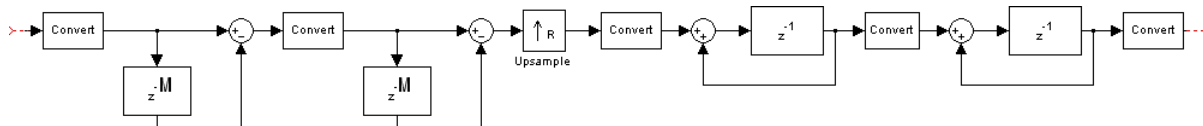
- $secNT = numericity(1, secWL, secFL)$
- $outNT = numericity(1, outWL, outFL)$
- $secWL$  is the section word length you specify through the “SectionWordLengths” on page 4-0 property.
- $secFL$  is the section fraction length you specify through the “SectionFractionLengths” on page 4-0 property.
- $outWL$  is the output word length you specify through the “OutputWordLength” on page 4-0 property.
- $outFL$  is the output fraction length you specify through the “OutputFractionLength” on page 4-0 property.

The value of NumSections in this diagram is 2.

## Algorithms

### CIC Interpolation Filter

The CIC interpolation filter in More About on page 4-267 is realized as a cascade of  $N$  sections of comb filters followed by a rate change by a factor  $R$ , followed by  $N$  sections of cascaded integrators.



This diagram shows two sections of cascaded comb filters and two sections of cascaded integrators. The unit delay in the integrator portion of the CIC filter can be located in either the feedforward or the feedback path. These two configurations yield identical filter frequency response. However, the numerical outputs from these two configurations are different due to the latency. This algorithm puts the unit delay in the feedforward path of the integrator since it is a preferred configuration for HDL implementation.

## Version History

Introduced in R2012a

## References

- [1] Hogenauer, E.B. "An Economical Class of Digital Filters for Decimation and Interpolation." *IEEE Transactions on Acoustics, Speech and Signal Processing*. Volume 29, Number 2, 1981, 155-162.
- [2] Meyer-Baese, U. *Digital Signal Processing with Field Programmable Gate Arrays*. New York: Springer, 2001.
- [3] Harris, Fredric J. *Multirate Signal Processing for Communication Systems*. Indianapolis, IN: Prentice Hall PTR, 2004.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

This object supports HDL code generation with the Filter Design HDL Coder product. For workflows and limitations, see "Generate HDL Code for Filter System Objects" (Filter Design HDL Coder).

## See Also

### Functions

generatehdl | impz | freqz | phasez | fvtool | gain | getFixedPointInfo | info

### Objects

dsp.CICDecimator | dsp.CICCompensationDecimator |  
dsp.CICCompensationInterpolator

### Blocks

CIC Decimation | CIC Interpolation | CIC Compensation Interpolator | CIC Compensation Decimator

### Topics

"Analysis Methods for Filter System Objects" on page 3-2

## dsp.ColoredNoise

**Package:** dsp

Generate colored noise signal

### Description

The `dsp.ColoredNoise` System object generates a colored noise signal with a power spectral density (PSD) of  $1/|f|^\alpha$  over its entire frequency range. The inverse frequency power,  $\alpha$ , can be any value in the interval  $[-2 \ 2]$ . The type of colored noise the object generates depends on the “Color” on page 4-0 you choose. When you set `Color` to 'custom', you can specify the power density of the noise through the “InverseFrequencyPower” on page 4-0 property.

This object uses the default MATLAB random stream, `RandStream`. Reset the default stream for repeatable simulations.

To generate colored noise signal:

- 1 Create the `dsp.ColoredNoise` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

### Creation

#### Syntax

```
cn = dsp.ColoredNoise
cn = dsp.ColoredNoise(Name=Value)
cn = dsp.ColoredNoise(pow,samp,numChan,Name=Value)
cn = dsp.ColoredNoise(color,samp,numChan,Name=Value)
```

#### Description

`cn = dsp.ColoredNoise` creates a colored noise object, `cn`, that outputs a noise signal one sample or frame at a time, with a  $1/|f|^\alpha$  spectral characteristic over its entire frequency range. Typical values for  $\alpha$  are  $\alpha = 1$  (pink noise) and  $\alpha = 2$  (brownian noise).

`cn = dsp.ColoredNoise(Name=Value)` creates a colored noise object with each specified property set to the specified value. Enclose each property name in single quotes.

Example: `dsp.ColoredNoise(Color='pink');`

`cn = dsp.ColoredNoise(pow,samp,numChan,Name=Value)` creates a colored noise object with the `InverseFrequencyPower` property set to `pow`, the `SamplesPerFrame` property set to `samp`, and the `NumChannels` property set to `numChan`.

Example: `dsp.ColoredNoise(1,44.1e3,1,OutputDataType='single');`

`cn = dsp.ColoredNoise(color,samp,numChan,Name=Value)` creates a colored noise object with the `Color` property set to `color`, the `SamplesPerFrame` property set to `samp`, and the `NumChannels` property set to `numChan`.

Example: `dsp.ColoredNoise('pink',1024,2,OutputDataType='single');`

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

### Color — Noise color

'custom' (default) | 'pink' | 'white' | 'brown' | 'blue' | 'purple'

Noise color, specified as one of the following. Each color is associated with a specific inverse frequency power of the generated noise sequence.

- 'pink' -- The inverse frequency power  $\alpha$  equals 1.
- 'white' --  $\alpha = 0$ .
- 'brown' --  $\alpha = 2$ . Also known as red or Brownian noise.
- 'blue' --  $\alpha = -1$ . Also known as azure noise.
- 'purple' --  $\alpha = -2$ . Also known as violet noise.
- 'custom' -- For noise with a custom inverse frequency power,  $\alpha$  equals the value of the `InverseFrequencyPower` property.

`InverseFrequencyPower`  $\alpha$  can be any value in the interval  $[-2, 2]$ .

Data Types: `char` | `string`

### InverseFrequencyPower — Inverse frequency power

1 (default) | real scalar in  $[-2, 2]$

Inverse frequency power  $\alpha$ , specified as a real scalar in the interval  $[-2, 2]$ . The inverse exponent defines the PSD of the random process as  $1/|f|^\alpha$ . Values of `InverseFrequencyPower` greater than 0 generate lowpass noise with a singularity (pole) at  $f = 0$ . These processes exhibit long memory. Values of `InverseFrequencyPower` less than 0 generate highpass noise with increments that are negatively correlated. These processes are referred to as anti-persistent. Special cases include:

- 1 -- Pink noise
- 2 -- Brown, red, or Brownian noise
- 0 -- White noise process with a flat PSD
- -1 -- Blue or azure noise
- -2 -- Purple or violet noise

In a log-log plot of power as a function of frequency, processes generated by this object exhibit an approximate linear relationship with slope equal to  $-\alpha$ .

**Dependencies**

This property applies only when you set `Color` to `'custom'`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**SamplesPerFrameSource — Samples per frame source**

`'Property'` (default) | `'Input port'`

Source of the number of samples per frame, specified as `'Property'` or `'Input port'`.

Data Types: `char` | `string`

**SamplesPerFrame — Number of samples per output channel**

1024 (default) | positive integer

Number of samples per output channel, specified as a positive integer. This property determines the number of rows in the generated signal.

**Dependencies**

To enable this property, set `SamplesPerFrameSource` to `'Property'`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**MaxSamplesPerFrame — Maximum samples per frame**

192000 (default) | positive integer

Maximum number of samples per frame, specified as a positive integer.

The value you specify in the `MaxSamplesPerFrame` property acts as an upper limit to the number of samples in each frame that the object generates. If you specify a value greater than `MaxSamplesPerFrame`, the number of samples in each frame that the object generates equals the value in the `MaxSamplesPerFrame` property.

**Dependencies**

To enable this property, set `SamplesPerFrameSource` to `'Input port'`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**NumChannels — Number of output channels**

1 (default) | positive integer

Number of output channels, specified as an integer. This property determines the number of columns of the generated signal.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**RandomStream — Source of random number stream**

`'Global stream'` (default) | `'mt19937ar with seed'`

Source of the random number stream, specified as one of the following:

- `'Global stream'` -- The current global random number stream is used for normally distributed random number generation.
- `'mt19937ar with seed'` -- The `mt19937ar` algorithm is used for normally distributed random number generation. The `reset` function reinitializes the random number stream to the value of the `Seed` property.

Data Types: char | string

### Seed — Initial seed

67 (default) | nonnegative integer

Initial seed of mt19937ar random number stream generator algorithm, specified as a nonnegative integer. The reset function reinitializes the random number stream to the value of the Seed property.

### Dependencies

This property applies only when you set the RandomStream property to 'mt19937ar with seed'.

Data Types: double

### BoundedOutput — Set output bounds to +1 and -1

false (default) | true

Specify the output to be bounded between +1 and -1, specified as:

- `true` -- The internal random source that generates the noise is uniform instead of Gaussian, and a gain is applied so that the absolute maximum output never exceeds 1.

If `Color` is set to 'white', there is no color filter applied to the output of the random source. The output is uniform noise of amplitude between +1 and -1. The output distribution of filtered noise (such as pink noise) is quasi-Gaussian.

If `Color` is set to any other option, then a coloring filter is applied to the output of the random source, followed by a gain which ensures that the absolute maximum output never exceeds 1.

- `false` -- The internal random source is Gaussian. The output is not bounded.

Data Types: logical

### OutputDataType — Output data type

'double' (default) | 'single'

Output data type, specified as either 'double' or 'single'.

Data Types: char | string

## Usage

### Syntax

```
noiseOut = cn()
noiseOut = cn(L)
```

### Description

`noiseOut = cn()` outputs one sample or one frame of colored noise data.

`noiseOut = cn(L)` outputs one frame of colored noise data of length L, where L is a nonnegative integer.

This syntax applies only when you set `SamplesPerFrameSource` to 'Input port'.

## Output Arguments

### **noiseOut** — Colored noise output

vector | matrix

Colored noise output, returned as a vector or a matrix. The `SamplesPerFrame`, `NumChannels`, and the `OutputDataType` properties specify the size and data type of the output.

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Common to All System Objects

`step` Run System object algorithm

`release` Release resources and allow changes to System object property values and input characteristics

`reset` Reset internal states of System object

## Examples

### Measure Pink Noise Power in Octave Bands

The output from this example shows that pink noise has approximately equal power in octave bands.

Generate a single-channel signal of pink noise that is 44,100 samples in length. Set the random number generator to the default settings for reproducible results.

```
pinkNoise = dsp.ColoredNoise(1,44.1e3,1)

pinkNoise =
  dsp.ColoredNoise with properties:
           Color: 'custom'
  InverseFrequencyPower: 1
           BoundedOutput: false
           NumChannels: 1
  SamplesPerFrameSource: 'Property'
           SamplesPerFrame: 44100
           OutputDataType: 'double'
           RandomStream: 'Global stream'
```

```
rng default;
x = pinkNoise();
```

Set the sample rate to 44.1 kHz. Measure the power in octave bands beginning with 100-200 Hz and ending with 6.400-12.8 kHz. Display the results in a table.

```
beginfreq = 100;
endfreq = 200;
count = 1;
```



```

freqinterval = zeros(7,2);
Pwr = zeros(7,1);
while(endfreq<=44.1e3/2)
    freqinterval(count,:) = [beginfreq endfreq];
    Pwr(count) = bandpower(x,44.1e3,[beginfreq endfreq]);
    beginfreq = endfreq;
    endfreq = 2*endfreq;
    count = count+1;
end
Pwr = Pwr(:);
table(freqinterval,Pwr)

```

```

ans=7x2 table
    freqinterval      Pwr
    _____      _____
    100      200      0.17549
    200      400      0.20313
    400      800      0.2438
    800      1600     0.2503
    1600     3200     0.25233
    3200     6400     0.26828
    6400     12800    0.25211

```

The pink noise has roughly equal power in octave bands.

Rerun the preceding code with 'InverseFrequencyPower' equal to 0, which generates a white noise signal. A white noise signal has a flat power spectral density, or equal power per unit frequency. Set the random number generator to the default settings for reproducible results.

```

whiteNoise = dsp.ColoredNoise(0,44.1e3,1);
rng default;
x = whiteNoise();

```

Set the sample rate to 44.1 kHz. Measure the power in octave bands beginning with 100-200 Hz and ending with 6.400-12.8 kHz. Display the results in a table.

```

beginfreq = 100;
endfreq = 200;
count = 1;
while(endfreq<=44.1e3/2)
    freqinterval(count,:) = [beginfreq endfreq];
    Pwr(count) = bandpower(x,44.1e3,[beginfreq endfreq]);
    beginfreq = endfreq;
    endfreq = 2*endfreq;
    count = count+1;
end
Pwr = Pwr(:);
table(freqinterval,Pwr)

```

```

ans=7x2 table
    freqinterval      Pwr
    _____      _____
    100      200      0.0031417
    200      400      0.0073833
    400      800      0.017421

```

|      |       |          |
|------|-------|----------|
| 800  | 1600  | 0.035926 |
| 1600 | 3200  | 0.071139 |
| 3200 | 6400  | 0.15183  |
| 6400 | 12800 | 0.28611  |

White noise has approximately equal power per unit frequency, so octave bands have an unequal distribution of power. Because the width of an octave band increases with increasing frequency, the power per octave band increases for white noise.

### PSD of Pink Noise Realization

Generate a pink noise signal 2048 samples in length. The sample rate is 1 Hz. Obtain an estimate of the power spectral density using Welch's overlapped segment averaging.

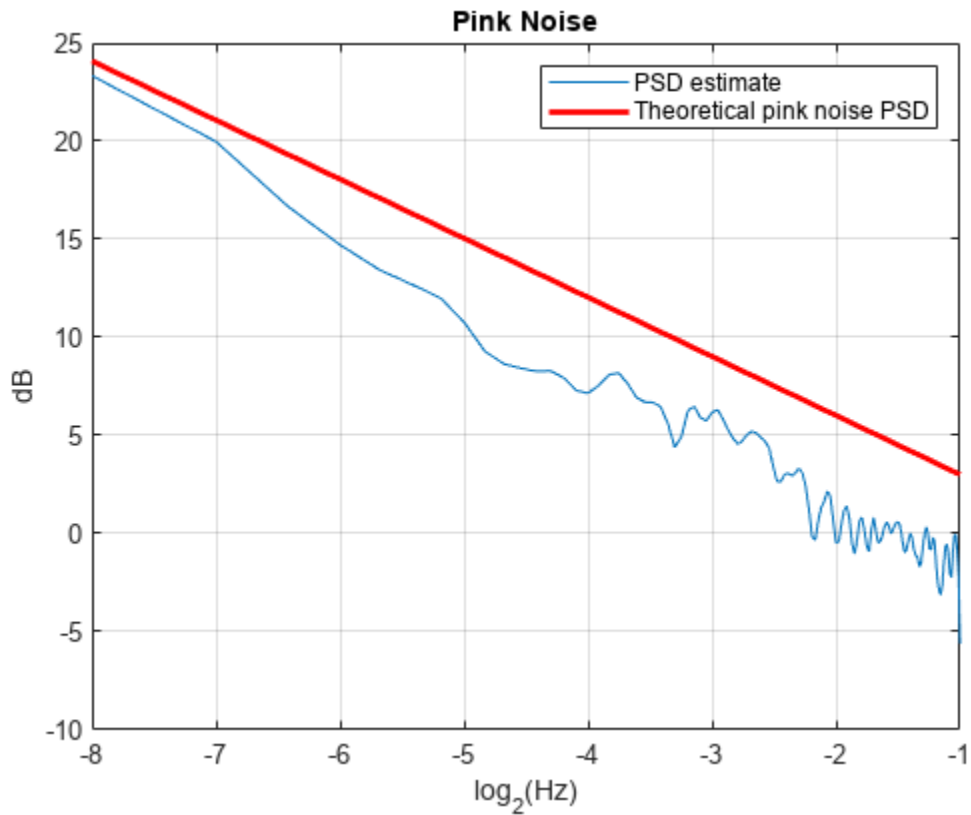
```
cn = dsp.ColoredNoise('pink',SamplesPerFrame=2048);  
x = cn();  
Fs = 1;  
[Pxx,F] = pwelch(x,hamming(128),[],[],Fs,'psd');
```

Construct the theoretical PSD of the pink noise process.

```
PSDPink = 1./F(2:end);
```

Display the Welch PSD estimate of the noise along with the theoretical PSD on a log-log plot. Plot the frequency axis with a base-2 logarithmic scale to clearly show the octaves. Plot the PSD estimate in dB,  $10\log_{10}$ .

```
plot(log2(F(2:end)),10*log10(Pxx(2:end)))  
hold on  
plot(log2(F(2:end)),10*log10(PSDPink),'r',linewidth=2)  
xlabel('log_2(Hz)')  
ylabel('dB')  
title('Pink Noise')  
grid on  
legend('PSD estimate','Theoretical pink noise PSD')  
hold off
```



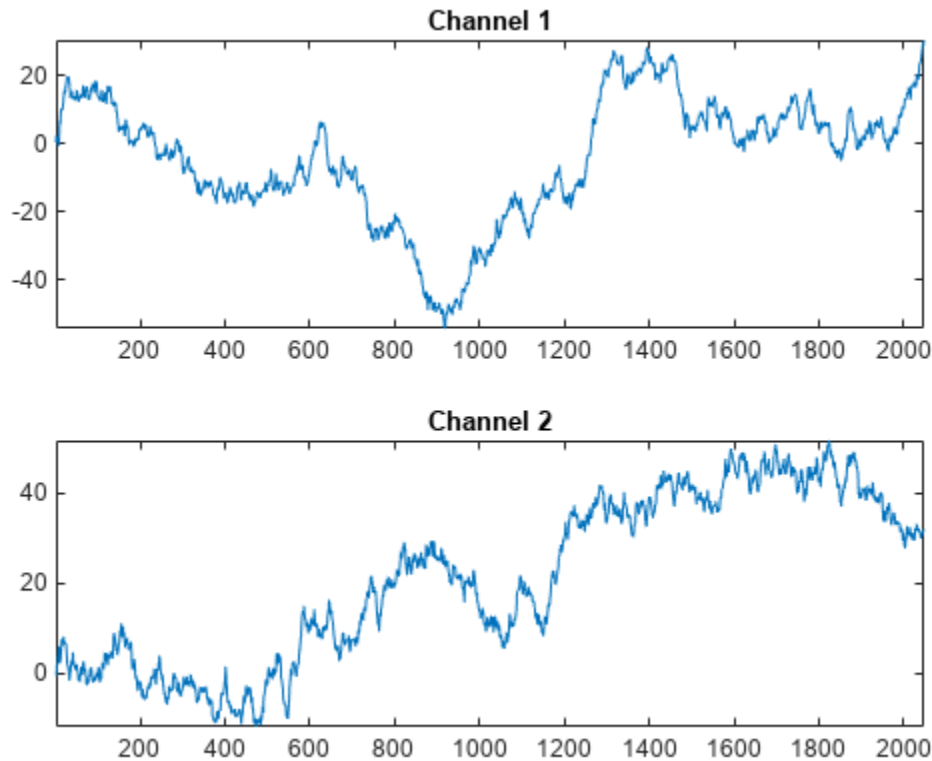
### Two-Channel Brownian Noise

Generate two channels of Brownian noise by setting `Color` to `'brown'` and `NumChannels` to 2. Specify the number of samples in each frame as an input while running the object algorithm.

```
cn = dsp.ColoredNoise('brown',SamplesPerFrameSource='Input port',...
    NumChannels=2)
```

```
cn =
    dsp.ColoredNoise with properties:
        Color: 'brown'
        BoundedOutput: false
        NumChannels: 2
        SamplesPerFrameSource: 'Input port'
        MaxSamplesPerFrame: 192000
        OutputDataType: 'double'
        RandomStream: 'Global stream'
```

```
x = cn(2048);
subplot(2,1,1)
plot(x(:,1)); title('Channel 1'); axis tight;
subplot(2,1,2)
plot(x(:,2)); title('Channel 2'); axis tight;
```

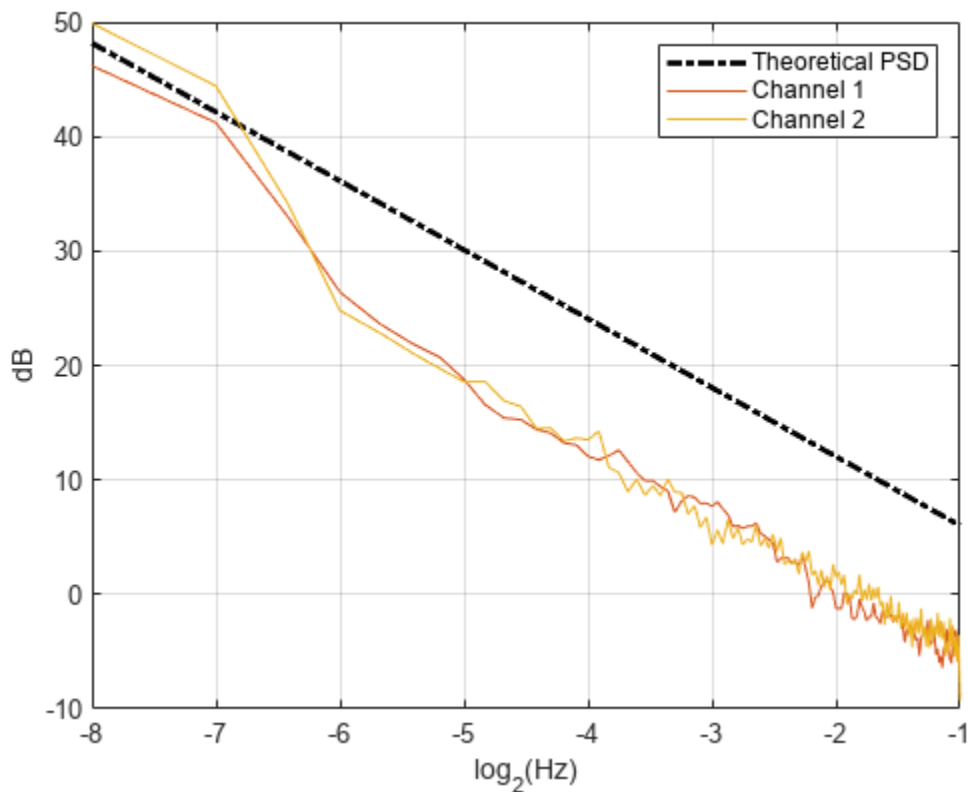


The sample rate is 1 Hz. Obtain Welch PSD estimates for both channels. The fourth argument of `pwelch`, `NFFT`, which is the number of FFT points, is empty. Hence, `NFFT` is set to 256. For even `NFFT`, The number of FFT points used to calculate the PSD estimate is  $(NFFT/2+1)$ , which equals 129.

```
Fs = 1;
Pxx = zeros(129,size(x,2));
for nn = 1:size(x,2)
[Pxx(:,nn),F] = pwelch(x(:,nn),hamming(128),[],[],Fs,'psd');
end
```

Construct the theoretical PSD of a Brownian process. Plot the theoretical PSD along with both realizations on a log-log plot. Use a base-2 logarithmic scale for the frequency axis and plot the power spectral densities in dB.

```
PSDBrownian = 1./F(2:end).^2;
figure;
plot(log2(F(2:end)),10*log10(PSDBrownian),'k-.',linewidth=2);
hold on;
plot(log2(F(2:end)),10*log10(Pxx(2:end,:)));
xlabel('log_2(Hz)'); ylabel('dB');
grid on;
legend('Theoretical PSD','Channel 1', 'Channel 2');
```



### Add Pink Noise at 0 dB SNR

**Note:** The `audioDeviceWriter` System object™ is not supported in MATLAB Online.

This example shows how to stream in an audio file and add pink noise at a 0 dB signal-to-noise ratio (SNR). The example reads in frames of an audio file 1024 samples in length, measures the root mean square (RMS) value of the audio frame, and adds pink noise with the same RMS value as the audio frame.

Set up the System objects. Set 'SamplesPerFrame' for both the file reader and the colored noise generator to 1024 samples. Set `Color` to 'pink' to generate pink noise with a  $1/|f|$  power spectral density.

```
N = 1024;
afr = dsp.AudioFileReader(FileName='speech_dft.mp3', ...
    SamplesPerFrame=N);
adw = audioDeviceWriter(SampleRate=afr.SampleRate);
cn = dsp.ColoredNoise('pink', SamplesPerFrame=N);
```

Stream the audio file in 1024 samples at a time. Measure the signal RMS value for each frame, generate a frame of pink noise equal in length, and scale the RMS value of the pink noise to match the signal. Add the scaled noise to the signal and play the output.

```
while ~isDone(afr)
    audio = afr();
```

```
    speechRMS = rms(audio);
    noise = cn();
    noiseRMS = rms(noise);
    noise = noise*(speechRMS/noiseRMS);
    sigPlusNoise = audio+noise;
    adw(sigPlusNoise);
end
release(afr);
release(adw);
```

### **Averaged Power Spectrum of Pink Noise**

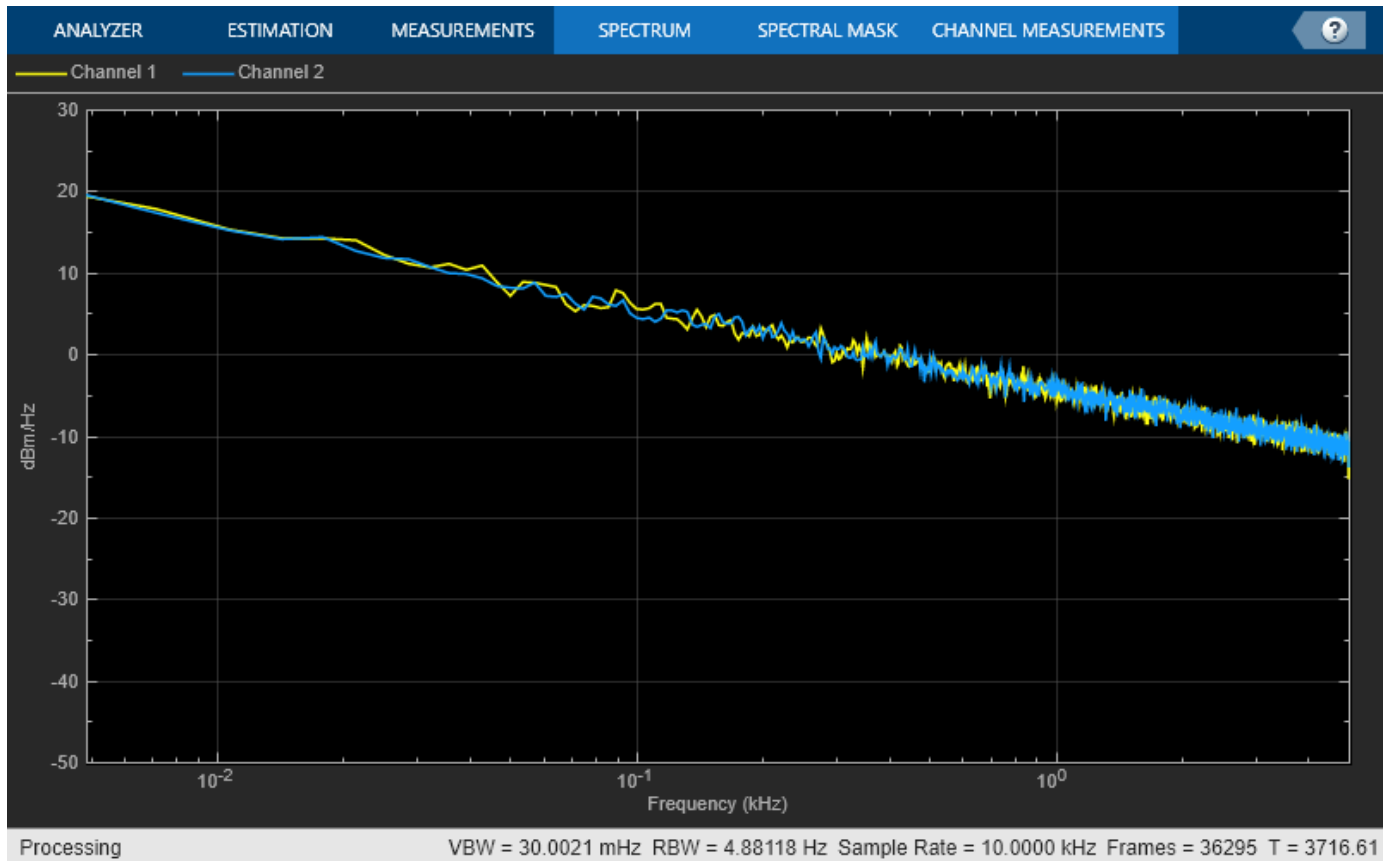
Generate two-channels of pink noise and compute the power density spectrum.

Set up the colored noise generator to generate two channels of pink noise with 1024 samples. Set up the spectrum analyzer to compute modified periodograms using a Hamming window and 50% overlap.

```
pinkNoise = dsp.ColoredNoise('pink',1024,2);
sa = spectrumAnalyzer(SpectrumType='power-density',...
    Method='welch',...
    AveragingMethod='exponential',...
    ForgettingFactor=0.95,...
    OverlapPercent=50,Window='hamming',...
    PlotAsTwoSidedSpectrum=false, ...
    FrequencyScale='log',YLimits=[-50 30]);
```

Run the simulation for 30 seconds.

```
tic
while toc < 30
    pink = pinkNoise();
    sa(pink);
end
```



## More About

### Colored Noise Processes

Many phenomena in diverse fields, such as hydrology and finance, produce time series with PSD functions that follow a power law of the form

$$S(f) = \frac{L(f)}{|f|^\alpha}$$

where  $\alpha$  is a real number in the interval  $[-2, 2]$  and  $L(f)$  is a positive, slowly-varying or constant function. Plotting the PSD of such processes on a log-log plot displays an approximate linear relationship between the log frequency and log PSD with slope equal to  $-\alpha$

$$\ln S(f) = -\alpha \ln |f| + \ln L(f).$$

It is often convenient to plot the PSD in dB as a function of the frequency on a base-2 logarithmic scale. The slope of the plot is then dB/octave. Rewriting the preceding equation, you obtain

$$10 \log S(f) = -10\alpha \frac{\ln(2) \log_2(f)}{\ln(10)} + 10 \frac{\ln(L(f))}{\ln(10)}$$

with the slope in dB/octave given by

$$-10\alpha \frac{\ln(2)\log_2(f)}{\ln(10)}$$

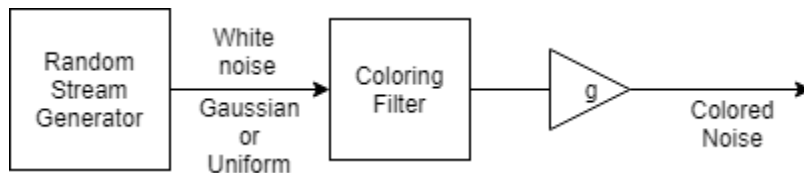
If  $\alpha > 0$ ,  $S(f)$  goes to infinity as the frequency,  $f$ , approaches 0. Stochastic processes with PSDs of this form exhibit long memory. Long-memory processes have autocorrelations that persist for a long time as opposed to decaying exponentially like many common time-series models. If  $\alpha < 0$ , the process is antipersistent and exhibits negative correlation between increments [1].

Special examples of  $\frac{1}{|f|^\alpha}$  processes include:

- $\alpha = 0$  — White noise, where  $L(f)$  is a constant proportional to the process variance.
- $\alpha = 1$  — Pink, or flicker noise. Pink noise has equal energy per octave. See “Measure Pink Noise Power in Octave Bands” on page 4-274 for a demonstration. The power spectral density of pink noise decreases 3 dB per octave.
- $\alpha = 2$  — brown noise, or Brownian motion. Brownian motion is a nonstationary process with stationary increments. You can think of Brownian motion as the integral of a white noise process. Even though Brownian motion is nonstationary, you can still define a generalized power spectrum, which behaves like  $\frac{1}{|f|^2}$ . Accordingly, power in a brown noise decreases 6 dB per octave.
- $\alpha = -1$  — blue noise. The power spectral density of blue noise increases 3 dB per octave.
- $\alpha = -2$  — violet, or purple noise. The power spectral density of violet noise increases 6 dB per octave. You can think of violet noise as the derivative of white noise process.

## Algorithms

The figure shows the overall process of generating the colored noise.



The random stream generator produces a stream of white noise that is either Gaussian or uniform in distribution. A coloring filter applied to the white noise generates colored noise with a power spectral density (PSD) function given by:

$$S(f) = \frac{L(f)}{|f|^\alpha}$$

When  $\alpha$ , the inverse frequency power, equals 0, no coloring filter is applied to the output of the random stream generator. If the bounded option is enabled, the output is uniform white noise with amplitude between +1 and -1. If the bounded output is not enabled, the output is a Gaussian white noise and the values are not bounded between +1 and -1. If  $\alpha$  is set to any other value, then a coloring filter is applied to the output of the random stream generator. If the bounded output option is enabled, a gain  $g$  is applied to the output of the coloring filter to ensure that the absolute maximum output never exceeds 1.

For details on colored noise processes and how the value of  $\alpha$  affects the PSD of the colored noise, see “Colored Noise Processes” on page 4-281.



When the inverse frequency power  $\alpha$  is positive, the colored noise is generated using an autoregressive (AR) model of order 63. The AR coefficients are:

$$a_0 = 1,$$

$$a_k = (k - 1 - \frac{\alpha}{2}) \frac{a_{k-1}}{k}, \quad k = 1, 2, \dots, 63$$

Pink and brown noises are special cases, which are generated from specially tuned SOS filters of orders 12 and 10, respectively. These filters are optimized for better performance.

When the inverse frequency power  $\alpha$  is negative, the colored noise is generated using a moving average (MA) model of order 255. The MA coefficients are:

$$b_0 = 1,$$

$$b_k = (k - 1 + \frac{\alpha}{2}) \frac{b_{k-1}}{k}, \quad k = 1, 2, \dots, 255$$

Purple noise is generated from a first order filter,  $B = [1 \ -1]$ .

The coloring filters applied (except pink, brown, and purple) are detailed on pp. 820-822 in [2].

## Version History

### Introduced in R2014a

### R2022b: Variable-sized output frames in dsp.ColoredNoise object

Starting in R2022b, dsp.ColoredNoise object can generate colored noise signals of variable-sized output frames.

You can specify the frame size of the output colored noise signal as a nonnegative scalar value that is less than or equal to the MaxSamplesPerFrame property.

## References

- [1] Beran, J., Y. Feng, S. Ghosh, and R. Kulik, *Long-Memory Processes: Probabilistic Properties and Statistical Methods*. New York: Springer, 2013.
- [2] Kasdin, N.J. "Discrete Simulation of Colored Noise and Stochastic Processes and  $1/f^\alpha$  Power Law Noise Generation." *Proceedings of the IEEE*, Vol. 83, No. 5, 1995, pp. 802-827.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

## **See Also**

### **Functions**

randn

### **Blocks**

Colored Noise

# dsp.ComplexBandpassDecimator

**Package:** dsp

Extract a frequency subband using a one-sided (complex) bandpass decimator

## Description

The `dsp.ComplexBandpassDecimator` System object extracts a specific sub-band of frequencies using a one-sided, multistage, complex bandpass decimator. The object determines the bandwidth of interest using the specified `CenterFrequency`, `DecimationFactor` and `Bandwidth` values.

To extract a frequency subband using a complex bandpass decimator:

- 1 Create the `dsp.ComplexBandpassDecimator` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
bpdecim = dsp.ComplexBandpassDecimator
bpdecim = dsp.ComplexBandpassDecimator(d)
bpdecim = dsp.ComplexBandpassDecimator(d,Fc)
bpdecim = dsp.ComplexBandpassDecimator(d,Fc,Fs)
bpdecim = dsp.ComplexBandpassDecimator(Name,Value)
```

### Description

`bpdecim = dsp.ComplexBandpassDecimator` creates a System object that filters each channel of the input over time using a one-sided, multistage, complex bandpass decimation filter. The object determines the bandwidth of interest using the default center frequency, decimation factor, and bandwidth values.

`bpdecim = dsp.ComplexBandpassDecimator(d)` creates a complex bandpass decimator object with the `DecimationFactor` property set to `d`.

`bpdecim = dsp.ComplexBandpassDecimator(d,Fc)` creates a complex bandpass decimator object with the `DecimationFactor` property set to `d`, and the `CenterFrequency` property set to `Fc`.

`bpdecim = dsp.ComplexBandpassDecimator(d,Fc,Fs)` creates a complex bandpass decimator object with the `DecimationFactor` property set to `d`, the `CenterFrequency` property set to `Fc`, and the `SampleRate` property set to `Fs`.

Example: `dsp.ComplexBandpassDecimator(48e3/1e3,2e3,48e3);`

`bpdecim = dsp.ComplexBandpassDecimator(Name, Value)` creates a complex bandpass decimator object with each specified property set to the specified value. Enclose each property name in quotes. You can use this syntax with any previous input argument combinations.

Example: `dsp.ComplexBandpassDecimator(48e3/1e3, 2e3, 48e3, 'CenterFrequency', 1e3);`

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see *System Design in MATLAB Using System Objects*.

### CenterFrequency — Center frequency in Hz

0 (default) | real scalar

Center frequency of the desired band in Hz, specified as a real, finite numeric scalar in the range  $[-\text{SampleRate}/2, \text{SampleRate}/2]$ .

**Tunable:** Yes

Data Types: `single` | `double`

### Specification — Filter design parameters

'Decimation factor' (default) | 'Bandwidth' | 'Decimation factor and bandwidth'

Filter design parameters, specified as:

- 'Decimation factor' -- The object specifies the decimation factor through the `Decimation Factor` property. The bandwidth of interest ( $BW$ ) is computed using the following equation:

$$BW = Fs/D$$

where

- $Fs$  -- Sample rate specified through `SampleRate` property.
- $D$  -- Decimation factor.
- 'Bandwidth' -- The object specifies the bandwidth through the `Bandwidth` property. The decimation factor ( $D$ ) is computed using the following equation:

$$D = \text{floor}\left(\frac{Fs}{BW + TW}\right)$$

where

- $Fs$  -- Sample rate specified through `SampleRate` property.
- $BW$  -- Bandwidth of interest.
- $TW$  -- Transition width specified through the `TransitionWidth` property.
- 'Decimation factor and bandwidth' -- The decimation factor and the bandwidth of interest are specified through the `DecimationFactor` and `Bandwidth` properties.

**DecimationFactor — Decimation factor**

2 (default) | positive integer

Factor by which to reduce the bandwidth of the input signal, specified as a positive integer. The frame size (number of rows) of the input signal must be a multiple of the decimation factor.

**Dependencies**

This property applies when you set `Specification` to either 'Decimation factor' or 'Decimation factor and bandwidth'.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**StopbandAttenuation — Stopband attenuation in dB**

80 (default) | positive scalar

Stopband attenuation of the filter in dB, specified as finite positive scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**TransitionWidth — Transition width in Hz**

100 (default) | positive scalar

Transition width of the filter in Hz, specified as a positive scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**Bandwidth — Bandwidth (Hz)**

5000 (default) | real positive scalar

Width of the frequency band of interest, specified as a real positive scalar in Hz.

**Dependencies**

This property applies when you set `Specification` to either 'Bandwidth' or 'Decimation factor and bandwidth'.

Data Types: `single` | `double`

**PassbandRipple — Passband ripple (dB)**

1 (default) | positive scalar

Passband ripple of the filter, specified as a positive scalar in dB.

**Dependencies**

This property applies when you set `Specification` to either 'Bandwidth' or 'Decimation factor and bandwidth'.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**MinimizeComplexCoefficients — Flag to minimize number of complex coefficients**`true` (default) | `false`

Flag to minimize the number of complex filter coefficients, specified as:

- `true` -- The first stage of the multistage filter is bandpass (complex coefficients) centered at the specified center frequency. The first stage is followed by a mixing stage that heterodynes the signal to DC. The remaining filter stages, all with real coefficients, follow.

- `false` -- The input signal is first passed through the different stages of the multistage filter. All stages are bandpass (complex coefficients). The signal is then heterodyned to DC if `MixToBaseband` is `true`, and the frequency offset resulting from the decimation is nonzero.

**MixToBaseband — Flag to mix signal to baseband**`true (default) | false`

Flag to mix the signal to baseband, specified as:

- `true` -- The object heterodynes the filtered, decimated signal to DC. This mixing stage runs at the output sample rate of the filter.
- `false` -- The object skips the mixing stage.

**Dependencies**

This property applies when you set `MinimizeComplexCoefficients` to `false`.

**SampleRate — Input sample rate in Hz**`44100 (default) | real positive scalar`

Sampling rate of the input signal in Hz, specified as a real positive scalar.

Data Types: `single` | `double`

**Usage****Syntax**`y = bpdecim(x)`**Description**

`y = bpdecim(x)` filters the real or complex input signal, `x`, to produce the output, `y`. The output contains the subband of frequencies specified by the System object properties. The System object filters each channel of the input signal independently over time. The frame size (first dimension) of `x` must be a multiple of the decimation factor.

**Input Arguments****x — Data input**`vector | matrix`

Data input, specified as a vector or a matrix of size  $P$ -by- $Q$ . The number of input rows  $P$  can be arbitrary and does not have to be a multiple of the decimation factor.

This object supports variable-size input signals, that is, the frame length (number of rows) of the signal can change even when the object is locked. However, the number of channels (columns) must remain constant.

Data Types: `single` | `double`

**Output Arguments****y — Filtered output**`vector | matrix`

Output of the complex bandpass decimator, returned as a vector or a matrix. The output contains the subband of frequencies specified by the System object properties.

When the input is of size  $P$ -by- $Q$ , and  $P$  is not a multiple of the decimation factor  $D$ , the output signal has an upper bound size of  $\text{ceil}(P/D)$ -by- $Q$ , where  $D$  is the decimation factor. If  $P$  is a multiple of the decimation factor, then the output is of size  $(P/D)$ -by- $Q$ . The number of channels (columns) does not change.

The data type of the output is same as the data type of the input. The output signal is always complex.

Data Types: `single` | `double`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Specific to dsp.ComplexBandpassDecimator

|                                    |   |
|------------------------------------|---|
| <code>cost</code>                  | Implementation cost of the complex bandpass decimator |
| <code>freqz</code>                 | Frequency response of the multirate multistage filter |
| <code>info</code>                  | Information about filter System object                |
| <code>visualizeFilterStages</code> | Visualize filter stages                               |

## Common to All System Objects

|                      |  |
|----------------------|--|
| <code>step</code>    | Run System object algorithm  |
| <code>release</code> | Release resources and allow changes to System object property values and input characteristics |
| <code>reset</code>   | Reset internal states of System object   |

## Examples

### Compute Cost of Complex Bandpass Decimator

Compute the implementation cost of a complex bandpass decimator using the `cost` function.

Create a `dsp.ComplexBandpassDecimator` object. Set the `DecimationFactor` to 12, the `CenterFrequency` to 5000 Hz, and the `SampleRate` to 44,100 Hz.

```
cbp = dsp.ComplexBandpassDecimator(12,5000,44100)
```

```
cbp =
```

```
  dsp.ComplexBandpassDecimator with properties:
```

```

        CenterFrequency: 5000
        Specification: 'Decimation factor'
        DecimationFactor: 12
        StopbandAttenuation: 80
        TransitionWidth: 100
        MinimizeComplexCoefficients: true
        SampleRate: 44100
```

Compute the implementation cost of `cbp` using the `cost` function.

```
c = cost(cbp)

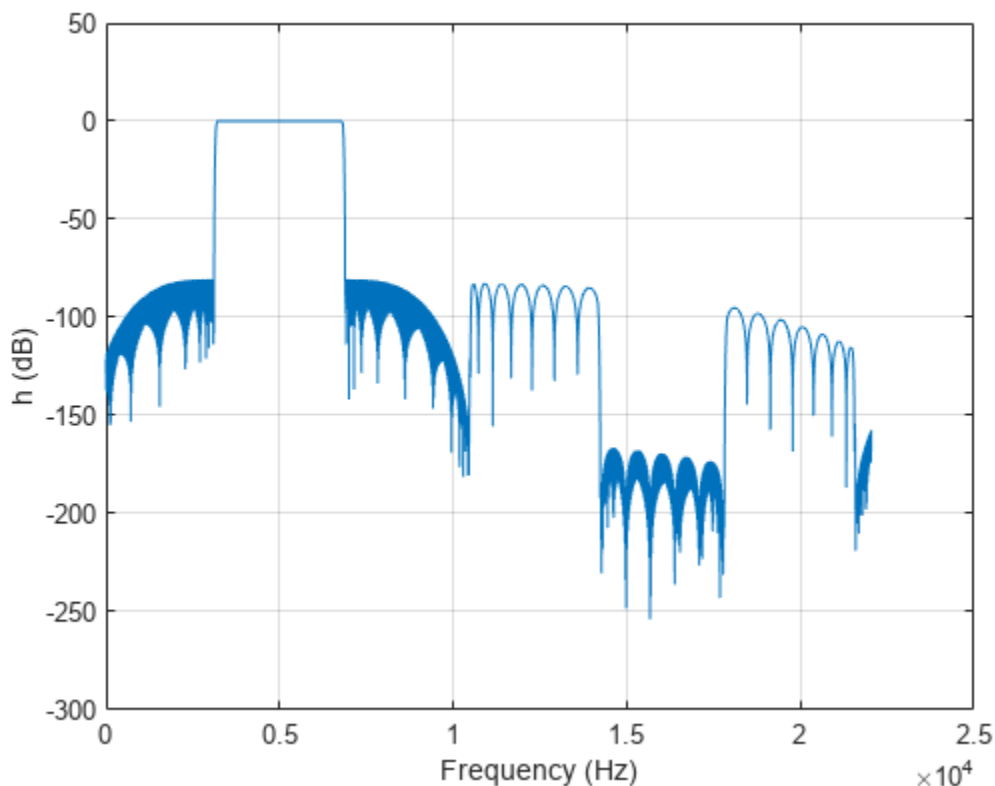
c = struct with fields:
    NumCoefficients: 201
    NumStates: 379
    RealMultiplicationsPerInputSample: 44.3333
    RealAdditionsPerInputSample: 43.8333
```

### Compute Frequency Response of Complex Bandpass Decimator

Compute the complex frequency response of a complex bandpass decimator using the `freqz` function.

Create a `dsp.ComplexBandpassDecimator` object. Set the `DecimationFactor` to 12, the `CenterFrequency` to 5000 Hz, and the `SampleRate` to 44100 Hz. Compute and display the frequency response.

```
cbp = dsp.ComplexBandpassDecimator(12,5000,44100);
[h,f] = freqz(cbp);
plot(f,20*log10(abs(h)))
grid on
xlabel('Frequency (Hz)')
ylabel('h (dB)')
```





## Filter Signal Through Complex Bandpass Decimator

Filter an input signal through a complex bandpass decimator and visualize the filtered spectrum in a spectrum analyzer.

### Initialization

Create a `dsp.ComplexBandpassDecimator System` object™ with center frequency set to 2000 Hz, bandwidth of interest set to 1000 Hz, and sample rate set to 48 kHz. The decimation factor is computed as the ratio of the sample rate to the bandwidth of interest. The input to the decimator is a sine wave with a frame length of 1200 samples with tones at 1625 Hz, 2000 Hz, and 2125 Hz. Create a `spectrumAnalyzer` scope to visualize the signal spectrum.

```
Fs = 48e3;
CF = 2000;
BW = 1000;
D = Fs/BW;
FrameLength = 1200;
bpdecim = dsp.ComplexBandpassDecimator(D,CF,Fs);

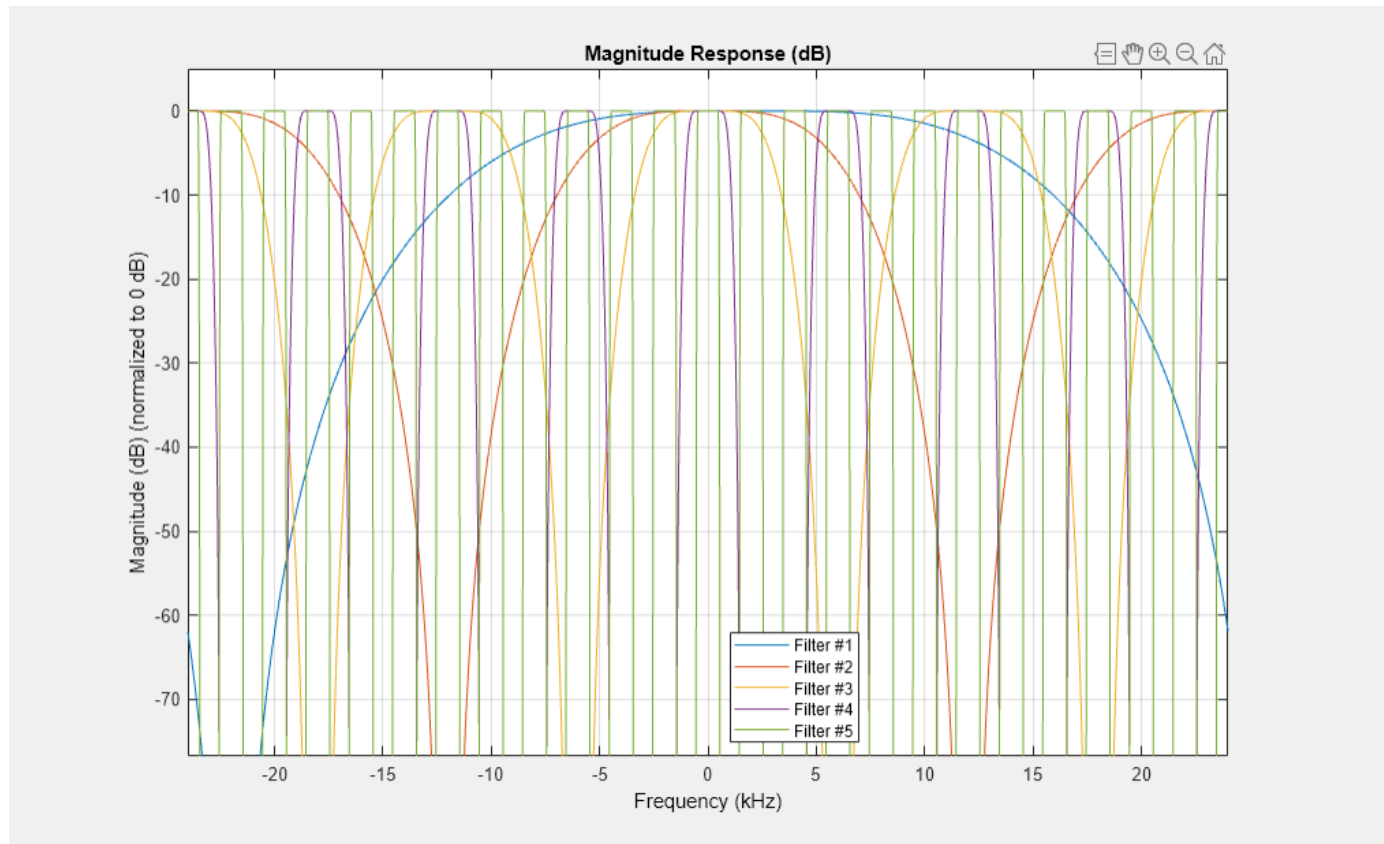
sa = spectrumAnalyzer('SampleRate',Fs/D,...
    'Method','welch',...
    'YLimits',[-120 40],...
    'FrequencyOffset',CF);

tones = [1625 2000 2125];
sin = dsp.SineWave('SampleRate',Fs,'Frequency',tones,...
    'SamplesPerFrame',FrameLength);
```

### Visualize Filter Stages

Using the `visualizeFilterStages` function, you can visualize the response of each individual filter stage using FVTool.

```
visualizeFilterStages(bpdecim)
```



### Display Filter info

The `info` function displays information about the bandpass decimator.

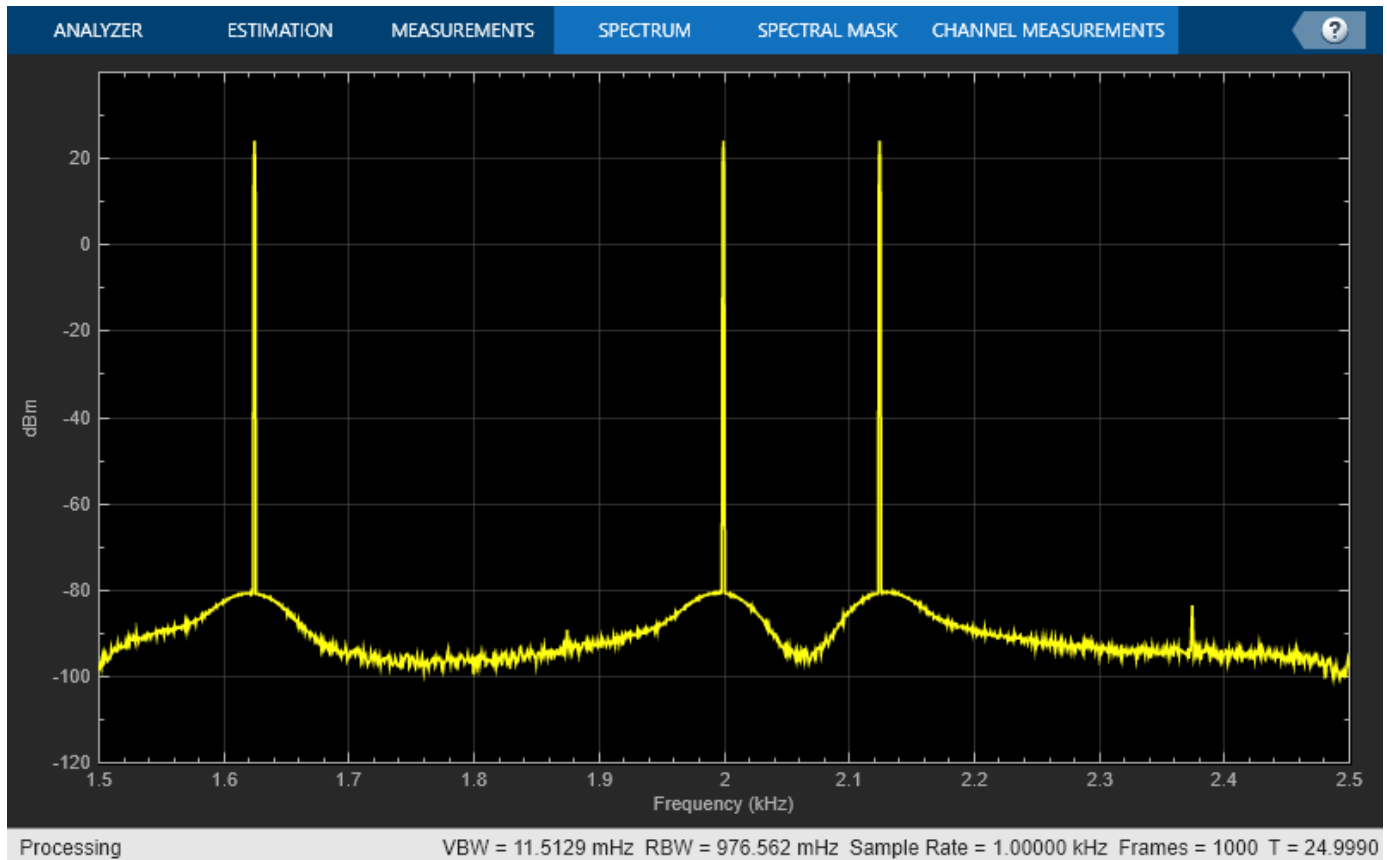
```
fprintf('%s',info(bpdecim))
```

```
Overall Decimation Factor      : 48
Bandwidth                      : 1000 Hz
Number of Filters              : 5
Real multiplications per Input Sample: 14.708333
Real additions per Input Sample  : 13.833333
Number of Coefficients         : 89
Filters:
  Filter 1:
  dsp.FIRDecimator            - Decimation Factor   : 2
  Filter 2:
  dsp.FIRDecimator            - Decimation Factor   : 2
  Filter 3:
  dsp.FIRDecimator            - Decimation Factor   : 2
  Filter 4:
  dsp.FIRDecimator            - Decimation Factor   : 3
  Filter 5:
  dsp.FIRDecimator            - Decimation Factor   : 2
```

### Stream In and Filter Signal

Construct a for-loop to run for 1000 iterations. In each iteration, stream in 1200 samples (one frame) of the noisy sine wave and apply the complex bandpass decimator on each frame of the input signal. Visualize the input and output spectrum in the spectrum analyzer, sa.

```
for index = 1:1000
    x = sum(sin(),2) + 1e-4*randn(FrameLength,1);
    z = bpdecim(x);
    sa(z);
end
```



The bandpass decimator with center frequency at 2000 Hz and a bandwidth of 1000 Hz passes the three sine wave tones at 1625 Hz, 2000 Hz, and 2125 Hz.

Change the center frequency of the decimator to 2400 Hz and filter the signal.

```
release(bpdecim);
bpdecim.CenterFrequency = 2400

bpdecim =
    dsp.ComplexBandpassDecimator with properties:

        CenterFrequency: 2400
        Specification: 'Decimation factor'
        DecimationFactor: 48
        StopbandAttenuation: 80
```

```

TransitionWidth: 100
MinimizeComplexCoefficients: true
SampleRate: 48000

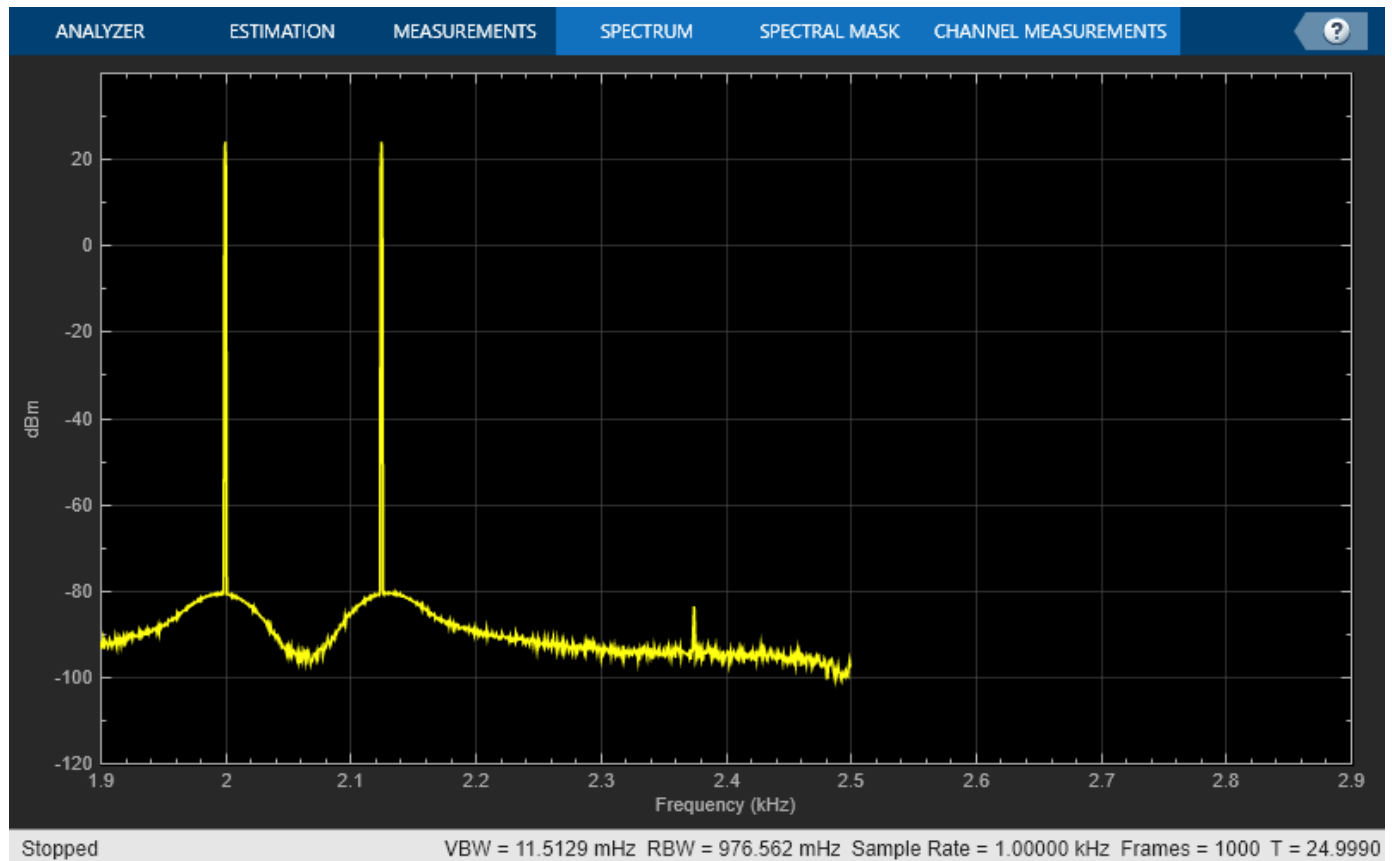
```

Configure the spectrum analyzer to show the bandwidth of interest, [-1900, 2900] Hz.

```

release(sa)
sa.FrequencyOffset = 2400;

```

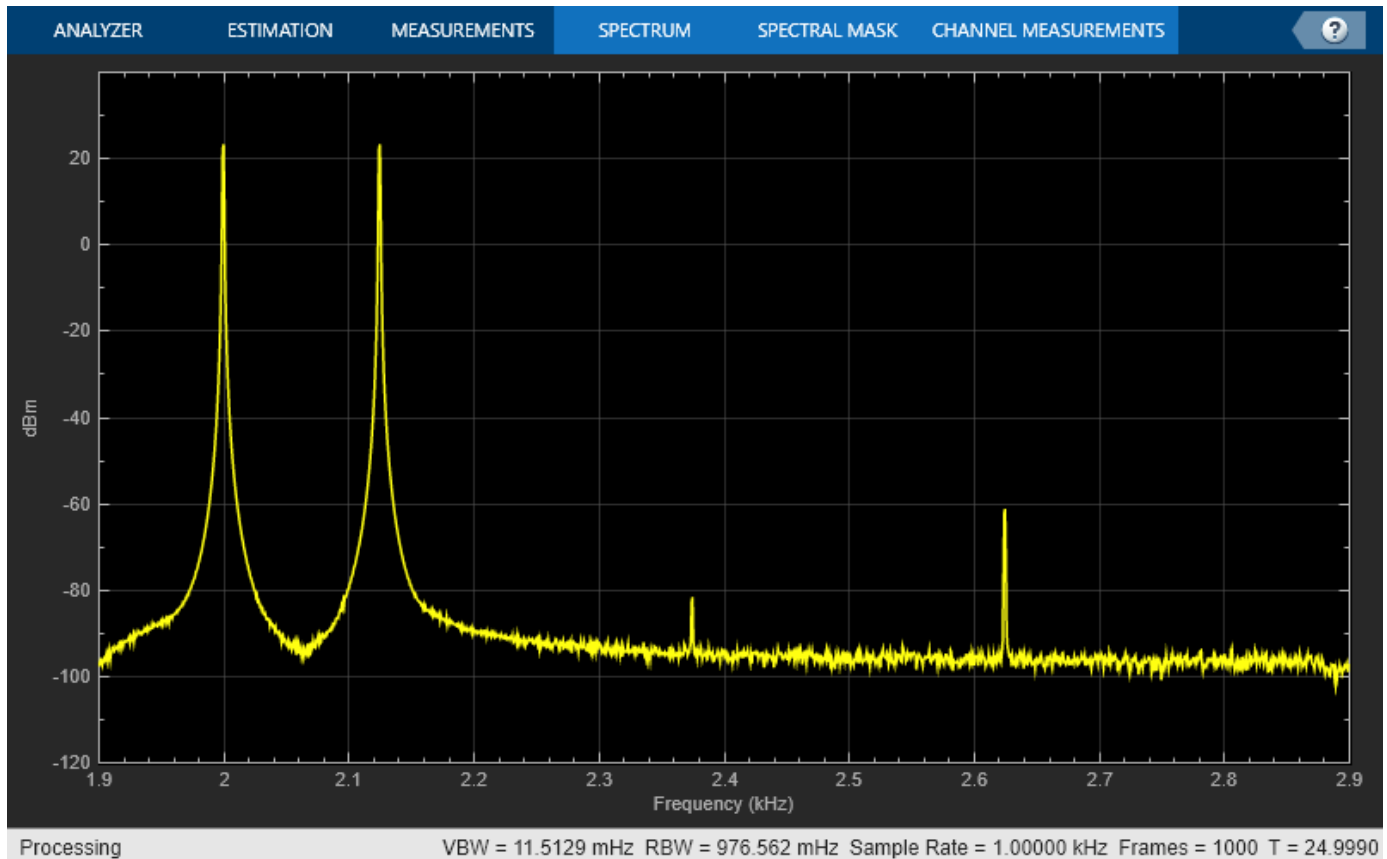


Stream in the data and filter the signal.

```

for index = 1:1000
    x = sum(sin(),2) + 1e-4 * randn(FrameLength,1);
    z = bpdecim(x);
    sa(z);
end

```



The tones at 2000 Hz and 2125 Hz are passed through the decimator, while the tone at 1625 Hz is filtered out.

## Algorithms

The complex bandpass decimator is designed by applying a complex frequency shift transformation on a lowpass prototype filter. The lowpass prototype in this case is a multirate, multistage finite impulse response (FIR) filter. The desired frequency shift applies only to the first stage. Subsequent stages scale the desired frequency shift by their respective cumulative decimation factors. For details, see “Complex Bandpass Filter Design” and “Zoom FFT”.

## Version History

**Introduced in R2018a**

### **R2023a: Support for arbitrary input frame length**

This object supports an input signal with an arbitrary frame length, so the input frame length does not have to be a multiple of the decimation factor.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- See “System Objects in MATLAB Code Generation” (MATLAB Coder).
- This object also supports SIMD code generation using Intel AVX2 technology under these conditions:
  - Input signal is complex-valued.
  - Input signal has a data type of `single` or `double`.

The SIMD technology significantly improves the performance of the generated code.

- In the code you generate from this object, the decimator output is a fixed-size signal if the input signal is fixed size and is a multiple of the decimation factor. If the input is not a multiple of the decimation factor or if the input is a variable-size signal, then the decimator output is of variable-size.

## See Also

### Functions

`cost` | `freqz` | `info` | `visualizeFilterStages`

### Blocks

Complex Bandpass Decimator

### Topics

“IF Subsampling with Complex Multirate Filters”

“Complex Bandpass Filter Design”

“Zoom FFT”

# dsp.Convolver

**Package:** dsp

(Removed) Convolution of two signals

---

**Note** `dsp.Convolver` has been removed. Use `conv` instead. For more information, see “Compatibility Considerations”.

---

## Description

The `dsp.Convolver` System object convolves the first dimension of an  $N$ -D input array,  $u$ , with the first dimension of an  $N$ -D input array,  $v$ . You can convolve the inputs in the time domain or frequency domain. In the time domain, the object convolves the first input with the second input. In the frequency domain, the object multiplies the Fourier transforms of both the inputs, and computes the inverse Fourier transform of the product. In this domain, depending on the input length, the object can require fewer computations. For more information on the two computation methods, see “Algorithms” on page 4-303.

To convolve two inputs:

- 1 Create the `dsp.Convolver` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
cnv = dsp.Convolver
cnv = dsp.Convolver(Name, Value)
```

### Description

`cnv = dsp.Convolver` creates a convolution System object, `cnv`, to convolve two inputs in the time domain or frequency domain.

`cnv = dsp.Convolver(Name, Value)` creates a convolution System object, `cnv`, with each specified property set to the specified value. Enclose each property name in single quotes.

Example: `cnv = dsp.Convolver('Method', 'Frequency Domain')`

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

**Method — Domain for computing convolutions**

'Time Domain' (default) | 'Frequency Domain' | 'Fastest'

Domain in which the System object computes convolutions, specified as one of the following:

- 'Time Domain' -- Computes the convolutions in the time domain, which minimizes memory usage. For more information, see “Time-Domain Computation” on page 4-303.
- 'Frequency Domain' -- Computes the convolutions in the frequency domain, which can require fewer computations depending on the input length. For more information, see “Frequency-Domain Computation” on page 4-304.
- 'Fastest' -- Computes the convolutions in the domain that minimizes the number of computations.

---

**Note** Fixed-point signals are supported for the time domain only. To use the following fixed-point properties, set Method to 'Time Domain'.

---

**Fixed-Point Properties****FullPrecisionOverride — Full-precision override for fixed-point arithmetic**

true (default) | false

Flag to use full-precision rules for fixed-point arithmetic, specified as one of the following:

- true -- The object computes all internal arithmetic and output data types using the full-precision rules. These rules provide the most accurate fixed-point numerics. In this mode, other fixed-point properties do not apply. No quantization occurs within the object. Bits are added, as needed, to ensure that no roundoff or overflow occurs.
- false -- Fixed-point data types are controlled through individual fixed-point property settings.

For more information, see “Full Precision for Fixed-Point System Objects” and “Set System Object Fixed-Point Properties”.

**RoundingMethod — Rounding method**

'Floor' (default) | 'Ceiling' | 'Convergent' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'

Select the rounding mode for fixed-point operations.

**Dependencies**

This property applies when you set the FullprecisionOverride property to false and at least one of the ProductDataType, AccumulatorDataType, and OutputDataType properties to any option other than 'Full precision'.

**OverflowAction — Overflow action**

'Wrap' (default) | 'Saturate'

The overflow action for fixed-point operations, specified as one of the following:

- 'Wrap' -- The object wraps the result of its fixed-point operations.



- 'Saturate' -- The object saturates the result of its fixed-point operations.

For more details on overflow actions, see overflow mode for fixed-point operations.

#### Dependencies

This property applies when you set the `FullprecisionOverride` property to `false` and at least one of the `ProductDataType`, `AccumulatorDataType`, and `OutputDataType` properties to any option other than 'Full precision'.

#### ProductDataType — Product output data type

'Full precision' (default) | 'Custom' | 'Same as first input'

Data type of the output of a product operation in the `dsp.Convolver` object, specified as one of the following:

- 'Full precision' -- The object computes the product output data type using the full-precision rules. These rules provide the most accurate fixed-point numerics. No quantization occurs. Bits are added, as needed, to ensure that no roundoff or overflow occurs.
- 'Custom' -- The product output data type is specified as a custom numeric type through the `CustomProductDataType` property. The rounding method and the overflow action are specified through the `RoundingMethod` and `OverflowAction` properties.
- 'Same as first input' -- The object specifies the product output data type to be the same as the first input data type.

For more information on the product output data type, see the “Fixed Point” on page 4-302 section.

#### CustomProductDataType — Product word and fraction lengths

`numericType([],32,30)` (default)

The product output data type, specified as an autosigned numeric type with a word length of 32 and a fraction length of 30.

#### Dependencies

This property applies only when you set `ProductDataType` to 'Custom'.

#### AccumulatorDataType — Accumulator data type

'Full precision' (default) | 'Custom' | 'Same as first input' | 'Same as product'

Data type of the output of an accumulation operation in the `dsp.Convolver` object, specified as one of the following:

- 'Full precision' -- The object computes the accumulator data type using the full precision rules. These rules provide the most accurate fixed-point numerics. No quantization occurs. Bits are added, as needed, to ensure that no roundoff or overflow occurs.
- 'Custom' -- The accumulator data type is specified as a custom numeric type through the `CustomAccumulatorDataType` property. The rounding method and the overflow action are specified through the `RoundingMethod` and the `OverflowAction` properties.
- 'Same as first input' -- The object specifies the accumulator data type to be the same as the first input data type.
- 'Same as product' -- The object specifies the accumulator data type to be the same as the product data type.

For more information on the accumulator data type, see the “Fixed Point” on page 4-302 section.

**CustomAccumulatorDataType — Accumulator word and fraction lengths**

`numericType([],32,30)` (default)

The accumulator data type, specified as an auto signed numeric type with a word length of 32 and a fraction length of 30.

**Dependencies**

This property applies only when you set `AccumulatorDataType` to 'Custom'.

**OutputDataType — Output data type**

'Same as accumulator' (default) | 'Custom' | 'Same as first input' | 'Same as product'

Data type of the output of the `dsp.Convolver` object, specified as one of the following:

- 'Same as accumulator' -- The object specifies the output data type to be the same as the accumulator data type. For more details on the accumulator data type, see the “AccumulatorDataType” on page 4-0 property.
- 'Custom' -- The output data type is specified as a custom numeric type through the `CustomOutputDataType` property. The rounding method and the overflow action are specified through the `RoundingMethod` and the `OverflowAction` properties.
- 'Same as first input' -- The object specifies the output data type to be the same as the first input data type.
- 'Same as product' -- The object specifies the output data type to be the same as the product data type.

For more information on the output data type, see the “Fixed Point” on page 4-302 section.

**CustomOutputDataType — Output word and fraction lengths**

`numericType([],16,15)` (default)

The output data type, specified as an autosigned numeric type with a word length of 16 and a fraction length of 15.

**Dependencies**

This property applies only when you set `OutputDataType` to 'Custom'.

**Usage****Syntax**

```
cnvOut = cnv(input1, input2)
```

**Description**

`cnvOut = cnv(input1, input2)` convolves the two inputs, `input1` and `input2`, along their first dimensions, and returns the convolved output, `cnvOut`.

## Input Arguments

### input1 — First data input

vector | matrix |  $N$ -D array

First data input, specified as a vector, matrix, or  $N$ -D array. If the input is a matrix or an array, all the dimensions of both the inputs, except for the first dimension, must be the same.

Example: `ones(10,3,2)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fi`

### input2 — Second data input

vector | matrix |  $N$ -D array

Second data input, specified as a vector, matrix, or  $N$ -D array. If the input is a matrix or an array, all the dimensions of both the inputs, except for the first dimension, must be the same.

Example: `randn(4,3,2)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fi`

## Output Arguments

### cnvOut — Convolved output

vector | matrix |  $N$ -D array

The convolved output of the two inputs, returned as a vector, matrix, or  $N$ -D array.

- When both inputs are  $N$ -D arrays, the size of their first dimension can differ, but the size of all other dimensions must be equal. For example, when  $u$  is an  $M_u$ -by- $N$ -by- $P$  array, and  $v$  is an  $M_v$ -by- $N$ -by- $P$  array, the output is an  $(M_u+M_v-1)$ -by- $N$ -by- $P$  array.
- When one input is a column vector and the other is an  $N$ -D array, the object independently convolves the vector with the first dimension of the  $N$ -D input array. For example, when  $u$  is an  $M_u$ -by-1 column vector and  $v$  is an  $M_v$ -by- $N$  matrix, the output is an  $(M_u+M_v-1)$ -by- $N$  matrix.
- When  $u$  and  $v$  are column vectors with lengths  $M_u$  and  $M_v$ , the object performs the vector convolution. The output is an  $(M_u+M_v-1)$ -by-1 column vector.

When both the inputs are real, the output is real. When one or both inputs are complex, the output is complex.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fi`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Common to All System Objects

|                      |  |
|----------------------|--|
| <code>step</code>    | Run System object algorithm  |
| <code>release</code> | Release resources and allow changes to System object property values and input characteristics |

reset     Reset internal states of System object

## Examples

### Convolve Two Rectangular Sequences

Create a `dsp.Convolver` object.

```
conv = dsp.Convolver
conv =
  dsp.Convolver with properties:
    Method: 'Time Domain'
  Show all properties
```

Convolve two rectangular sequences.

```
x = ones(10,1);
y = conv(x,x);
```

Plot the resulting convolved sequence, which is a triangular sequence.

```
plot(y)
```

## More About

### Convolution

The convolution of two signals is the integral that measures the amount of overlap of one signal as it is shifted over another signal.

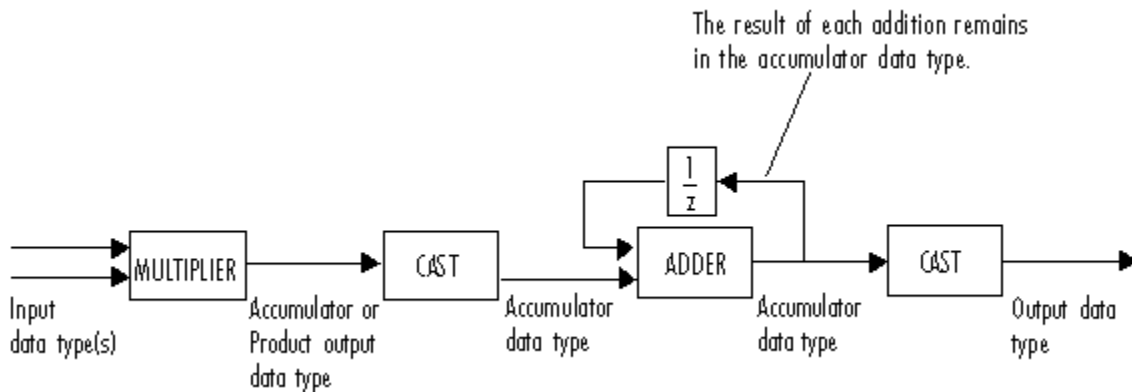
The convolution of two discrete time sequences,  $u[n]$  and  $v[n]$ , is given by the following equation:

$$y(k) = \sum_n u(n-k)v(k)$$

### Fixed Point

The following diagram shows the data types used within the `dsp.Convolver` System object for fixed-point signals.

Fixed-point signals are supported for the time domain only.



The output of the multiplier is in the product output data type when the input is real. When the input is complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication performed, see “Multiplication Data Types”.

When one or both of the inputs are signed fixed-point signals, all internal object data types are signed fixed point. The internal data types are unsigned fixed point only when *both* inputs are unsigned fixed-point signals.

## Algorithms

### Time-Domain Computation

When you set the computation domain to *time*, the algorithm computes the convolution of the two inputs in the time domain.

When the two inputs,  $u$  and  $v$ , are of size  $M_u$ -by- $N$  and  $M_v$ -by- $N$  respectively, the  $j$ th column of the convolution output is given by the following equation:

$$y_{i,j} = \sum_{k=0}^{\max(M_u, M_v) - 1} u_{k,j} v_{(i-k),j} \quad 0 \leq i \leq (M_u + M_v - 2)$$

Inputs  $u$  and  $v$  are zero when they are indexed outside their valid ranges.

When  $u$  is an  $M_u$ -by-1 column and  $v$  an  $M_v$ -by- $N$  matrix, the output is an  $(M_u + M_v - 1)$ -by- $N$  matrix whose  $j$ th column is computed using the following equation:

$$y_{i,j} = \sum_{k=0}^{\max(M_u, M_v) - 1} u_k v_{(i-k),j} \quad 0 \leq i \leq (M_u + M_v - 2)$$

When both the inputs are column vectors with lengths  $M_u$  and  $M_v$ , the object performs the vector convolution given by the following equation:

$$y_i = \sum_{k=0}^{\max(M_u, M_v) - 1} u_k v_{(i-k)} \quad 0 \leq i \leq (M_u + M_v - 2)$$

The output is an  $(M_u + M_v - 1)$ -by-1 column vector.

## Frequency-Domain Computation

When you set the computation domain to *frequency*, the algorithm computes the convolution in the frequency domain.

In this domain, the algorithm computes the convolution sequence by taking the Fourier transform of both the input signals, multiplying the Fourier transforms, and taking the inverse Fourier transform of the product. In this domain, depending on the input length, the algorithm can require fewer computations.

## Version History

### Introduced in R2012a

#### R2021b: `dsp.Convolver` System object will be removed

*Warns starting in R2021b*

`dsp.Convolver` System object will be removed in a future release. Use the `conv` function instead.

#### R2023a: `dsp.Convolver` System object has been removed

*Errors starting in R2023a*

The `dsp.Convolver` System object has been removed. Use the `conv` function instead.

### Update Code

This table shows how to update existing code to use the `conv` function.

| Discouraged Usage  | Recommended Replacement  |
|--|--|
| <pre>cv = dsp.Convolver; x = ones(10,1); yobj = cv(x,x);</pre> | <pre>yfn = conv(x,x); isequal(yobj,yfn)  ans =      logical       1</pre> <p>Run the code and plot the vectors <i>yobj</i> and <i>yfn</i> using the <code>plot</code> function. Verify that the plots match.</p> |

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

## See Also

### Functions

conv

### Blocks

Convolution

# dsp.Counter

**Package:** dsp

(Removed) Count up or down through specified range of numbers

---

**Note** dsp.Counter has been removed. Alternatively, you can create a variable in MATLAB® and increment the variable by 1.

---

## Description

The Counter object counts up or down through a specified range of numbers.

To count up or down through a specified range of numbers:

- 1 Create the dsp.Counter object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
count = dsp.Counter  
count = dsp.Counter(Name, Value)
```

### Description

count = dsp.Counter returns a counter System object, count, that counts up when the input is nonzero.

count = dsp.Counter(Name, Value) returns a counter System object, count, with each specified property set to the specified value.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

### Direction — Count up or down

Up (default) | Down

Specify the counter direction as Up or Down.



**Tunable:** Yes

### **CountEventInputPort — Add input to specify a count event**

true (default) | false

Set this property to true to enable a count event input for the internal counter. The internal counter increments or decrements whenever the count event input satisfies the condition you specify in the CountEventCondition property. When you set this property to false, the internal counter is free running, that is, the counter increments or decrements on every call to the object algorithm.

### **CountEventCondition — Condition that increments, decrements, or resets internal counter**

Non-zero (default) | Rising edge | Falling edge | Either edge

Specify the event at the count event input that increments or decrements the counter as Rising edge, Falling edge, Either edge or Non-zero.

If you set the ResetInputPort and CountEventInputPort properties to true, the counter is reset when the event you specify for the CountEventCondition occurs.

#### **Dependencies**

This property applies only when you set the CountEventInputPort property to true.

### **CounterSizeSource — Source of counter size data type**

Property (default) | Input port

Specify the source of the counter size data type as Property or Input port.

### **CounterSize — Range of integer values to count through**

Maximum (default) | 8 bits | 16 bits | 32 bits

Specify the range of integer values to count through before recycling to zero as 8 bits, 16 bits, 32 bits or Maximum.

### **MaximumCount — Counter's maximum value**

255 (default) | 0 | positive integer

Specify the counter's maximum value as a numeric scalar value.

**Tunable:** Yes

#### **Dependencies**

This property applies only when you set the CounterSizeSource property to Property and the CounterSize property to Maximum.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### **InitialCount — Counter initial value**

0 (default) | 0 | positive integer

Specify the initial value for the counter.

**Tunable:** Yes

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**CountOutputPort — Output count**`true (default) | false`

Set this property to `true` to enable output of the internal count. The default is `true`. You cannot set both `CountOutputPort` and `HitOutputPort` to `false` at the same time.

**HitOutputPort — Output hit events**`true (default) | false`

Set this property to `true` to enable output of the hit events. You cannot set both `CountOutputPort` and `HitOutputPort` to `false` at the same time.

**HitValues — Values whose occurrence in count produce a true hit output**`32 (default) | scalar integer | vector of integers`

Specify an integer scalar or a vector of integers, whose occurrences in the count you want flagged as a hit.

**Dependencies**

This property applies only when you set the `HitOutputPort` property to `true`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**ResetInputPort — Add input to enable internal counter reset**`true (default) | false`

When you set this property to `true`, specify a reset input to the object algorithm. When the reset input receives the event you specify for the `CountEventCondition` property, the counter resets. If you set the `CountEventInputPort` property to `false`, the counter resets whenever the reset input is not zero.

**SamplesPerFrame — Number of samples in each output frame**`1 (default) | positive integer`

Specify the number of samples in each output frame.

**Dependencies**

This property applies only when you set the `CountEventInputPort` property to `false`, indicating a free-running counter.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**CountOutputDataType — Data type of count output**`double (default) | single | int8 | uint8 | int16 | uint16 | int32 | uint32`

Specify the data type of the count output, `cnt`, as `double`, `single`, `int8`, `uint8`, `int16`, `uint16`, `int32` or `uint32`.

**Dependencies**

This property applies when you set the `CountOutputPort` property to `true`.

## Usage

### Syntax

```
[cnt, hit] = count(event, reset)
cnt = count(event, reset)
hit = count(event, reset)
[ ___ ] = count()
[ ___ ] = count(event)
```

### Description

`[cnt, hit] = count(event, reset)` increments, decrements, or resets the internal counter as specified by the values of the `event` and `reset` inputs. The output argument `cnt` denotes the present value of the counter. A trigger event at the `event` input causes the counter to increment or decrement. A trigger event at the `reset` input resets the counter to its initial state.

`cnt = count(event, reset)` returns the current value of the count when you set the `CountOutputPort` property to `true` and the `HitOutputPort` property to `false`.

`hit = count(event, reset)` returns a Boolean value indicating whether the count has reached any of the values specified by the `HitValues` property. This condition applies when you set the `HitOutputPort` property to `true` and the `CountOutputPort` property to `false`.

`[ ___ ] = count()` increments or decrements the free-running internal counter when you set the `CountEventInputPort` property to `false` and the `ResetInputPort` property to `false`.

`[ ___ ] = count(event)` increments or decrements the internal counter when the `event` input matches the event you specify for the `CountEventCondition` property and you set the `ResetInputPort` property to `false`.

### Input Arguments

#### **event** — Event that causes counter to increment or decrement

scalar

Event that causes the counter to increment or decrement, specified as a scalar. The `CountEventCondition` property specifies the event under which the counter value changes.

#### **Dependencies**

This input is valid only when `CountEventInputPort` is set to `true`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `logical`

#### **reset** — Reset of internal counter

scalar

A trigger event at the reset input resets the counter to the initial state.

#### **Dependencies**

This input is valid only when `ResetInputPort` is set to `true`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `logical`

## Output Arguments

### **cnt** — Present value of counter

scalar | column vector

Current value of the count, returned as a scalar. The data type of this output is set by the `CountOutputDataType` property. If `CountEventInputPort` is false, the number of elements in this output vector is determined by the value you specify in the `SamplesPerFrame` property.

### Dependencies

This output is enabled only when you set the `CountOutputPort` property to `true`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

### **hit** — Indicates if counter matches hit values

scalar | column vector

Boolean value indicating whether the count has reached any of the values specified by the `HitValues` property. If `CountEventInputPort` is false, the number of elements in this output vector is determined by the value you specify in the `SamplesPerFrame` property.

### Dependencies

This output is enabled when you set the `HitOutputPort` property to `true`.

Data Types: `logical`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Common to All System Objects

|                      |  |
|----------------------|--|
| <code>step</code>    | Run System object algorithm  |
| <code>release</code> | Release resources and allow changes to System object property values and input characteristics |
| <code>reset</code>   | Reset internal states of System object   |

## Examples

### Count the Rising Edges

Use `dsp.Counter` System object to count at every rising edge of the input signal.

```
count = dsp.Counter('MaximumCount', 5, ...  
    'CountOutputPort', true, ...  
    'HitOutputPort', false, ...  
    'ResetInputPort', false);  
sgnl = [0 1 0 1 0 1 0 1 0 1 0 1];  
cnt = zeros(1,12);  
for ii = 1:length(sgnl)
```

```

        cnt(ii) = count(sgnl(ii));
end
disp(cnt);

    0     1     1     2     2     3     3     4     4     5     5     0

```

## Algorithms

This object implements the algorithm, inputs, and outputs described on the Counter block reference page. The object properties correspond to the block parameters.

- The CountEventCondition object property does not have a free-running option. Set the CountEventInputPort property to `false` to obtain the free-running option.
- The CounterSizeSource and CounterSize object properties correspond to the **Counter size** block parameter.
- The CountOutputPort and HitOutputPort correspond to the **Output** block parameter.
- There is no object property that corresponds to the **Hit data type** block parameter. The output type is logical in MATLAB. (This logical is different from the popup logical in the block. For the object, logical corresponds to Boolean in the block.)

## Version History

### Introduced in R2012a

#### **R2021b: dsp.Counter System object will be removed**

*Warns starting in R2021b*

`dsp.Counter` System object will be removed in a future release. Alternatively, you can create a variable in MATLAB and increment the variable by 1.

#### **R2023a: dsp.Counter System object has been removed**

*Errors starting in R2023a*

The `dsp.Counter` System object has been removed. Alternatively, you can create a variable in MATLAB and increment the variable by 1.

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

## dsp.CoupledAllpassFilter

**Package:** dsp

Coupled allpass IIR filter

### Description

The `dsp.CoupledAllpassFilter` object implements a coupled allpass filter structure composed of two allpass filters connected in parallel. Each allpass branch can contain multiple sections. The overall filter output is computed by adding the output of the two respective branches. An optional second output can also be returned, which is power complementary to the first. For example, from the frequency domain perspective, if the first output implements a lowpass filter, the second output implements the power complementary highpass filter. For real signals, the power complementary output is computed by subtracting the output of the second branch from the first.

`dsp.CoupledAllpassFilter` supports double- and single-precision floating point and allows you to choose between different realization structures. This System object also supports complex coefficients, multichannel variable length input, and tunable filter coefficient values.

To filter each channel of the input:

- 1 Create the `dsp.CoupledAllpassFilter` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
caf = dsp.CoupledAllpassFilter
caf = dsp.CoupledAllpassFilter(AllpassCoeffs1,AllpassCoeffs2)
caf = dsp.CoupledAllpassFilter(struc,AllpassCoeffs1,AllpassCoeffs2)
caf = dsp.CoupledAllpassFilter(Name,Value)
```

### Description

`caf = dsp.CoupledAllpassFilter` returns a coupled allpass filter System object, `caf`, that filters each channel of the input signal independently. The coupled allpass filter uses the default inner structures and coefficients.

`caf = dsp.CoupledAllpassFilter(AllpassCoeffs1,AllpassCoeffs2)` returns a coupled allpass filter System object, `caf`, with `Structure` set to 'Minimum multiplier', `AllpassCoefficients1` set to `AllpassCoeffs1`, and `AllpassCoefficients2` set to `AllpassCoeffs2`.

`caf = dsp.CoupledAllpassFilter(struc,AllpassCoeffs1,AllpassCoeffs2)` returns a coupled allpass filter System object, `caf`, with `Structure` set to `struc` and the relevant coefficients set to `AllpassCoeffs1` and `AllpassCoeffs2`. `struc` can be 'Minimum multiplier' | 'Wave Digital Filter' | 'Lattice'.

`caf = dsp.CoupledAllpassFilter(Name,Value)` returns a Coupled allpass filter System object, `caf`, with each property set to the specified value.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see *System Design in MATLAB Using System Objects*.

### Structure — Internal structure of allpass branches

'Minimum multiplier' (default) | 'Wave Digital Filter' | 'Lattice'

Specify the internal structure of allpass branches as one of 'Minimum multiplier', 'Wave Digital Filter', or 'Lattice'. Each structure uses a different pair of coefficient values, independently stored in the relevant object property.

### AllpassCoefficients1 — Allpass polynomial coefficients of branch 1

[0 0.5] (default) | row vector | cell array

Specify the polynomial filter coefficients for the first allpass branch. This property can accept values either in the form of a row vector (single-section configuration) or a cell array with as many cells as filter sections.

**Tunable:** Yes

#### Dependencies

This property is applicable only if you set the Structure property to 'Minimum multiplier'.

Data Types: `single` | `double`

### WDFCoefficients1 — Wave Digital Filter coefficients of branch 1

[0.5 0] (default) | row vector | cell array

Specify the Wave Digital Filter coefficients for the first allpass branch. This property can accept values either in the form of a row vector (single-section configuration) or a cell array with as many cells as filter sections.

**Tunable:** Yes

#### Dependencies

This property is applicable only if you set the Structure property to 'Wave Digital Filter'.

Data Types: `single` | `double`

### LatticeCoefficients1 — Lattice coefficients of branch 1

[0.5 0] (default) | row vector | cell array

Specify the allpass lattice coefficients for the first allpass branch. This property can accept values either in the form of a row vector (single-section configuration) or a cell array with as many cells as filter sections.

**Tunable:** Yes

**Dependencies**

This property is applicable only if you set the Structure property to 'Lattice'.

Data Types: `single` | `double`

**Delay — Length in samples for branch 1**

0 (default) | positive integer scalar

Integer number of the delay taps in the top branch, specified as a positive integer scalar.

**Tunable:** Yes

**Dependencies**

This property is applicable only if you set the PureDelayBranch property to `true`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**Gain1 — Independent Branch 1 Phase Gain**

1 (default) | '-1' | '0+i' | '0-i'

Gain1 is the individual branch phase gain. This property can accept only values equal to '1', '-1', '0+i', or '0-i'. This property is nontunable.

Data Types: `char`

**AllpassCoefficients2 — Allpass polynomial coefficients of branch 2**

{[ ]} (default) | row vector | cell array

Specify the polynomial filter coefficients for the second allpass branch. This property can accept values either in the form of a row vector (single-section configuration) or a cell array with as many cells as filter sections.

**Tunable:** Yes

**Dependencies**

This property is applicable only if you set the Structure property to 'Minimum multiplier'.

Data Types: `single` | `double`

**WDFCoefficients2 — Wave Digital Filter coefficients of branch 2**

{[ ]} (default) | row vector | cell array

Specify the Wave Digital Filter coefficients for the second allpass branch. This property can accept values either in the form of a row vector (single-section configuration) or a cell array with as many cells as filter sections.

**Tunable:** Yes

**Dependencies**

This property is applicable only if you set the Structure property to 'Wave Digital Filter'.

Data Types: `single` | `double`



**LatticeCoefficients2 — Lattice coefficients of branch 2**

{[ ]} (default) | row vector | cell array

Specify the allpass lattice coefficients for the second allpass branch. This property can accept values either in the form of a row vector (single-section configuration) or a cell array with as many cells as filter sections.

**Tunable:** Yes

**Dependencies**

This property is applicable only if you set the Structure property to 'Lattice'.

Data Types: single | double

**Gain2 — Independent Branch 2 Phase Gain**

'1' (default) | '-1' | '0+1i' | '0-1i'

Specify the value of the independent phase gain applied to branch 2. This property can accept only values equal to '1', '-1', '0+i', or '0-i'. This property is nontunable.

Data Types: char

**Beta — Coupled phase gain**

1 (default) | complex value with magnitude equal to 1

Specify the value of the phasor gain in complex conjugate form, in each of the two branches, and in complex coefficient configuration. The absolute value of this property should be 1 and its default value is 1.

**Tunable:** Yes

**Dependencies**

This property is applicable only when the selected Structure property supports complex coefficients.

Data Types: single | double

**PureDelayBranch — Replace allpass filter in first branch with pure delay**

false (default) | true

If you set PureDelayBranch to true, the property holding the coefficients for the first allpass branch is disabled and Delay becomes enabled. You can use this property to improve performance, when one of the two allpass branches is known to be a pure delay (e.g. for halfband filter designs)

Data Types: logical

**ComplexConjugateCoefficients — Allow inferring coefficients of second allpass branch as complex conjugate of first**

false (default) | true

When the input signal is real, this property triggers the use of an optimized structural realization. This property enables providing complex coefficients for only the first branch. The coefficients for the second branch are automatically inferred as complex conjugate values of the first branch coefficients

### Dependencies

This property is only enabled if the currently selected structure supports complex coefficients. Use it only if the filter coefficients are actually complex.

Data Types: `logical`

### Usage

### Syntax

```
y = caf(x)
[y, ypc] = caf(x)
```

### Description

`y = caf(x)` filters the input signal `x` to produce the output `y`. When `x` is a matrix, each column is filtered independently as a separate channel over time.

`[y, ypc] = caf(x)` also returns `ypc`, the power complementary signal to the primary output `y`.

### Input Arguments

#### **x** — Data input

vector | matrix

Data input, specified as a vector or a matrix. This object also accepts variable-size inputs. Once the object is locked, you can change the size of each input channel, but you cannot change the number of channels.

Data Types: `single` | `double`

### Output Arguments

#### **y** — Lowpass filtered output

vector | matrix

Lowpass filtered output, returned as a vector or a matrix. The size and data type of the output signal matches that of the input signal.

Data Types: `double` | `single`

#### **ypc** — Highpass filtered output

vector | matrix

Power complimentary highpass filtered output, returned as a vector or a matrix. The size and data type of the output signal matches that of the input signal.

Data Types: `double` | `single`

### Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Specific to dsp.CoupledAllpassFilter

|             |  |
|-------------|--|
| fvtool      | Visualize frequency response of coupled allpass filter       |
| getBranches | Return internal allpass branches                             |
| freqz       | Frequency response of discrete-time filter System object     |
| impz        | Impulse response of discrete-time filter System object       |
| info        | Information about filter System object                       |
| coeffs      | Returns the filter System object coefficients in a structure |
| cost        | Estimate cost of implementing filter System object           |
| grpdelay    | Group delay response of discrete-time filter System object   |

## Common to All System Objects

|         |  |
|---------|--|
| step    | Run System object algorithm  |
| release | Release resources and allow changes to System object property values and input characteristics |
| reset   | Reset internal states of System object   |

## Examples

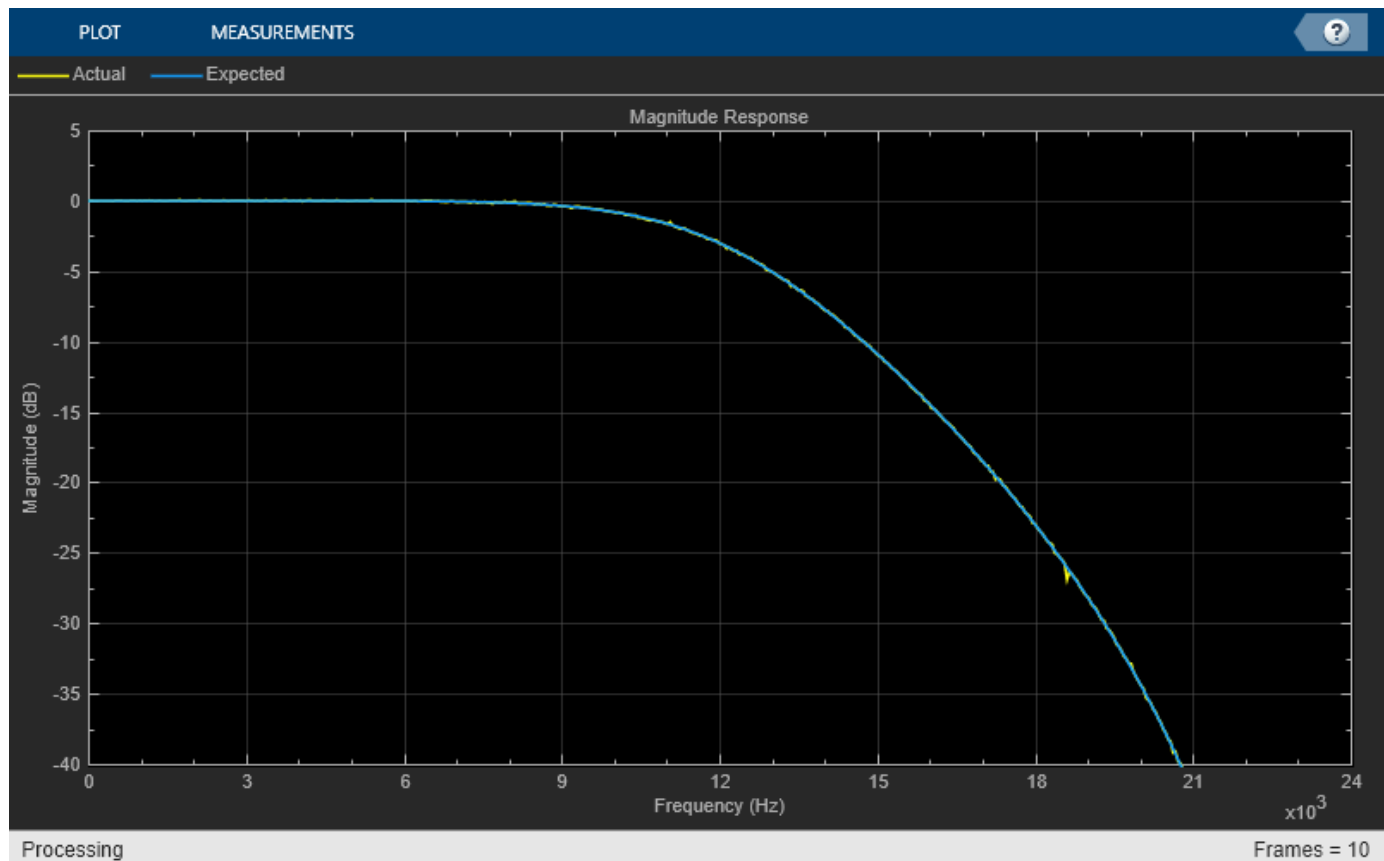
### Allpass Realization of a Butterworth Lowpass Filter

Realize a Butterworth lowpass filter of order 3. Use a coupled allpass structure with inner minimum multiplier structure.

```

Fs = 48000;    % in Hz
Fc = 12000;   % in Hz
frameLength = 1024;
[b, a] = butter(3,2*Fc/Fs);
AExp = [freqz(b,a,frameLength/2); NaN];
[c1, c2] = tf2ca(b,a);
caf = dsp.CoupledAllpassFilter(c1(2:end),c2(2:end));
tfe = dsp.TransferFunctionEstimator('FrequencyRange', 'onesided',...
    'SpectralAverages', 2);
aplot = dsp.ArrayPlot('PlotType', 'Line',...
    'YLimits', [-40 5],...
    'YLabel', 'Magnitude (dB)',...
    'SampleIncrement', Fs/frameLength,...
    'XLabel', 'Frequency (Hz)',...
    'Title', 'Magnitude Response',...
    'ShowLegend', true,'ChannelNames',{'Actual','Expected'});
Niter = 200;
for k = 1:Niter
    in = randn(frameLength,1);
    out = caf(in);
    A = tfe(in,out);
    aplot(db([A,AExp]));
end

```



### Allpass Realization of an Elliptic Highpass Filter

Remove a low-frequency sinusoid using an elliptic highpass filter design implemented through a coupled allpass structure.

#### Initialize

```

Fs = 1000;
f1 = 50; f2 = 100;
Fpass = 70; Apass = 1;
Fstop = 60; Astop = 80;
filtSpecs = fdesign.highpass(Fstop,Fpass,Astop,Apass,Fs);
hpSpec = design(filtSpecs,'ellip','FilterStructure','cascadeallpass',...
    'SystemObject',true);
frameLength = 1000;
nFrames = 100;
sine = dsp.SineWave('Frequency',[f1,f2],'SampleRate',Fs,...
    'SamplesPerFrame',frameLength); % Input composed of two sinusoids.

sa = spectrumAnalyzer('SampleRate',Fs,...
    'PlotAsTwoSidedSpectrum',false,'ShowLegend',true,...
    'Method','welch','AveragingMethod','exponential',...
    'Title','Original (Channel 1) Filtered (Channel 2)',...
    'ChannelNames',{'Original','Filtered'});

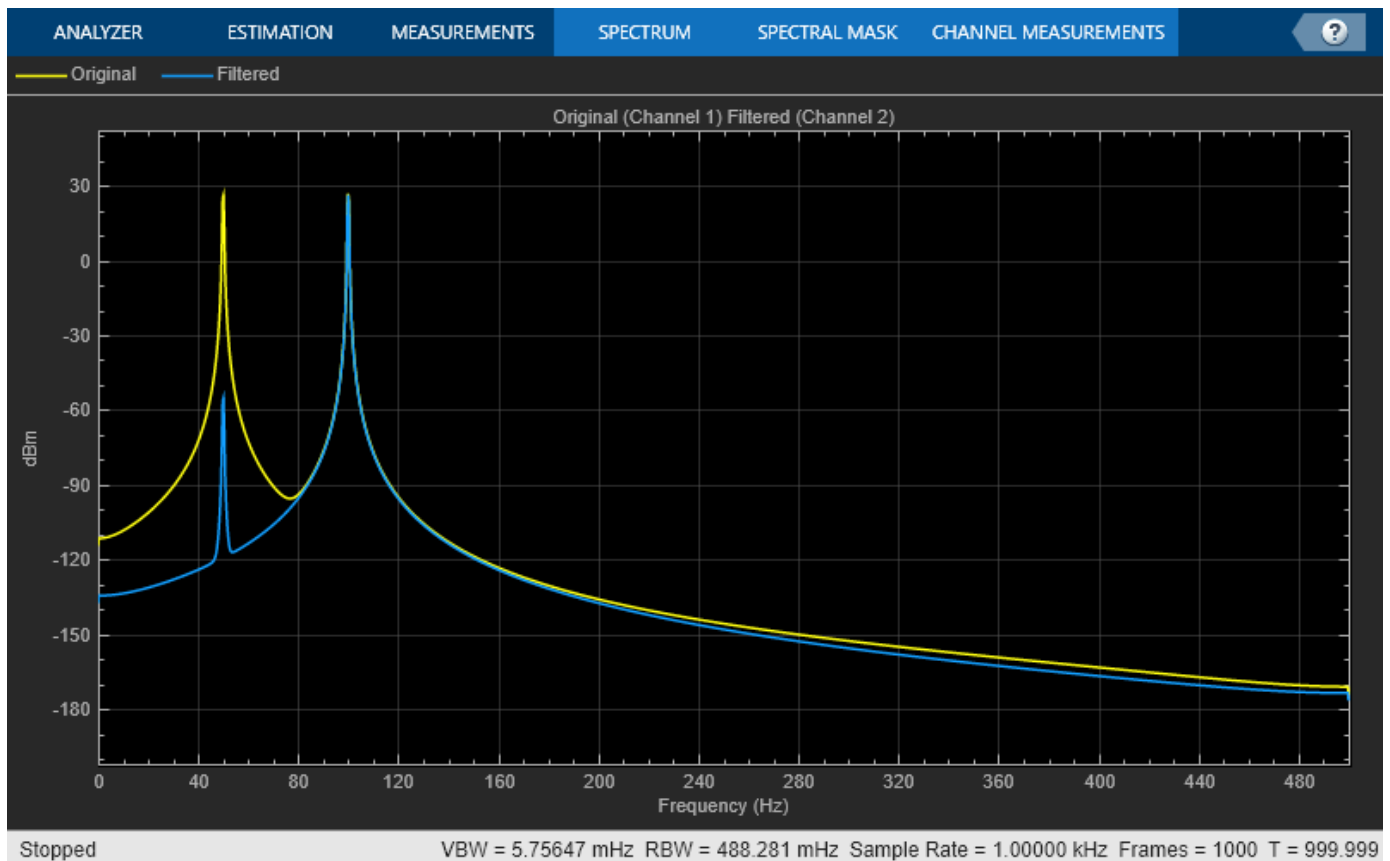
```

## Simulate

```

for k = 1:1000
    original = sum(sine(),2); % Add the two sinusoids together
    filtered = hpSpec(original);
    sa([original,filtered]);
end
release(sa)

```



## View Power Complementary Output of Coupled Allpass Filter

Design a Butterworth lowpass filter of order 3. Use a coupled allpass structure with inner minimum multiplier structure.

```

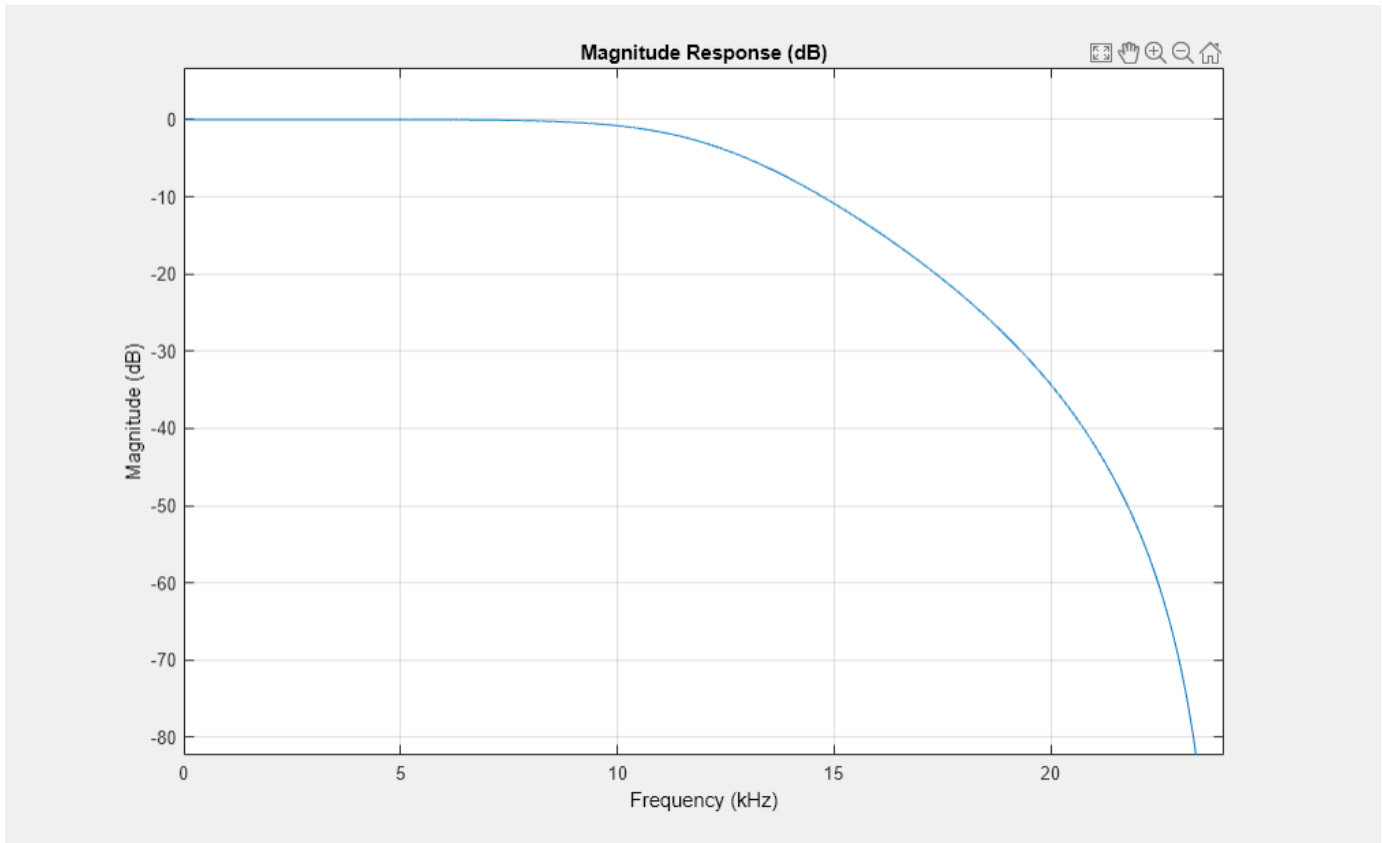
Fs = 48000; % in Hz
Fc = 12000; % in Hz
frameLength = 1024;
[b,a] = butter(3,2*Fc/Fs);
AExp = [freqz(b,a,frameLength/2); NaN];
[c1,c2] = tf2ca(b,a);
caf = dsp.CoupledAllpassFilter(c1(2:end),c2(2:end));

```

Using the 'SubbandView' option of the `dsp.CoupledAllpassFilter`, you can visualize the lowpass filter output, the power complementary highpass filter output, or both using the `fvtool`.

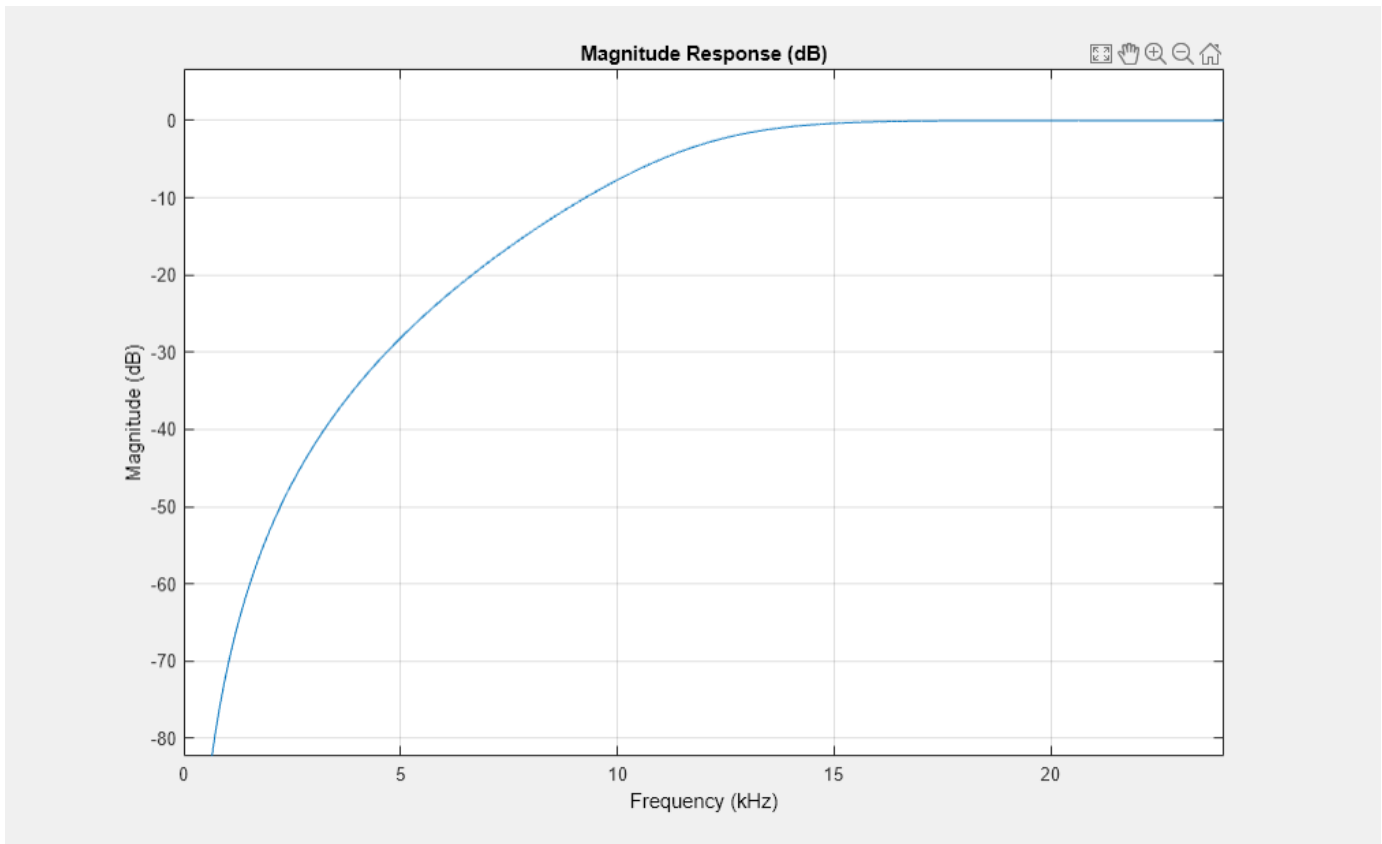
To view the lowpass filter output, set 'SubbandView' to 1.

```
fvtool(caf, 'SubbandView', 1, 'Fs', Fs)
```



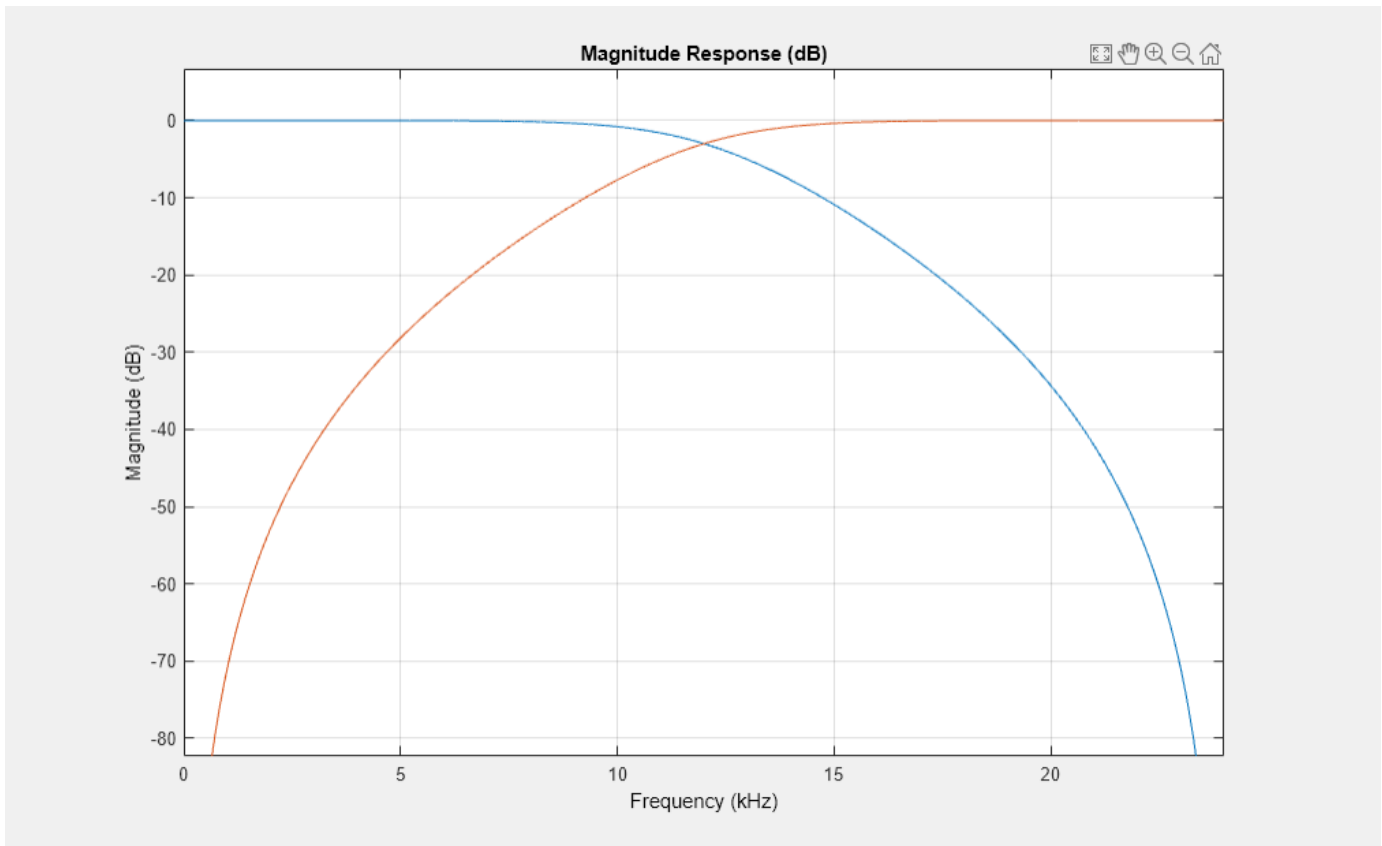
To view the highpass filter output, set 'SubbandView' to 2.

```
fvtool(caf, 'SubbandView', 2, 'Fs', Fs)
```



To view both the outputs, set 'SubbandView' to 'all', [1 2] or [1;2].

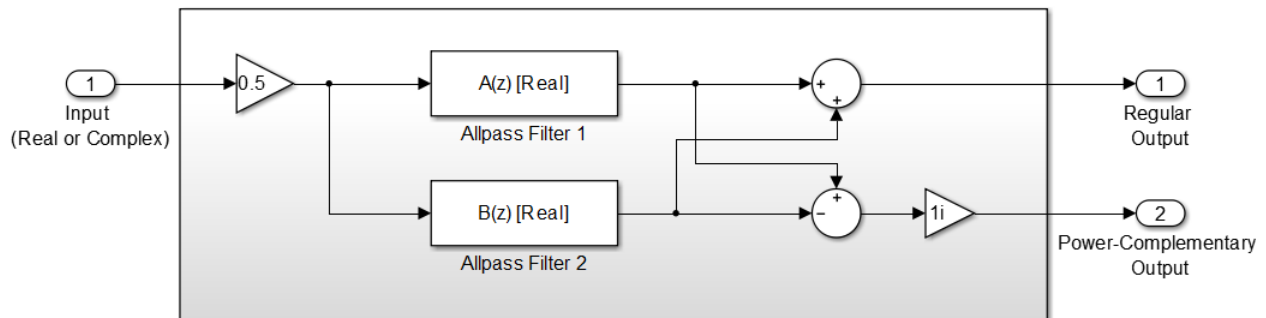
```
fvtool(caf, 'SubbandView', 'all', 'Fs', Fs);
```



## Algorithms

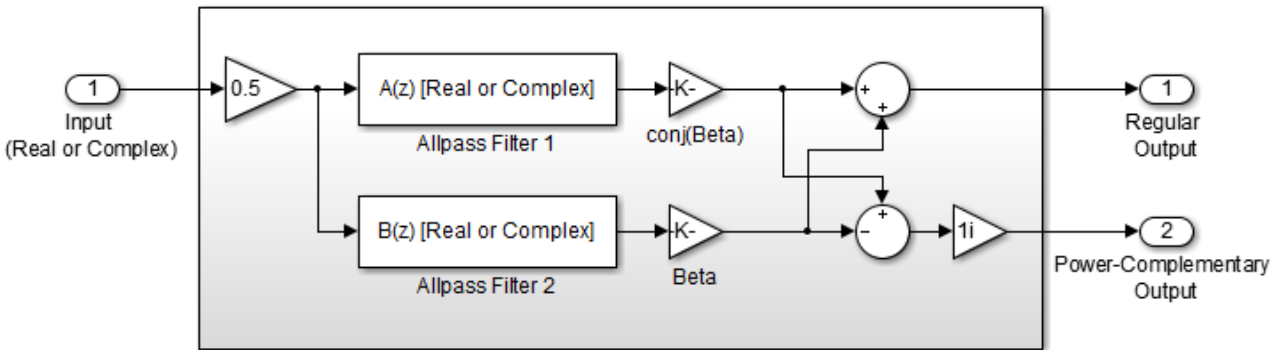
The following three figures summarize the main structures supported by `dsp.CoupledAllpassFilter`.

- Minimum Multiplier and WDF

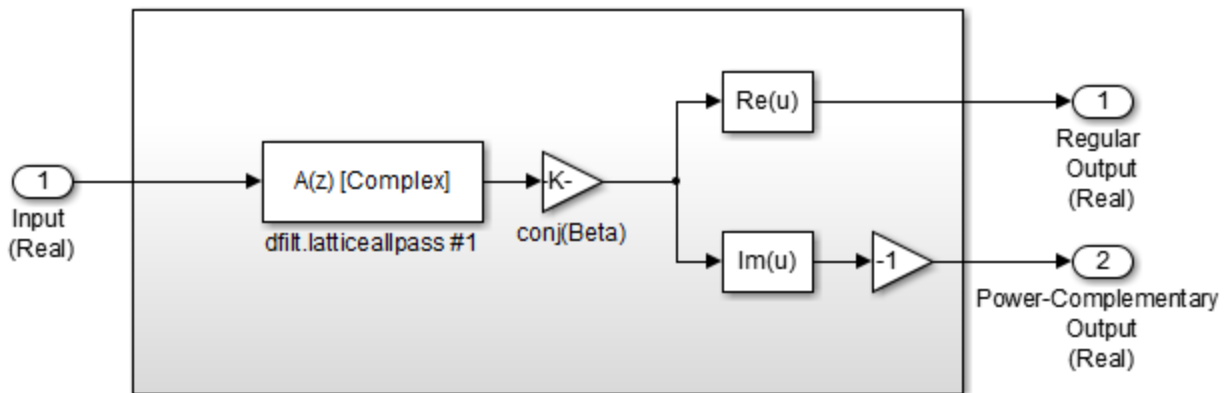


- Lattice





- Lattice with Complex Conjugate Coefficients



## Version History

Introduced in R2013b

## References

- [1] Regalia, Philip A., Mitra, Sanjit K., and P.P Vaidyanathan " The Digital All-Pass Filter: A Versatile Signal Processing Building Block." *Proceedings of the IEEE 1988*, Vol. 76, No. 1, pp. 19–37.
- [2] Mitra, Sanjit K., and James F. Kaiser, "*Handbook for Digital Signal Processing*" New York: John Wiley & Sons, 1993.

## See Also

### Functions

fvtool | getBranches | freqz | impz | info | coeffs | cost | grpdelay | allpass2wdf

### Objects

dsp.AllpassFilter | dsp.SOSFilter | dsp.IIRFilter

**Topics**

“Analysis Methods for Filter System Objects” on page 3-2

# dsp.CrossSpectrumEstimator

**Package:** dsp

Estimate cross-spectral density

## Description

The `dsp.CrossSpectrumEstimator` System object computes the cross-spectrum density of a signal, using the Welch's averaged periodogram method.

To implement the cross-spectrum estimation object:

- 1 Create the `dsp.CrossSpectrumEstimator` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
cse = dsp.CrossSpectrumEstimator
cse = dsp.CrossSpectrumEstimator(Name, Value)
```

### Description

`cse = dsp.CrossSpectrumEstimator` returns a System object, `cse`, that computes the cross-power spectrum of real or complex signals using the periodogram method and Welch's averaged, modified periodogram method.

`cse = dsp.CrossSpectrumEstimator(Name, Value)` returns a `dsp.CrossSpectrumEstimator` System object, `cse`, with each specified property name set to the specified value. Unspecified properties have default values.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

### FFTLengthSource — Source of FFT length value

'Auto' (default) | 'Property'

Specify the source of the FFT length value as either 'Auto' or 'Property'. If you set this property to 'Auto', the cross-spectrum estimator sets the FFT length to the input frame size. If you set this property to 'Property', then specify the number of FFT points using the FFTLength property.

**FFTLength — FFT Length**

128 (default) | positive integer

Specify the length of the FFT that the cross-spectrum estimator uses to compute cross-spectral estimates as a positive, integer scalar.

**Dependencies**

This property applies when you set the FFTLengthSource property to 'Property'.

Data Types: double

**Window — Window function**

'Hann' (default) | 'Rectangular' | 'Chebyshev' | 'Flat Top' | 'Hamming' | 'Kaiser'

Specify a window function for the cross-spectrum estimator as one of 'Rectangular', 'Chebyshev', 'Flat Top', 'Hamming', 'Hann', or 'Kaiser'.

**SidelobeAttenuation — Side lobe attenuation of window**

60 (default) | positive scalar

Specify the side lobe attenuation of the window as a real, positive scalar, in decibels (dB).

**Dependencies**

This property applies when you set the Window property to 'Chebyshev' or 'Kaiser'.

Data Types: double

**FrequencyRange — Frequency range of the cross-spectrum estimate**

'Twosided' (default) | 'onesided' | 'centered'

Specify the frequency range of the cross-spectrum estimator as one of 'twosided', 'onesided', or 'centered'.

If you set the FrequencyRange to 'onesided', the cross-spectrum estimator computes the one-sided spectrum of real input signals,  $x$  and  $y$ . If the FFT length,  $N_{FFT}$ , is even, the length of the cross-spectral estimate is  $N_{FFT}/2+1$  and is computed over the interval  $[0, SampleRate/2]$ . If  $N_{FFT}$  is odd, the length of the cross-spectrum estimate is equal to  $(N_{FFT}+1)/2$ , and the interval is  $[0, SampleRate/2]$ .

If you set the FrequencyRange to 'twosided', the cross-spectrum estimator computes the two-sided spectrum of complex or real input signals,  $x$  and  $y$ . The length of the cross-spectrum estimate is equal to  $N_{FFT}$ . This value is computed over  $[0, SampleRate]$ .

If you set the FrequencyRange to 'centered', the cross-spectrum estimator computes the centered two-sided spectrum of complex or real input signals,  $x$  and  $y$ . The length of the cross-spectrum estimate is equal to  $N_{FFT}$ , and the estimate is computed between  $[-SampleRate/2, SampleRate/2]$  and  $(-SampleRate/2, SampleRate/2)$  for even and odd lengths, respectively.

**AveragingMethod — Averaging method**

'Running' (default) | 'Exponential'

Specify the averaging method as 'Running' or 'Exponential'. In the running averaging method, the object computes an equally weighted average of a specified number of spectrum estimates defined by the `SpectralAverages` property. In the exponential method, the object computes the average over samples weighted by an exponentially decaying forgetting factor.

### **SpectralAverages — Number of spectral averages**

8 (default) | positive integer

Specify the number of spectral averages as a positive, integer scalar. The object computes the current cross-spectral estimate by averaging the last N estimates. N is the number of spectral averages defined in the `SpectralAverages` property.

#### **Dependencies**

This property applies when you set `AveragingMethod` to 'Running'.

Data Types: double

### **ForgettingFactor — Forgetting factor**

0.9 (default) | scalar in the range (0,1]

Specify the exponential weighting forgetting factor as a scalar value greater than zero and smaller than or equal to one.

**Tunable:** Yes

#### **Dependencies**

This property applies when you set `AveragingMethod` to 'Exponential'.

Data Types: single | double

### **SampleRate — Sample rate of input**

1 (default) | positive scalar

Specify the sample rate of the input, in hertz, as a finite numeric scalar. The sample rate is the rate at which the signal is sampled in time.

Data Types: single | double

## **Usage**

### **Syntax**

`pxy = cse(x,y)`

#### **Description**

`pxy = cse(x,y)` computes the cross power spectrum density, `pxy`, of the input signals, `x` and `y`.

#### **Input Arguments**

##### **x — First data input**

vector | matrix

First data input, specified as a vector or a matrix. The inputs, `x` and `y` must have the same size and data type.

Data Types: `single` | `double`

### **y** — Second data input

`vector` | `matrix`

Second data input, specified as a vector or a matrix. The inputs, `x` and `y` must have the same size and data type.

Data Types: `single` | `double`

### **Output Arguments**

#### **pxy** — Cross-power spectrum density output

`vector` | `matrix`

Cross-power spectrum density output, returned as a vector or a matrix. The output has the same size and data type as the input signals.

Data Types: `single` | `double`

## **Object Functions**

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

### **Specific to `dsp.CrossSpectrumEstimator`**

`getFrequencyVector`    Vector of frequencies at which estimation is done  
`getRBW`                Resolution bandwidth of spectrum

### **Common to All System Objects**

`step`        Run System object algorithm  
`release`    Release resources and allow changes to System object property values and input characteristics  
`reset`      Reset internal states of System object

## **Examples**

### **Compute Cross-Power Spectrum of Two Noisy Sine Waves**

Generate two sine waves.

```
sin1 = dsp.SineWave('Frequency',200, 'SampleRate', 1000);  
sin1.SamplesPerFrame = 1000;  
sin2 = dsp.SineWave('Frequency',100, 'SampleRate', 1000);  
sin2.SamplesPerFrame = 1000;
```

Use the `dsp.CrossSpectrumEstimator` System object™ to compute the cross-spectrum of the signals. Also, use the `dsp.ArrayPlot` object to display the spectra.

```
cse = dsp.CrossSpectrumEstimator('SampleRate', sin1.SampleRate,...  
    'FrequencyRange','centered');
```

```

aplot = dsp.ArrayPlot('PlotType','Line','XOffset',-500,'YLimits',...
    [-150 -60],'YLabel','Power Spectrum Density (Watts/Hz)',...
    'XLabel','Frequency (Hz)',...
    'Title','Cross Power Spectrum of Two Signals');

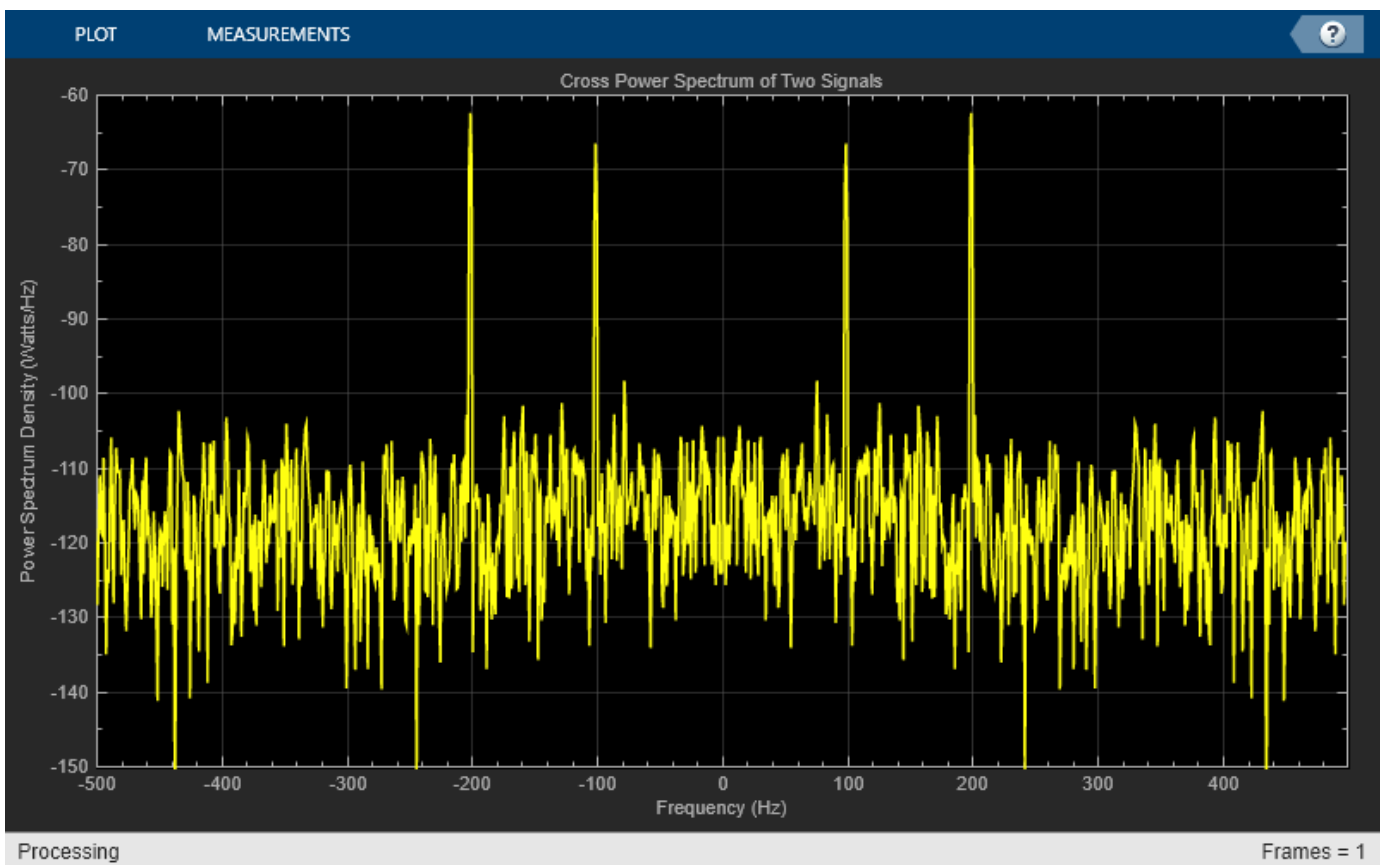
```

Add random noise to the sine waves. Stream in the data, and plot the cross-power spectrum of the two signals.

```

for ii = 1:10
x = sin1() + 0.05*randn(1000,1);
y = sin2() + 0.05*randn(1000,1);
Pxy = cse(x, y);
aplot(20*log10(abs(Pxy)));
end

```



## Algorithms

### Welch's Method of Averaged Modified Periodograms

Give two signal inputs,  $x$  and  $y$ :

- 1 Multiply the inputs by the window and scale the result by the window power.
- 2 Compute FFT of the signals,  $X$  and  $Y$ , and multiply  $X$  with  $\text{conj}(Y)$  using  $Z = X \cdot \text{conj}(Y)$ .
- 3 Compute the current cross power spectrum estimate by taking the moving average of the last  $N$  number of  $Z$ 's and scaling the answer by the sample rate.

For further information on the algorithms, refer to the “Averaging Method” on page 2-1343 section in Spectrum Analyzer.

## Version History

Introduced in R2013b

## References

- [1] Hayes, Monson H. *Statistical Digital Signal Processing and Modeling*. Hoboken, NJ: John Wiley & Sons, 1996.
- [2] Kay, Steven M. *Modern Spectral Estimation: Theory and Application*. Englewood Cliffs, NJ: Prentice Hall, 1999.
- [3] Stoica, Petre, and Randolph L. Moses. *Spectral Analysis of Signals*. Englewood Cliffs, NJ: Prentice Hall, 2005.
- [4] Welch, P. D. "The Use of Fast Fourier Transform for the Estimation of Power Spectra: A Method Based on Time Averaging Over Short Modified Periodograms." *IEEE Transactions on Audio and Electroacoustics*. Vol. 15, No. 2, June 1967, pp. 70-73.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- “System Objects in MATLAB Code Generation” (MATLAB Coder)

## See Also

### Functions

`getFrequencyVector` | `getRBW`

### Objects

`spectrumAnalyzer` | `dsp.TransferFunctionEstimator` | `dsp.SpectrumEstimator`

### Blocks

Cross Spectrum Estimator



# dsp.Crosscorrelator

**Package:** dsp

(Removed) Cross-correlation of two inputs

---

**Note** `dsp.Crosscorrelator` has been removed. Use `xcorr` instead. For more information, see “Compatibility Considerations”.

---

## Description

The `dsp.Crosscorrelator` System object computes the cross-correlation of two  $N$ -D input arrays along the first dimension. The computation can be done in the time domain or frequency domain. You can specify the domain through the “Method” on page 4-0 property. In the time domain, the object convolves the first input signal,  $u$ , with the time-reversed complex conjugate of the second input signal,  $v$ . To compute the cross-correlation in the frequency domain, the object:

- 1 Takes the Fourier transform of both input signals, resulting in  $U$  and  $V$ .
- 2 Multiplies  $U$  and  $V^*$ , where  $*$  denotes the complex conjugate.
- 3 Computes the inverse Fourier transform of the product.

If you set `Method` to 'Fastest', the object chooses the domain that minimizes the number of computations. For information on these computation methods, see “Algorithms” on page 4-338.

To obtain the cross-correlation for two discrete-time deterministic inputs:

- 1 Create the `dsp.Crosscorrelator` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

## Creation

### Syntax

```
xcorr = dsp.Crosscorrelator
xcorr = dsp.Crosscorrelator(Name, Value)
```

### Description

`xcorr = dsp.Crosscorrelator` returns a cross-correlator object, `xcorr`, that computes the cross-correlation of two inputs in the time domain or frequency domain.

`xcorr = dsp.Crosscorrelator(Name, Value)` returns a cross-correlator object with each specified property set to the specified value. Enclose each property name in single quotes.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

### Method — Domain for computing correlations

'Time Domain' (default) | 'Frequency Domain' | 'Fastest'

Domain in which the System object computes the correlation, specified as one of the following:

- 'Time Domain' -- Computes the cross-correlation in the time domain, which minimizes the memory usage.
- 'Frequency Domain' -- Computes the cross-correlation in the frequency domain. For more information, see “Algorithms” on page 4-338.
- 'Fastest' -- Computes the cross-correlation in the domain that minimizes the number of computations.

To cross-correlate fixed-point signals, set this property to 'Time Domain'.

### Fixed-Point Properties

---

**Note** Fixed-point signals are supported for the time domain only. To use these properties, set Method to 'Time Domain'.

---

### FullPrecisionOverride — Full-precision override for fixed-point arithmetic

true (default) | false

Flag to use full-precision rules for fixed-point arithmetic, specified as one of the following:

- true -- The object computes all internal arithmetic and output data types using the full-precision rules. These rules provide the most accurate fixed-point numerics. In this mode, other fixed-point properties do not apply. No quantization occurs within the object. Bits are added, as needed, to ensure that no roundoff or overflow occurs.
- false -- Fixed-point data types are controlled through individual fixed-point property settings.

For more information, see “Full Precision for Fixed-Point System Objects” and “Set System Object Fixed-Point Properties”.

### RoundingMethod — Rounding method for fixed-point operations

'Floor' (default) | 'Ceiling' | 'Convergent' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'

Rounding method for fixed-point operations. For more details, see rounding mode.

### Dependencies

This property is not visible and has no effect on the numerical results when the following conditions are met:

- FullPrecisionOverride set to true.
- FullPrecisionOverride set to false, OutputDataType set to 'Same as accumulator', ProductDataType set to 'Full precision', and AccumulatorDataType set to 'Full precision'

Under these conditions, the object operates in full precision mode.

In addition, if Method is set to either 'Frequency Domain' or 'Fastest', the RoundingMethod property does not apply.

### **OverflowAction — Overflow action for fixed-point operations**

'Wrap' (default) | 'Saturate'

Overflow action for fixed-point operations, specified as one of the following:

- 'Wrap' -- The object wraps the result of its fixed-point operations.
- 'Saturate' -- The object saturates the result of its fixed-point operations.

For more details on overflow actions, see overflow mode for fixed-point operations.

### **Dependencies**

This property is not visible and has no effect on the numerical results when the following conditions are met:

- FullPrecisionOverride set to true.
- FullPrecisionOverride set to false, OutputDataType set to 'Same as accumulator', ProductDataType set to 'Full precision', and AccumulatorDataType set to 'Full precision'

Under these conditions, the object operates in full precision mode.

In addition, if Method is set to either 'Frequency Domain' or 'Fastest', the OverflowAction property does not apply.

### **ProductDataType — Data type of product output**

'Full precision' (default) | 'Custom' | 'Same as first input'

Data type of the product output in this object, specified as one of the following:

- 'Full precision' -- The product output data type has full precision.
- 'Same as first input' -- The object specifies the product output data type to be the same as that of the first input data type.
- 'Custom' -- The product output data type is specified as a custom numeric type through the "CustomProductDataType" on page 4-0 property.

For more information on the product output data type, see "Multiplication Data Types" and the "Fixed Point" on page 4-337 section.

### **Dependencies**

This property applies when you set FullPrecisionOverride to false.

### **CustomProductDataType — Word and fraction lengths of product data type**

numericType([],32,30) (default)

Word and fraction lengths of the product data type, specified as an autosigned numeric type with a word length of 32 and a fraction length of 30.

**Dependencies**

This property applies only when you set `FullPrecisionOverride` to `false` and “`ProductDataType`” on page 4-0 to `'Custom'`.

**AccumulatorDataType — Data type of accumulation operation**

`'Full precision' (default) | 'Same as first input' | 'Same as product' | 'Custom'`

Data type of an accumulation operation in this object, specified as one of the following:

- `'Full precision'` -- The accumulation operation has full precision.
- `'Same as product'` -- The object specifies the accumulator data type to be the same as that of the product output data type.
- `'Same as first input'` -- The object specifies the accumulator data type to be the same as that of the first input data type.
- `'Custom'` -- The accumulator data type is specified as a custom numeric type through the “`CustomAccumulatorDataType`” on page 4-0 property.

For more information on the accumulator data type this object uses, see the “Fixed Point” on page 4-337 section.

**Dependencies**

This property applies when you set `FullPrecisionOverride` to `false`.

**CustomAccumulatorDataType — Word and fraction lengths of accumulator data type**

`numericType([],32,30) (default)`

Word and fraction lengths of the accumulator data type, specified as an autosigned numeric type with a word length of 32 and a fraction length of 30.

**Dependencies**

This property applies only when you set `FullPrecisionOverride` to `false` and “`AccumulatorDataType`” on page 4-0 to `'Custom'`.

**OutputDataType — Data type of object output**

`'Same as accumulator' (default) | 'Same as first input' | 'Same as product' | 'Custom'`

Data type of the object output, specified as one of the following:

- `'Same as accumulator'` -- The output data type is the same as that of the accumulator output data type.
- `'Same as first input'` -- The output data type is the same as that of the first input data type.
- `'Same as product'` -- The output data type is the same as that of the product output data type.
- `'Custom'` -- The output data type is specified as a custom numeric type through the “`CustomOutputDataType`” on page 4-0 property.

For more information on the output data type this object uses, see the “Fixed Point” on page 4-337 section.

**Dependencies**

This property applies when you set `FullPrecisionOverride` to `false`.

**CustomOutputDataType — Word and fraction lengths of output data type**

`numericType([],16,15)` (default)

Word and fraction lengths of the output data type, specified as an autosigned numeric type with a word length of 16 and a fraction length of 15.

**Dependencies**

This property applies only when you set `FullPrecisionOverride` to `false` and “`OutputDataType`” on page 4-0 to 'Custom'.

**Usage****Syntax**

```
y = xcorr(u,v)
```

**Description**

`y = xcorr(u,v)` computes the cross-correlation of the two input signals, `u` and `v`.

**Input Arguments****u — First data input signal**

vector | matrix | *N*-D array

First data input signal, specified as a vector, matrix, or an *N*-D array. The object accepts real-valued or complex-valued multichannel and multidimensional inputs. The input can be a fixed-point signal when you set the `Method` property to 'Time Domain'. When one or both of the input signals are complex, the output signal is also complex. Both data inputs must have the same data type.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

**v — Second data input signal**

scalar | column vector | matrix

Second data input signal, specified as a vector, matrix, or an *N*-D array. The object accepts real-valued or complex-valued multichannel and multidimensional inputs. The input can be a fixed-point signal when you set the `Method` property to 'Time Domain'. When one or both of the input signals are complex, the output signal is also complex. Both data inputs must have the same data type.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

**Output Arguments****y — Cross-correlated output**

vector | matrix | *N*-D array

Cross-correlated output of the two input signals.

When the inputs are  $N$ -D arrays, the object outputs an  $N$ -D array. All the dimensions of the output array, except for the first dimension, match the input array. For example:

- When the inputs  $u$  and  $v$  have dimensions  $M_u$ -by- $N$ -by- $P$  and  $M_v$ -by- $N$ -by- $P$ , respectively, the object outputs an  $(M_u + M_v - 1)$ -by- $N$ -by- $P$  array.
- When the inputs  $u$  and  $v$  have the dimensions  $M_u$ -by- $N$  and  $M_v$ -by- $N$ , the object outputs an  $(M_u + M_v - 1)$ -by- $N$  matrix.

If one input is a column vector and the other input is an  $N$ -D array, the object computes the cross-correlation of the vector with each column in the  $N$ -D array. For example:

- When the input  $u$  is an  $M_u$ -by-1 column vector and  $v$  is an  $M_v$ -by- $N$  matrix, the object outputs an  $(M_u + M_v - 1)$ -by- $N$  matrix.
- Similarly, when  $u$  and  $v$  are column vectors with lengths  $M_u$  and  $M_v$ , respectively, the object performs the vector cross-correlation.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Common to All System Objects

|                      |  |
|----------------------|--|
| <code>step</code>    | Run System object algorithm  |
| <code>release</code> | Release resources and allow changes to System object property values and input characteristics |
| <code>reset</code>   | Reset internal states of System object   |

## Examples

### Cross-Correlation of Input Noise and Delayed Version

Compute the cross-correlation of a noisy input signal with its delayed version. The peak of the correlation output occurs at the lag, which corresponds to the delay between the signals.

Use `randn` to create the white Gaussian noisy input, `x`. Create a delayed version of this input, `x1`, using the `dsp.Delay` object.

```
S = rng('default');  
x = randn(100,1);  
delay = dsp.Delay(10);  
x1 = delay(x);
```

Compute the cross-correlation between the two inputs. Plot the correlation output with respect to the lag between the inputs.

```
xcorr = dsp.Crosscorrelator;  
y = xcorr(x1,x);
```

```
lags = 0:99;
stem(lags,y(100:end),'markerfacecolor',[0 0 1])
axis([0 99 -125 125])
xlabel('Lags')
title('Cross-Correlation of Input Noise and Delayed Version')
```

The correlation sequence peaks when the lag is 10, indicating that the correct delay between the two signals is 10 samples.

## More About

### Cross-Correlation

Cross-correlation is the measure of similarity of two discrete-time sequences as a function of the lag of one relative to the other.

For two length- $N$  deterministic inputs or realizations of jointly wide-sense stationary (WSS) random processes,  $x$  and  $y$ , the cross-correlation is computed using the following relationship:

$$r_{xy}(h) = \begin{cases} \sum_{n=0}^{N-h-1} x(n+h)y^*(n) & 0 \leq h \leq N-1 \\ r_{yx}^*(h) & -(N-1) \leq h \leq 0 \end{cases}$$

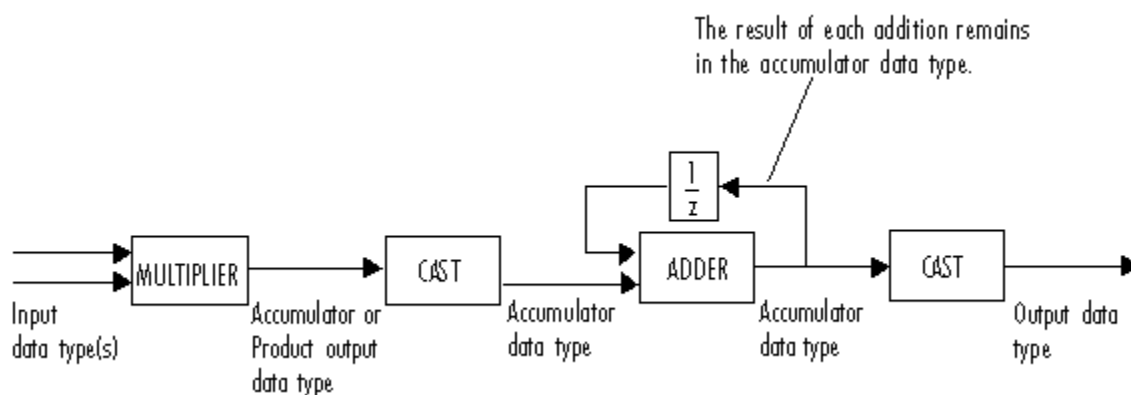
where  $h$  is the lag and  $*$  denotes the complex conjugate. If the inputs are realizations of jointly WSS stationary random processes,  $r_{xy}(h)$  is an unnormalized estimate of the theoretical cross-correlation:

$$\rho_{xy}(h) = E\{x(n+h)y^*(n)\}$$

where  $E\{ \}$  is the expectation operator.

### Fixed Point

The diagram shows the data types the `dsp.Crosscorrelator` object uses for fixed-point signals (time domain only).



You can set the product output, accumulator, and output data types using the corresponding fixed-point properties of the object.

When the input is real, the output of the multiplier is in the product output data type. When the input is complex, the output of the multiplier is in the accumulator data type. For details on the complex multiplication performed, see “Multiplication Data Types”.

---

**Note** When one or both of the inputs are signed fixed-point signals, all internal object data types are signed fixed point. The internal object data types are unsigned fixed point only when both inputs are unsigned fixed-point signals.

---

## Algorithms

### Time-Domain Computation

When you set the computation domain to time, the algorithm computes the cross-correlation of two signals in the time domain. The input signals can be fixed-point signals in this domain.

#### Correlate Two 2-D Arrays

When the inputs are two 2-D arrays, the  $j$ th column of the output,  $y_{u,v}$ , has these elements:

$$y_{uv}(i, j) = \sum_{k=0}^{\max(M_u, M_v) - 1} u_{k, j}^* v_{(k+i), j} \quad 0 \leq i < M_v$$

$$y_{uv}(i, j) = y_{vu}^*(-i, j) \quad -M_u < i < 0$$

where:

- \* denotes the complex conjugate.
- $u$  is an  $M_u$ -by- $N$  input matrix.
- $v$  is an  $M_v$ -by- $N$  input matrix.
- $y_{u,v}$  is an  $(M_u + M_v - 1)$ -by- $N$  matrix.

Inputs  $u$  and  $v$  are zero when indexed outside their valid ranges.

#### Correlate a Column Vector with a 2-D Array

When one input is a column vector and the other input is a 2-D array, the algorithm independently cross-correlates the input vector with each column of the 2-D array. The  $j$ th column of the output,  $y_{u,v}$ , has these elements:

$$y_{uv}(i, j) = \sum_{k=0}^{\max(M_u, M_v) - 1} u_k^* v_{(k+i), j} \quad 0 \leq i < M_v$$

$$y_{uv}(i, j) = y_{vu}^*(-i, j) \quad -M_u < i < 0$$

where:

- \* denotes the complex conjugate.
- $u$  is an  $M_u$ -by-1 column vector.
- $v$  is an  $M_v$ -by- $N$  matrix.



- $y_{uv}$  is an  $(M_u + M_v - 1)$ -by- $N$  matrix.

Inputs  $u$  and  $v$  are zero when indexed outside their valid ranges.

### Correlate Two Column Vectors

When the inputs are two column vectors, the  $j$ th column of the output,  $y_{uv}$ , has these elements:

$$y_{uv(i)} = \sum_{k=0}^{\max(M_u, M_v) - 1} u_k^* v_{(k+i)} \quad 0 \leq i < M_v$$

$$y_{uv(i)} = y_{vu}^*(-i) \quad -M_u < i < 0$$

where:

- $*$  denotes the complex conjugate.
- $u$  is an  $M_u$ -by-1 column vector.
- $v$  is an  $M_v$ -by-1 column vector.
- $y_{uv}$  is an  $(M_u + M_v - 1)$ -by-1 column vector.

Inputs  $u$  and  $v$  are zero when indexed outside their valid ranges.

### Frequency-Domain Computation

When you set the computation domain to frequency, the algorithm computes the cross-correlation in the frequency domain.

To compute the cross-correlation, the algorithm:

- 1 Takes the Fourier transform of both input signals,  $U$  and  $V$ .
- 2 Multiplies  $U$  and  $V^*$ , where  $*$  denotes the complex conjugate.
- 3 Computes the inverse Fourier transform of the product.

In this domain, depending on the input length, the algorithm can require fewer computations.

## Version History

### Introduced in R2012a

#### **R2021b: dsp.Crosscorrelator System object will be removed**

*Warns starting in R2021b*

`dsp.Crosscorrelator` System object will be removed in a future release. Use the `xcorr` function instead.

#### **R2023a: dsp.Crosscorrelator System object has been removed**

*Errors starting in R2023a*

The `dsp.Crosscorrelator` System object has been removed. Use the `xcorr` function instead.

### Update Code

This table shows how to update existing code to use the `xcorr` function.

| Discouraged Usage  | Recommended Replacement  |
|--|--|
| <pre> xc0bj = dsp.Crosscorrelator; t = 0:0.001:1; x1 = sin(2*pi*2*t) + 0.05*sin(2*pi*50*t); x2 = sin(2*pi*2*t); % Computes cross-correlation of x1 and x2 y = xc0bj(x1',x2'); figure; plot(t,x1,'b',t,x2,'g'); legend('Input signal 1', 'Input signal 2') figure; plot(y); title('Correlated output') </pre> | <pre> [yfn,lags] = xcorr(x1',x2'); figure; plot(yfn) </pre> <p>Plot the signal with respect to the lags.</p> <pre> figure; plot(lags,yfn) </pre> |

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

## See Also

### Functions

xcorr

### Blocks

Correlation

# dsp.CumulativeProduct

**Package:** dsp

(Removed) Cumulative product of channel, column, or row elements

---

**Note** dsp.CumulativeProduct has been removed. Use the cumprod function instead. For more information, see “Compatibility Considerations”.

---

## Description

The dsp.CumulativeProduct System object computes the cumulative product of channel, column, or row elements.

To compute the cumulative product of channel, column, or row elements:

- 1 Create the dsp.CumulativeProduct object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

## Creation

### Syntax

```
cprod = dsp.CumulativeProduct  
cprod = dsp.CumulativeProduct(Name,Value)
```

### Description

cprod = dsp.CumulativeProduct returns a cumulative product object, cprod, that computes the cumulative product of input matrix or input vector elements along the default dimension.

cprod = dsp.CumulativeProduct(Name,Value) returns a cumulative product object with each specified property set to the specified value.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the release function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

### Dimension — Computation dimension for cumulative product

Channels (running product) (default) | Rows | Columns

Specify the computation dimension as `Channels (running product)`, `Rows`, or `Columns`.

**ResetInputPort — Enable resetting cumulative product through input port**

`false` (default) | `true`

Set this property to `true` to enable resetting the cumulative product. When you set this property to `true`, specify a reset signal to the object algorithm to reset the cumulative product.

**Dependencies**

You can access this property when the “Dimension” on page 4-0 property is set to `Channels (running product)`.

**ResetCondition — Reset condition for cumulative product**

`Rising edge` (default) | `Falling edge` | `Either edge` | `Non-zero`

Specify the event on the reset input port that causes resetting the cumulative product to `Rising edge`, `Falling edge`, `Either edge`, or `Non-zero`.

**Dependencies**

This property applies when you set the “ResetInputPort” on page 4-0 property to `true` and the “Dimension” on page 4-0 property to `Channels (running product)`.

**Fixed-Point Properties****RoundingMethod — Rounding method for fixed-point operations**

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

Specify the rounding method as one of `Ceiling`, `Convergent`, `Floor`, `Nearest`, `Round`, `Simplest`, or `Zero`.

**OverflowAction — Overflow action for fixed-point operations**

`Wrap` (default) | `Saturate`

Specify the overflow action as one of `Wrap` or `Saturate`.

**IntermediateProductDataType — Intermediate product word and fraction lengths**

`Same as input` (default) | `Custom`

Specify the intermediate product fixed-point data type as `Same as input` or `Custom`.

**CustomIntermediateProductDataType — Intermediate product word and fraction lengths**

`numerictype([], 16, 15)` (default) | `numerictype`

Specify the intermediate product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`.

**Dependencies**

This property applies when you set the `IntermediateProductDataType` property to `Custom`.

**ProductDataType — Product output word and fraction lengths**

`Same as input` (default) | `Custom`

Specify the product output fixed-point data type as one of | `Same as input` | `Custom` |.

**CustomProductDataType — Custom product output word and fraction lengths**

numericity([], 32, 30) (default) | numericity

Specify the product output fixed-point type as a scaled numericity object with a Signedness of Auto.

**Dependencies**

This property applies when you set the ProductDataType property to Custom.

**AccumulatorDataType — Accumulator word and fraction lengths**

Same as input (default) | Same as product output | Custom

Specify the accumulator fixed-point data type as Same as product output, Same as input, or Custom.

**CustomAccumulatorDataType — Custom accumulator word and fraction lengths**

numericity([], 32, 30) (default) | numericity

Specify the accumulator fixed-point type as a scaled numericity object with a Signedness of Auto.

**Dependencies**

This property applies when you set the AccumulatorDataType property to Custom.

**OutputDataType — Output word and fraction lengths**

Same as input (default) | Same as product output | Custom

Specify the output fixed-point data type as one of | Same as product output | Same as input | Custom |.

**CustomOutputDataType — Custom output word and fraction lengths**

numericity([], 16, 15) (default) | numericity

Specify the output fixed-point type as a scaled numericity object with a Signedness of Auto.

**Dependencies**

This property applies when you set the “OutputDataType” on page 4-0 property to Custom.

**Usage****Syntax**

```
y = cprod(x)
y = cprod(x,r)
```

**Description**

$y = \text{cprod}(x)$  computes the cumulative product along the specified dimension for the input  $x$ .

$y = \text{cprod}(x,r)$  resets the cumulative product object's state based on the ResetCondition property value and the value of the reset signal,  $r$ , when the ResetInputPort property is true.

## Input Arguments

### **x — Data input**

vector | matrix

Data input, specified as a vector or a matrix.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

### **r — Reset signal**

scalar

Reset signal used to reset the running cumulative product, specified as a scalar. The object resets the running cumulative product if the reset signal satisfies the `ResetCondition`.

## Dependencies

This input is applicable only when `Dimension` is set to `'Channels (running product)'` and `ResetInputPort` is set to `true`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `logical` | `fi`

## Output Arguments

### **y — Cumulative product**

vector | matrix

Cumulative product of the input signal, returned as a vector or a matrix.

The size, data type, and complexity characteristics of the output signal match that of the input signal.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Common to All System Objects

|                      |  |
|----------------------|--|
| <code>step</code>    | Run System object algorithm  |
| <code>release</code> | Release resources and allow changes to System object property values and input characteristics |
| <code>reset</code>   | Reset internal states of System object   |

## Examples

### Cumulative Product of Matrix

Use the `dsp.CumulativeProduct` object to compute the cumulative product of a matrix.

```
cprod = dsp.CumulativeProduct;  
x = magic(2)
```

```
x = 2×2
```

```
    1    3  
    4    2
```

```
y = cprod(x)
```

```
y = 2×2
```

```
    1    3  
    4    6
```

The cumulative product is computed column-wise along each channel.

## Algorithms

This object implements the algorithm, inputs, and outputs described on the Cumulative Product block reference page. The object properties correspond to the block parameters, except the **Reset port** block parameter corresponds to both the ResetCondition and ResetInputPort object properties.

## Version History

### Introduced in R2012a

### R2021b: dsp.CumulativeProduct System object will be removed

*Warns starting in R2021b*

dsp.CumulativeProduct System object will be removed in a future release. Use the cumprod function instead.

### R2023a: dsp.CumulativeProduct System object has been removed

*Errors starting in R2023a*

The dsp.CumulativeProduct System object has been removed. Use the cumprod function instead.

## Update Code

This table shows how to update existing code to use the cumprod function.

| Discouraged Usage   | Recommended Replacement  |
|---|--|
| <p><b>Column-wise cumulative product</b></p> <p>By default, the object computes the running product along the columns (channels) since the Dimension property of the object is set to 'Channels (running product)'.<br/> <code>cprod = dsp.CumulativeProduct;</code><br/> <code>x = magic(2)</code></p> <p><code>x = 2x2</code></p> <pre> 1     3 4     2 </pre> <p><code>y = cprod(x)</code></p> <p><code>y = 2x2</code></p> <pre> 1     3 4     6 </pre> <p><b>Row-wise cumulative product</b></p> <p>To compute the row-wise cumulative product, set the Dimension property to 'Rows' and run the object algorithm again.</p> <pre> release(cprod); cprod.Dimension = 'Rows'; y = cprod(x) </pre> <p><code>y = 2x2</code></p> <pre> 1     3 4     8 </pre> | <p><b>Column-wise cumulative product</b></p> <pre> y = cumprod(x,1) </pre> <p><code>y = 2x2</code></p> <pre> 1     3 4     6 </pre> <p><b>Row-wise cumulative product</b></p> <pre> y = cumprod(x,2) </pre> <p><code>y = 2x2</code></p> <pre> 1     3 4     8 </pre> |

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

## See Also

### Functions

`cumprod`



# dsp.CumulativeSum

**Package:** dsp

(Removed) Cumulative sum of channel, column, or row elements

---

**Note** dsp.CumulativeSum has been removed. Use the cumsum function instead. For more information, see “Compatibility Considerations”.

---

## Description

The dsp.CumulativeSum System object computes the cumulative sum of channel, column, or row elements.

To compute the cumulative sum of channel, column, or row elements:

- 1 Create the dsp.CumulativeSum object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

## Creation

### Syntax

```
csum = dsp.CumulativeSum  
csum = dsp.CumulativeSum(Name,Value)
```

### Description

csum = dsp.CumulativeSum returns a cumulative sum System object, csum, which computes the running cumulative sum for each channel in the input.

csum = dsp.CumulativeSum(Name,Value) returns a cumulative sum object, csum, with each specified property set to the specified value. Enclose each property name in single quotes. Unspecified properties have default values.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the release function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

### Dimension — Computation dimension for cumulative sum

Channels (running sum) (default) | Rows | Columns

Specify the computation dimension as `Channels` (running `sum`), `Rows`, or `Columns`.

**ResetInputPort — Enable resetting cumulative sum through input port**

`false` (default) | `true`

Set this property to `true` to enable resetting the cumulative sum. When you set this property to `true`, you also specify a reset input to the object algorithm to reset the cumulative sum.

**ResetCondition — Reset condition for cumulative sum**

`Rising edge` (default) | `Falling edge` | `Either edge` | `Non-zero`

Specify the event on the reset input port that resets the cumulative sum as `Rising edge`, `Falling edge`, `Either edge`, or `Non-zero`.

**Dependencies**

This property applies when you set the “ResetInputPort” on page 4-0 property to `true`.

**Fixed-Point Properties****RoundingMethod — Rounding method for fixed-point operations**

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

Specify the rounding method as one of `Ceiling`, `Convergent`, `Floor`, `Nearest`, `Round`, `Simplest`, or `Zero`.

**OverflowAction — Overflow action for fixed-point operations**

`Wrap` (default) | `Saturate`

Specify the overflow action as one of `Wrap` or `Saturate`.

**AccumulatorDataType — Accumulator word and fraction lengths**

`Same as input` (default) | `Custom`

Specify the accumulator fixed-point data type as `Same as input` or `Custom`.

**CustomAccumulatorDataType — Custom accumulator word and fraction lengths**

`numerictype([], 32, 30)` (default) | `numerictype`

Specify the accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`.

**Dependencies**

This property applies when you set the `AccumulatorDataType` property to `Custom`.

**OutputDataType — Output word and fraction lengths**

`Same as accumulator` (default) | `Same as input` | `Custom`

Specify the output fixed-point data type as `Same as accumulator`, `Same as input`, or `Custom`.

**CustomOutputDataType — Custom output word and fraction lengths**

`numerictype([], 16, 15)` (default) | `numerictype`

Specify the output fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`.

**Dependencies**

This property applies when you set the “OutputDataType” on page 4-0 property to Custom.

**Usage****Syntax**

```
y = csum(x)
y = csum(x, r)
```

**Description**

$y = \text{csum}(x)$  computes the cumulative sum along the specified dimension for the input  $x$ .

$y = \text{csum}(x, r)$  resets the System object state based on the ResetCondition property value and the value of the reset signal,  $r$ . You can only reset the state if the ResetInputPort property is true.

**Input Arguments****x — Data input**

vector | matrix

Data input, specified as a vector or a matrix.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fi

**r — Reset signal**

scalar

Reset signal used to reset the running cumulative sum, specified as a scalar. The object resets the running cumulative sum if the reset signal satisfies the ResetCondition.

**Dependencies**

This input is applicable only when Dimension is set to 'Channels (running sum)' and ResetInputPort is set to true.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | logical | fi

**Output Arguments****y — Cumulative sum**

vector | matrix

Cumulative sum of the input signal, returned as a vector or a matrix.

The size, data type, and complexity characteristics of the output signal match that of the input signal.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fi

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Common to All System Objects

|         |  |
|---------|--|
| step    | Run System object algorithm  |
| release | Release resources and allow changes to System object property values and input characteristics |
| reset   | Reset internal states of System object   |

## Examples

### Cumulative Sum of Matrix

Use the `dsp.CumulativeSum` object to compute the cumulative sum of a matrix.

```
csum = dsp.CumulativeSum;  
x = magic(2)
```

```
x = 2×2
```

```
    1    3  
    4    2
```

```
y = csum(x)
```

```
y = 2×2
```

```
    1    3  
    5    5
```

The cumulative sum is computed column-wise along each channel.

## Algorithms

This object implements the algorithm, inputs, and outputs described on the Cumulative Sum block reference page. The object properties correspond to the block properties, except the **Reset port** block parameter corresponds to both the `ResetCondition` and the `ResetInputPort` object properties.

## Version History

**Introduced in R2012a**

**R2021b: `dsp.CumulativeSum` System object will be removed**

*Warns starting in R2021b*

`dsp.CumulativeSum` System object will be removed in a future release. Use the `cumsum` function instead.

### R2023a: `dsp.CumulativeSum` System object has been removed

*Errors starting in R2023a*

The `dsp.CumulativeSum` System object has been removed. Use the `cumsum` function instead.

### Update Code

This table shows how to update existing code to use the `cumsum` function.

| Discouraged Usage   | Recommended Replacement   |
|---|---|
| <p><b>Column-wise cumulative sum</b></p> <p>By default, the object computes the running sum along the columns (channels) since the <code>Dimension</code> property of the object is set to <code>'Channels (running sum)'</code>.</p> <pre>csum = dsp.CumulativeSum; x = magic(2)</pre> <p><math>x = 2 \times 2</math></p> <pre>    1    3     4    2</pre> <p><code>y = csum(x)</code></p> <p><math>y = 2 \times 2</math></p> <pre>    1    3     5    5</pre> | <p><b>Column-wise cumulative sum</b></p> <pre>y = cumsum(x,1)</pre> <p><math>y = 2 \times 2</math></p> <pre>    1    3     5    5</pre> |
| <p><b>Row-wise cumulative sum</b></p> <p>To compute row-wise cumulative sum, set the <code>Dimension</code> property to <code>'Rows'</code> and run the object algorithm again.</p> <pre>release(csum); csum.Dimension = 'Rows'; y = csum(x)</pre> <p><math>y = 2 \times 2</math></p> <pre>    1    4     4    6</pre>  | <p><b>Row-wise cumulative sum</b></p> <pre>y = cumsum(x,2)</pre> <p><math>y = 2 \times 2</math></p> <pre>    1    4     4    6</pre>    |

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

### **See Also**

#### **Functions**

cumsum

# dsp.DCBlocker

**Package:** dsp

Block DC component (offset) from input signal

## Description

The `dsp.DCBlocker` System object removes the DC offset from each channel (column) of the input signal. The operation runs over time to continually estimate and remove the DC offset.

To block the DC component of the input signal:

- 1 Create the `dsp.DCBlocker` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
dcblk = dsp.DCBlocker
dcblk = dsp.DCBlocker(Name, Value)
```

### Description

`dcblk = dsp.DCBlocker` creates a DC blocker System object, `dcblk`, to block the DC component from each channel (column) of the input signal.

`dcblk = dsp.DCBlocker(Name, Value)` creates a DC blocker System object, `dcblk`, with each specified property set to the specified value. Enclose each property name in single quotes.

Example: `dcblk = dsp.DCBlocker('Algorithm','FIR')`

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

### Algorithm — Algorithm for estimating DC offset

'IIR' (default) | 'FIR' | 'CIC' | 'Subtract mean'

Algorithm for estimating DC offset, specified as one of the following:

- 'IIR' -- The object uses a recursive estimate based on a narrow, lowpass elliptic filter. The "Order" on page 4-0 property sets the order of the filter, and the "NormalizedBandwidth" on page 4-0 property sets its bandwidth. This algorithm typically uses less memory than the FIR algorithm and is more efficient.
- 'FIR' -- The object uses a nonrecursive, moving average estimate based on a finite number of past input samples. The "Length" on page 4-0 property sets the number of samples. The FIR filter has a linear phase response and does not cause any phase distortion to the signal. The IIR filter requires less memory and is more efficient to implement.
- 'CIC' -- The object uses a CIC decimator, with a decimation factor of 1, whose differential delay is calculated using the NormalizedBandwidth property. It uses two sections to ensure that the first sidelobe attenuation is at least 25 dB below the main lobe of the filter. This algorithm requires fixed-point inputs and can be used for HDL code generation.
- 'Subtract mean' -- The object computes the means of the columns of the input matrix, and subtracts the means from the input. This method does not retain state between inputs.

You can visualize the IIR, FIR, and CIC responses by using the `fvtool` function.

### **NormalizedBandwidth — Normalized bandwidth of lowpass IIR or CIC filter**

0.001 (default) | real scalar greater than 0 and less than 1

Normalized bandwidth of the IIR or CIC filter, specified as a real scalar greater than 0 or less than 1. The normalized bandwidth is used to estimate the DC component of the input signal.

#### **Dependencies**

This property applies only when you set the "Algorithm" on page 4-0 property to 'IIR' or 'CIC'.

### **Order — Order of lowpass IIR elliptic filter**

6 (default) | integer greater than 3

Order of the lowpass IIR elliptic filter that is used to estimate the DC level, specified as an integer greater than 3.

#### **Dependencies**

This property applies only when you set the "Algorithm" on page 4-0 property to 'IIR'.

### **Length — Number of past input samples used in FIR algorithm**

50 (default) | positive integer

Number of past input samples used in the FIR algorithm to estimate the running mean, specified as a positive integer.

#### **Dependencies**

This property applies only when you set the "Algorithm" on page 4-0 property to 'FIR'.

## **Usage**

### **Syntax**

```
dcblkOut = dcblk(input)
```



## Description

`dcblkerOut = dcblker(input)` removes the DC component from each channel (column) of the input and returns the output.

## Input Arguments

### **input** — Input signal

vector | matrix | *N*-D array

Data input to the DC blocker object, specified as a vector, matrix, or *N*-D array.

Example: `t = (0:0.001:100)'`; `x = sin(30*pi*t) + 1`;

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `fi`

## Output Arguments

### **dcblkerOut** — Signal with DC component removed

vector | matrix | *N*-D array

Signal with DC component removed, returned as a vector, matrix, or *N*-D array. The output dimensions match the input dimensions.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `fi`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Visualization Specific to DSP System Toolbox

`fvtool` Visualize frequency response of DSP filters

## Common to All System Objects

`step` Run System object algorithm

`release` Release resources and allow changes to System object property values and input characteristics

`reset` Reset internal states of System object

## Examples

### Remove DC Component and Display Results

Remove the DC component of an input signal using the IIR, FIR, and subtract mean estimation algorithms.

Create a signal composed of a 15 Hz tone, a 25 Hz tone, and a DC offset.

```
t = (0:0.001:100)';
x = sin(30*pi*t) + 0.33*cos(50*pi*t) + 1;
```

Create three DC blocker objects for the three estimation algorithms.

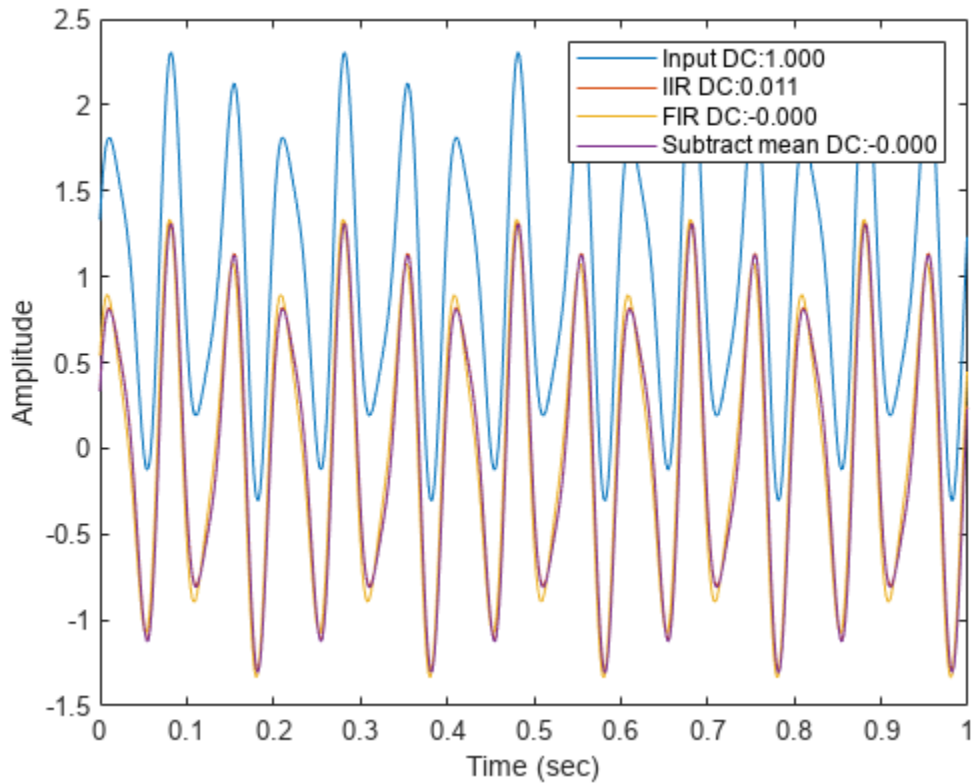
```
dc1 = dsp.DCBlocker('Algorithm','IIR','Order', 6);  
dc2 = dsp.DCBlocker('Algorithm','FIR','Length', 100);  
dc3 = dsp.DCBlocker('Algorithm','Subtract mean');
```

For each second of time, pass the input signal through the DC blockers. By implementing the DC blockers in 1-second increments, you can observe differences in the convergence times.

```
for idx = 1 : 100  
    range = (1:1000) + 1000*(idx-1);  
    y1 = dc1(x(range));           % IIR estimate  
    y2 = dc2(x(range));           % FIR estimate  
    y3 = dc3(x(range));           % Subtract mean  
end
```

Plot the input and output data for the three DC blockers for the first second of time, and show the mean value for each signal. The mean values for the three algorithm types show that the FIR and Subtract mean algorithms converge more quickly.

```
plot(t(1:1000),x(1:1000), ...  
     t(1:1000),y1, ...  
     t(1:1000),y2, ...  
     t(1:1000),y3);  
xlabel('Time (sec)')  
ylabel('Amplitude')  
legend(sprintf('Input DC:%.3f',mean(x)), ...  
        sprintf('IIR DC:%.3f',mean(y1)), ...  
        sprintf('FIR DC:%.3f',mean(y2)), ...  
        sprintf('Subtract mean DC:%.3f',mean(y3)));
```



### Frequency Response Before and After DC Blocker

Compare the spectrum of an input signal with a DC offset to the spectrum of the same signal after applying a DC blocker. Enable the DC blocker to use the FIR estimation algorithm.

Create an input signal composed of three tones and that has a DC offset of 1. Set the sampling frequency to 1 kHz and set the signal duration to 100 seconds.

```
fs = 1000;
t = (0:1/fs:100)';
x = sin(30*pi*t) + 0.67*sin(40*pi*t) + 0.33*sin(50*pi*t) + 1;
```

Create a DC blocker object that uses the FIR algorithm to estimate the DC offset.

```
dcblker = dsp.DCBlocker('Algorithm','FIR','Length',100);
```

Create a spectrum analyzer with power units set to dBW and a frequency range of [-30 30] to display the frequency response of the input signal. Using the clone function, create a second spectrum analyzer to display the response of the output. Then, use the Title property of the spectrum analyzers to label them.

```
hsa = spectrumAnalyzer('SampleRate',fs, ...
    'Method','welch',...
    'AveragingMethod','exponential',...
```

```
'SpectrumUnits','dBW','FrequencySpan','start-and-stop-frequencies',...
'StartFrequency',-30,'StopFrequency',30,'YLimits',[-200 20],...
'Title','Signal Spectrum');
```

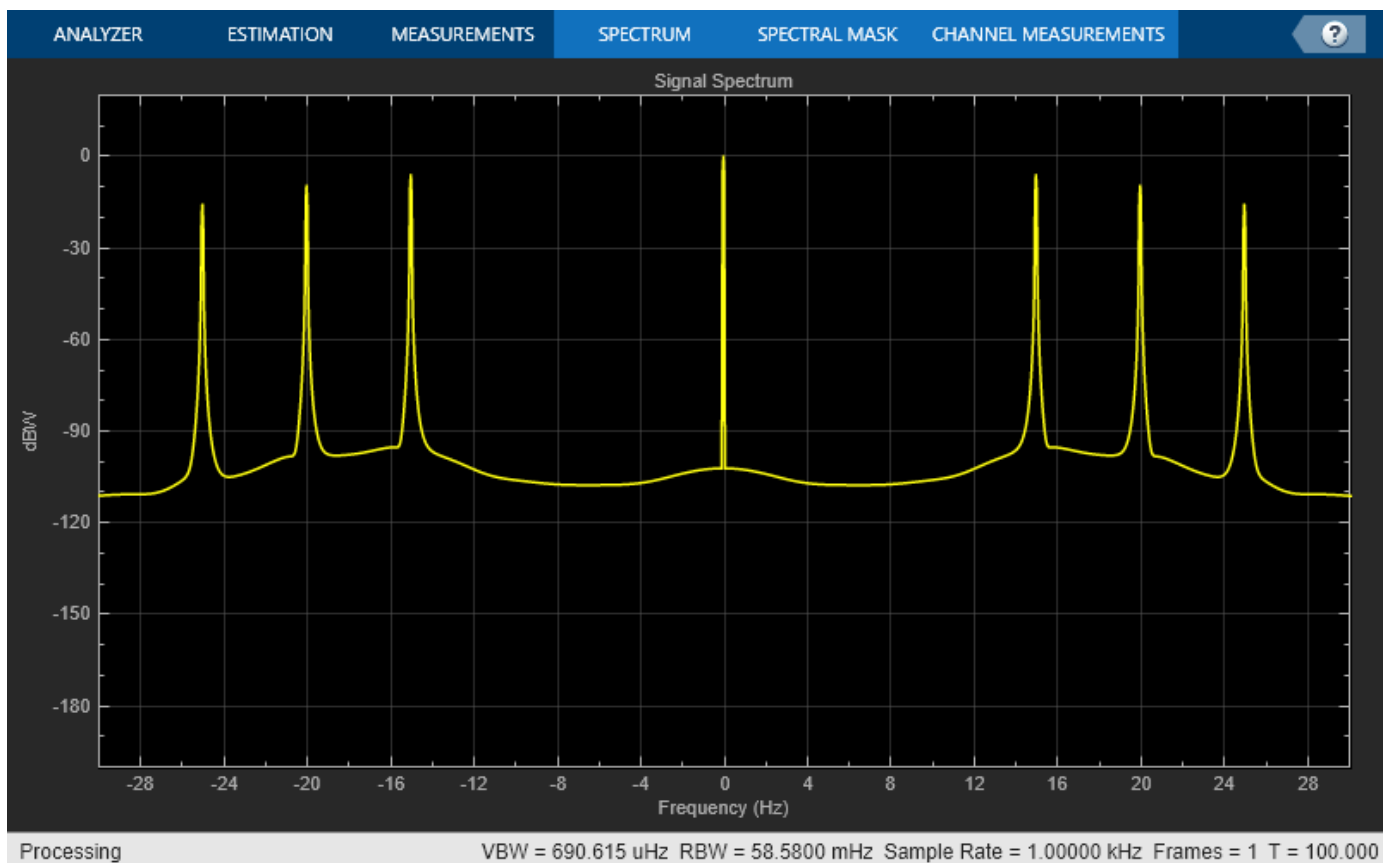
```
hsb = clone(hsa);
hsb.Title = 'Signal Spectrum After DC Blocker';
```

Pass the input signal,  $x$ , through the DC blocker to generate the output signal,  $y$ .

```
y = dcbkcr(x);
```

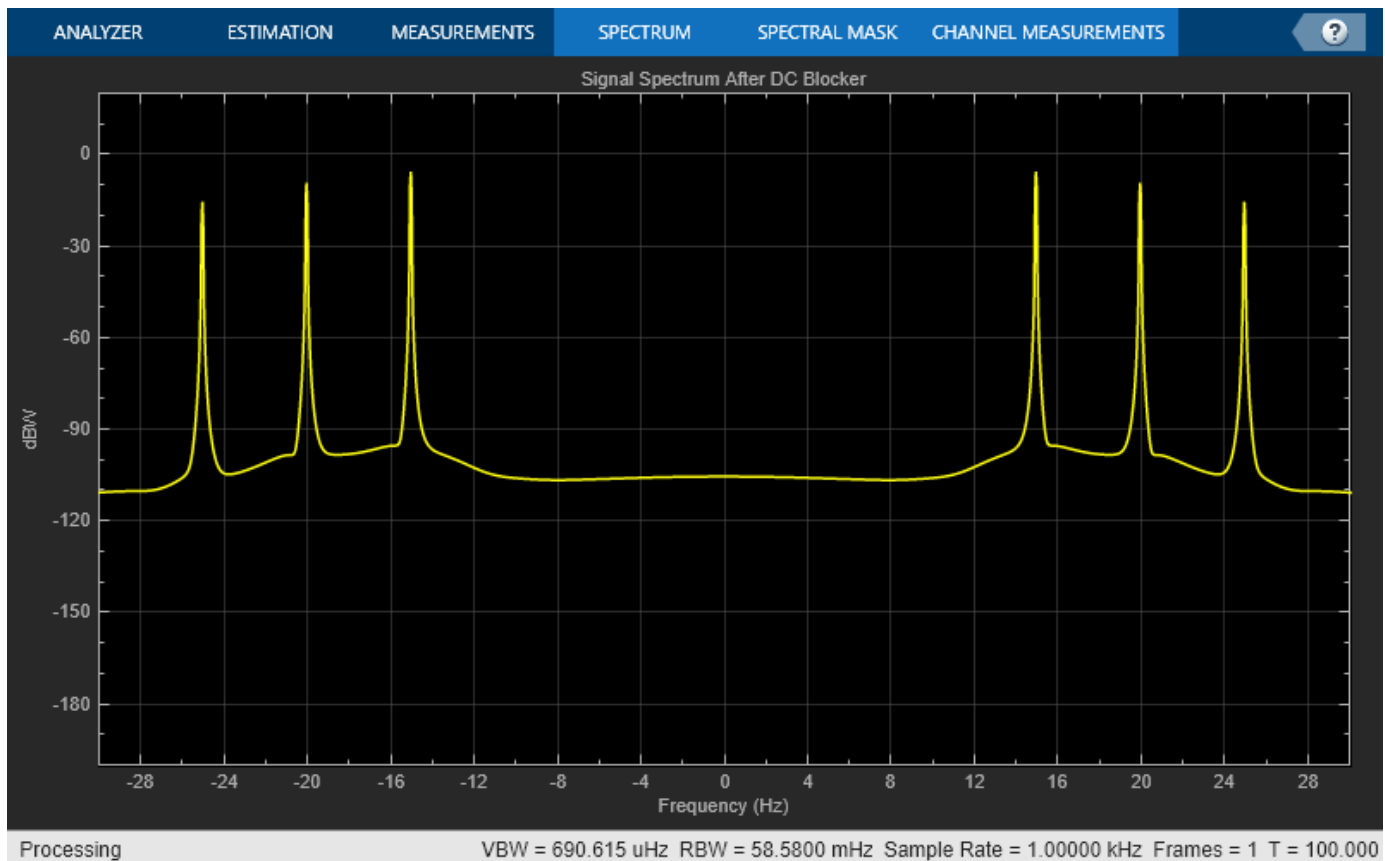
Use the first spectrum analyzer to display the frequency characteristics of the input signal. The tones at 15, 20, and 25 Hz, and the DC component, are clearly visible.

```
hsa(x)
```



Use the second spectrum analyzer to display the frequency characteristics of the output signal. The DC component has been removed.

```
hsb(y)
```



## Algorithms

The DC blocker subtracts the DC component from the input signal. You can estimate the DC component by using the IIR, FIR, CIC, or subtract mean algorithm.

### IIR

Pass the input signal through an IIR lowpass elliptical filter.

The elliptical IIR filter has a passband ripple of 0.1 dB and a stopband attenuation of 60 dB. You specify the normalized bandwidth and the filter order.

### FIR

Pass the input signal through an FIR filter that uses a nonrecursive moving average from a finite number of past input samples.

The FIR filter coefficients are given as  $\text{ones}(1, \text{Length}) / \text{Length}$ , where  $\text{Length}$  is the number of past input samples for FIR algorithm. The FIR filter structure is a direct form I transposed structure.

### CIC

Pass the input signal through a CIC filter. Because the CIC filter amplifies the signal, the filter gain is estimated and subtracted from the DC estimate.

The Cascaded Integrator-Comb (CIC) filter consists of two integrator-comb pairs. These pairs help to ensure that the peak of the first sidelobe of the filter response is attenuated by at least 25 dB relative to the peak of the main lobe. The normalized 3 dB bandwidth is used to calculate the differential delay. The delay is used to determine the gain of the CIC filter. The inverse of the filter gain is used as a multiplier, which is applied to the output of the CIC filter. This ensures that the aggregate gain of the DC estimate is 0 dB.

The following equation characterizes the aggregate magnitude response of the filter and the multiplier:

$$|H(e^{j\omega})| = \left| \frac{\sin(M\frac{\pi}{2}B_{norm})}{M\sin(\frac{\pi}{2}B_{norm})} \right|^N$$

- $B_{norm}$  is the normalized bandwidth such that  $0 < B_{norm} < 1$ .
- $M$  is the differential delay in samples.
- $N$  is the number of sections, equal to 2.

Set the differential delay,  $M$ , to the smallest integer such that  $|H(e^{j\omega})| < 1/\sqrt{2}$ . Once  $M$  is known, the gain of the CIC filter is calculated as  $M^N$ . Therefore, to precisely compensate for the filter gain, the multiplier is set to  $(1/M)^N$ .

### Subtract mean

Compute the mean value of each column of the input signal and subtract the mean from the input. For example, if the input is [1 2 3 4; 3 4 5 6], then a DC Blocker set to this mode outputs [-1 -1 -1 -1; 1 1 1 1].

## Version History

Introduced in R2014a

## References

- [1] Nezami, M. "Performance Assessment of Baseband Algorithms for Direct Conversion Tactical Software Defined Receivers: I/Q Imbalance Correction, Image Rejection, DC Removal, and Channelization." MILCOM, 2002.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

This object also supports SIMD code generation using Intel AVX2 technology when the input signal has a data type of `single` or `double`.

The SIMD technology significantly improves the performance of the generated code.

**HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

To generate HDL code from predefined System objects, see “HDL Code Generation from Viterbi Decoder System Object” (HDL Coder).

**See Also****Functions**

`fvtool`

**Objects**

`dsp.SOSFilter` | `dsp.FIRFilter` | `dsp.CICDecimator`

**Blocks**

DC Blocker

## dsp.Delay

**Package:** dsp

Delay input signal by fixed samples

### Description

The `dsp.Delay` System object delays the input by a specified number of samples along each channel (column) of the input. You can specify the initial output of the object through the “InitialConditions” on page 4-0 property. To reset the delay, enable the “ResetCondition” on page 4-0 through the “ResetInputPort” on page 4-0 .

To delay the input:

- 1 Create the `dsp.Delay` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

### Creation

#### Syntax

```
delay = dsp.Delay
delay = dsp.Delay(Name,Value)
delay = dsp.Delay(len,Name,Value)
```

#### Description

`delay = dsp.Delay` creates a System object that delays the input by 1 sample.

`delay = dsp.Delay(Name,Value)` creates a delay System object with each specified property set to the specified value. Enclose each property name in single quotes.

Example: `delay = dsp.Delay('InitialConditionsPerChannel',true);`

`delay = dsp.Delay(len,Name,Value)` creates a delay System object, `delay`, with the `Length` property set to `len`, and other specified properties set to the specified values. Enclose each property name in single quotes.

Example: `delay = dsp.Delay(10,'ResetInputPort',true,'ResetCondition','Rising edge');`

#### Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.



For more information on changing property values, see System Design in MATLAB Using System Objects.

### **Length — Amount of delay to apply to input signal**

1 (default) | scalar positive integer | vector of positive integers

Amount of delay in samples to apply to the input signal, specified as one of the following:

- Scalar positive integer -- The object applies equal delay to all the channels.
- Vector of positive integers -- The length of the vector must equal the number of input channels (columns). The object delays each channel by the amount specified by the respective element in the delay vector.

### **InitialConditionsPerChannel — Enable different initial conditions per channel**

false (default) | true

Enable different initial conditions per channel, specified as either:

- false -- The object applies the same initial conditions for all channels.
- true -- The object applies different initial conditions for each channel.

The value of this property must be the same as the value you choose for the `InitialConditionsPerSample` property. This value determines the size of the initial conditions array. For more details, see the “InitialConditions” on page 4-0 property.

### **InitialConditionsPerSample — Enables different initial conditions per sample**

false (default) | true

Different initial conditions per sample, specified as either:

- false -- The object applies the same initial conditions for all samples.
- true -- The object applies different initial conditions for each sample.

The value of this property must be the same as the value you choose for the `InitialConditionsPerChannel` property. This value determines the size of the initial conditions array. For more details, see the “InitialConditions” on page 4-0 property.

### **InitialConditions — Initial output of the object**

0 (default) | scalar | vector | matrix

Initial output of System object, specified as a scalar, vector, or a matrix. The dimensions of the initial condition matrix must be (Length property value)-by-(number of input channels).

If the input is an  $M$ -by- $N$  matrix, the dimensions of the `InitialConditions` property value must be as follows:

| <code>InitialConditionsPerChannel</code> | <code>InitialConditionsPerSample</code> | <code>InitialConditions</code> |
|--|---|--------------------------------|
| false                                    | false                                   | scalar                         |
| true                                     | true                                    | Length-by- $N$ matrix          |

### **ResetInputPort — Enable reset condition**

false (default) | true

Enable the reset condition so that you can pass the reset control input to the object, specified as either:

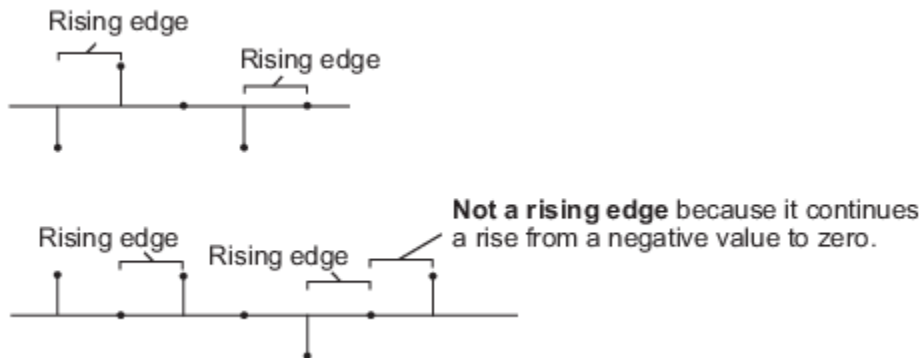
- `false` -- The object does not reset the delay states.
- `true` -- You must pass a reset control input to the object by using the “ResetCondition” on page 4-0 property. The object resets the delay states based on the values of the `ResetCondition` property and the reset control that is input to the object.

#### ResetCondition — Event that triggers reset of delay

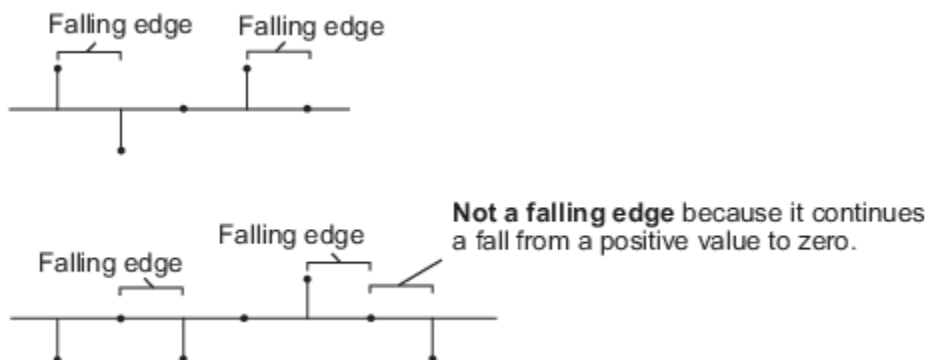
'Non-zero' (default) | 'Rising edge' | 'Falling edge' | 'Either edge'

Event that triggers the reset of the delay, specified as one of the following. The object resets the delay whenever a reset event is detected in its reset input.

- 'Non-zero' -- Triggers a reset operation at each sample, when the reset input is not zero.
- 'Rising edge' -- Triggers a reset operation when the reset input does one of the following:
  - Rises from a negative value to either a positive value or zero.
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero.



- 'Falling edge' -- Triggers a reset operation when the reset input does one of the following:
  - Falls from a positive value to a negative value or zero.
  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero.



- 'Either edge' -- Triggers a reset operation when the reset input is a rising edge or a falling edge.

## Dependencies

This property applies only when you set the `ResetInputPort` property to `true`.

## Usage

## Syntax

```
delayOut = delay(dataInput)
delayOut = delay(dataInput, resetInput)
```

## Description

`delayOut = delay(dataInput)` adds delay to the data input and returns the delayed output. Each column of the input is treated as an independent channel.

`delayOut = delay(dataInput, resetInput)` adds delay to the data input and selectively resets the state of the System object based on the value of the reset input and the value of the “ResetCondition” on page 4-0 property.

To pass the reset input, set “ResetInputPort” on page 4-0 property to `true`.

```
delay = dsp.Delay('ResetInputPort', true);
...
delayOut = delay(dataInput, resetInput);
```

## Input Arguments

### **dataInput** — Data input

vector | matrix

Data input that is delayed by the object, specified as a vector or a matrix.

Example: `[1;2;3;4;5]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `fi`

### **resetInput** — Reset input

scalar

Reset input, specified as a scalar.

Example: `2`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `logical`

## Output Arguments

### **delayOut** — Delayed output

vector | matrix

Delayed output, returned as a vector or matrix. The size and data type of the output match the size and data type of the data input.

Example: `[0;0;1;2;3]`

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical | fi

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Common to All System Objects

|         |  |
|---------|--|
| step    | Run System object algorithm  |
| release | Release resources and allow changes to System object property values and input characteristics |
| reset   | Reset internal states of System object   |

## Examples

### Delay Input

Delay input by four samples by using the `dsp.Delay` System object™. By default, the initial conditions are 0.

```
delay = dsp.Delay(4);  
input = [(1:10)' (11:20)'];  
delayOut = delay(input) %#ok
```

```
delayOut = 10×2
```

```
0     0  
0     0  
0     0  
0     0  
1     11  
2     12  
3     13  
4     14  
5     15  
6     16
```

```
release(delay);
```

Specify initial conditions for each channel and for each sample. The `InitialConditions` property must be a (*Length*)-by-(*NumChannels*) matrix.

```
delay.InitialConditionsPerChannel = true;  
delay.InitialConditionsPerSample = true;  
delay.InitialConditions = [(0.1:0.1:0.4)' (0.5:0.1:0.8)'];  
delayOut = delay(input) %#ok
```

```
delayOut = 10×2
```

```
0.1000    0.5000
```

```

0.2000    0.6000
0.3000    0.7000
0.4000    0.8000
1.0000   11.0000
2.0000   12.0000
3.0000   13.0000
4.0000   14.0000
5.0000   15.0000
6.0000   16.0000

```

```
release(delay);
```

Reset the delay by setting the reset event to 'Rising edge'. In this mode, a reset event occurs when the reset input:

- Rises from a negative value to 0.
- Rises from a negative value to a positive value.
- Rises from 0 to a positive value, where the rise is not a continuation from a negative value to 0.

Pass an initial reset input of 0.

```

delay.ResetInputPort = true;
delay.ResetCondition = 'Rising edge';
delayOut = delay(input,0) %#ok

```

```
delayOut = 10x2
```

```

0.1000    0.5000
0.2000    0.6000
0.3000    0.7000
0.4000    0.8000
1.0000   11.0000
2.0000   12.0000
3.0000   13.0000
4.0000   14.0000
5.0000   15.0000
6.0000   16.0000

```

Continue to run the delay. The delay samples now contain the rest of the input vector.

```
delayOut = delay(input,0) %#ok
```

```
delayOut = 10x2
```

```

7    17
8    18
9    19
10   20
1    11
2    12
3    13
4    14
5    15
6    16

```

Change the reset input to 2, indicating a rising edge.

```
delayOut = delay(input,2)
```

```
delayOut = 10×2
```

```
0.1000    0.5000
0.2000    0.6000
0.3000    0.7000
0.4000    0.8000
1.0000   11.0000
2.0000   12.0000
3.0000   13.0000
4.0000   14.0000
5.0000   15.0000
6.0000   16.0000
```

The delay values are reset to the initial conditions.

## Version History

Introduced in R2012a

### Extended Capabilities

#### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

#### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

To generate HDL code from predefined System objects, see “HDL Code Generation from Viterbi Decoder System Object” (HDL Coder).

### See Also

#### Functions

`designFracDelayFIR`

#### Objects

`dsp.VariableIntegerDelay` | `dsp.VariableFractionalDelay`.

#### Blocks

Delay

**Topics**

“Three-Channel Wavelet Transmultiplexer”

“Envelope Detection in MATLAB”

## **dsp.DelayLine**

**Package:** dsp

(Removed) Rebuffer sequence of inputs with one-sample shift

---

**Note** `dsp.DelayLine` has been removed. There is no direct replacement. You can use the `dsp.AsyncBuffer` object to achieve a delay line.

---

### **Description**

The `dsp.DelayLine` System object rebuffers a sequence of inputs with one-sample shift.

To rebuffer a sequence of inputs with one-sample shift:

- 1 Create the `dsp.DelayLine` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

### **Creation**

#### **Syntax**

```
dline = dsp.DelayLine
dline = dsp.DelayLine(delaysize,initial)
dline = dsp.DelayLine(Name,Value)
```

#### **Description**

`dline = dsp.DelayLine` returns a delay line System object, `dline`, that buffers the input samples into a sequence of overlapping or underlapping matrix outputs.

`dline = dsp.DelayLine(delaysize,initial)` returns a delay line System object, `dline`, with the `Length` property set to `delaysize` and the `InitialConditions` property set to `initial`.

`dline = dsp.DelayLine(Name,Value)` returns a delay line object with each specified property set to the specified value. Enclose each property name in single quotes. Unspecified properties have default values.

#### **Properties**

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).



**Length — Number of rows in output matrix**

64 (default) | positive integer

Specify the number of rows in the output matrix as a scalar positive integer.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**InitialConditions — Initial delay line output**

0 (default) | scalar | vector | matrix

Set the value of the object's initial output as one of `scalar`, `vector`, or `matrix`.

For vector outputs, you can use these options for the `InitialConditions` property:

- A vector of the same size
- A scalar value that you want repeated across all elements of the initial output

For matrix outputs, you can use these options for the `InitialConditions` property:

- A matrix of the same size
- A vector (equal to the length of the number of matrix rows) that repeats across all columns of the initial output
- A scalar that repeats across all elements of the initial output

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**DirectFeedthrough — Enable passing input data to output without extra frame delay**`false` (default) | `true`

When you set this property to `true`, the input data is available immediately at the output. When you set this property to `false`, the output has a delay of one frame.

**EnableOutputInputPort — Enable selective output linearization**`false` (default) | `true`

The object uses a circular buffer, even though the output is linear. To obtain a valid output, the object must linearize the circular buffer. When this property is `true`, the object uses an additional Boolean input to determine if a valid output calculation is needed. If the input value is `true`, the object's output is linearized and thus valid. If the input value is `false`, the output is not linearized and is invalid. This allows the object to be more efficient when each step does not require the tapped delay line output. When you set this property to `false`, the output is always linearized and valid.

**HoldPreviousValue — Hold previous valid value for invalid output**`false` (default) | `true`

If you set this property to `true`, the most recent, valid value is held on the output. If you set this property to `false`, the signal on the output is invalid data.

**Dependencies**

This property applies only when you set the `EnableOutputInputPort` property to `true`.

## Usage

### Syntax

```
y = dline(x)
y = dline(x,en)
```

### Description

`y = dline(x)` returns the delayed version of input `x`. `y` is an output matrix with the same number of rows as the delay line size. Each column of `x` is treated as a separate channel.

`y = dline(x,en)` selectively outputs the delayed version of input `x` depending on the Boolean input `en`. This occurs only when you set the `EnableOutputInputPort` property to `true`. If `en` is `false`, use the `HoldPreviousValue` property to specify if the object should hold the previous output value(s).

### Input Arguments

#### **x** — Data input

vector | matrix

Data input, specified as a vector or a matrix.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `fi`

#### **en** — Enable output input port

`true` | `false`

Enable output input port signal, specified as a logical value.

If `en` is `false`, use the `HoldPreviousValue` property to specify if the object should hold the previous output value(s).

### Dependencies

This input is valid only when you set the `EnableOutputInputPort` property to `true`.

Data Types: `logical`

### Output Arguments

#### **y** — Delay line output

vector | matrix

Delay line output, returned as a vector or a matrix.

When the input is an  $M_i$ -by- $N$  matrix, the System object rebuffers the input into a sequence of  $M_o$ -by- $N$  matrix outputs, where  $M_o$  is the output frame size specified by the `Length` property. Depending on whether  $M_o$  is greater than, less than, or equal to the input frame size,  $M_i$ , the output frames can be underlapped or overlapped. Each of the  $N$  input channels is rebuffered independently:

- When  $M_o > M_i$ , the output frame overlap is the difference between the output and input frame size,  $M_o - M_i$ .

- When  $M_o < M_i$ , the output is underlapped; the object discards the first  $M_i - M_o$  samples of each input frame so that only the last  $M_o$  samples are buffered into the corresponding output frame.
- When  $M_o = M_i$ , the output data is identical to the input data, but is delayed by the latency of the object.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical | fi

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Common to All System Objects

`step` Run System object algorithm  
`release` Release resources and allow changes to System object property values and input characteristics  
`reset` Reset internal states of System object

## Examples

### Delay Line

Use a delay line object with a delay line size of 4 samples.

```
delayline = dsp.DelayLine( ...
    'Length', 4, ...
    'DirectFeedthrough', true, ...
    'InitialConditions', -2, ...
    'EnableOutputInputPort', true, ...
    'HoldPreviousValue', true);
en = logical([1 1 0 1 0]);
y = zeros(4,5);
for ii = 1:5
    y(:,ii) = delayline(ii, en(ii));
end
disp(y)
```

```
-2    -2    -2     1     1
-2    -2    -2     2     2
-2     1     1     3     3
 1     2     2     4     4
```

## Algorithms

This object implements the algorithm, inputs, and outputs described on the Delay Line block reference page. The object properties correspond to the block properties.

## Version History

**Introduced in R2012a**

### **R2021b: dsp.DelayLine System object will be removed**

*Warns starting in R2021b*

`dsp.DelayLine` System object will be removed in a future release. There is no direct replacement for this object. You can use the `dsp.AsyncBuffer` object to achieve a delay line.

### **R2023a: dsp.DelayLine System object has been removed**

*Errors starting in R2023a*

The `dsp.DelayLine` System object has been removed. To create a delay line, use the `dsp.AsyncBuffer` object.

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

## See Also

### **Objects**

`dsp.AsyncBuffer`

# dsp.Differentiator

**Package:** dsp

Direct form FIR fullband differentiator filter

## Description

The `dsp.Differentiator` System object applies a fullband differentiator filter on the input signal to differentiate all its frequency components. This object uses an FIR equiripple filter design to design the differentiator filter. The ideal frequency response of the differentiator is  $D(\omega) = j\omega$  for  $-\pi \leq \omega \leq \pi$ . You can design the filter with minimum order with a specified order. This object supports fixed-point operations.

To filter each channel of your input:

- 1 Create the `dsp.Differentiator` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
DF = dsp.Differentiator
DF = dsp.Differentiator(Name,Value)
```

### Description

`DF = dsp.Differentiator` returns a differentiator, `DF`, which independently filters each channel of the input over time using the given design specifications.

`DF = dsp.Differentiator(Name,Value)` sets each property name to the specified value. Unspecified properties have default values.

### Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

#### **DesignForMinimumOrder** — Design minimum order filter

true (default) | false

Option to design a minimum-order filter, specified as a logical scalar. The filter has 2 degrees of freedom. When you set this property to

- `true` — The object designs the filter with the minimum order that meets the `PassbandRipple` value.
- `false` — The object designs the filter with order that you specify in the `FilterOrder` property.

This property is not tunable.

### **FilterOrder — Order of the filter**

31 (default) | odd positive integer

Order of the filter, specified as an odd positive integer.

This property is not tunable.

#### **Dependencies**

You can specify the filter order only when `'DesignForMinimumOrder'` is set to `false`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **PassbandRipple — Maximum passband ripple**

0.1 (default) | positive real scalar

Maximum passband ripple in dB, specified as a positive real scalar.

This property is not tunable.

#### **Dependencies**

You can specify the passband ripple only when `'DesignForMinimumOrder'` is set to `true`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **ScaleCoefficients — Scale filter coefficients**

`false` (default) | `true`

Option to scale the filter coefficients, specified as a logical scalar. When you set this property to `true`, the object scales the filter coefficients to preserve the input dynamic range.

This property is not tunable.

#### **Fixed-Point Properties**

### **CoefficientsDataType — Word and fraction lengths of coefficients**

`numericType(1,16)` (default) | `numericType` object

Word and fraction lengths of coefficients, specified as a signed or unsigned `numericType` object. The default, `numericType(1,16)`, corresponds to a signed numeric type object with 16-bit coefficients. To give the best possible precision, the fraction length is computed based on the coefficient values.

This property is not tunable.

The word length of the output is the same as the word length of the input. The object computes the fraction length of the output such that the entire dynamic range of the output can be represented

without overflow. For details on how the object computes the fraction length of the output, see “Fixed-Point Precision Rules for Avoiding Overflow in FIR Filters”.

### **RoundingMethod — Rounding method for output fixed-point operations**

'Floor' (default) | 'Ceiling' | 'Convergent' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'

Rounding method for output fixed-point operations, specified as a character vector. For more information on the rounding modes, see “Precision and Range”.

This property is not tunable.

## **Usage**

### **Syntax**

$y = DF(x)$

### **Description**

$y = DF(x)$  applies a fullband differentiator filter to the input signal,  $x$ .  $y$  is a differentiated version of  $x$ .

### **Input Arguments**

#### **x — Data input**

vector | matrix

Data input, specified as a vector or a matrix. If the input signal is a matrix, each column of the matrix is treated as an independent channel. The number of rows in the input signal denotes the channel length. The data type characteristics (double, single, or fixed-point) and the real-complex characteristics (real or complex valued) must be the same for the input data and output data.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fi

### **Output Arguments**

#### **y — Differentiated signal**

vector | matrix

Differentiated signal, returned as a vector or matrix of the same size, data type, and complexity as the input signal,  $x$ .

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fi

## **Object Functions**

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Specific to dsp.Differentiator

getFilter Get underlying FIR filter

## Common to All System Objects

step Run System object algorithm

release Release resources and allow changes to System object property values and input characteristics

reset Reset internal states of System object

## Examples

### Group Delay Estimation

Estimate the group delay of a linear phase FIR filter using a dsp.TransferFunctionEstimator object followed by dsp.PhaseExtractor and dsp.Differentiator objects. The group delay of a linear phase FIR filter is given by  $GD = -\frac{d\theta(\omega)}{d\omega} = -\frac{N}{2}$ , where  $\theta(\omega)$  is the phase information of the filter,  $\omega$  is the frequency vector, and  $N$  is the order of the filter.

### Set Up the Objects

Create a linear phase FIR lowpass filter. Set the order to 200, the passband frequency to 255 Hz, the passband ripple to 0.1 dB, and the stopband attenuation to 80 dB. Specify a sample rate of 512 Hz.

```
Fs = 512;
LPF = dsp.LowpassFilter('SampleRate',Fs,'PassbandFrequency',255,...
    'DesignForMinimumOrder',false,'FilterOrder',200);
```

To estimate the transfer function of the lowpass filter, create a transfer function estimator. Specify the window to be Hann. Set the FFT length to 1024 and the number of spectral averages to 200.

```
TFE = dsp.TransferFunctionEstimator('FrequencyRange','twosided',...
    'SpectralAverages',200,'FFTLengthSource','Property',...
    'FFTLength',1024);
```

To extract the unwrapped phase from the frequency response of the filter, create a phase extractor.

```
PE = dsp.PhaseExtractor;
```

To differentiate the phase  $\theta$ , create a differentiator filter. This value is used in computing the group delay.

```
DF = dsp.Differentiator;
```

To smoothen the input, create a variable bandwidth FIR filter.

```
Gain1 = 512/pi;
Gain2 = -1;
VBFilter = dsp.VariableBandwidthFIRFilter('CutoffFrequency',10,...
    'SampleRate',Fs);
```

To view the group delay of the filter, create an array plot object.

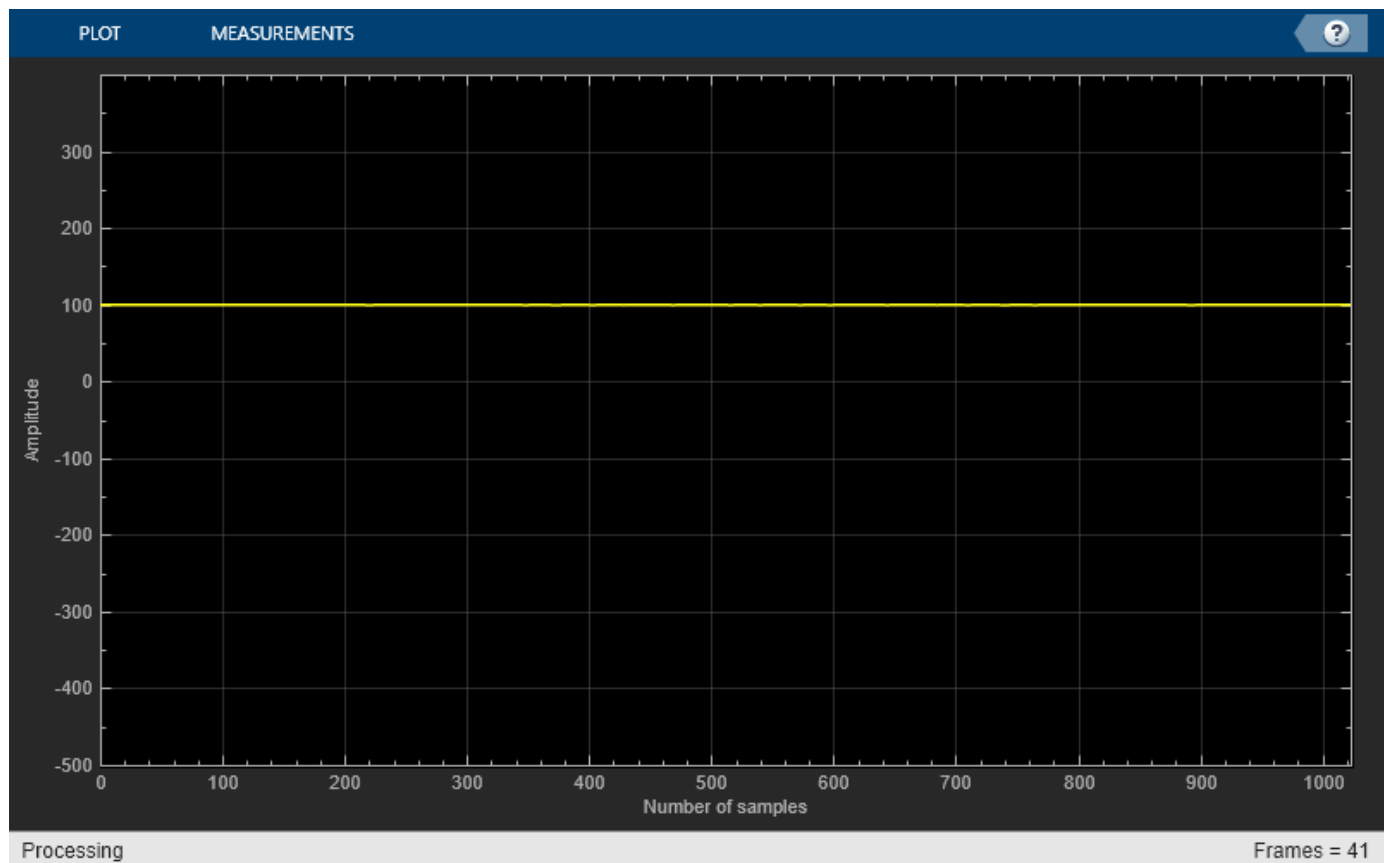
```
AP = dsp.ArrayPlot('PlotType','Line','YLimits',[-500 400],...
    'YLabel','Amplitude','XLabel','Number of samples');
```



## Run the Algorithm

The for-loop is the streaming loop that estimates the group delay of the filter. In the loop, the algorithm filters the input signal, estimates the transfer function of the filter, and differentiates the phase of the filter to compute the group delay.

```
Niter = 1000; % Number of iterations
for k = 1:Niter
    x = randn(512,1); % Input signal = white Gaussian noise
    y = LPF(x); % Filter noise with Lowpass FIR filter
    H = TFE(x,y); % Compute transfer function estimate
    Phase = PE(H); % Extract the Unwrapped phase
    phaseaftergain1 = Gain1*Phase;
    DiffOut = DF(phaseaftergain1); % Differentiate the phase
    phaseaftergain2 = Gain2 * DiffOut;
    VBFOut = VBFiter(phaseaftergain2); % Smooth the group delay
    AP(VBFOut); % Display the group delay
end
```



As you can see, the group delay of the lowpass filter is 100.

## Convert FM Signal to AM Signal

Create an FM wave on a 100 Hz carrier signal sampled at 1.5 kHz.

```
Fc = 1e2; % Carrier
Fs = 1.5e3; % Sample rate
sinewave = dsp.SineWave('Frequency',10,...
                        'SamplesPerFrame',1e3,...
                        'SampleRate',Fs);
```

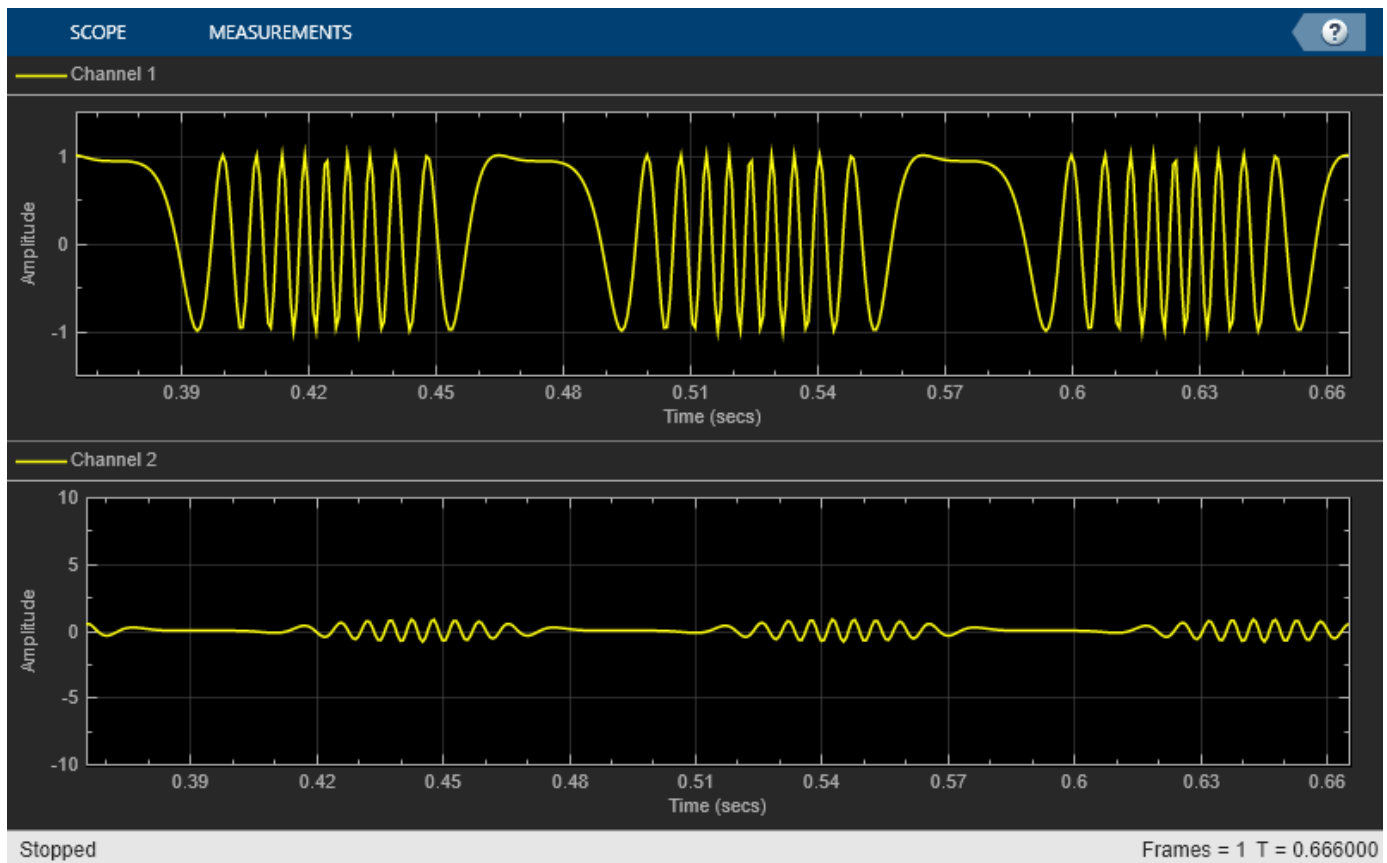
Convert the FM signal to an AM signal.

```
ts = timescope('TimeSpanSource','Property',...
               'TimeSpan',0.3,...
               'BufferLength',10*Fs,...
               'SampleRate',Fs,...
               'ShowGrid',true,...
               'YLimits',[-1.5 1.5],...
               'LayoutDimensions',[2 1]);
```

```
df = dsp.Differentiator;
```

```
tic
while toc<2.2
    x = step(sinewave);
    fm_y = modulate(x,Fc,Fs,'fm');
    am_y = step(df,fm_y);
    step(ts,fm_y,am_y);
end
```

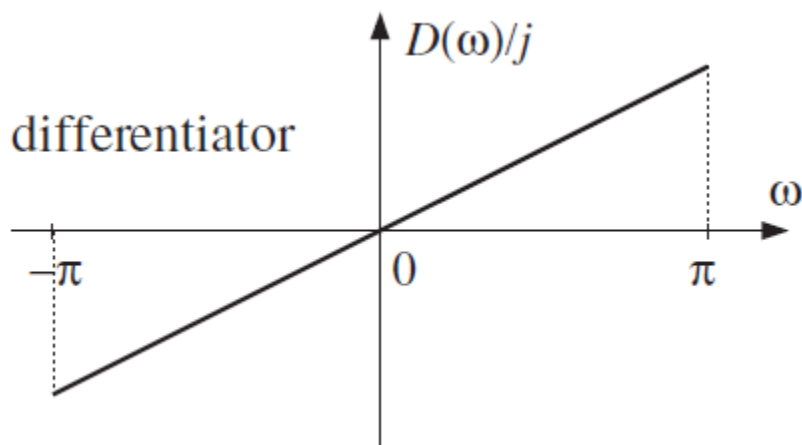
```
release(df);
release(ts);
```



## Algorithms

### Differentiator Filter

Differentiator computes the derivative of a signal. The frequency response of an ideal differentiator filter is given by  $D(\omega) = j\omega$ , defined over the Nyquist interval  $-\pi \leq \omega \leq \pi$ .



The frequency response is antisymmetric and is linearly proportional to the frequency.

`dsp.Differentiator` object acts as a differentiator filter. This object condenses the two-step process into one. For the minimum order design, the object uses generalized Remez FIR filter design algorithm. For the specified order design, the object uses the Parks-McClellan optimal equiripple FIR filter design algorithm. The filter is designed as a linear phase Type-IV FIR filter with a Direct form structure.

The ideal differentiator has an antisymmetric impulse response given by  $d(n) = -d(-n)$ . Hence  $d(0) = 0$ . The differentiator must have zero response at zero frequency.

### Linear-Phase FIR Differentiator Filter

The impulse response of an antisymmetric linear-phase FIR filter is given by  $h(n) = -h(M - 1 - n)$ , where  $M$  is the length of the filter. Because the filter is antisymmetric, you can use this type of FIR filter to design the linear-phase FIR differentiators.

Consider the design of linear-phase FIR differentiators based on the Chebyshev approximation criterion.

If  $M$  is odd, the real-valued frequency response of the FIR filter,  $H_r(\omega)$ , has the characteristics that  $H_r(0) = 0$  and  $H_r(\pi) = 0$ . This filter satisfies the condition of zero response at zero frequency. However, it is not fullband because  $H_r(\pi) = 0$ . This differentiator has a linear response over the limited frequency range  $[0, 2\pi f_p]$ , where  $f_p$  is the bandwidth of the differentiator. The absolute error between the desired response and the Chebyshev approximation increases as  $\omega$  increases from 0 to  $2\pi f_p$ .

If  $M$  is even, the real-valued frequency response of the FIR filter,  $H_r(\omega)$ , has the characteristics that  $H_r(0) = 0$  and  $H_r(\pi) \neq 0$ . This filter satisfies the condition of zero response at zero frequency. It is fullband and this design results in a significantly smaller approximation error than comparable odd-length differentiators. Hence, even-length (odd order) differentiators are preferred in practical systems.

## Version History

Introduced in R2016a

## References

- [1] Orfanidis, Sophocles J. *Introduction to Signal Processing*. Upper Saddle River, NJ: Prentice-Hall, 1996.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

This object also supports SIMD code generation using Intel AVX2 technology when the input signal has a data type of `single` or `double`.

The SIMD technology significantly improves the performance of the generated code.

## See Also

### Functions

getFilter

### Objects

dsp.HighpassFilter | dsp.VariableBandwidthFIRFilter |  
dsp.VariableBandwidthIIRFilter | dsp.FIRFilter | dsp.SOSFilter

### Blocks

Differentiator Filter

# **dsp.DigitalDownConverter**

**Package:** dsp

Translate digital signal from intermediate frequency (IF) band to baseband and decimate it

## **Description**

The `dsp.DigitalDownConverter` object translates digital signal from intermediate frequency (IF) band to baseband and decimates it.

To digitally downconvert the input signal:

- 1 Create the `dsp.DigitalDownConverter` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## **Creation**

### **Syntax**

```
dwnConv = dsp.DigitalDownConverter  
dwnConv = dsp.DigitalDownConverter(Name,Value)
```

### **Description**

`dwnConv = dsp.DigitalDownConverter` returns a digital downconverter (DDC) System object, `dwnConv`.

`dwnConv = dsp.DigitalDownConverter(Name,Value)` returns a DDC object, `dwnConv`, with the specified property `Name` set to the specified `Value`. You can specify additional name-value pair arguments in any order as `(Name1,Value1,...,NameN,ValueN)`.

### **Properties**

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

#### **SampleRate — Sample rate of input signal**

30000000 (default) | positive scalar

Set this property to a positive scalar value, greater than or equal to twice the value of the `CenterFrequency` property.

Data Types: `single` | `double`

### **DecimationFactor — Decimation factor**

100 (default) | positive integer scalar | vector of positive integers

Set this property to a positive integer scalar, or to a 1-by-2 or 1-by-3 vector of positive integers.

When you set this property to a scalar, the object automatically chooses the decimation factors for each of the three filtering stages.

When you set this property to a 1-by-2 vector, the object bypasses the third filter stage and sets the decimation factor of the first and second filtering stages to the values in the first and second vector elements respectively. Both elements of the `DecimationFactor` vector must be greater than one.

When you set this property to a 1-by-3 vector, the  $i$ th element of the vector specifies the decimation factor for the  $i$ th filtering stage. The first and second elements of the `DecimationFactor` vector must be greater than one, and the third element must be 1 or 2.

Data Types: `double`

### **MinimumOrderDesign — Minimum order filter design**

`true` (default) | `false`

When you set this property to `true`, the object designs filters with the minimum order that meets the passband ripple, stopband attenuation, passband frequency, and stopband frequency specifications that you set using the `PassbandRipple`, `StopbandAttenuation`, `Bandwidth`, `StopbandFrequencySource`, and `StopbandFrequency` properties.

When you set this property to `false`, the object designs filters with orders that you specify in the `NumCICSections`, `SecondFilterOrder`, and `ThirdFilterOrder` properties. The filter designs meet the passband and stopband frequency specifications that you set using the `Bandwidth`, `StopbandFrequencySource`, and `StopbandFrequency` properties.

Data Types: `logical`

### **NumCICSections — Number of sections of CIC decimator**

3 (default) | positive integer scalar

Number of sections of CIC decimator, specified as a positive integer scalar.

#### **Dependencies**

This property applies when you set the `MinimumOrderDesign` property to `false`.

Data Types: `double`

### **SecondFilterOrder — Order of CIC compensation filter stage**

12 (default) | positive integer scalar

Order of CIC compensation filter stage, specified as a positive integer scalar.

#### **Dependencies**

This property applies when you set the `MinimumOrderDesign` property to `false`.

Data Types: `double`

**ThirdFilterOrder — Order of third filter stage**

10 (default) | even positive integer

Order of third filter stage, specified as an even positive integer scalar. When you set the `DecimationFactor` property to a 1-by-2 vector, the object ignores the `ThirdFilterOrder` property because the third filter stage is bypassed.

**Dependencies**

This property applies when you set the `MinimumOrderDesign` property to `false`.

Data Types: `double`

**Bandwidth — Two-sided bandwidth of input signal in Hz**

200000 (default) | positive integer scalar

Two-sided bandwidth of input signal in Hz, specified as a positive integer scalar. The object sets the passband frequency of the cascade of filters to one-half of the value that you specify in the `Bandwidth` property. Set the value of this property to less than `SampleRate/DecimationFactor`.

Data Types: `double`

**StopbandFrequencySource — Source of stopband frequency**

Auto (default) | Property

Specify the source of the stopband frequency as one of `Auto` | `Property`. When you set this property to `Auto`, the object places the cutoff frequency of the cascade filter response at approximately  $F_c = \text{SampleRate}/M/2$  Hz, where  $M$  is the total decimation factor that you specify in the `DecimationFactor` property. The object computes the stopband frequency as  $F_{stop} = F_c + TW/2$ .  $TW$  is the transition bandwidth of the cascade response computed as  $2 \times (F_c - F_p)$ , and the passband frequency,  $F_p$ , equals `Bandwidth/2`.

**StopbandFrequency — Stopband frequency in Hz**

150000 (default) | positive scalar

Stopband frequency in Hz, specified as a positive scalar.

**Dependencies**

This property applies when you set the `StopbandFrequencySource` property to `Property`.

Data Types: `double`

**PassbandRipple — Passband ripple of cascade response in dB**

0.1 (default) | positive scalar

Passband ripple of cascade response in dB, specified as a positive scalar. When you set the `MinimumOrderDesign` property to `true`, the object designs the filters so that the cascade response meets the passband ripple that you specify in the `PassbandRipple` property.

**Dependencies**

This property applies when you set the `MinimumOrderDesign` property to `true`.

Data Types: `double`

**StopbandAttenuation — Stopband attenuation of cascade response in dB**

60 (default) | positive scalar



Stopband attenuation of cascade response in dB, specified as a positive scalar. When you set the `MinimumOrderDesign` property to `true`, the object designs the filters so that the cascade response meets the stopband attenuation that you specify in the `StopbandAttenuation` property.

#### Dependencies

This property applies when you set the `MinimumOrderDesign` property to `true`.

Data Types: `double`

#### Oscillator — Type of oscillator

`Sine wave` (default) | `NCO` | `Input port` | `None`

Specify the oscillator as one of `Sine wave` | `NCO` | `Input port` | `None`. When you set this property to `Sine wave`, the object frequency down converts the input signal using a complex exponential obtained from samples of a sinusoidal trigonometric function. When you set this property to `NCO`, the object performs frequency down conversion with a complex exponential obtained using a numerically controlled oscillator (NCO). When you set this property to `Input port`, the object performs frequency down conversion using the complex oscillator signal, `z`, that you pass as an input to the object. When you set this property to `None`, the mixer stage in the object is not present and the object acts as three stage cascaded decimator.

#### CenterFrequency — Center frequency of input signal in Hz

`14000000` (default) | positive scalar

Center frequency of input signal in Hz, specified as a positive scalar that is less than or equal to half the value of the `SampleRate` property. The object down converts the input signal from the passband center frequency you specify in the `CenterFrequency` property, to 0 Hz.

#### Dependencies

This property applies when you set the `Oscillator` property to `Sine wave` or `NCO`.

Data Types: `double`

#### NumAccumulatorBits — Number of NCO accumulator bits

`16` (default) | positive integer

Number of NCO accumulator bits, specified as a positive integer in the range [1 128].

#### Dependencies

This property applies when you set the `Oscillator` property to `NCO`.

Data Types: `double`

#### NumQuantizedAccumulatorBits — Number of NCO quantized accumulator bits

`12` (default) | positive integer

Number of NCO quantized accumulator bits, specified as an integer scalar in the range [1 128]. The value you specify in this property must be less than the value you specify in the `NumAccumulatorBits` property.

#### Dependencies

This property applies when you set the `Oscillator` property to `NCO`.

Data Types: `double`

**Dither — Dither control for NCO**`true (default) | false`

When you set this property to `true`, a number of dither bits specified in the `NumDitherBits` property will be used to apply dither to the NCO signal.

**Dependencies**

This property applies when you set the `Oscillator` property to `NCO`.

**NumDitherBits — Number of NCO dither bits**`4 (default) | positive integer`

Specify this property as an integer scalar smaller than the number of accumulator bits that you specify in the `NumAccumulatorBits` property.

**Dependencies**

This property applies when you set the `Oscillator` property to `NCO` and the `Dither` property to `true`.

Data Types: `double`

**Fixed-Point Properties****FiltersInputDataType — Data type of input of each filter stage**`Same as input (default) | Custom`

Specify the data type at the input of the first, second, and third (if it has not been bypassed) filter stages as one of `Same as input` | `Custom`. The object casts the data at the input of each filter stage according to the value you set in this property.

**CustomFiltersInputDataType — Fixed-point data type of input of each filter stage**`numericType([], 16, 15) (default) | numeric type`

Specify the filters input fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`.

**Dependencies**

This property applies when you set the `FiltersInputDataType` property to `Custom`.

**OutputDataType — Data type of output**`Same as input (default) | Custom`

Specify the data type of output as `Same as input` | `Custom`.

**CustomOutputDataType — Fixed-point data type of output**`numericType([], 16, 15) (default) | numeric type`

Specify the output fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`.

**Dependencies**

This property applies when you set the `OutputDataType` property to `Custom`.

## Usage

### Syntax

```
y = dwnConv(x)
y = dwnConv(x,z)
```

### Description

`y = dwnConv(x)` takes an input `x` and outputs a signal, `y` that is frequency downconverted and downsampled.

`y = dwnConv(x,z)` uses the complex input, `z`, as the oscillator signal used to frequency down convert input `x` when you set the `Oscillator` property to `Input port`.

### Input Arguments

#### **x** — Data input

column vector | matrix

Data input, specified as a column vector or a matrix. The length of input `x` must be a multiple of the decimation factor. When the data type of `x` is `double` or `single` precision, the data type of `y` is the same as that of `x`. When the data type of `x` is of a fixed-point type, the data type of `y` is defined by the `OutputDataType` property.

The input can have multiple channels only if its data type is `double` or `single`. The input can be of data type `double`, `single`, signed integer, or signed fixed-point (`fi` objects).

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `fi`

#### **z** — Oscillator signal

column vector | matrix

Oscillator signal used to frequency down convert the input signal, specified as a column vector or a matrix. This input must be complex. The length of `z` must be equal to the length of `x`. `z` can be `double`, `single`, signed integer, or signed fixed-point (`fi` objects).

### Dependencies

This input applies when you set the `Oscillator` property to `Input port`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `fi`

### Output Arguments

#### **y** — Down converted and down sampled signal

column vector | matrix

Down converted and down sampled signal, returned as a column vector or a matrix. The length of `y` is equal to the length of `x` divided by the `DecimationFactor`. When the data type of `x` is `double` or `single` precision, the data type of `y` is the same as that of `x`. When the data type of `x` is of a fixed point type, the data type of `y` is defined by the `OutputDataType` property.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `fi`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

### Specific to `dsp.DigitalDownConverter`

|                                    |   |
|------------------------------------|---|
| <code>getDecimationFactors</code>  | Get decimation factors of each filter stage of a digital down converter                       |
| <code>getFilterOrders</code>       | Get orders of digital down converter or digital up converter filter cascade                   |
| <code>getFilters</code>            | Get handles to digital down converter or digital up converter filter cascade objects          |
| <code>fvtool</code>                | Visualize frequency response of digital down converter or digital up converter filter cascade |
| <code>groupDelay</code>            | Group delay of digital down converter or digital up converter filter cascade                  |
| <code>visualizeFilterStages</code> | Display response of digital down converter or digital up converter filter cascade             |
| <code>generatehdl</code>           | Generate HDL code for quantized DSP filter (requires Filter Design HDL Coder)                 |

### Common to All System Objects

|                      |  |
|----------------------|--|
| <code>step</code>    | Run System object algorithm  |
| <code>release</code> | Release resources and allow changes to System object property values and input characteristics |
| <code>reset</code>   | Reset internal states of System object   |

## Examples

### Upconvert and Downconvert a Sine Wave Signal

Create a digital up converter object that up samples a 1 KHz sinusoidal signal by a factor of 20 and up converts it to 50 KHz. Create a digital down converter object that down converts the signal to 0 Hz and down samples it by a factor of 20.

Create a sine wave generator to obtain the 1 KHz sinusoidal signal with a sample rate of 6 KHz.

```
Fs = 6e3; % Sample rate
sine = dsp.SineWave('Frequency',1000,'SampleRate',...
Fs,'SamplesPerFrame',1024);
x = sine(); % generate signal
```

Create a `DigitalUpConverter` object. Use minimum order filter designs and set passband ripple to 0.2 dB and the stopband attenuation to 55 dB. Set the double sided signal bandwidth to 2 KHz.

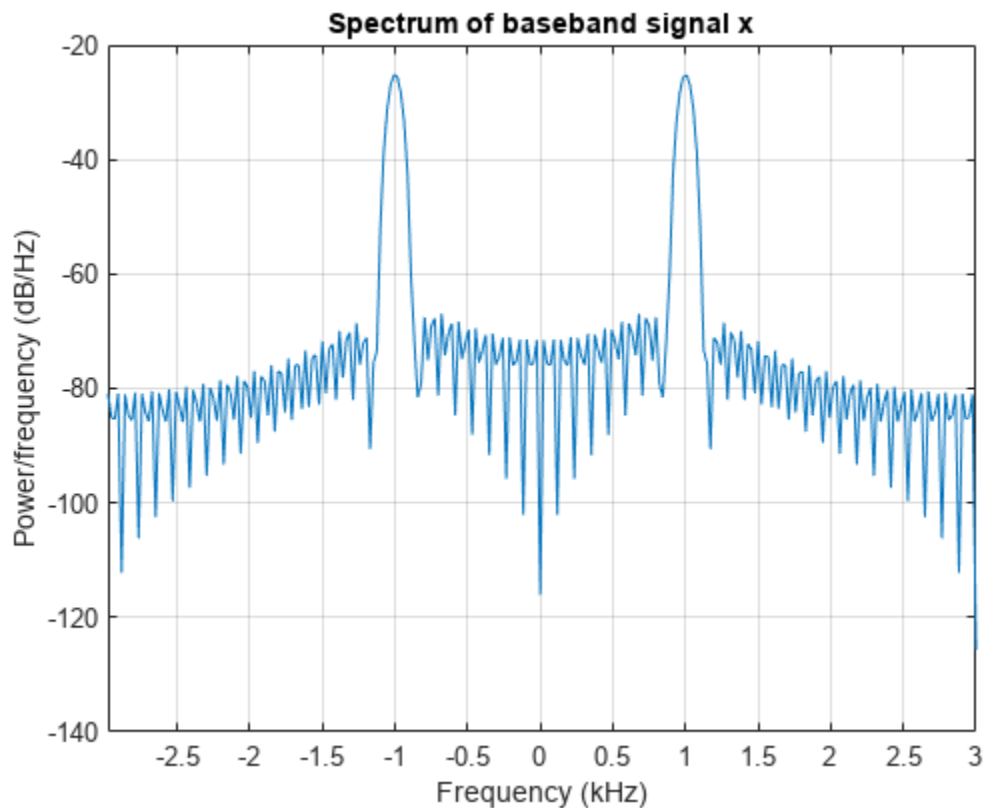
```
upConv = dsp.DigitalUpConverter(...
'InterpolationFactor', 20,...
'SampleRate', Fs,...
'Bandwidth', 2e3,...
'StopbandAttenuation', 55,...
'PassbandRipple', 0.2,...
'CenterFrequency', 50e3);
```

Create a `DigitalDownConverter` object. Use minimum order filter designs and set the passband ripple to 0.2 dB and the stopband attenuation to 55 dB.

```
dwnConv = dsp.DigitalDownConverter(...
    'DecimationFactor',20,...
    'SampleRate', Fs*20,...
    'Bandwidth', 3e3,...
    'StopbandAttenuation', 55,...
    'PassbandRipple',0.2,...
    'CenterFrequency',50e3);
```

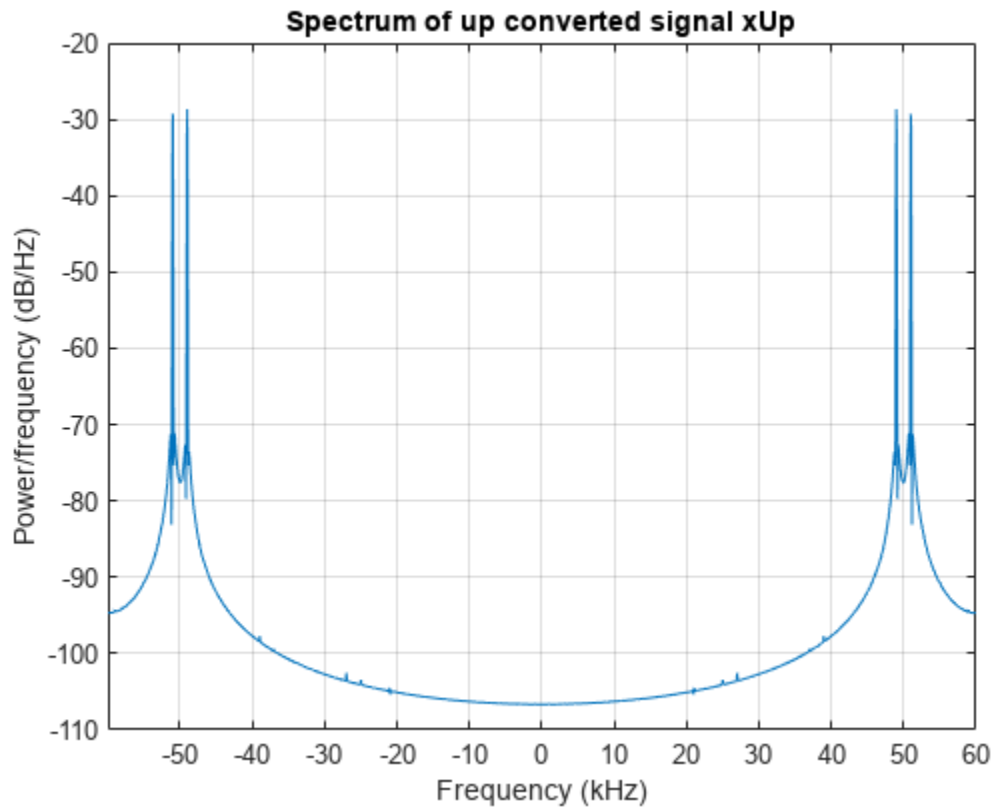
Create a spectrum estimator to visualize the signal spectrum before up converting, after up converting, and after down converting.

```
window = hamming(floor(length(x)/10));
figure; pwelch(x>window, [], [], Fs, 'centered')
title('Spectrum of baseband signal x')
```



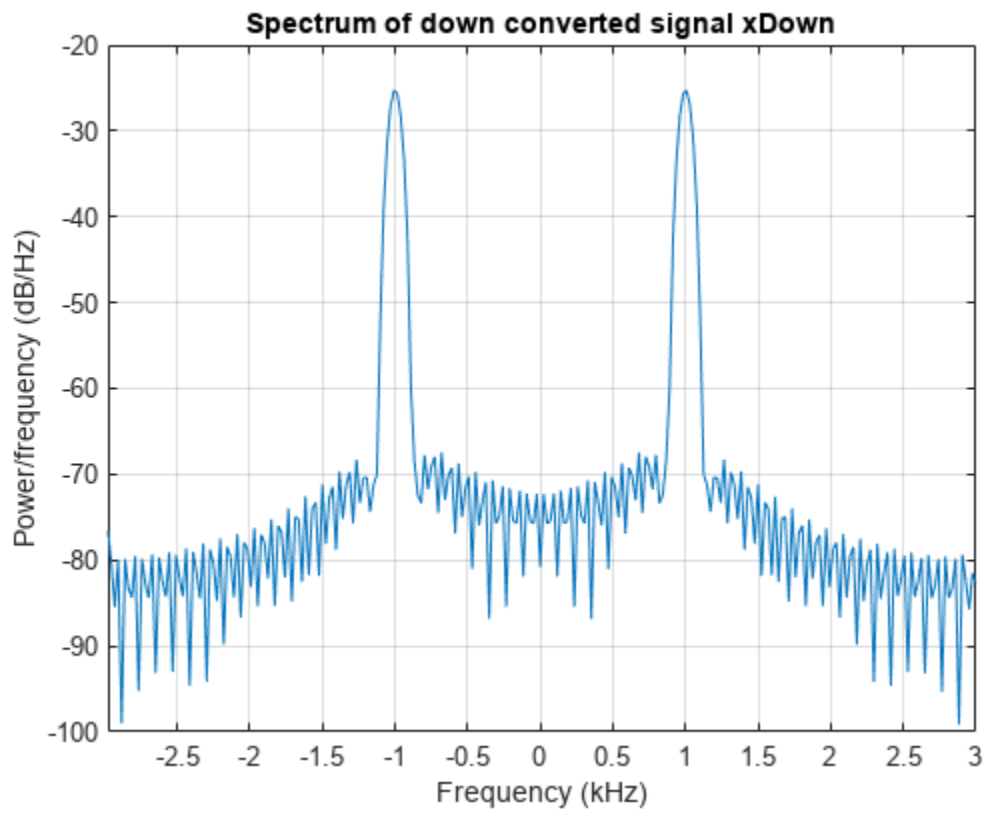
Up convert the signal and visualize the spectrum

```
xUp = upConv(x); % up convert
window = hamming(floor(length(xUp)/10));
figure; pwelch(xUp>window, [], [], 20*Fs, 'centered');
title('Spectrum of up converted signal xUp')
```



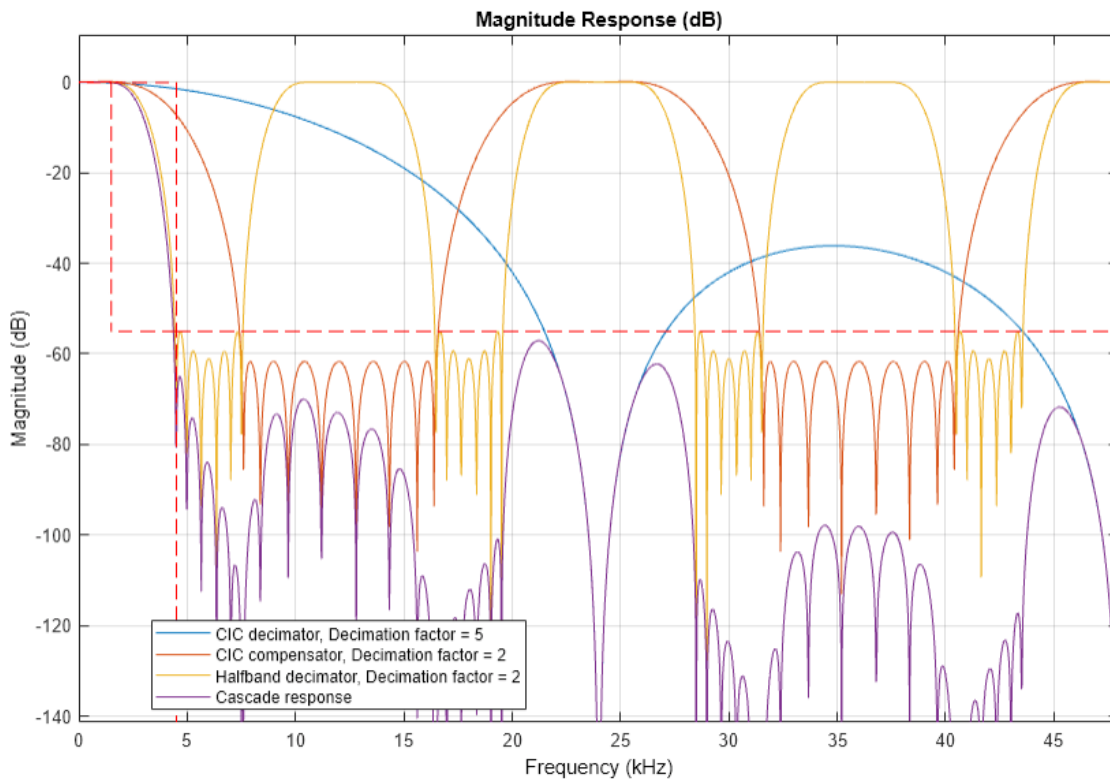
Down convert the signal and visualize the spectrum

```
xDown = dwnConv(xUp); % down convert  
window = hamming(floor(length(xDown)/10));  
figure; pwelch(xDown,window,[],[],Fs,'centered')  
title('Spectrum of down converted signal xDown')
```



Visualize the response of the decimation filters

```
visualizeFilterStages(dwnConv)
```



### Get Decimation Factors

Get decimation factors of each filter stage of the `dsp.DigitalDownConverter` System object™.

Create a `dsp.DigitalDownConverter` System object with the default settings. Using the `getDecimationFactors` function, obtain the decimation factors of each stage of the object.

```
dwnConv = dsp.DigitalDownConverter
```

```
dwnConv =  
dsp.DigitalDownConverter with properties:
```

```
DecimationFactor: 100  
MinimumOrderDesign: true  
Bandwidth: 200000  
StopbandFrequencySource: 'Auto'  
PassbandRipple: 0.1000  
StopbandAttenuation: 60  
Oscillator: 'Sine wave'  
CenterFrequency: 14000000  
SampleRate: 30000000
```

```
Show all properties
```



```
M = getDecimationFactors(dwnConv) %#ok
```

```
M = 1×3
```

```
    25     2     2
```

The `DecimationFactor` property of the object is set to 100. The output `M` is by default a 1-by-3 vector, where each element in the vector is a factor of the overall decimation factor.

When you set the `DecimationFactor` to a 1-by-2 vector, the object bypasses the third filter stage and sets the decimation factor of the first and second filtering stages to the values in the first and second vector elements respectively.

```
dwnConv.DecimationFactor = [10 10]
```

```
dwnConv =
    dsp.DigitalDownConverter with properties:
```

```
    DecimationFactor: [10 10]
    MinimumOrderDesign: true
    Bandwidth: 200000
    StopbandFrequencySource: 'Auto'
    PassbandRipple: 0.1000
    StopbandAttenuation: 60
    Oscillator: 'Sine wave'
    CenterFrequency: 14000000
    SampleRate: 30000000
```

```
Show all properties
```

```
M = getDecimationFactors(dwnConv)
```

```
M = 1×2
```

```
    10    10
```

The output of the `getDecimationFactors` function is now a 1-by-2 vector.

## More About

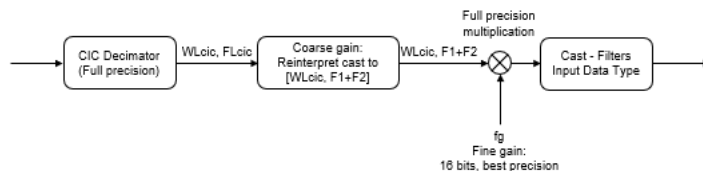
### Fixed Point

The following block diagram represents the DDC arithmetic with signed fixed-point inputs.



- WL is the word length of the input, and FL is the fraction length of the input.
- The input of each filter is cast to the filter input data type. In the `dsp.DigitalDownConverter` object, you can specify the filter input data type through the `FiltersInputDataType` and `CustomFiltersInputDataType` properties. In the Digital Down-Converter block, you can specify the filter input data type through the **Stage input** parameter.
- The oscillator output is cast to a word length equal to the input word length plus one. The fraction length is equal to the input word length minus one.
- The scaling at the output of the CIC decimator consists of coarse- and fine-gain adjustments. The coarse gain is achieved using the `reinterpretcast` function on the CIC decimator output. The fine gain is achieved using full-precision multiplication.

The following figure depicts the coarse-gain and fine-gain operations.

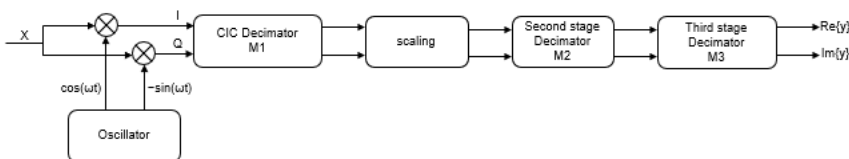


If the normalization gain is  $G$  (where  $0 < G \leq 1$ ), then:

- $WLCic$  is the word length of the CIC decimator output and  $FLCic$  is the fraction length of the CIC decimator output.
- $F1 = \text{abs}(\text{nextpow2}(G))$ , indicating the part of  $G$  achieved using bit shifts (coarse gain).
- $F2 = \text{fraction length specified by the filter input data type}$ .
- $fg = \text{fi}((2^{F1}) * G, \text{true}, 16)$ , which indicates that the remaining gain cannot be achieved with a bit shift (fine gain).

## Algorithms

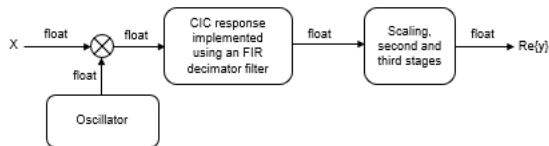
The digital down converter downconverts the input signal by multiplying it with a complex exponential that has the specified center frequency. The algorithm downsamples the frequency downconverted signal using a cascade of three decimation filters. In this case, the filter cascade consists of a CIC decimator, a CIC compensator, and a third FIR decimation stage. The following block diagram shows the architecture of the digital down converter.



The scaling section normalizes the CIC gain and the oscillator power. It can also contain a correction factor to achieve the desired ripple specification. When you specify an oscillator signal through the input port, the normalization factor does not include the oscillator power factor. Depending on how you set the decimation factor, the block bypasses the third filter stage. When the input data type is double or single, the algorithm implements an  $N$ -section CIC decimation filter as an FIR filter with a response that corresponds to a cascade of  $N$  boxcar filters. The algorithm emulates a CIC filter with

an FIR filter so that you can run simulations with floating-point data. When the input data type is fixed-point, the algorithm implements a true CIC filter with actual comb and integrator sections.

This block diagram represents the DDC arithmetic with single or double-precision, floating-point inputs.



For details about fixed-point operation, see “Fixed Point” on page 4-395.

## Version History

Introduced in R2012a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

This object also supports SIMD code generation using Intel AVX2 technology when the input signal has a data type of `single` or `double`.

The SIMD technology significantly improves the performance of the generated code.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

This object supports HDL code generation with the Filter Design HDL Coder product. For workflows and limitations, see “Generate HDL Code for Filter System Objects” (Filter Design HDL Coder).

## See Also

### Functions

`getDecimationFactors` | `getFilterOrders` | `getFilters` | `fvtool` | `groupDelay` | `visualizeFilterStages` | `generatehdl`

### Objects

`dsp.DigitalUpConverter`

### Blocks

Digital Down-Converter | Digital Up-Converter

### Topics

“Digital Up and Down Conversion for Family Radio Service in MATLAB”

“Design and Analysis of a Digital Down Converter”

# dsp.DigitalUpConverter

**Package:** dsp

Interpolate digital signal and translate it from baseband to IF band

## Description

The `dsp.DigitalUpConverter` System object interpolates a digital signal, and translates it from baseband to intermediate frequency (IF) band.

To digitally upconvert the input signal:

- 1 Create the `dsp.DigitalUpConverter` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
upConv = dsp.DigitalUpConverter
upConv = dsp.DigitalUpConverter(Name,Value)
```

### Description

`upConv = dsp.DigitalUpConverter` returns a digital up-converter (DUC) System object, `upConv`.

`upConv = dsp.DigitalUpConverter(Name,Value)` returns a DUC System object with the specified property `Name` set to the specified value `Value`. You can specify one or more name-value pair arguments in any order as `(Name1,Value1,...,NameN,ValueN)`. Enclose each property name in single quotes. For example, create an object that upsamples the input signal by a factor of 20, using a filter with the specified qualities.

```
upConv = dsp.DigitalUpConverter('InterpolationFactor',20,...
'SampleRate',Fs,...
'Bandwidth',2e3,...
'StopbandAttenuation',55,...
'PassbandRipple',0.2,...
'CenterFrequency',50e3);
```

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

**SampleRate — Sample rate of input signal**

30000000 (default) | positive scalar

Set this property to a positive scalar value. The value of this property multiplied by the total interpolation factor must be greater than or equal to twice the value of the `CenterFrequency` property.

Data Types: `single` | `double`

**InterpolationFactor — Interpolation factor**

100 (default) | positive integer | vector of positive integers

Interpolation factor, specified as a positive integer, or a 1-by-2 or 1-by-3 vector of positive integers.

When you set this property to a scalar the object automatically chooses the interpolation factors for each of the three filtering stages.

When you set this property to a 1-by-2 vector, the object bypasses the first filter stage and sets the interpolation factor of the second and third filtering stages to the values in the first and second vector elements, respectively. Both elements of this `InterpolationFactor` vector must be greater than 1.

When you set this property to a 1-by-3 vector, the *i*th element of the vector specifies the interpolation factor for the *i*th filtering stage. The second and third elements of this `InterpolationFactor` vector must be greater than 1 and the first element must equal 1 or 2.

Data Types: `double`

**MinimumOrderDesign — Minimum order filter design**

`true` (default) | `false`

Minimum order filter design, specified as `true` or `false`.

When you set this property to `true`, the object designs filters with the minimum order that meets the passband ripple, stopband attenuation, passband frequency, and stopband frequency specifications that you set using the `PassbandRipple`, `StopbandAttenuation`, `Bandwidth`, `StopbandFrequencySource`, and `StopbandFrequency` properties.

When you set this property to `false`, the object designs filters with orders that you specify in the `FirstFilterOrder`, `SecondFilterOrder`, and `NumCICSections` properties. The filter designs meet the passband and stopband frequency specifications that you set using the `Bandwidth`, `StopbandFrequencySource`, and `StopbandFrequency` properties.

Data Types: `logical`

**SecondFilterOrder — Order of CIC compensation filter stage**

12 (default) | positive integer

Order of CIC compensation filter stage, specified as a positive integer.

**Dependencies**

To enable this property, set the `MinimumOrderDesign` property to `false`.

Data Types: double

### **FirstFilterOrder — Order of first filter stage**

10 (default) | positive even integer

Order of first filter stage, specified as a positive even integer.

#### **Dependencies**

To enable this property, set the `MinimumOrderDesign` property to `false`. When you set the `InterpolationFactor` property to a 1-by-2 vector, the object ignores the `FirstFilterOrder` property, because the first filter stage is bypassed.

Data Types: double

### **NumCICSections — Number of sections of CIC interpolator**

3 (default) | positive integer

Number of sections of CIC interpolator, specified as a positive integer.

#### **Dependencies**

To enable this property, set the `MinimumOrderDesign` property to `false`.

Data Types: double

### **Bandwidth — Two-sided bandwidth of input signal in Hz**

200000 (default) | positive integer

Two-sided bandwidth of input signal in Hz, specified as a positive integer. The object sets the passband frequency of the cascade of filters to half of the value that you specify in this `Bandwidth` property.

Data Types: double

### **StopbandFrequencySource — Source of stopband frequency**

Auto (default) | Property

Source of stopband frequency, specified as `Auto` or `Property`. When you set this property to `Auto`, the object places the cutoff frequency of the cascade filter response at approximately  $F_c = \text{SampleRate}/2$  Hz and computes the stopband frequency as  $F_{\text{stop}} = F_c + TW/2$ .  $TW$  is the transition bandwidth of the cascade response, computed as  $2 \times (F_c - F_p)$ .  $F_p$  is the passband frequency computed by `Bandwidth/2`.

### **StopbandFrequency — Stopband frequency in Hz**

150000 (default) | positive scalar

Stopband frequency in Hz, specified as a positive scalar.

#### **Dependencies**

To enable this property, set the `StopbandFrequencySource` property to `Property`.

Data Types: double

**PassbandRipple — Passband ripple of cascade response in dB**

0.1 (default) | positive scalar

Passband ripple of cascade response in dB, specified as a positive scalar. When you set the `MinimumOrderDesign` property to `true`, the object designs the filters so that the cascade response meets the passband ripple that you specify in this `PassbandRipple` property.

**Dependencies**

To enable this property, set the `MinimumOrderDesign` property to `true`.

Data Types: double

**StopbandAttenuation — Stopband attenuation of cascade response in dB**

60 (default) | positive scalar

Stopband attenuation of cascade response in dB, specified as a positive scalar. When you set the `MinimumOrderDesign` property to `true`, the object designs the filters so that the cascade response meets the stopband attenuation that you specify in this `StopbandAttenuation` property.

**Dependencies**

To enable this property, set the `MinimumOrderDesign` property to `true`.

Data Types: double

**Oscillator — Type of oscillator**

Sine wave (default) | NCO

Type of oscillator, specified as `Sine wave` or `NCO`. When you set this property to `Sine wave`, the object frequency-upconverts the output of the interpolation filter cascade by using a complex exponential signal obtained from samples of a sinusoidal trigonometric function. When you set this property to `NCO`, the object frequency-upconverts the output by using a complex exponential obtained from a numerically controlled oscillator (NCO).

**CenterFrequency — Center frequency of output signal in Hz**

14000000 (default) | positive scalar

Center frequency of output signal in Hz, specified as a positive scalar. The value of this property must be less than or equal to half the product of the `SampleRate` property and the total interpolation factor. The object up converts the input signal so that the output spectrum centers at this frequency you specify in the `CenterFrequency` property.

Data Types: double

**NCO Properties****NumAccumulatorBits — Number of NCO accumulator bits**

16 (default) | integer in the range [1, 128]



Number of NCO accumulator bits, specified as an integer in the range [1, 128]. For more details, see the `dsp.NCO` System object.

**Dependencies**

To enable this property, set the `Oscillator` property to `NCO`.

Data Types: `double`

**NumQuantizedAccumulatorBits — Number of NCO accumulator bits**

12 (default) | integer in the range [1, 128]

Number of NCO accumulator bits, specified as an integer in the range [1, 128]. The value you specify for this property must be less than the value you specify in the `NumAccumulatorBits` property. For more details, see the `dsp.NCO` System object.

**Dependencies**

To enable this property, set the `Oscillator` property to `NCO`.

Data Types: `double`

**Dither — Dither control for NCO**

true (default) | false

Dither control for NCO, specified as `true` or `false`. When you set this property to `true`, the object uses the number of dither bits specified in the `NumDitherBits` property when applying dither to the NCO signal. When this property is `false`, the NCO does not apply dither to the signal. For more details, see the `dsp.NCO` System object.

**Dependencies**

To enable this property, set the `Oscillator` property to `NCO`.

Data Types: `logical`

**NumDitherBits — Number of NCO dither bits**

4 (default) | positive integer

Number of NCO dither bits, specified as a positive integer scalar smaller than the number of accumulator bits that you specify in the `NumAccumulatorBits` property. For more details, see the `dsp.NCO` System object.

**Dependencies**

To enable this property, set the `Oscillator` property to `NCO` and the `Dither` property to `true`.

Data Types: `double`

**Fixed-Point Properties****FiltersOutputDataType — Data type at output of each filter stage**

Same as `input` (default) | `Custom`

Data type at the output of the first (if it has not been bypassed), second, and third filter stages, specified as `Same as input` or `Custom`. The object casts the data at the output of each filter stage according to the value you set in this property. For the CIC stage, the casting is done after the signal is scaled by the normalization factor.

**CustomFiltersOutputDataType — Fixed-point data type at output of each filter stage**

`numericType([],16,15)` (default) | `numericType` object

Fixed-point data type at output of each filter stage, specified as a scaled `numericType` object with the `Signedness` property set to `Auto`.

**Dependencies**

To enable this property, set the `FiltersOutputDataType` property to `Custom`.

**OutputDataType — Data type of output**

`Same as input` (default) | `Custom`

Data type of output, specified as `Same as input` or `Custom`.

**CustomOutputDataType — Fixed-point data type of output**

`numericType([],16,15)` (default) | `numericType` object

Fixed-point data type of output, specified as a scaled `numericType` object the `Signedness` property set to `Auto`.

**Dependencies**

To enable this property, set the `OutputDataType` property to `Custom`.

**Usage****Syntax**

`y = upConv(x)`

**Description**

`y = upConv(x)` returns an upsampled and frequency-upconvertedred signal `y`, for a real or complex input column vector `x`.

**Input Arguments****x — Input signal**

column vector

Input signal, specified as a column vector of real or complex values.

When the data type of `x` is `double` or `single`, the data type of `y` is the same as that of `x`. When the data type of `x` is of a fixed-point type, the data type of `y` is defined by the `OutputDataType` property.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `fi`

## Output Arguments

### y — Upconverted and upsampled signal

column vector

Upconverted and upsampled signal, returned as a column vector. The length of y is equal to the length of x multiplied by the value in the `InterpolationFactor` property. When the data type of x is `double` or `single`, the data type of y is the same as that of x. When the data type of x is of a fixed-point type, the data type of y is defined by the `OutputDataType` property.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `fi`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Specific to dsp.DigitalUpConverter

|                                      |   |
|--------------------------------------|---|
| <code>getInterpolationFactors</code> | Get interpolation factors of each filter stage of digital upconverter                         |
| <code>getFilterOrders</code>         | Get orders of digital down converter or digital up converter filter cascade                   |
| <code>getFilters</code>              | Get handles to digital down converter or digital up converter filter cascade objects          |
| <code>fvtool</code>                  | Visualize frequency response of digital down converter or digital up converter filter cascade |
| <code>groupDelay</code>              | Group delay of digital down converter or digital up converter filter cascade                  |
| <code>visualizeFilterStages</code>   | Display response of digital down converter or digital up converter filter cascade             |
| <code>generatehdl</code>             | Generate HDL code for quantized DSP filter (requires Filter Design HDL Coder)                 |

## Common to All System Objects

|                      |  |
|----------------------|--|
| <code>step</code>    | Run System object algorithm  |
| <code>release</code> | Release resources and allow changes to System object property values and input characteristics |
| <code>reset</code>   | Reset internal states of System object   |

## Examples

### Upconvert Sine Wave Signal

Create a DUC System object™ that upsamples a 1-kHz sinusoidal signal by a factor of 20 and upconverts it to 50 kHz.

Create a sine wave generator to obtain the 1-kHz sinusoidal signal with a sample rate of 6 kHz.

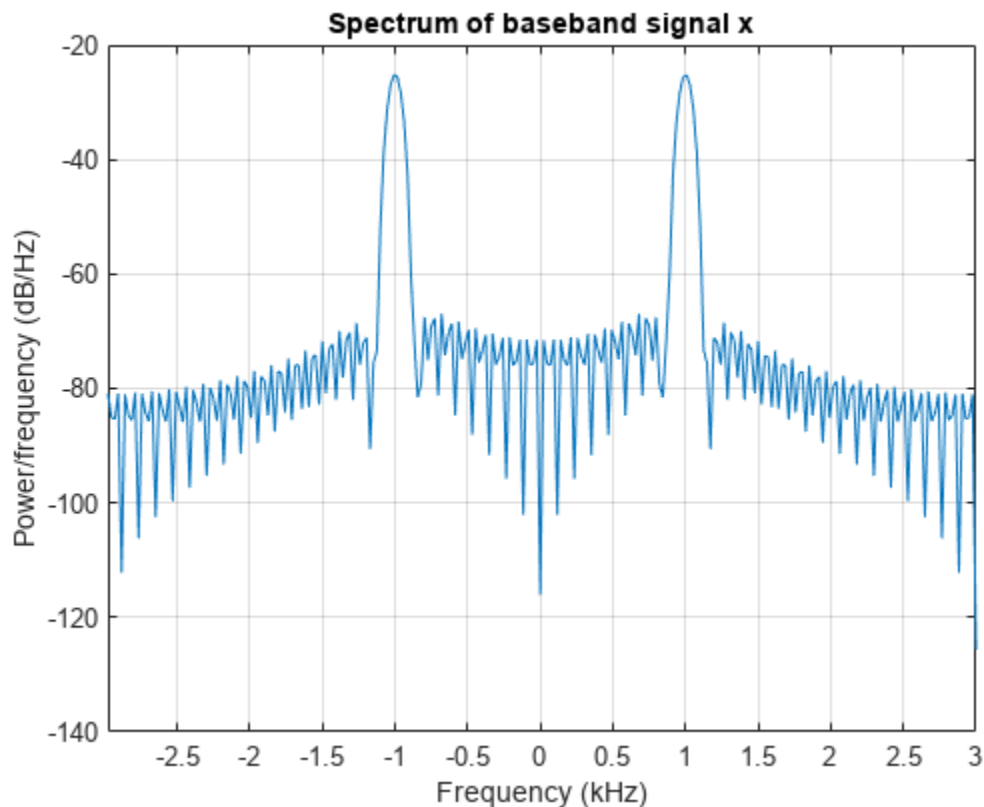
```
Fs = 6e3; % Sample rate
sine = dsp.SineWave('Frequency',1000,...
    'SampleRate',Fs,...
    'SamplesPerFrame',1024);
x = sine(); % generate signal
```

Create a DUC System object. Use minimum order filter designs and set the passband ripple to 0.2 dB and stopband attenuation to 55 dB. Set the double-sided signal bandwidth to 2 kHz.

```
upConv = dsp.DigitalUpConverter(...
    'InterpolationFactor',20,...
    'SampleRate',Fs,...
    'Bandwidth',2e3,...
    'StopbandAttenuation',55,...
    'PassbandRipple',0.2,...
    'CenterFrequency',50e3);
```

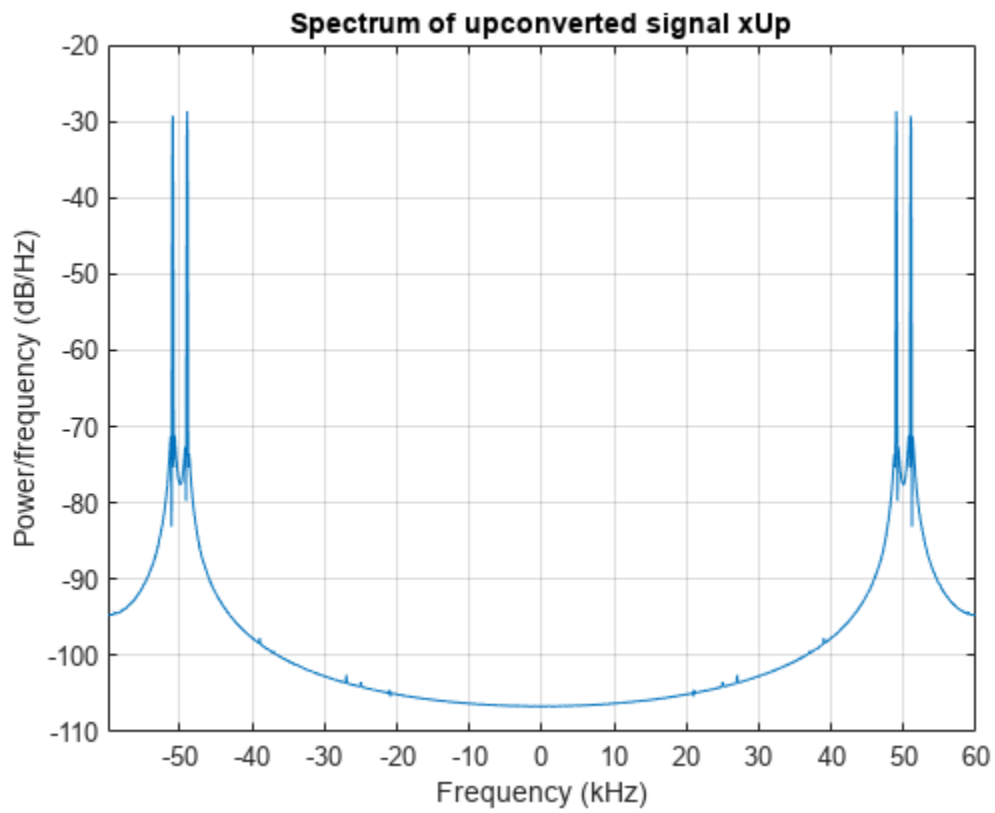
Create a spectrum estimator to visualize the signal spectrum before and after upconverting.

```
window = hamming(floor(length(x)/10));
figure; pwelch(x>window, [], [], Fs, 'centered')
title('Spectrum of baseband signal x')
```



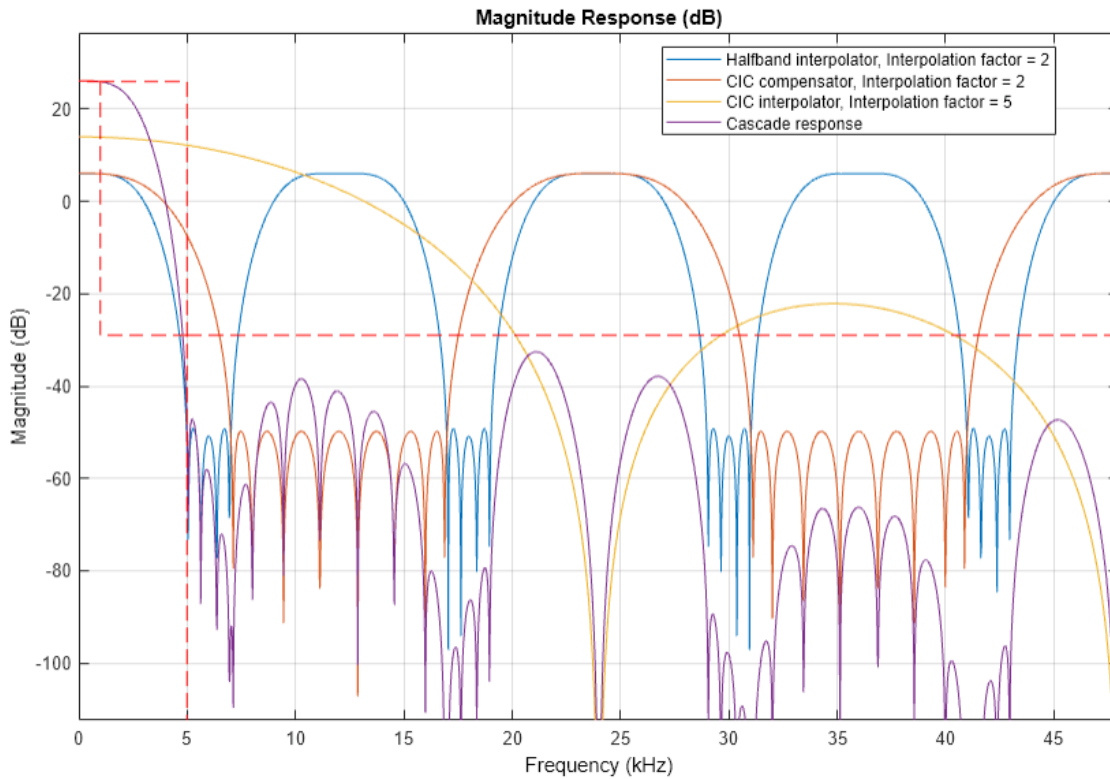
Upconvert the signal and visualize the spectrum.

```
xUp = upConv(x);
window = hamming(floor(length(xUp)/10));
figure;
pwelch(xUp>window, [], [], 20*Fs, 'centered')
title('Spectrum of upconverted signal xUp')
```



Visualize the response of the interpolation filters.

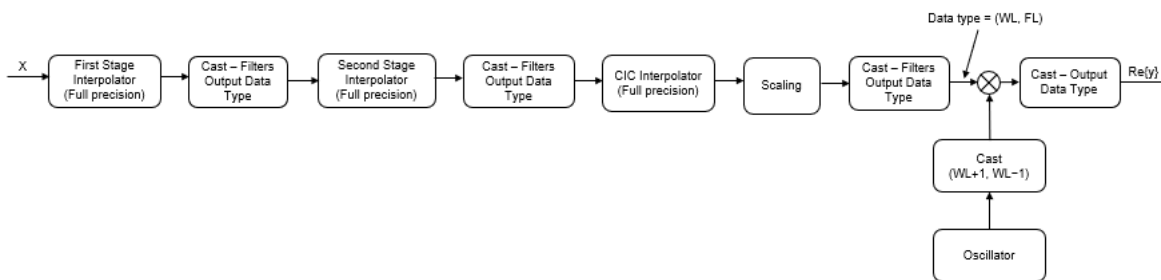
```
visualizeFilterStages(upConv)
```



## More About

### Fixed Point

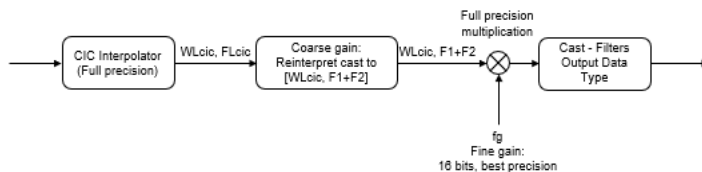
The block diagram represents the DUC arithmetic with signed fixed-point inputs.



- $WL$  is the word length of the input, and  $FL$  is the fraction length of the input.
- The output of each filter is cast to the filter output data type. In the `dsp.DigitalUpConverter` object, you can specify the filter output data type through the `FiltersOutputDataType` and `CustomFiltersOutputDataType` properties. In the Digital Up-Converter block, you can specify the filter output data type through the **Stage output** parameter. The casting of the CIC output occurs after the scaling factor is applied.

- The oscillator output is cast to a word length equal to the filter output data type word length plus one. The fraction length is equal to the filter output data type word length minus one.
- The scaling at the output of the CIC interpolator consists of coarse-gain and fine-gain adjustments. The coarse gain is achieved using the `reinterpretcast` function on the CIC interpolator output. The fine gain is achieved using full-precision multiplication.

The figure shows the coarse-gain and fine-gain operations.

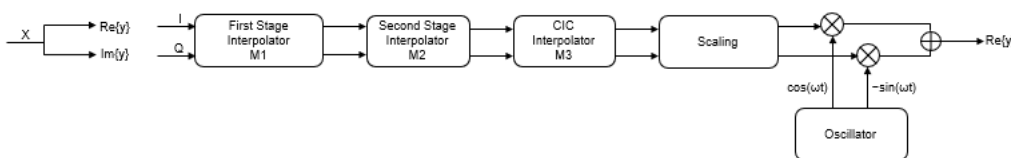


If the normalization gain is  $G$  (where  $0 < G \leq 1$ ), then:

- $WL_{cic}$  is the word length of the CIC interpolator output, and  $FL_{cic}$  is the fraction length of the CIC interpolator output.
- $F1 = \text{abs}(\text{nextpow2}(G))$ , indicating the part of  $G$  achieved by using bit shifts (coarse gain).
- $F2$  is the fraction length specified by the filter output data type.
- $fg = \text{fi}((2^{F1}) * G, \text{true}, 16)$ , which indicates that the remaining gain cannot be achieved with a bit shift (fine gain).

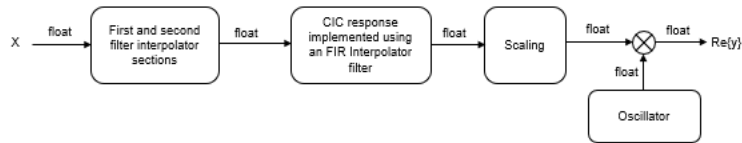
## Algorithms

The digital up converter upsamples the input signal using a cascade of three interpolation filters. This algorithm frequency-upconverts the upsampled signal by multiplying it with a complex exponential that has the specified center frequency. In this case, the filter cascade consists of an FIR interpolation stage, a second stage for CIC compensation, and a CIC interpolator. The block diagram shows the architecture of the digital up converter.



The scaling section normalizes the CIC gain and the oscillator power. It can also contain a correction factor to achieve the desired ripple specification. Depending on how you set the interpolation factor, the block bypasses the first filter stage. When the input data type is floating point, the algorithm implements an  $N$ -section CIC interpolation filter as a FIR filter with a response that corresponds to a cascade of  $N$  boxcar filters. The algorithm emulates a CIC filter with an FIR filter so that you can run simulations with floating-point data. When the input data type is a fixed-point type, the algorithm implements a true CIC filter with actual comb and integrator sections.

This block diagram represents the DUC arithmetic with floating-point inputs.



For details about fixed-point operation, see “Fixed Point” on page 4-408.

## Version History

Introduced in R2012a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

This object also supports SIMD code generation using Intel AVX2 technology when the input signal has a data type of `single` or `double`.

The SIMD technology significantly improves the performance of the generated code.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

This object supports HDL code generation with the Filter Design HDL Coder product. For workflows and limitations, see “Generate HDL Code for Filter System Objects” (Filter Design HDL Coder).

## See Also

### Functions

`getDecimationFactors` | `getFilterOrders` | `getFilters` | `fvtool` | `groupDelay` | `visualizeFilterStages` | `generatehdl`

### Objects

`dsp.DigitalDownConverter`

### Blocks

Digital Up-Converter | Digital Down-Converter

### Topics

“Digital Up and Down Conversion for Family Radio Service in MATLAB”

“Design and Analysis of a Digital Down Converter”



# getInterpolationFactors

**Package:** dsp

Get interpolation factors of each filter stage of digital upconverter

## Syntax

```
M = getInterpolationFactors(upConv)
```

## Description

`M = getInterpolationFactors(upConv)` returns a vector, `M`, with the interpolation factors of each filter stage of digital up converter `upConv`. If the first filter stage is bypassed, then `M` is a 1-by-2 vector that contains the interpolation factors of the second and third stages. If the first filter stage is not bypassed, then `M` is a 1-by-3 vector containing the interpolation factors of the first, second, and third filter stages.

## Examples

### Get Interpolation Factors

Get interpolation factors of each filter stage of a `dsp.DigitalUpConverter` System object™.

Create a `dsp.DigitalUpConverter` System object with the default settings. Using the `getInterpolationFactors` function, obtain the interpolation factors of each stage of the object.

```
upConv = dsp.DigitalUpConverter

upConv =
  dsp.DigitalUpConverter with properties:

    InterpolationFactor: 100
    MinimumOrderDesign: true
    Bandwidth: 200000
    StopbandFrequencySource: 'Auto'
    PassbandRipple: 0.1000
    StopbandAttenuation: 60
    Oscillator: 'Sine wave'
    CenterFrequency: 14000000
    SampleRate: 300000
```

Show all properties

```
M = getInterpolationFactors(upConv) %#ok
```

```
M = 1×3
     2     2    25
```

The `InterpolationFactor` property of the object is set to 100. By default, the output `M` is a 1-by-3 vector, where each element in the vector is a factor of the overall interpolation factor.

When you set the `InterpolationFactor` property to a 1-by-2 vector, the object bypasses the first filter stage and sets the interpolation factor of the second and third filtering stages to the values in the first and second vector elements, respectively. The output of the `getInterpolationFactors` function is now a 1-by-2 vector.

```
upConv.InterpolationFactor = [10 10]
```

```
upConv =  
dsp.DigitalUpConverter with properties:
```

```
    InterpolationFactor: [10 10]  
    MinimumOrderDesign: true  
                Bandwidth: 200000  
StopbandFrequencySource: 'Auto'  
    PassbandRipple: 0.1000  
    StopbandAttenuation: 60  
                Oscillator: 'Sine wave'  
    CenterFrequency: 14000000  
    SampleRate: 300000
```

```
Show all properties
```

```
M = getInterpolationFactors(upConv)
```

```
M = 1×2
```

```
    10    10
```

## Input Arguments

### **upConv** — Digital up converter

`dsp.DigitalUpConverter` System object

Digital up converter, specified as a `dsp.DigitalUpConverter` System object.

## Output Arguments

### **M** — Interpolation factors

vector

Interpolation factors of each filter stage, returned as a 1-by-2 or 1-by-3 vector. If the first filter stage is bypassed, then `M` is a 1-by-2 vector containing the interpolation factors of the second and third stages. If the first filter stage is not bypassed, then `M` is a 1-by-3 vector containing the interpolation factors of the first, second, and third filter stages.

Data Types: double

## Version History

Introduced in R2012a

## See Also

### Objects

`dsp.FilterCascade`

### Functions

`addStage` | `generateFilteringCode` | `getNumStages` | `releaseStages` | `removeStage`

# **dsp.DyadicAnalysisFilterBank**

**Package:** dsp

Dyadic analysis filter bank

## **Description**

The `dsp.DyadicAnalysisFilterBank` System object decomposes a broadband signal into a collection of subbands with smaller bandwidths and slower sample rates. The System object uses a series of highpass and lowpass FIR filters to provide approximate octave band frequency decompositions of the input. Each filter output is downsampled by a factor of two. With the appropriate analysis filters and tree structure, the dyadic analysis filter bank is a discrete wavelet transform (DWT) or discrete wavelet packet transform (DWPT).

To obtain approximate octave band frequency decompositions of the input:

- 1 Create the `dsp.DyadicAnalysisFilterBank` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## **Creation**

### **Syntax**

```
dydan = dsp.DyadicAnalysisFilterBank  
dydan = dsp.DyadicAnalysisFilterBank(Name, Value)
```

### **Description**

`dydan = dsp.DyadicAnalysisFilterBank` constructs a dyadic analysis filter bank object, `dydan`, that computes the level-two discrete wavelet transform (DWT) of a column vector input. For a 2-D matrix input, the object transforms the columns using the Daubechies third-order extremal phase wavelet. The length of the input along the first dimension must be a multiple of 4.

`dydan = dsp.DyadicAnalysisFilterBank(Name, Value)` returns a dyadic analysis filter bank object, with each property set to the specified value.

### **Properties**

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

**Filter — Type of filter used in subband decomposition**

Custom (default) | Biorthogonal | Coiflets | Daubechies | Discrete Meyer | Haar | Reverse Biorthogonal | Symlets

Specify the type of filter used to determine the high and lowpass FIR filters in the dyadic analysis filter bank as Custom, Haar, Daubechies, Symlets, Coiflets, Biorthogonal, Reverse Biorthogonal, or Discrete Meyer. All property values except Custom require Wavelet Toolbox software. If the value of this property is Custom, the filter coefficients are specified by the values of the CustomLowpassFilter and CustomHighpassFilter properties. Otherwise, the dyadic analysis filter bank object uses the Wavelet Toolbox function `wfilters` to construct the filters. The following table lists supported wavelet filters and example syntax to construct the filters:

| Filter                                | Example Setting                   | Syntax for Analysis Filters                                      |
|---------------------------------------|-----------------------------------|--|
| Haar                                  | N/A                               | <code>[Lo_D,Hi_D]=wfilters('haar');</code>                       |
| Daubechies extremal phase             | <code>WaveletOrder=3;</code>      | <code>[Lo_D,Hi_D]=wfilters('db3');</code>                        |
| Symlets (Daubechies least-asymmetric) | <code>WaveletOrder=4;</code>      | <code>[Lo_D,Hi_D]=wfilters('sym4');</code>                       |
| Coiflets                              | <code>WaveletOrder=1;</code>      | <code>[Lo_D,Hi_D]=wfilters('coif1');</code>                      |
| Biorthogonal                          | <code>FilterOrder='[3/1]';</code> | <code>[Lo_D,Hi_D,Lo_R,Hi_R]=...<br/>wfilters('bior3.1');</code>  |
| Reverse biorthogonal                  | <code>FilterOrder='[3/1]';</code> | <code>[Lo_D,Hi_D,Lo_R,Hi_R]=...<br/>wfilters('rbior3.1');</code> |
| Discrete Meyer                        | N/A                               | <code>[Lo_D,Hi_D]=wfilters('dmey');</code>                       |

**CustomLowpassFilter — Lowpass FIR filter coefficients**

`[0.0352 -0.0854 -0.1350 0.4599 0.8069 0.3327]` (default) | row vector

Specify a vector of lowpass FIR filter coefficients, in powers of  $z^{-1}$ . Use a half-band filter that passes the frequency band stopped by the filter specified in the CustomHighpassFilter property. The default specifies a Daubechies third-order extremal phase scaling (lowpass) filter.

**Dependencies**

This property applies when you set the Filter property to Custom.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**CustomHighpassFilter — Highpass FIR filter coefficients**

`[-0.3327 0.8069 -0.4599 -0.1350 0.0854 0.0352]` (default) | row vector

Specify a vector of highpass FIR filter coefficients, in powers of  $z^{-1}$ . Use a half-band filter that passes the frequency band stopped by the filter specified in the CustomLowpassFilter property. The default specifies a Daubechies 3rd-order extremal phase wavelet (highpass) filter.

**Dependencies**

This property applies when you set the Filter property to Custom.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**WaveletOrder — Order for orthogonal wavelets**

2 (default) | positive integer

Specify the order of the wavelet selected in the `Filter` property.

#### Dependencies

This property applies when you set the `Filter` property to an orthogonal wavelet: `Daubechies` (Daubechies extremal phase), `Symlets` (Daubechies least-asymmetric), or `Coiflets`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `fi`

#### FilterOrder — Analysis and synthesis filter orders for biorthogonal filters

1 / 1 (default) | 1 / 3 | 1 / 5 | 2 / 2 | 2 / 4 | 2 / 6 | 2 / 8 | 3 / 1 | 3 / 3 | 3 / 5 | 3 / 7 | 3 / 9 | 4 / 4 | 5 / 5 | 6 / 8

Specify the order of the analysis and synthesis filter orders for biorthogonal filter banks as 1 / 1, 1 / 3, 1 / 5, 2 / 2, 2 / 4, 2 / 6, 2 / 8, 3 / 1, 3 / 3, 3 / 5, 3 / 7, 3 / 9, 4 / 4, 5 / 5, or 6 / 8. Unlike orthogonal wavelets, biorthogonal wavelets require different filters for the analysis (decomposition) and synthesis (reconstruction) of an input. The first number indicates the order of the synthesis (reconstruction) filter. The second number indicates the order of the analysis (decomposition) filter.

#### Dependencies

This property applies when you set the `Filter` property to `Biorthogonal` or `ReverseBiorthogonal`.

Data Types: `char`

#### NumLevels — Number of filter bank levels used in analysis (decomposition)

2 (default) | integer greater than or equal to 1

Specify the number of filter bank analysis levels a positive integer greater than or equal to 1. A level- $N$  asymmetric structure produces  $N+1$  output subbands. A level- $N$  symmetric structure produces  $2^N$  output subbands. The size of the input along the first dimension must be a multiple of  $2^N$ , where  $N$  is the number of levels.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

#### TreeStructure — Structure of filter bank

`Asymmetric` (default) | `Symmetric`

Specify the structure of the filter bank as `Asymmetric` or `Symmetric`. The asymmetric structure decomposes only the lowpass filter output from each level. The symmetric structure decomposes the highpass and lowpass filter outputs from each level. If the analysis filters are scaling (lowpass) and wavelet (highpass) filters, the asymmetric structure is the discrete wavelet transform, while the symmetric structure is the discrete wavelet packet transform.

When this property is `Symmetric`, the output has  $2^N$  subbands each of size  $M/2^N$ . In this case,  $M$  is the length of the input along the first dimension and  $N$  is the value of the `NumLevels` property. When this property is `Asymmetric`, the output has  $N+1$  subbands. The following equation gives the length of the output in the  $k$ th subband in the asymmetric case:

$$M_k = \begin{cases} \frac{M}{2^k} & 1 \leq k \leq N \\ \frac{M}{2^N} & k = N + 1 \end{cases}$$

## Usage

### Syntax

```
y = dydan(x)
```

### Description

`y = dydan(x)` computes the subband decomposition of the input `x` and outputs the dyadic subband decomposition in `y` as a single concatenated column vector or matrix of coefficients.

### Input Arguments

#### **x** — Data input

column vector | matrix

Data input, specified as a column vector or a matrix. Each column of `x` is treated as an independent input, and the number of rows of `x` must be a multiple of  $2^N$ , where  $N$  is the number of levels specified by the `NumLevels` property.

Data Types: `single` | `double`

### Output Arguments

#### **y** — Dyadic subband decomposition output

column vector | matrix

Dyadic subband decomposition output, returned as a column vector or a matrix. The elements of `y` are ordered with the highest-frequency subband first followed by subbands in decreasing frequency.

When `TreeStructure` is set to `Symmetric`, the output has  $2^N$  subbands each of size  $M/2^N$ . In this case,  $M$  is the length of the input along the first dimension, and  $N$  is the value of the `NumLevels` property. When `TreeStructure` is set to `Asymmetric`, the output has  $N+1$  subbands. The following equation gives the length of the output in the  $k$ th subband in the asymmetric case:

$$M_k = \begin{cases} \frac{M}{2^k} & 1 \leq k \leq N \\ \frac{M}{2^N} & k = N + 1 \end{cases}$$

Data Types: `single` | `double`

## Object Functions

To use an object function, specify the `System` object as the first input argument. For example, to release system resources of a `System` object named `obj`, use this syntax:

```
release(obj)
```

## Common to All System Objects

`step` Run System object algorithm

release Release resources and allow changes to System object property values and input characteristics  
reset Reset internal states of System object

## Examples

### Filter Square Wave Using Dyadic Filter Banks

Denoise square wave input using dyadic analysis and synthesis filter banks.

```
t = 0:.0001:.0511;  
x= square(2*pi*30*t);  
xn = x' + 0.08*randn(length(x),1);  
dydanl = dsp.DyadicAnalysisFilterBank;
```

The filter coefficients correspond to a haar wavelet.

```
dydanl.CustomLowpassFilter = [1/sqrt(2) 1/sqrt(2)];  
dydanl.CustomHighpassFilter = [-1/sqrt(2) 1/sqrt(2)];  
dydsyn = dsp.DyadicSynthesisFilterBank;  
dydsyn.CustomLowpassFilter = [1/sqrt(2) 1/sqrt(2)];  
dydsyn.CustomHighpassFilter = [1/sqrt(2) -1/sqrt(2)];  
C = dydanl(xn);
```

Subband outputs.

```
C1 = C(1:256); C2 = C(257:384); C3 = C(385:512);
```

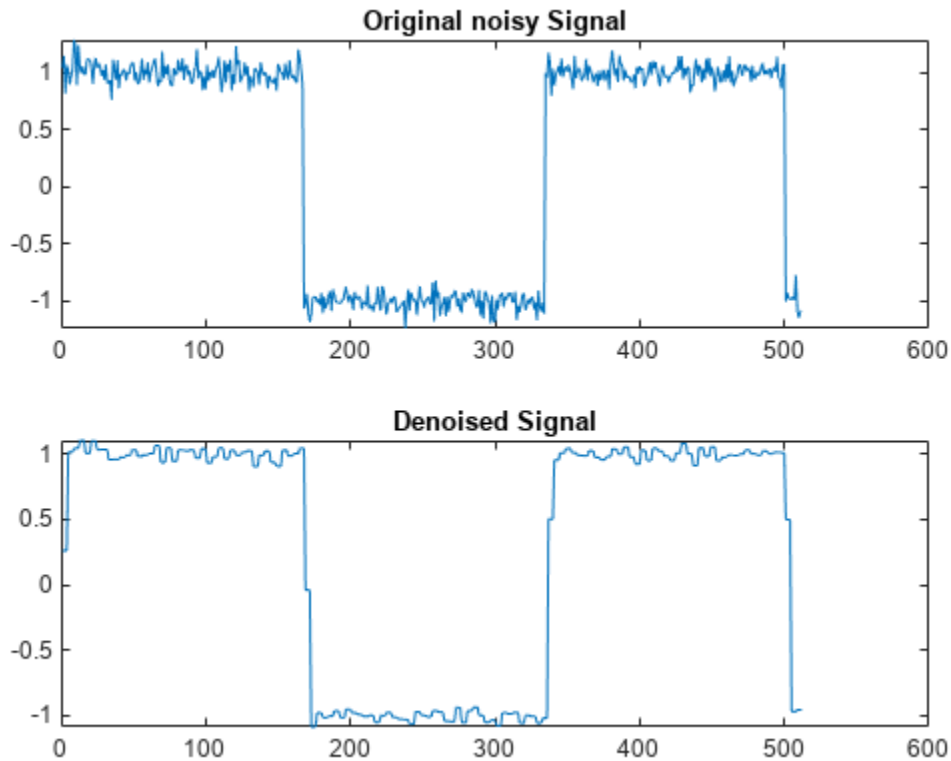
Set higher frequency coefficients to zero to remove the noise.

```
x_den = dydsyn([zeros(length(C1),1);...  
              zeros(length(C2),1);C3]);
```

Plot the original and denoised signals.

```
subplot(2,1,1), plot(xn); title('Original noisy Signal');  
subplot(2,1,2), plot(x_den); title('Denoised Signal');
```





### Subband Ordering For Asymmetric Tree Structure Using Dyadic Analysis Filter Bank

Sampling frequency 1 kHz input length 1024

```
t = 0:.001:1.023;
x = square(2*pi*30*t);
xn = x' + 0.08*randn(length(x),1);
```

Default asymmetric structure with order 3 extremal phase wavelet

```
dydan = dsp.DyadicAnalysisFilterBank;
Y = dydan(xn);
```

Level 2 yields 3 subbands (two detail-one approximation) Nyquist frequency is 500 Hz

```
D1 = Y(1:512); % subband approx. [250, 500] Hz
D2 = Y(513:768); % subband approx. [125, 250] Hz
Approx = Y(769:1024); % subband approx. [0,125] Hz
```

### Subband Ordering For Symmetric Tree Structure Using Dyadic Analysis Filter Bank

Sampling frequency 1 kHz input length 1024.

```
t = 0:.001:1.023;
x = square(2*pi*30*t);
xn = x' + 0.08*randn(length(x),1);

dydan = dsp.DyadicAnalysisFilterBank('TreeStructure',...
'Symmetric');
Y = dydan(xn);
D1 = Y(1:256); % subband approx. [375,500] Hz
D2 = Y(257:512); % subband approx. [250,375] Hz
D3 = Y(513:768); % subband approx. [125,250] Hz
Approx = Y(769:1024); % subband approx. [0, 125] Hz
```

### Algorithms

This object implements the algorithm, inputs, and outputs described on the Dyadic Analysis Filter Bank block reference page. The object properties correspond to the block parameters, except: The dyadic analysis filter bank object always concatenates the subbands into a single column vector for a column vector input, or into the columns of a matrix for a matrix input. This behavior corresponds to the block's behavior when you set the **Output** parameter to `Single port`.

### Version History

Introduced in R2012a

### See Also

#### Objects

`dsp.DyadicSynthesisFilterBank` | `dsp.SubbandAnalysisFilter`

# dsp.DyadicSynthesisFilterBank

**Package:** dsp

Reconstruct signals from subbands

## Description

The `dsp.DyadicSynthesisFilterBank` System object reconstructs signals from subbands with smaller bandwidths and lower sample rates. The filter bank uses a series of highpass and lowpass FIR filters to repeatedly reconstruct the signal.

To reconstruct signals from subbands with smaller bandwidths and lower sample rates:

- 1 Create the `dsp.DyadicSynthesisFilterBank` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
dydsyn = dsp.DyadicSynthesisFilterBank
dydsyn = dsp.DyadicSynthesisFilterBank(Name, Value)
```

### Description

`dydsyn = dsp.DyadicSynthesisFilterBank` returns a synthesis filter bank, `dydsyn`, that reconstructs a signal from its subbands with smaller bandwidths and smaller sample rates.

`dydsyn = dsp.DyadicSynthesisFilterBank(Name, Value)` returns a dyadic synthesis filter bank object, with each property set to the specified value.

### Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

#### Filter — Type of filter used in filter bank

Custom (default) | Biorthogonal | Coiflets | Daubechies | Discrete Meyer | Haar | Reverse Biorthogonal | Symlets

Specify the type of filter used to determine the highpass and lowpass FIR filters in the filter bank as one of Custom, Haar, Daubechies, Symlets, Coiflets, Biorthogonal, Reverse

Biorthogonal, or Discrete Meyer. If you set this property to Custom, the CustomLowpassFilter and CustomHighpassFilter properties specify the filter coefficients. Otherwise, the object uses the wfilters function to construct the filters. Depending on the filter, the WaveletOrder or “FilterOrder” on page 4-0 property might apply. For a list of the supported wavelets, see the following table.

| Filter               | Sample Setting for Related Filter Specification Properties | Corresponding Wavelet Toolbox Function Syntax |
|----------------------|--|---|
| Haar                 | None   | wfilters('haar')                              |
| Daubechies           | H.WaveletOrder = 4   | wfilters('db4')                               |
| Symlets              | H.WaveletOrder = 3   | wfilters('sym3')                              |
| Coiflets             | H.WaveletOrder = 1   | wfilters('coif1')                             |
| Biorthogonal         | H.FilterOrder = '[3/1]'                                    | wfilters('bior3.1')                           |
| Reverse Biorthogonal | H.FilterOrder = '[3/1]'                                    | wfilters('rbior3.1')                          |
| Discrete Meyer       | None   | wfilters('dmey')                              |

In order to automatically design wavelet-based filters, install the Wavelet Toolbox product. Otherwise, use the CustomLowpassFilter and CustomHighpassFilter properties to specify lowpass and highpass FIR filters.

#### CustomLowpassFilter — Lowpass FIR filter coefficients

[0.3327 0.8069 0.4599 -0.1350 -0.0854 0.0352] (default) | row vector

Specify a vector of lowpass FIR filter coefficients, in descending powers of  $z$ . Use a half-band filter that passes the frequency band stopped by the filter specified in the CustomHighpassFilter property. To perfectly reconstruct a signal decomposed by the dsp.DyadicAnalysisFilterBank object, design the filters in the synthesis filter bank to perfectly reconstruct the outputs of the analysis filter bank. Otherwise, the reconstruction is imperfect. The default values of this property specify a perfect reconstruction filter for the default settings of the analysis filter bank (based on a third-order Daubechies wavelet).

#### Dependencies

This property applies when you set the Filter property to Custom.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

#### CustomHighpassFilter — Highpass FIR filter coefficients

[0.0352 0.0854 -0.1350 -0.4599 0.8069 -0.3327] (default) | row vector

Specify a vector of highpass FIR filter coefficients, in descending powers of  $z$ . Use a half-band filter that passes the frequency band stopped by the filter specified in the CustomLowpassFilter property. To perfectly reconstruct a signal decomposed by the dsp.DyadicAnalysisFilterBank object, design the filters in the synthesis filter bank to perfectly reconstruct the outputs of the analysis filter bank. Otherwise, the reconstruction is imperfect. The default values of this property specify a perfect reconstruction filter for the default settings of the analysis filter bank (based on a third-order Daubechies wavelet).

#### Dependencies

This property applies when you set the Filter property to Custom.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### WaveletOrder — Wavelet order

2 (default) | positive integer

Specify the order of the wavelet selected in the `Filter` property.

#### Dependencies

This property applies when you set the `Filter` property to an orthogonal wavelet: `Daubechies` (Daubechies extremal phase), `Symlets` (Daubechies least-asymmetric), or `Coiflets`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `fi`

### FilterOrder — Wavelet order for synthesis filter stage

1 / 1 (default) | 1 / 3 | 1 / 5 | 2 / 2 | 2 / 4 | 2 / 6 | 2 / 8 | 3 / 1 | 3 / 3 | 3 / 5 | 3 / 7 | 3 / 9 | 4 / 4 | 5 / 5 | 6 / 8

Specify the order of the wavelet for the synthesis filter stage as:

- First order: `'[1/1]'`, `'[1/3]'`, or `'[1/5]'`.
- Second order: `'[2/2]'`, `'[2/4]'`, `'[2/6]'`, or `'[2/8]'`.
- Third order: `'[3/1]'`, `'[3/3]'`, `'[3/5]'`, `'[3/7]'`, or `'[3/9]'`.
- Fourth order: `'[4/4]'`.
- Fifth order: `'[5/5]'`.
- Sixth order: `'[6/8]'`.

#### Dependencies

This property applies when you set the `Filter` property to `Biorthogonal` or `ReverseBiorthogonal`.

Data Types: `char`

### NumLevels — Number of filter bank levels

2 (default) | integer greater than or equal to 1

Specify the number of filter bank levels as a scalar integer. An  $N$ -level asymmetric structure has  $N + 1$  input subbands, and an  $N$ -level symmetric structure has  $2^N$  input subbands.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### TreeStructure — Structure of filter bank

`Asymmetric` (default) | `Symmetric`

Specify the structure of the filter bank as `Asymmetric` or `Symmetric`. In the asymmetric structure, the low-frequency subband input to each level is the output of the previous level, while the high-frequency subband input to each level is an input to the filter bank. In the symmetric structure, both the low- and high-frequency subband inputs to each level are outputs from the previous level.

## Usage

### Syntax

```
y = dydsyn(x)
```

### Description

`y = dydsyn(x)` reconstructs the concatenated subband input `x` to output `y`. Each column of input `x` contains the subbands for an independent signal. Upper rows contain the high-frequency subbands, and lower rows contain the low-frequency subbands.

### Input Arguments

#### **x** — Data input

column vector | matrix

Data input, specified as a column vector or a matrix. Each column of input `x` contains the subbands for an independent signal. Upper rows contain the high-frequency subbands, and lower rows contain the low-frequency subbands. The number of rows of `x` must be a multiple of  $2^N$ , where  $N$  is the value of the `NumLevels` property.

Data Types: `single` | `double`

### Output Arguments

#### **y** — Reconstructed signal

column vector | matrix

Reconstructed signal, returned as a column vector or a matrix.

Data Types: `single` | `double`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Common to All System Objects

|                      |  |
|----------------------|--|
| <code>step</code>    | Run System object algorithm  |
| <code>release</code> | Release resources and allow changes to System object property values and input characteristics |
| <code>reset</code>   | Reset internal states of System object   |

## Examples

### Filter Square Wave Using Dyadic Filter Banks

Denoise square wave input using dyadic analysis and synthesis filter banks.

```

t = 0:.0001:.0511;
x= square(2*pi*30*t);
xn = x' + 0.08*randn(length(x),1);
dydanl = dsp.DyadicAnalysisFilterBank;

```

The filter coefficients correspond to a haar wavelet.

```

dydanl.CustomLowpassFilter = [1/sqrt(2) 1/sqrt(2)];
dydanl.CustomHighpassFilter = [-1/sqrt(2) 1/sqrt(2)];
dydsyn = dsp.DyadicSynthesisFilterBank;
dydsyn.CustomLowpassFilter = [1/sqrt(2) 1/sqrt(2)];
dydsyn.CustomHighpassFilter = [1/sqrt(2) -1/sqrt(2)];
C = dydanl(xn);

```

Subband outputs.

```

C1 = C(1:256); C2 = C(257:384); C3 = C(385:512);

```

Set higher frequency coefficients to zero to remove the noise.

```

x_den = dydsyn([zeros(length(C1),1);...
               zeros(length(C2),1);C3]);

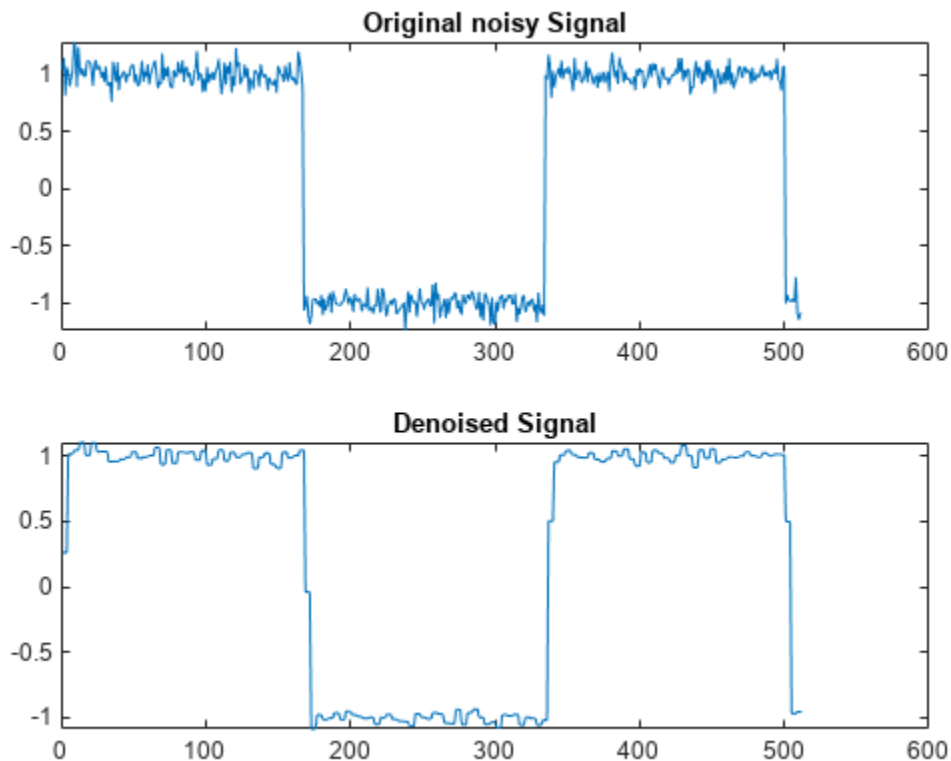
```

Plot the original and denoised signals.

```

subplot(2,1,1), plot(xn); title('Original noisy Signal');
subplot(2,1,2), plot(x_den); title('Denoised Signal');

```



### Algorithms

This object implements the algorithm, inputs, and outputs described on the Dyadic Synthesis Filter Bank block reference page. The object properties correspond to the block parameters, except: The object only receives data as a vector or matrix of concatenated subbands.

### Version History

Introduced in R2012a

### See Also

#### Objects

`dsp.DyadicAnalysisFilterBank` | `dsp.SubbandSynthesisFilter`



# dsp.FarrowRateConverter

**Package:** dsp

Polynomial sample rate converter with arbitrary conversion factor

## Description

The `dsp.FarrowRateConverter` System object implements a polynomial-fit sample rate conversion filter using a Farrow structure. You can use this object to convert the sample rate of a signal up or down by an arbitrary factor. This object supports fixed-point operations.

To convert the sample rate of a signal:

- 1 Create the `dsp.FarrowRateConverter` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
frc = dsp.FarrowRateConverter
frc = dsp.FarrowRateConverter(Name,Value)
frc = dsp.FarrowRateConverter(fsIn,fsOut,tol,np)
```

### Description

`frc = dsp.FarrowRateConverter` creates a polynomial filter-based sample rate converter System object, `frc`. For each channel of an input signal, `frc` converts the input sample rate to the output sample rate.

`frc = dsp.FarrowRateConverter(Name,Value)` sets properties using one or more name-value pairs. Enclose each property name in single quotes.

Example: `frc = dsp.FarrowRateConverter('Specification','Coefficients','Coefficients',[1 2; 3 4])` returns a filter that converts from 44.1 kHz to 48 kHz using custom coefficients that implement a 2nd-order polynomial filter.

`frc = dsp.FarrowRateConverter(fsIn,fsOut,tol,np)` returns a sample rate converter System object, `frc`, with `InputSampleRate` property set to `fsIn`, `OutputSampleRate` property set to `fsOut`, `OutputRateTolerance` property set to `tol`, and `PolynomialOrder` property set to `np`.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

### Filter Properties

#### **InputSampleRate — Sample rate of input signal**

44100 (default) | positive scalar in Hz

Sample rate of the input signal, specified as a positive scalar in Hz. The input sample rate must be greater than the bandwidth of interest.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

#### **OutputSampleRate — Sample rate of output signal**

48000 (default) | positive scalar in Hz

Sample rate of the output signal, specified as a positive scalar in Hz. The output sample rate can represent an upsample or downsample of the input signal.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

#### **OutputRateTolerance — Maximum tolerance for output sample rate**

0 (default) | positive scalar

Maximum tolerance for the output sample rate, specified as a positive scalar from 0 through 0.5, inclusive.

The actual output sample rate varies but is within the specified range. For example, if `OutputRateTolerance` is specified as `0.01`, then the actual output sample rate is in the range `OutputSampleRate ± 1%`. This flexibility often enables a simpler filter design.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

#### **Specification — Method for specifying filter coefficients**

'Polynomial order' (default) | 'Coefficients'

Method for specifying filter coefficients for the interpolator filter, specified as one of the following:

- 'Polynomial order' — Use the `PolynomialOrder` property to specify the order of the Lagrange-interpolation-filter polynomial. The object calculates coefficients that meet the rate and tolerance properties.
- 'Coefficients' — Use the `Coefficients` property to specify the polynomial coefficients directly.

#### **PolynomialOrder — Order of Lagrange-interpolation-filter polynomial**

3 (default) | positive integer less than or equal to 4

Order of the Lagrange-interpolation-filter polynomial, specified as a positive integer less than or equal to 4. The object calculates coefficients that meet the rate and tolerance properties.

**Dependencies**

This property applies only when you set Specification to 'Polynomial order'.

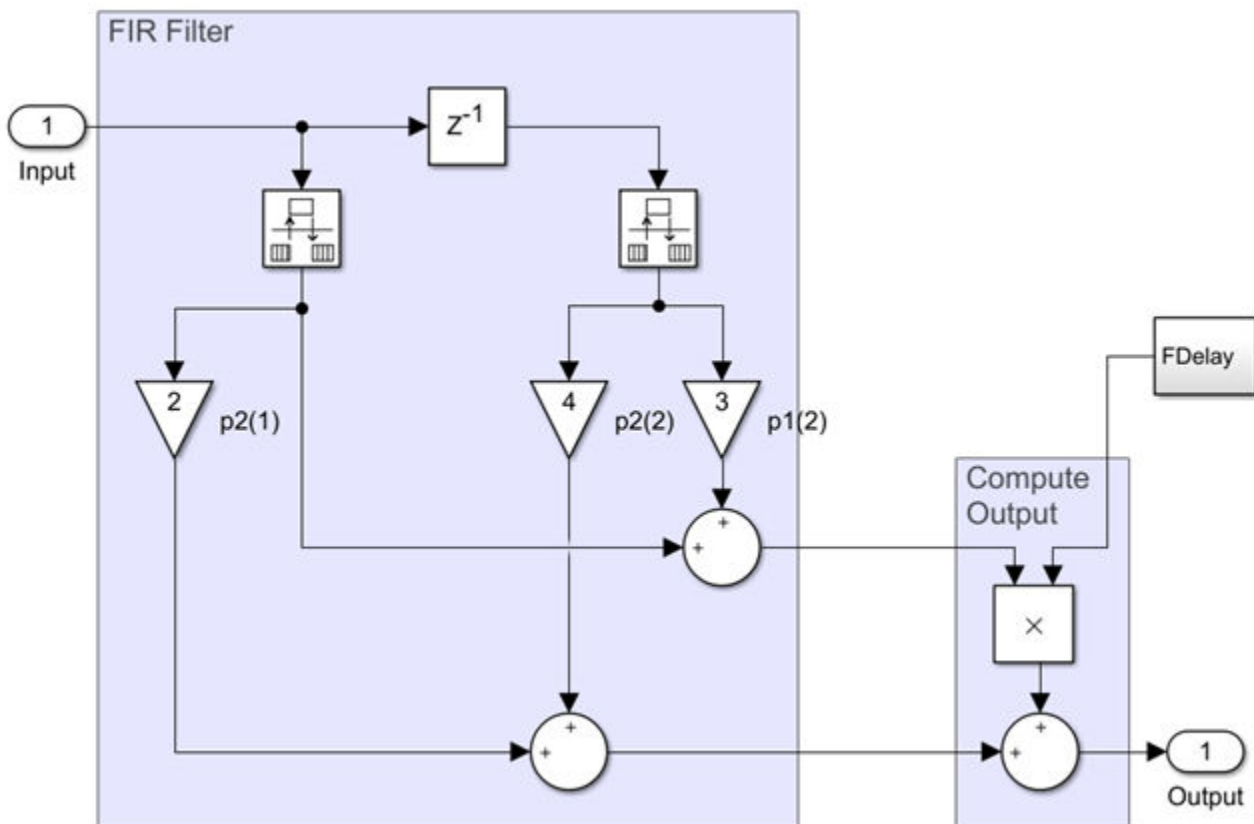
Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**Coefficients — Filter polynomial coefficients**

[-1 1; 1 0] (default) | real-valued square matrix

Filter polynomial coefficients, specified as a real-valued  $M$ -by- $M$  matrix, where  $M$  is the polynomial order.

The diagram shows the signal flow graph for a dsp.FarrowRateConverter object with coefficients set to [1 2; 3 4].



Each branch of the FIR filter corresponds to a row of the coefficient matrix.

**Dependencies**

This property applies only when you set Specification to 'Coefficients'.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**Fixed-Point Properties****RoundingMethod — Rounding method for fixed-point operations**

'Floor' (default) | 'Ceiling' | 'Convergent' | 'Nearest' | 'Round' | 'Zero'

Rounding method for fixed-point operations, specified as a character vector. For more information on the rounding methods see “Rounding Modes”.

**OverflowAction — Overflow action for fixed-point operations**

'Wrap' (default) | 'Saturate'

Overflow action for fixed-point operations, specified as either 'Wrap' or 'Saturate'. For more details on the overflow actions, see “Overflow Handling”.

**CoefficientsDataType — Data type of filter coefficients**

numerictype(1,16) (default) | numerictype object

Data type of the filter coefficients, specified as a signed numerictype object. The default data type is a signed, 16-bit numerictype object. You must specify a numerictype object without specific binary-point scaling. To maximize precision, the object determines the fraction length of this data type based on the coefficient values.

**FractionalDelayDataType — Data type of fractional delay**

numerictype(0,8) (default) | numerictype object

Data type of the fractional delay, specified as an unsigned numerictype object. The default data type is an unsigned, 8-bit numerictype object. You must specify a numerictype object without specific binary-point scaling. To maximize precision, the object determines the fraction length of this data type based on the fractional delay values.

**MultiplicandDataType — Data type of multiplicand**

numerictype(1,16,13) (default) | numerictype object

Data type of the multiplicand, specified as a signed numerictype object. The default data type is a signed 16-bit numerictype object with 13-bit fraction length. You must specify a numerictype object that has a specific binary point scaling.

**OutputDataType — Data type of output**

'Same word length as input' (default) | 'Same as accumulator' | numerictype object

Data type of the output, specified as one of the following:

- 'Same word length as input' — Output word length and fraction length are the same as the input.
- 'Same as accumulator' — Output word length and fraction length are the same as the accumulator.
- numerictype object — Signed fixed-point output data type. If you do not specify a fraction length, the object computes the fraction length based on the input range. The object preserves the dynamic range of the input.

## Usage

### Syntax

```
y = frc(x)
```

### Description

$y = \text{frc}(x)$  resamples input  $x$  to create output  $y$  according to the rate conversion defined by  $\text{frc}$ .

### Input Arguments

#### **x** — Input signal

vector | matrix

Input signal, specified as a vector or a matrix. Each column of the input is treated as a separate channel.

### Output Arguments

#### **y** — Resampled signal

vector | matrix

Resampled signal, returned as a vector or matrix.

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

### Specific to dsp.FarrowRateConverter

|  |   |
|--|---|
| <code>getPolynomialCoefficients</code> | Get polynomial coefficients of farrow rate conversion filter                  |
| <code>getActualOutputRate</code>       | Get actual output rate  |
| <code>getRateChangeFactors</code>      | Get overall interpolation and decimation factors                              |
| <code>generatehdl</code>               | Generate HDL code for quantized DSP filter (requires Filter Design HDL Coder) |

### Filter Analysis

|                     |  |
|---------------------|--|
| <code>freqz</code>  | Frequency response of discrete-time filter System object |
| <code>fvtool</code> | Visualize frequency response of DSP filters              |
| <code>info</code>   | Information about filter System object                   |
| <code>cost</code>   | Estimate cost of implementing filter System object       |

### Common to All System Objects

|                      |  |
|----------------------|--|
| <code>step</code>    | Run System object algorithm  |
| <code>release</code> | Release resources and allow changes to System object property values and input characteristics |
| <code>reset</code>   | Reset internal states of System object   |

## Examples

### Upsample an Audio Signal Using `dsp.FarrowRateConverter`

**Note:** The `dsp.AudioFileWriter` System object™ is not supported in MATLAB Online.

Create a `dsp.FarrowRateConverter` System object™ to convert an audio signal from 44.1 kHz to 96 kHz. Set the polynomial order for the filter.

```
FsIn = 44.1e3;
FsOut = 96e3;
LagrangeOrder = 2; % 1 = linear interpolation
frc = dsp.FarrowRateConverter('InputSampleRate',FsIn,...
                              'OutputSampleRate',FsOut,...
                              'PolynomialOrder',LagrangeOrder);
ar = dsp.AudioFileReader('guitar10min.ogg','SamplesPerFrame',14700);
aw = dsp.AudioFileWriter('guitar10min_96kHz.wav','SampleRate',FsOut);
```

Check the resulting interpolation and decimation factors.

```
[interp,decim] = getRateChangeFactors(frc)
```

```
interp = 320
```

```
decim = 147
```

Display the polynomial that the object uses to fit the input samples.

```
coeffs = getPolynomialCoefficients(frc)
```

```
coeffs = 3x3
```

```
    0.5000    -0.5000         0
   -1.0000         0     1.0000
    0.5000     0.5000         0
```

Convert 100 frames of the audio signal. Write the result to a file.

```
for n = 1:1:100
    x = ar();
    y = frc(x);
    aw(y);
end
```

Release the `dsp.AudioFileWriter` System object™ to complete creation of the output file.

```
release(aw)
release(ar)
```

Plot the input and output signals. The latency of the Farrow rate converter introduces a delay in the output signal.

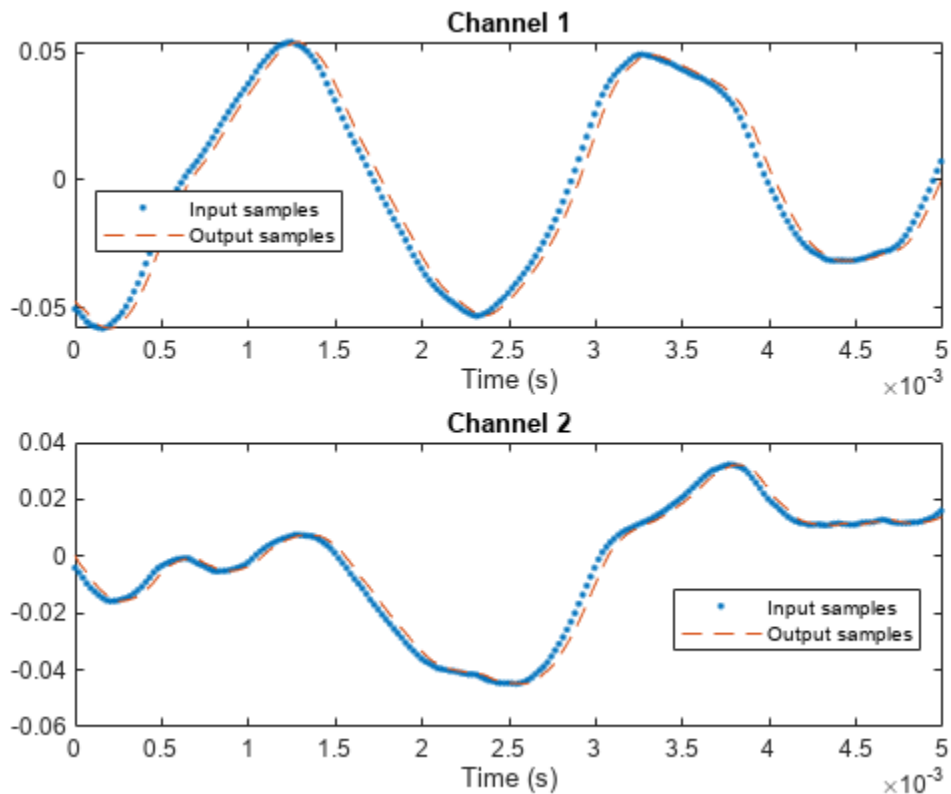
```
tx = (0:length(x)-1)./FsIn;
ty = (0:length(y)-1)./FsOut;
```

```
figure
```

```

subplot(2,1,1)
plot(tx,x(:,1),'.')
hold on
plot(ty,y(:,1),'--')
xlim([0 0.005])
xlabel('Time (s)')
legend('Input samples','Output samples','Location','best')
title('Channel 1')
subplot(2,1,2)
plot(tx,x(:,2),'.')
hold on
plot(ty,y(:,2),'--')
xlim([0 0.005])
xlabel('Time (s)')
legend('Input samples','Output samples','Location','best')
title('Channel 2')

```



Use the `outputDelay` function to determine this delay value. To account for this delay, shift the output by this delay value.

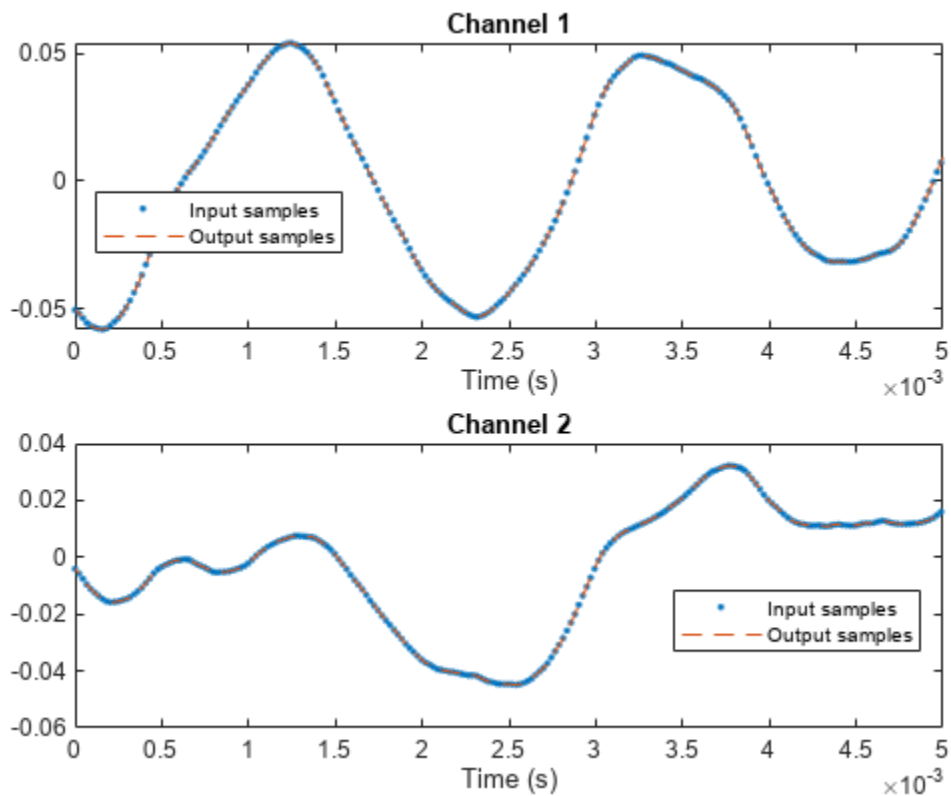
```
[delay,~,~] = outputDelay(frc,Fc=0)
```

```
delay = 4.5351e-05
```

```
tx = (0:length(x)-1)./FsIn;
ty = (0:length(y)-1)./FsOut;
```

figure

```
subplot(2,1,1)
plot(tx,x(:,1),'.')
hold on
plot(ty-delay,y(:,1),'--')
xlim([0 0.005])
xlabel('Time (s)')
legend('Input samples','Output samples','Location','best')
title('Channel 1')
subplot(2,1,2)
plot(tx,x(:,2),'.')
hold on
plot(ty-delay,y(:,2),'--')
xlim([0 0.005])
xlabel('Time (s)')
legend('Input samples','Output samples','Location','best')
title('Channel 2')
```



Zoom in to see the difference in sample rates.

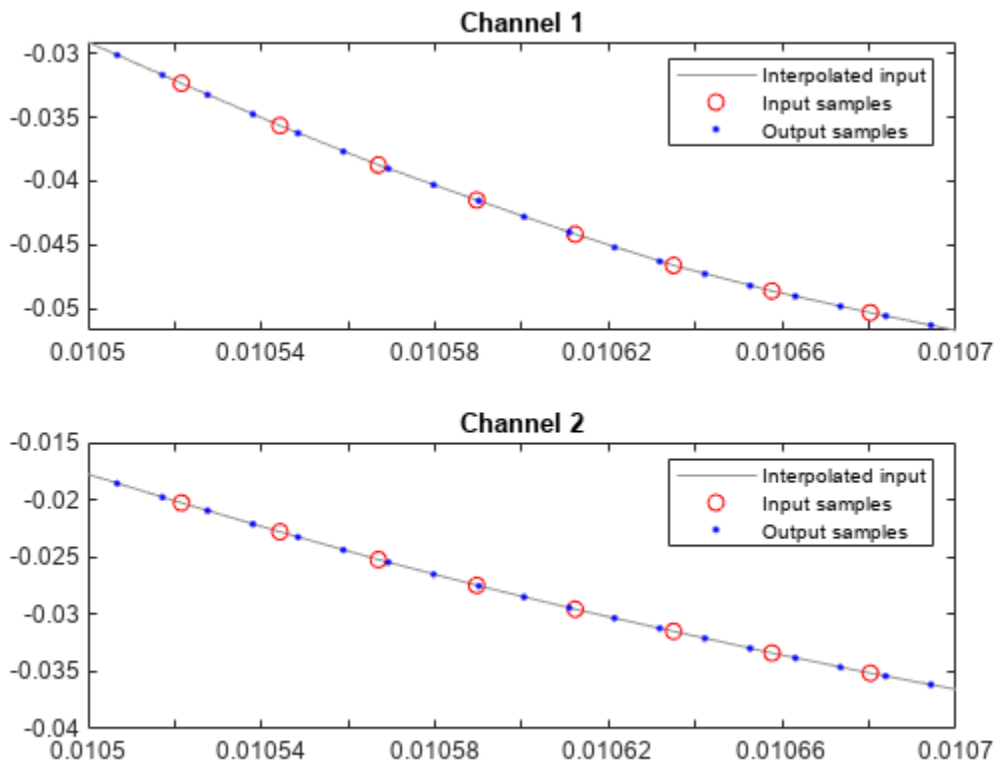
```
figure
subplot(2,1,1)
plot(tx,x(:,1),Color=[0.6 0.6 0.6])
hold on
plot(tx,x(:,1),'ro')
plot(ty-delay,y(:,1),'b.')
xlim([0.0105 0.0107])
```



```

legend('Interpolated input','Input samples','Output samples')
title('Channel 1')
subplot(2,1,2)
plot(tx,x(:,2),Color=[0.6 0.6 0.6])
hold on
plot(tx,x(:,2),'ro')
plot(ty-delay,y(:,2),'b.')
xlim([0.0105 0.0107])
legend('Interpolated input','Input samples','Output samples')
title('Channel 2')

```



### Reduce Input Size Restriction by Adjusting Tolerance

Create a `dsp.FarrowRateConverter` System object™ with 0% tolerance. The output rate is equal to the `OutputSampleRate` property. The input size must be a multiple of the decimation factor,  $M$ . In this case  $M$  is 320.

```

frc = dsp.FarrowRateConverter('InputSampleRate',96e3,...
    'OutputSampleRate',44.1e3);
FsOut = getActualOutputRate(frc)

FsOut = 44100

[L,M] = getRateChangeFactors(frc)

L = 147

```

```
M = 320
```

Allow a 1% tolerance on the output rate and observe the difference in decimation factor.

```
frc.OutputRateTolerance = 0.01;  
FsOut2 = getActualOutputRate(frc)
```

```
FsOut2 = 4.4308e+04
```

```
[L2,M2] = getRateChangeFactors(frc)
```

```
L2 = 6
```

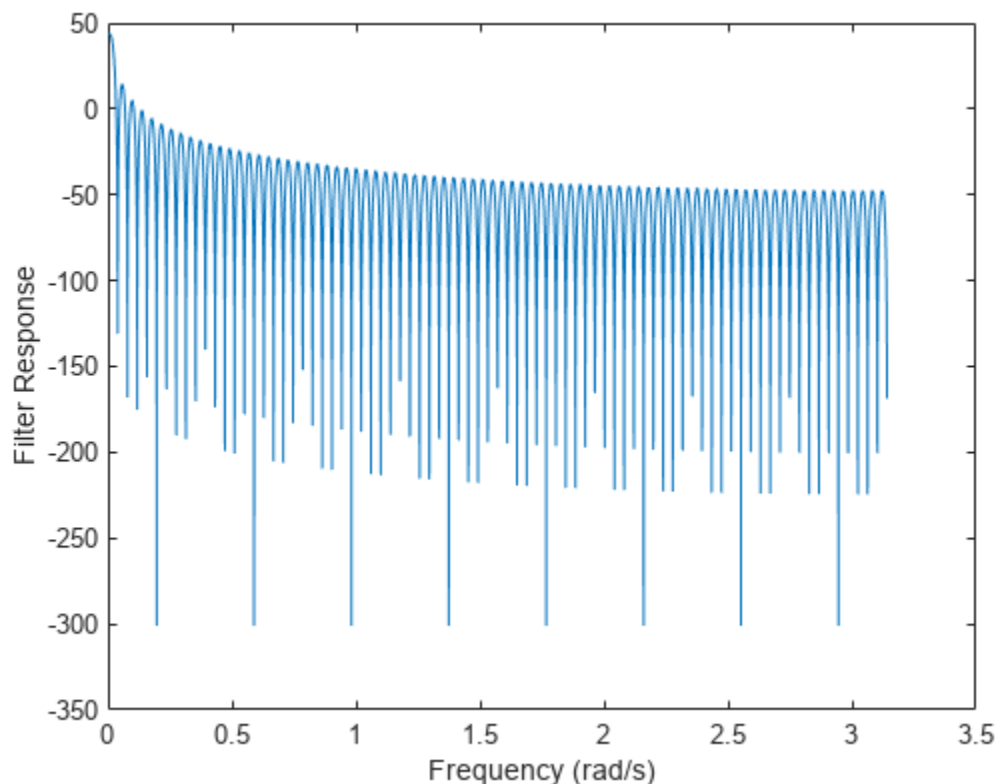
```
M2 = 13
```

The decimation factor is now only 13. The lower the decimation factor, the more flexibility in input size. The output rate is within the range `OutputSampleRate ± 1%`.

### Frequency Response of `dsp.FarrowRateConverter`

Create a `dsp.FarrowRateConverter` System object™ with default properties. Compute and display the frequency response.

```
frc = dsp.FarrowRateConverter;  
[h,f] = freqz(frc);  
plot(f,20*log10(abs(h)))  
ylabel('Filter Response')  
xlabel('Frequency (rad/s)')
```



## Determine Computational Cost of dsp.FarrowRateConverter

Create a dsp.FarrowRateConverter System object™ with default values. Determine its computational cost: the number of coefficients, number of states, multiplications per input sample, and additions per input sample.

```
frc = dsp.FarrowRateConverter;
cst = cost(frc)

cst = struct with fields:
    NumCoefficients: 16
    NumStates: 3
    MultiplicationsPerInputSample: 13.0612
    AdditionsPerInputSample: 11.9728
```

Repeat the computation, allowing for a 10% tolerance in the output sample rate.

```
frc.OutputRateTolerance = 0.1;
ctl = cost(frc)

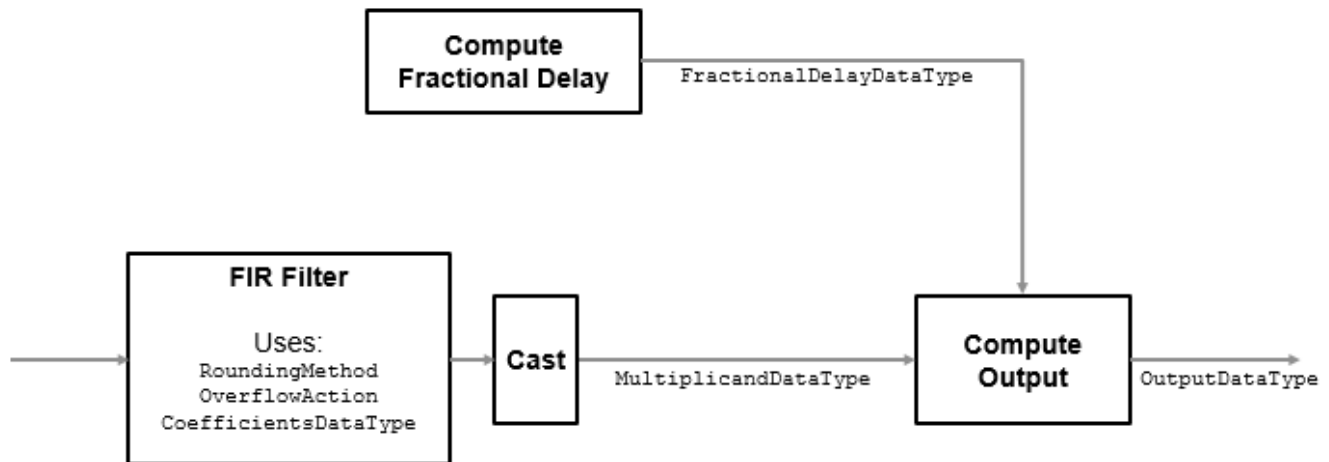
ctl = struct with fields:
    NumCoefficients: 16
    NumStates: 3
    MultiplicationsPerInputSample: 12
    AdditionsPerInputSample: 11
```

## More About

### Fixed Point

If the input is fixed point, it must be a signed integer or a signed fixed point value with a power-of-two slope and zero bias.

The diagram shows the data types that the dsp.FarrowRateConverter object uses for fixed-point signals and floating-point signals. You can specify these data types using the properties of the object, see “Fixed-Point Properties” on page 4-0 . If the input is floating point, all data types in filter are the same as the input data type, single or double.



If the input is fixed point, the FIR filter defines internal data types using the `RoundingMode`, `OverflowMode`, and `CoefficientsDataType` properties. The accumulators and products within the FIR filter use full precision data types. The object casts the output of the FIR filter to `MultiplicandDataType`.

## Algorithms

Farrow filters implement piecewise polynomial interpolation using Horner's rule to compute samples from the polynomial. The polynomial coefficients used to fit the input samples correspond to the Lagrange interpolation coefficients.

Once a polynomial is fitted to the input data, the value of the polynomial can be calculated at any point. Therefore, a polynomial filter enables interpolation at arbitrary locations between input samples.

You can use a polynomial of any order to fit to the existing samples. However, since large-order polynomials frequently oscillate, polynomials of order 1, 2, 3, or 4 are used in practice.

The algorithm computes interpolated values at the desired locations by varying only the fractional delay  $\mu$ . This value is the interval between the previous input sample and the current output sample. All filter coefficients remain constant.

- The input samples are filtered using  $M + 1$  FIR filters, where  $M$  is the polynomial order.
- The outputs of these filters are multiplied by the fractional delay,  $\mu$ .
- The output is the sum of the multiplication results.

## Version History

Introduced in R2014b

## References

- [1] Hentschel, T., and G. Fettweis. "Continuous-Time Digital Filters for Sample-Rate Conversion in Reconfigurable Radio Terminals." *Frequenz*. Vol. 55, Number 5-6, 2001, pp. 185-188.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

The `getRateChangeFactors` function supports C and C++ code generation.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

This object supports HDL code generation with the Filter Design HDL Coder product. For workflows and limitations, see “Generate HDL Code for Filter System Objects” (Filter Design HDL Coder).

## See Also

### Functions

`getPolynomialCoefficients` | `getActualOutputRate` | `getRateChangeFactors` | `generatehdl` | `freqz` | `fvtool` | `info` | `cost`

### Objects

`dsp.FIRRateConverter` | `dsp.SampleRateConverter`

### Blocks

Farrow Rate Converter

### Topics

“Efficient Sample Rate Conversion Between Arbitrary Factors”

# **dsp.FastTransversalFilter**

**Package:** dsp

Fast transversal least-squares FIR adaptive filter

## **Description**

The `dsp.FastTransversalFilter` computes output, error and coefficients using a fast transversal least-squares FIR adaptive filter.

To implement the adaptive FIR filter object:

- 1 Create the `dsp.FastTransversalFilter` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## **Creation**

### **Syntax**

```
ftf = dsp.FastTransversalFilter  
ftf = dsp.FastTransversalFilter(len)  
ftf = dsp.FastTransversalFilter(Name,Value)
```

### **Description**

`ftf = dsp.FastTransversalFilter` returns a System object, `ftf`, which is a fast transversal, least-squares FIR adaptive filter. This System object computes the filtered output and the filter error for a given input and desired signal.

`ftf = dsp.FastTransversalFilter(len)` returns a `dsp.FastTrasversalFilter` System object with the `Length` property set to `len`.

`ftf = dsp.FastTransversalFilter(Name,Value)` returns a `dsp.FastTransversalFilter` System object with each specified property set to the specified value.

### **Properties**

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

**Method — Method to calculate filter coefficients**

'Fast transversal least-squares' (default) | 'Sliding-window fast transversal least-squares'

Specify the method used to calculate filter coefficients as either 'Fast transversal least-squares' or 'Sliding-window fast transversal least-squares'. For algorithms used to implement these two different methods, refer to [1]. This property is nontunable.

**Length — Length of filter coefficients vector**

32 (default) | positive integer

Specify the length of the FIR filter coefficients vector as a positive integer value.

Data Types: double

**SlidingWindowBlockLength — Width of sliding window**

32 (default) | positive integer

Specify the width of the sliding window as a positive integer value greater than or equal to the Length property value. The default value is the value of the Length property.

**Dependencies**

This property applies only if the Method property is set to 'Sliding-window fast transversal least-squares'.

Data Types: double

**ForgettingFactor — Fast transversal filter forgetting factor**

1 (default) | positive scalar

Specify the fast transversal filter forgetting factor as a positive scalar in the range (0,1]. Setting this value to 1 denotes infinite memory while filter adaptation. Setting this property value to 1 denotes infinite memory while adapting to find the new filter. For best results, set this property to a value that lies in the range  $[1 - 0.5/L, 1]$ , where L is the Length property value.

**Tunable:** Yes

**Dependencies**

This property applies only if the Method property is set to 'Fast transversal least-squares'.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical

**InitialPredictionErrorPower — Initial prediction error power**

10 (default) | positive scalar

Specify the initial value of the forward and backward prediction error vectors as a positive numeric scalar. This scalar should be sufficiently large to maintain stability and prevent an excessive number of Kalman gain rescues.

**Tunable:** Yes

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**InitialConversionFactor — Initial conversion factor (gamma)**

1 (default) | scalar | 2-element row vector

Specify the initial value of the conversion factor of the fast transversal filter.

The value of this property depends on the `Method` property. If `Method` is set to:

- `'Fast transversal least-squares'` -- This property must be a positive numeric value less than or equal to 1
- `'Sliding-window fast transversal least-squares'` -- This property must be a two-element numeric vector. The first element of this vector must lie within the range `[0, 1]`, and the second element must be less than or equal to `-1`. In this case, the default value is `[1, -1]`.

**Tunable:** Yes

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

### **InitialCoefficients** — Initial coefficients of filter

`0` (default) | `scalar` | `vector`

Specify the initial values of the FIR adaptive filter coefficients as a scalar or a vector of length equal to the value of the `Length` property.

**Tunable:** Yes

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **LockCoefficients** — Locked status of coefficient updates

`false` (default) | `true`

Specify whether to lock the filter coefficient values. By default, the value of this property is `false`, and the object continuously updates the filter coefficients. If this property is set to `true`, the filter coefficients do not update, and their values remain the same.

**Tunable:** Yes

## **Usage**

### **Syntax**

```
[y,err] = ftf(x,d)
```

### **Description**

`[y,err] = ftf(x,d)` filters the input `x`, using `d` as the desired signal, and returns the filtered output in `y` and the filter error in `err`. The System object estimates the filter weights needed to minimize the error between the output signal and the desired signal.

### **Input Arguments**

#### **x** — Data input

`scalar` | `column vector`

The signal to be filtered by the fast transversal filter. The input, `x`, and the desired signal, `d` must have the same size and data type.



The data input can be a variable-size signal. You can change the number of elements in the column vector even when the object is locked. The System object locks when you call the object to run its algorithm.

Data Types: `single` | `double`

### **d — Desired signal**

scalar | column vector

The fast transversal filter adapts its filter weights, `wts`, to minimize the error, `err`, and converge the input signal `x` to the desired signal `d` as closely as possible. You can access the current filter weights by calling `ftf.Coefficients`, where `ftf` is the fast transversal filter object.

The input and the desired signal must have the same size and data type.

The desired signal can be a variable-size signal. You can change the number of elements in the column vector even when the object is locked. The System object locks when you call the object to run its algorithm.

Data Types: `single` | `double`

### **Output Arguments**

#### **y — Filtered output**

scalar | column vector

Filtered output, returned as a scalar or a column vector. The object adapts its filter weights to converge the input signal `x` to match the desired signal `d`. The filter outputs the converged signal.

Data Types: `single` | `double`

#### **err — Difference between output and desired signal**

scalar | column vector

Difference between the output signal `y` and the desired signal `d`, returned as a scalar or a column vector. The data type of `err` matches the data type of `y`. The objective of the adaptive filter is to minimize this error. The object adapts its weights to converge toward optimal filter weights that produce an output signal that matches closely with the desired signal.

Data Types: `single` | `double`

### **Object Functions**

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

### **Specific to dsp.FastTransversalFilter**

`mseSim` Estimated mean squared error for adaptive filters

### **Common to All System Objects**

`step` Run System object algorithm

release Release resources and allow changes to System object property values and input characteristics  
reset Reset internal states of System object

## Examples

### System Identification of FIR Filter Using Fast Transversal Filter

System identification is the process of identifying the coefficients of an unknown system using an adaptive filter. The general overview of the process is shown in “System Identification -- Using an Adaptive Filter to Identify an Unknown System”. The main components involved are:

- The adaptive filter algorithm.
- An unknown system or process to adapt to. In this example, the filter designed by `fir1` is the unknown system.
- Appropriate input data to exercise the adaptation process. In a generic system identification model, the desired signal  $d(k)$  and the input signal  $x(k)$  are used to exercise the adaptation process.

The objective of the adaptive filter is to minimize the error signal between the output of the adaptive filter  $y(k)$  and the output of the unknown system (or the system to be identified)  $d(k)$ . Once the error signal is minimized, the adapted filter resembles the unknown system. The coefficients of both the filters match closely.

### Unknown System

Create a `dsp.FIRFilter` object that represents the system to be identified. Use the `fir1` function to design the filter coefficients. The designed filter is a 10th order lowpass digital filter with a cutoff frequency of 0.25.

```
filt = dsp.FIRFilter;  
filt.Numerator = fir1(10, .25)
```

```
filt =  
dsp.FIRFilter with properties:  
  
Structure: 'Direct form'  
NumeratorSource: 'Property'  
Numerator: [-0.0039 1.7585e-18 0.0321 0.1167 0.2207 0.2687 0.2207 0.1167 0.0321 1.7585e-18]  
InitialConditions: 0
```

Show all properties

Pass the signal  $x$  to the FIR filter. The desired signal  $d$  is the sum of the output of the unknown system (FIR filter) and an additive noise signal  $n$ .

```
x = randn(1000,1);  
d = filt(x) + 0.01*randn(1000,1);
```

### Adaptive Filter

With the unknown filter designed and the desired signal in place, create and apply the fast transversal filter object to identify the unknown filter.

Create a `dsp.FastTransversalFilter` object to represent an adaptive filter. Set the length of the adaptive filter to 11 taps and a forgetting factor of 0.99.

```
ftf1 = dsp.FastTransversalFilter(11,...
    'ForgettingFactor',0.99)

ftf1 =
    dsp.FastTransversalFilter with properties:

                Method: 'Fast transversal least-squares'
                Length: 11
        ForgettingFactor: 0.9900
    InitialPredictionErrorPower: 10
        InitialCoefficients: 0
    InitialConversionFactor: 1
        LockCoefficients: false
```

Pass the primary input signal  $x$  and the desired signal  $d$  to the fast transversal filter. Run the adaptive filter to determine the unknown system. The output  $y$  of the adaptive filter is the signal converged to the desired signal  $d$ , thereby minimizing the error  $e$  between the two signals.

```
[y,e] = ftf1(x,d);
w = ftf1.Coefficients
```

```
w = 1×11
```

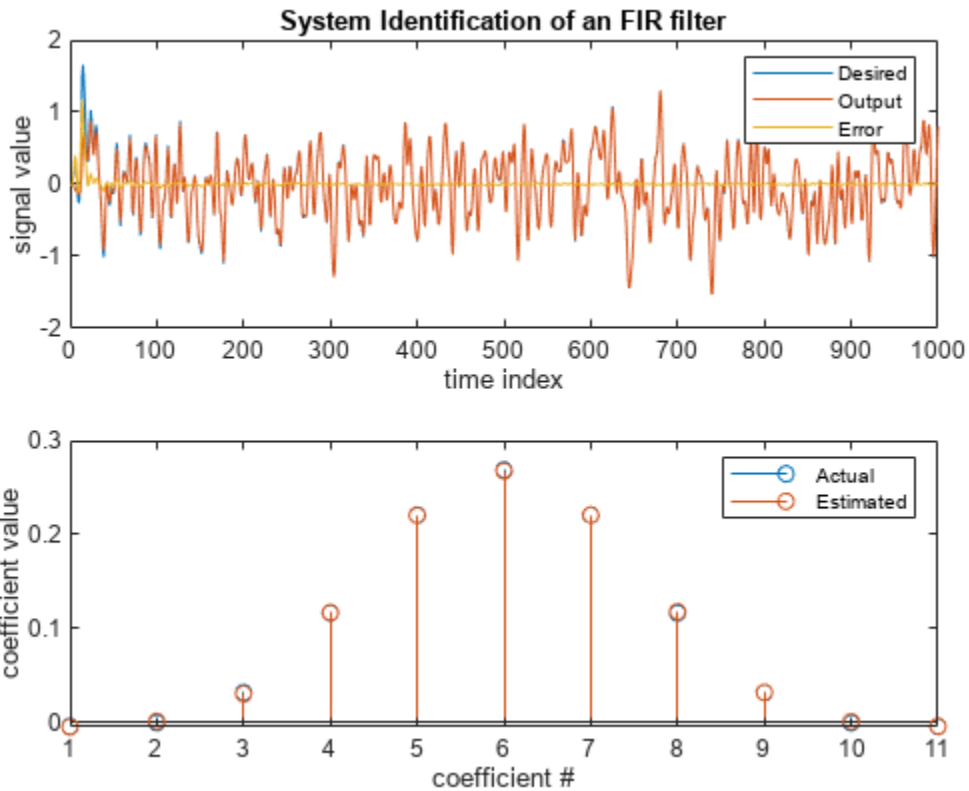
```
    -0.0043    0.0016    0.0308    0.1171    0.2204    0.2677    0.2210    0.1181    0.0323    0.0000
```

Plot the results. The output signal matches the desired signal very closely making the error between the two close to zero.

```
subplot(2,1,1);
plot(1:1000,[d,y,e])
title('System Identification of an FIR filter');
legend('Desired','Output','Error');
xlabel('time index');
ylabel('signal value');
```

The coefficients of the FIR filter match very closely with the coefficients of the adapted filter, thereby confirming the convergence.

```
subplot(2,1,2);
stem([filt.Numerator; w].');
legend('Actual','Estimated');
xlabel('coefficient #');
ylabel('coefficient value');
```



## Version History

Introduced in R2013b

## References

[1] Haykin, Simon. *Adaptive Filter Theory*, 4th Ed. Upper Saddle River, NJ: Prentice Hall, 2002.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

## See Also

### Objects

`dsp.LMSFilter` | `dsp.RLSFilter` | `dsp.AffineProjectionFilter` |  
`dsp.FrequencyDomainAdaptiveFilter` | `dsp.FilteredXLMSFilter` | `dsp.FIRFilter`

# dsp.FFT

**Package:** dsp

Discrete Fourier transform

## Description

The `dsp.FFT` System object computes the discrete Fourier transform (DFT) of an input using fast Fourier transform (FFT). The object uses one or more of the following fast Fourier transform (FFT) algorithms depending on the complexity of the input and whether the output is in linear or bit-reversed order:

- Double-signal algorithm
- Half-length algorithm
- Radix-2 decimation-in-time (DIT) algorithm
- Radix-2 decimation-in-frequency (DIF) algorithm
- An algorithm chosen from FFTW [1], [2]

To compute the DFT of an input:

- 1 Create the `dsp.FFT` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
ft = dsp.FFT
ft = dsp.FFT(Name, Value)
```

### Description

`ft = dsp.FFT` returns a FFT object that computes the discrete Fourier transform (DFT) of a real or complex  $N$ -D array input along the first dimension using fast Fourier transform (FFT).

`ft = dsp.FFT(Name, Value)` returns a FFT object with each specified property set to the specified value. Enclose each property name in single quotes. Unspecified properties have default values.

### Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

**FFTImplementation — FFT implementation**

Auto (default) | Radix-2 | FFTW

Specify the implementation used for the FFT as one of Auto, Radix-2, or FFTW. When you set this property to Radix-2, the FFT length must be a power of two.

**BitReversedOutput — Order of output elements relative to input elements**

false (default) | true

Designate order of output channel elements relative to order of input elements. Set this property to true to output the frequency indices in bit-reversed order. The default is false, which corresponds to a linear ordering of frequency indices.

**Normalize — Divide butterfly outputs by two**

false (default) | true

Set this property to true if the output of the FFT should be divided by the FFT length. This option is useful when you want the output of the FFT to stay in the same amplitude range as its input. This is particularly useful when working with fixed-point data types.

The default value of this property is false with no scaling.

**FFTLengthSource — Source of FFT length**

Auto (default) | Property

Specify how to determine the FFT length as Auto or Property. When you set this property to Auto, the FFT length equals the number of rows of the input signal.

**FFTLength — FFT length**

64 (default) | integer

FFT length, specified as an integer greater than or equal to 2.

This property must be a power of two if any of these conditions apply:

- The input is a fixed-point data type.
- The BitReversedOutput property is true.
- The FFTImplementation property is Radix-2.

**Dependencies**

This property applies when you set the FFTLengthSource property to Property.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**WrapInput — Boolean value of wrapping or truncating input**

true (default) | false

Wrap input data when FFT length is shorter than input length. If this property is set to true, modulo-length data wrapping occurs before the FFT operation, given FFT length is shorter than the input length. If this property is set to false, truncation of the input data to the FFT length occurs before the FFT operation.

## Fixed-Point Properties

### **RoundingMethod — Rounding method for fixed-point operations**

Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Specify the rounding method.

### **OverflowAction — Overflow action for fixed-point operations**

Wrap (default) | Saturate

Specify the overflow action as Wrap or Saturate.

### **SineTableDataType — Sine table word and fraction lengths**

Same word length as input (default) | Custom

Specify the sine table data type as Same word length as input or Custom.

### **CustomSineTableDataType — Sine table word and fraction lengths**

numericType([],16) (default) | numericType

Specify the sine table fixed-point type as an unscaled numericType object with a Signedness of Auto.

#### **Dependencies**

This property applies when you set the SineTableDataType property to Custom.

### **ProductDataType — Product word and fraction lengths**

Full precision (default) | Same as input | Custom

Specify the product data type as Full precision, Same as input, or Custom.

### **CustomProductDataType — Product word and fraction lengths**

numericType([],32,30) (default) | numericType

Specify the product fixed-point type as a scaled numericType object with a Signedness of Auto.

#### **Dependencies**

This property applies when you set the ProductDataType property to Custom.

### **AccumulatorDataType — Accumulator word and fraction lengths**

Full precision (default) | Same as input | Same as product | Custom

Specify the accumulator data type as Full precision, Same as input, Same as product, or Custom.

### **CustomAccumulatorDataType — Accumulator word and fraction lengths**

numericType([],32,30) (default) | numericType

Specify the accumulator fixed-point type as a scaled numericType object with a Signedness of Auto.

#### **Dependencies**

This property applies when you set the AccumulatorDataType property to Custom.

**OutputDataType — Output word and fraction lengths**

Full precision (default) | Same as input | Custom

Specify the output data type as one of Full precision, Same as input, Custom.

**CustomOutputDataType — Output word and fraction lengths**

numericType([],16,15) (default) | numericType

Specify the output fixed-point type as a scaled numericType object with a Signedness of Auto.

**Dependencies**

This property applies when you set the “OutputDataType” on page 4-0 property to Custom.

**Usage****Syntax** $y = ft(x)$ **Description**

$y = ft(x)$  computes the DFT,  $y$ , of the input  $x$  along the first dimension of  $x$ .

**Input Arguments****x — Time-domain input signal**vector | matrix |  $N$ -D array

Time-domain input signal, specified as a vector, matrix, or  $N$ -D array.

When the FFTLengthSource property is set to 'Auto', the length of  $x$  along the first dimension must be a positive integer power of two. This length is also the FFT length. When the FFTLengthSource property is 'Property', the value you specify in FFTLength property must be a positive integer power of two.

Variable-size input signals are only supported when the FFTLengthSource property is set to 'Auto'.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fi

**Output Arguments****y — Discrete Fourier transform of input signal**vector | matrix |  $N$ -D array

Discrete Fourier transform of input signal, returned as a vector, matrix, or an  $N$ -D array. When FFTLengthSource property is set to 'Auto', the FFT length is same as the number of rows in the input signal. When FFTLengthSource property is set to 'Property', the FFT length is specified through the FFTLength property.

To support non-power-of-two transform lengths with variable-size data, set the FFTImplementation property to 'FFTW'.



Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Common to All System Objects

|                      |  |
|----------------------|--|
| <code>step</code>    | Run System object algorithm  |
| <code>release</code> | Release resources and allow changes to System object property values and input characteristics |
| <code>reset</code>   | Reset internal states of System object   |

## Examples

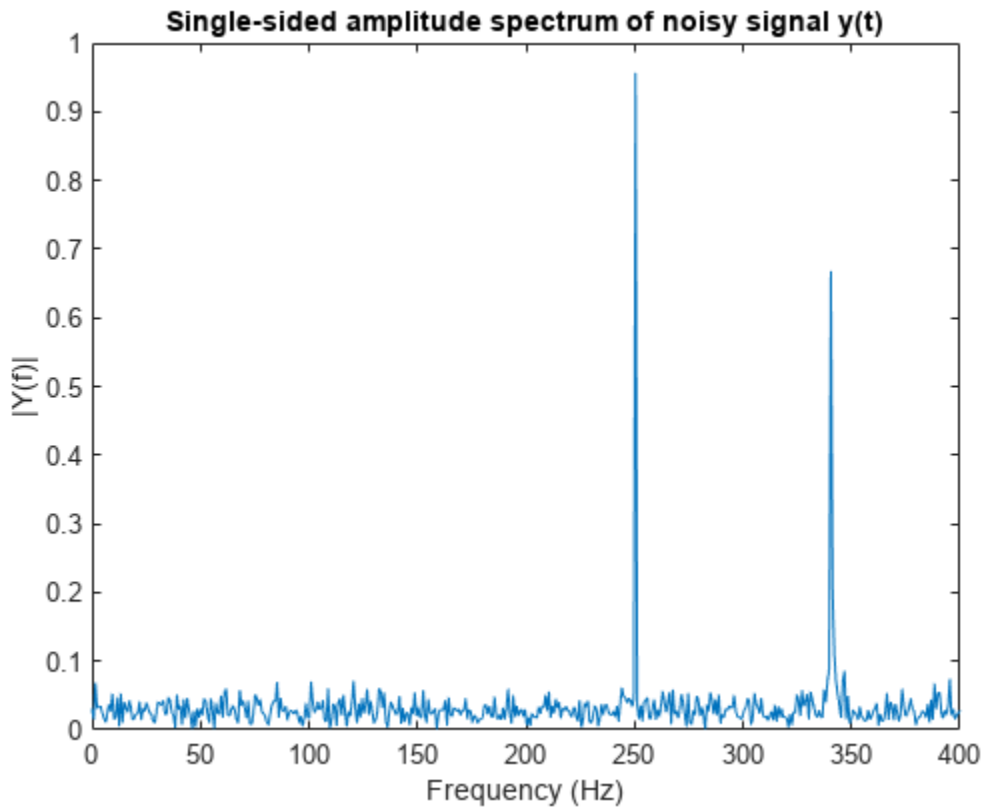
### Single-Sided Amplitude Spectrum of Signal

Find frequency components of a signal in additive noise.

```
Fs = 800; L = 1000;
t = (0:L-1)/Fs;
x = sin(2*pi*250*t) + 0.75*cos(2*pi*340*t);
y = x + .5*randn(size(x)); % noisy signal
ft = dsp.FFT('FFTLengthSource','Property', ...
    'FFTLength',1024);
Y = ft(y);
```

Plot the single-sided amplitude spectrum

```
plot(Fs/2*linspace(0,1,512), 2*abs(Y(1:512))/1024)
title('Single-sided amplitude spectrum of noisy signal y(t)')
xlabel('Frequency (Hz)'); ylabel('|Y(f)|')
```



### Construct a Sinusoidal Signal Using High Energy FFT Coefficients

Compute the FFT of a noisy sinusoidal input signal. The energy of the signal is stored as the magnitude square of the FFT coefficients. Determine the FFT coefficients which occupy 99.99% of the signal energy and reconstruct the time-domain signal by taking the IFFT of these coefficients. Compare the reconstructed signal with the original signal.

Consider a time-domain signal  $x[n]$ , which is defined over the finite time interval  $0 \leq n \leq N - 1$ . The energy of the signal  $x[n]$  is given by the following equation:

$$E_N = \sum_{n=0}^{N-1} |x[n]|^2$$

FFT coefficients,  $X[k]$ , are considered signal values in the frequency domain. The energy of the signal  $x[n]$  in the frequency-domain is therefore the sum of the squares of the magnitude of the FFT coefficients:

$$E_N = \frac{1}{N} \sum_{k=0}^{N-1} |X[k]|^2$$

According to Parseval's theorem, the total energy of the signal in time or frequency-domain is the same.

$$E_N = \sum_{n=0}^{N-1} |x[n]|^2 = \frac{1}{N} \sum_{k=0}^{N-1} |X[k]|^2$$

## Initialization

Initialize a `dsp.SineWave` System object to generate a sine wave sampled at 44.1 kHz and has a frequency of 1000 Hz. Construct a `dsp.FFT` and `dsp.IFFT` objects to compute the FFT and the IFFT of the input signal.

The `'FFTLengthSource'` property of each of these transform objects is set to `'Auto'`. The FFT length is hence considered as the input frame size. The input frame size in this example is 1020, which is not a power of 2, so select the `'FFTImplementation'` as `'FFTW'`.

```
L = 1020;
Sineobject = dsp.SineWave('SamplesPerFrame',L,...
    'PhaseOffset',10,...
    'SampleRate',44100,...
    'Frequency',1000);
ft = dsp.FFT('FFTImplementation','FFTW');
ift = dsp.IFFT('FFTImplementation','FFTW',...
    'ConjugateSymmetricInput',true);
rng(1);
```

## Streaming

Stream in the noisy input signal. Compute the FFT of each frame and determine the coefficients that constitute 99.99% energy of the signal. Take IFFT of these coefficients to reconstruct the time-domain signal.

```
numIter = 1000;
for Iter = 1:numIter
    Sinewave1 = Sineobject();
    Input = Sinewave1 + 0.01*randn(size(Sinewave1));
    FFTCoeff = ft(Input);
    FFTCoeffMagSq = abs(FFTCoeff).^2;

    EnergyFreqDomain = (1/L)*sum(FFTCoeffMagSq);
    [FFTCoeffSorted, ind] = sort((1/L)*FFTCoeffMagSq,...
        1, 'descend');

    CumFFTCoeffs = cumsum(FFTCoeffSorted);
    EnergyPercent = (CumFFTCoeffs/EnergyFreqDomain)*100;
    Vec = find(EnergyPercent > 99.99);
    FFTCoeffsModified = zeros(L,1);
    FFTCoeffsModified(ind(1:Vec(1))) = FFTCoeff(ind(1:Vec(1)));
    ReconstrSignal = ift(FFTCoeffsModified);
end
```

99.99% of the signal energy can be represented by the number of FFT coefficients given by `Vec(1)`:

```
Vec(1)
```

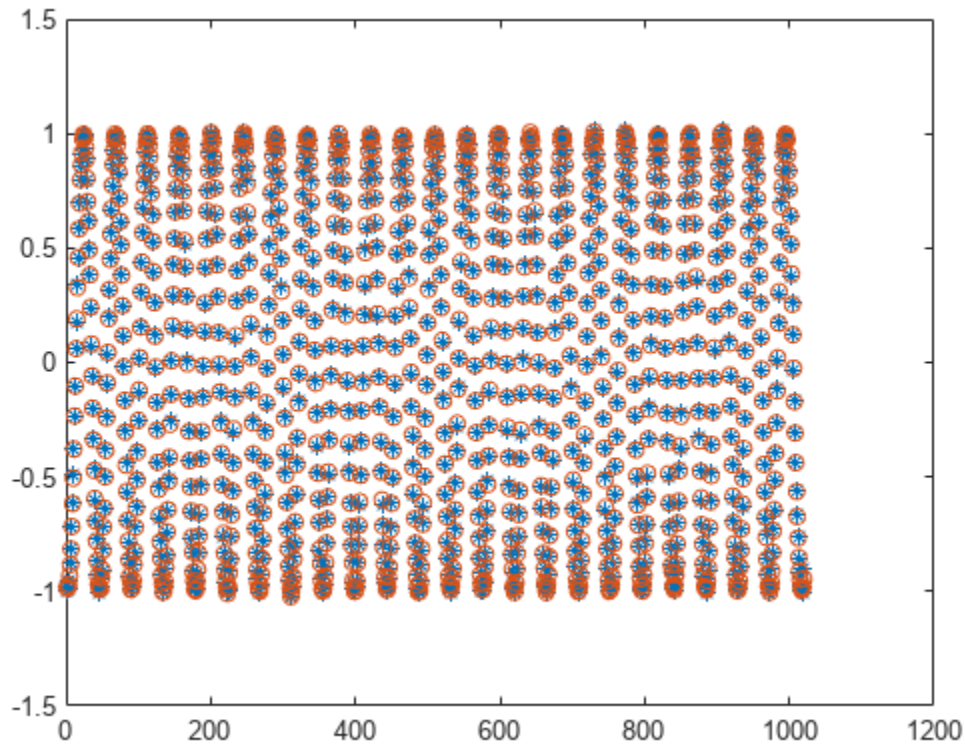
```
ans = 296
```

The signal is reconstructed efficiently using these coefficients. If you compare the last frame of the reconstructed signal with the original time-domain signal, you can see that the difference is very small and the plots match closely.

```
max(abs(Input-ReconstrSignal))
```

```
ans = 0.0431
```

```
plot(Input, '*');  
hold on;  
plot(ReconstrSignal, 'o');  
hold off;
```



## Algorithms

This object implements the algorithm, inputs, and outputs described on the FFT block reference page. The object properties correspond to the block parameters.

## Version History

Introduced in R2012a

## References

- [1] FFTW (<https://www.fftw.org>)
- [2] Frigo, M. and S. G. Johnson, "FFTW: An Adaptive Software Architecture for the FFT," *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Vol. 3, 1998, pp. 1381-1384.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- “System Objects in MATLAB Code Generation” (MATLAB Coder).
- When the following conditions apply, the executable generated from this System object relies on prebuilt dynamic library files (.dll files) included with MATLAB:
  - FFTImplementation is set to 'FFTW'.
  - FFTImplementation is set to 'Auto', FFTLengthSource is set to 'Property', and FFTLength is not a power of two.

Use the packNGo function to package the code generated from this System object and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see “How To Run a Generated Executable Outside MATLAB”.

- When the FFT length is a power of two, you can generate standalone C and C++ code from this System object.

## See Also

### Objects

dsp.IFFT

# **dsp.FilterCascade**

**Package:** dsp

Create cascade of filter System objects

## **Description**

The `dsp.FilterCascade` object creates a multistage System object that enables cascading of filter System objects, delays, and scalar gains. This object operates similar to the `cascade` function. However, the `cascade` function does not support delay as a filter stage.

You can pass the `dsp.FilterCascade` System object as a stage to another `dsp.FilterCascade` System object. You can also pass `dsp.FilterCascade` System object as an input to the `cascade` function.

When you call the object, the size, data type, and complexity of the input signal must be supported by all of the stages in the filter cascade. This object supports variable-size signals if the filter stages within the object support variable-size signals.

To filter a signal with a cascade of filters:

- 1 Create the `dsp.FilterCascade` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

Alternatively, you can generate a MATLAB function from the filter cascade object, and call that function to filter a signal. The generated function supports C/C++ code generation. For more details, see the `generateFilteringCode` function.

## **Creation**

### **Syntax**

```
FC = dsp.FilterCascade
FC = dsp.FilterCascade(filt1,filt2,...,filtn)
```

### **Description**

`FC = dsp.FilterCascade` returns a System object, `FC` that has a single stage, a `dsp.FIRFilter` System object with default properties.

`FC = dsp.FilterCascade(filt1,filt2,...,filtn)` returns a multistage System object, `FC`, with the first stage set to `filt1`, the second stage set to `filt2`, and so on. Each stage can be a filter System object, a `dsp.FilterCascade` System object, a `dsp.Delay` System object, or a scalar gain value.

For example, create a filter cascade that includes a lowpass filter, a highpass filter, and a gain stage.

```

lpFilt = dsp.LowpassFilter('StopbandFrequency',15000,...
                           'PassbandFrequency',12000);
hpFilt = dsp.HighpassFilter('StopbandFrequency',5000,...
                             'PassbandFrequency',8000);
gain = 2;
bpFilt = dsp.FilterCascade(lpFilt, hpFilt, 2);

```

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

### Stage*i* — Filter stage

`dsp.FIRFilter` System object with default properties (default) | filter System object | delay System object | scalar gain value

Filter stage, specified as a filter System object, delay System object, or a scalar gain value. To see which System objects you can add to a filter cascade, use:

```
dsp.FilterCascade.helpSupportedSystemObjects
```

You can modify an existing stage by modifying the associated property. For example:

```
FC = dsp.FilterCascade(dsp.FIRFilter,5)
```

```
FC =
```

```
  dsp.FilterCascade with properties:
```

```
  Stage1: [1×1 dsp.FIRFilter]
  Stage2: 5
```

```
K>> FC.Stage2 = dsp.FIRDecimator
```

```
FC =
```

```
  dsp.FilterCascade with properties:
```

```
  Stage1: [1×1 dsp.FIRFilter]
  Stage2: [1×1 dsp.FIRDecimator]
```

To change the number of stages in a cascade, use the `addStage` and `removeStage` functions.

## Usage

### Syntax

```
y = FC(x)
```

**Description**

$y = FC(x)$  filters input signal  $x$  by using the filter cascade defined in  $FC$  and returns filtered output  $y$ . The size, data type, and complexity of the input signal must be supported by all of the stages in the filter cascade. This object supports variable-size signals if the filter stages within the object support variable-size signals.

**Input Arguments****x — Data input**

vector | matrix

Data input, specified as a vector or a matrix. When the input is a matrix, each column of the matrix represents an independent data channel.

Data Types: `single` | `double`

**Output Arguments****y — Filtered output data**

vector | matrix

Filtered output data, returned as a vector or a matrix. The size, data type, and complexity of the output signal matches that of the input signal.

Data Types: `double` | `single`

**Object Functions**

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

**Specific to `dsp.FilterCascade`**

|                                    |   |
|------------------------------------|---|
| <code>addStage</code>              | Add filter stage to cascade                   |
| <code>generateFilteringCode</code> | Generate MATLAB code for a filter cascade     |
| <code>getNumStages</code>          | Get number of stages in filter cascade        |
| <code>releaseStages</code>         | Release locked state of all stages in cascade |
| <code>removeStage</code>           | Remove stage from filter cascade              |

**Common to All System Objects**

|                      |  |
|----------------------|--|
| <code>step</code>    | Run System object algorithm  |
| <code>release</code> | Release resources and allow changes to System object property values and input characteristics |
| <code>reset</code>   | Reset internal states of System object   |

**Examples****Filter Signal Using Cascaded Lowpass and Highpass Filters**

Design a bandpass filter by cascading:



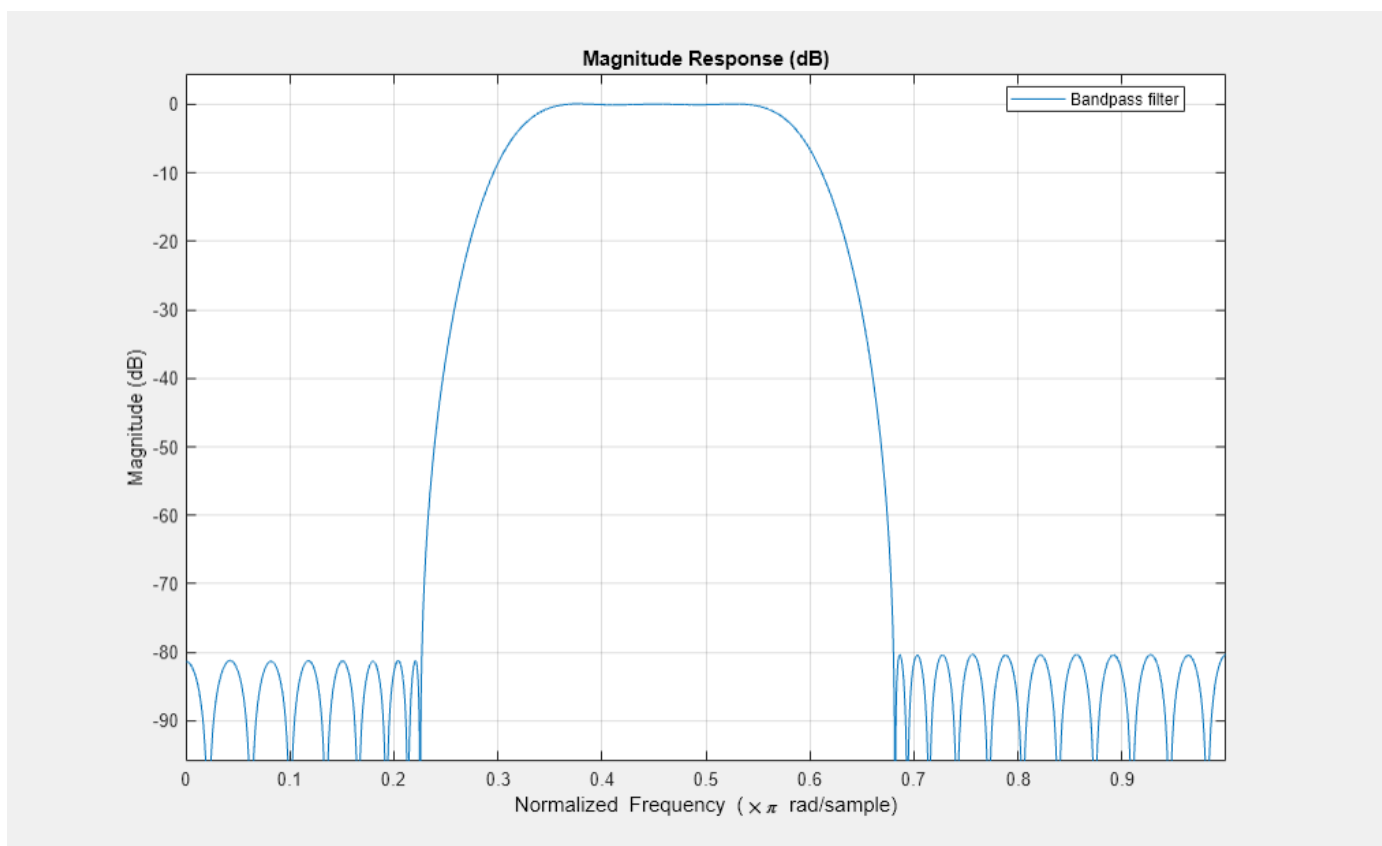
- A highpass filter with a stopband frequency of 5000 Hz and a passband frequency of 8000 Hz
- A lowpass filter with a passband frequency of 12,000 Hz and a stopband frequency of 15,000 Hz

Visualize the frequency response using `fvtool`.

```
lpFilt = dsp.LowpassFilter('StopbandFrequency',15000,...
    'PassbandFrequency',12000);
hpFilt = dsp.HighpassFilter('StopbandFrequency',5000,...
    'PassbandFrequency',8000);

bpFilt = dsp.FilterCascade(lpFilt,hpFilt);

hvft = fvtool(bpFilt);
legend(hvft,'Bandpass filter');
```



Pass a noisy sine wave as the input to the bandpass filter. The input is a sum of three sine waves with frequencies at 3 kHz, 10 kHz, and 15 kHz. The sampling frequency is 48 kHz. View the input and the filtered output on a spectrum analyzer.

The tones at 3 kHz and 15 kHz are attenuated, and the tone at 10 kHz is preserved by the bandpass filter.

```
Sine1 = dsp.SinWave('Frequency',3e3,...
    'SampleRate',48e3,...
    'SamplesPerFrame',6000);
Sine2 = dsp.SinWave('Frequency',10e3,...
    'SampleRate',48e3,...
```

```

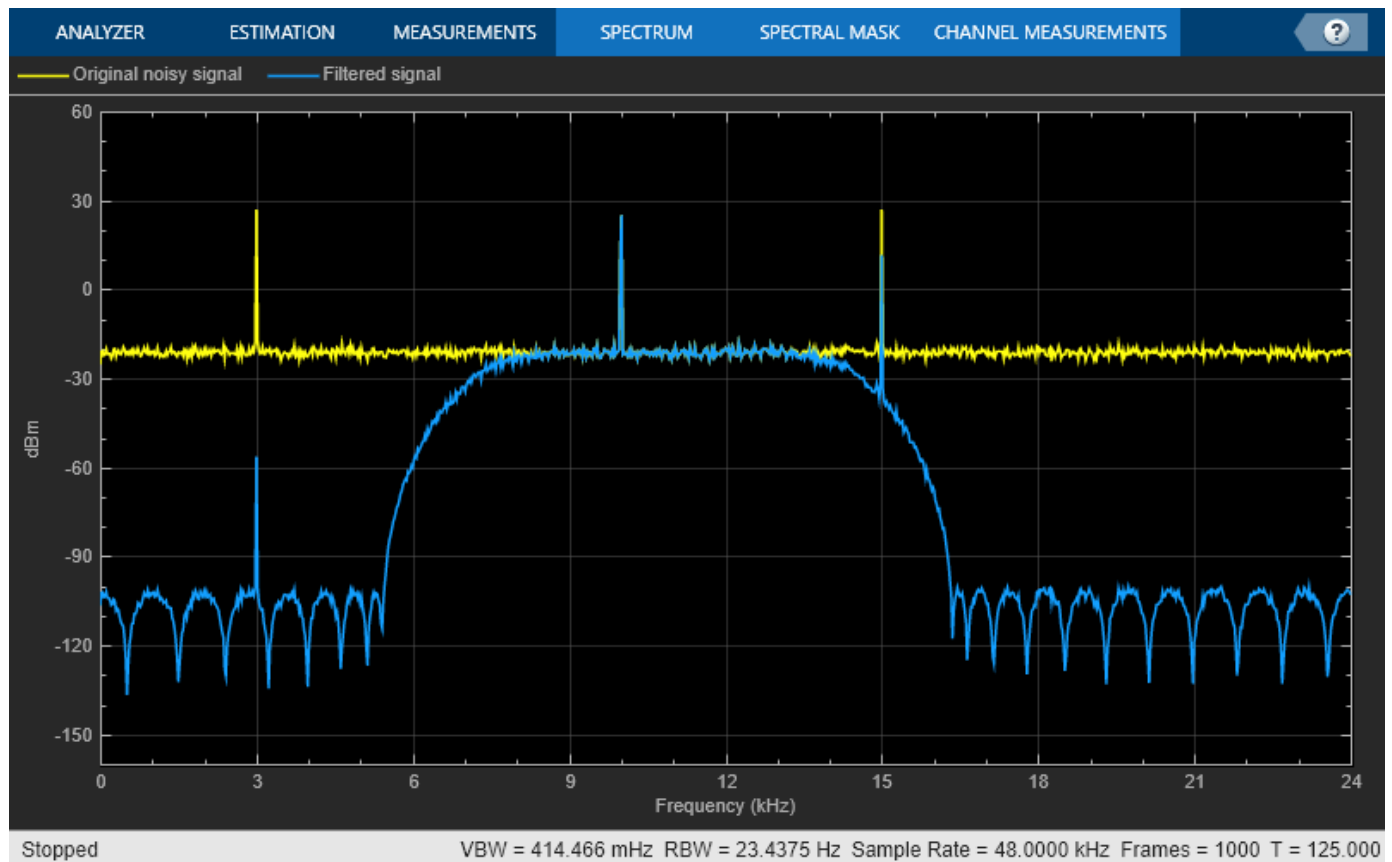
        'SamplesPerFrame',6000);
Sine3 = dsp.SineWave('Frequency',15e3,...
    'SampleRate',48e3,...
    'SamplesPerFrame',6000);

SpecAna = spectrumAnalyzer(...
    'PlotAsTwoSidedSpectrum',false,...
    'SampleRate',Sine1.SampleRate, ...
    'ShowLegend',true, ...
    'YLimits',[-160,60]);

SpecAna.ChannelNames = {'Original noisy signal','Filtered signal'};

for i = 1:1000
    x = Sine1()+Sine2()+Sine3()+0.1.*randn(Sine1.SamplesPerFrame,1);
    y = bpFilt(x);
    SpecAna(x,y);
end
release(SpecAna)

```



### Design Compensation Decimator for CIC Decimator

Create a CIC decimator. Cascade the decimator with a gain.

```

cicdecim = dsp.CICDecimator(...
    'DecimationFactor',6,...
    'NumSections',6);
decimcasc = dsp.FilterCascade(cicdecim,...
    1/gain(cicdecim));

```

Design a compensation decimator and cascade it with the filter cascade, `decimcasc`. This operation nests a `dsp.FilterCascade` object as a stage in another filter cascade. The CIC compensation decimator has an inherent gain, `gain(cicdecim)`. The factor of `1/gain(cicdecim)` from the decimation filter cascade, `decimcasc`, compensates for the compensation filter gain.

```

% Sample rate of input of compensation decimator
fs = 16e3;
% Passband frequency
fPass = 4e3;
% Stopband frequency
fStop = 4.5e3;
ciccomp = dsp.CICCompensationDecimator(cicdecim,...
    'DecimationFactor',2, ...
    'PassbandFrequency',fPass, ...
    'StopbandFrequency',fStop, ...
    'SampleRate',fs);
filtchain = dsp.FilterCascade(decimcasc,ciccomp);

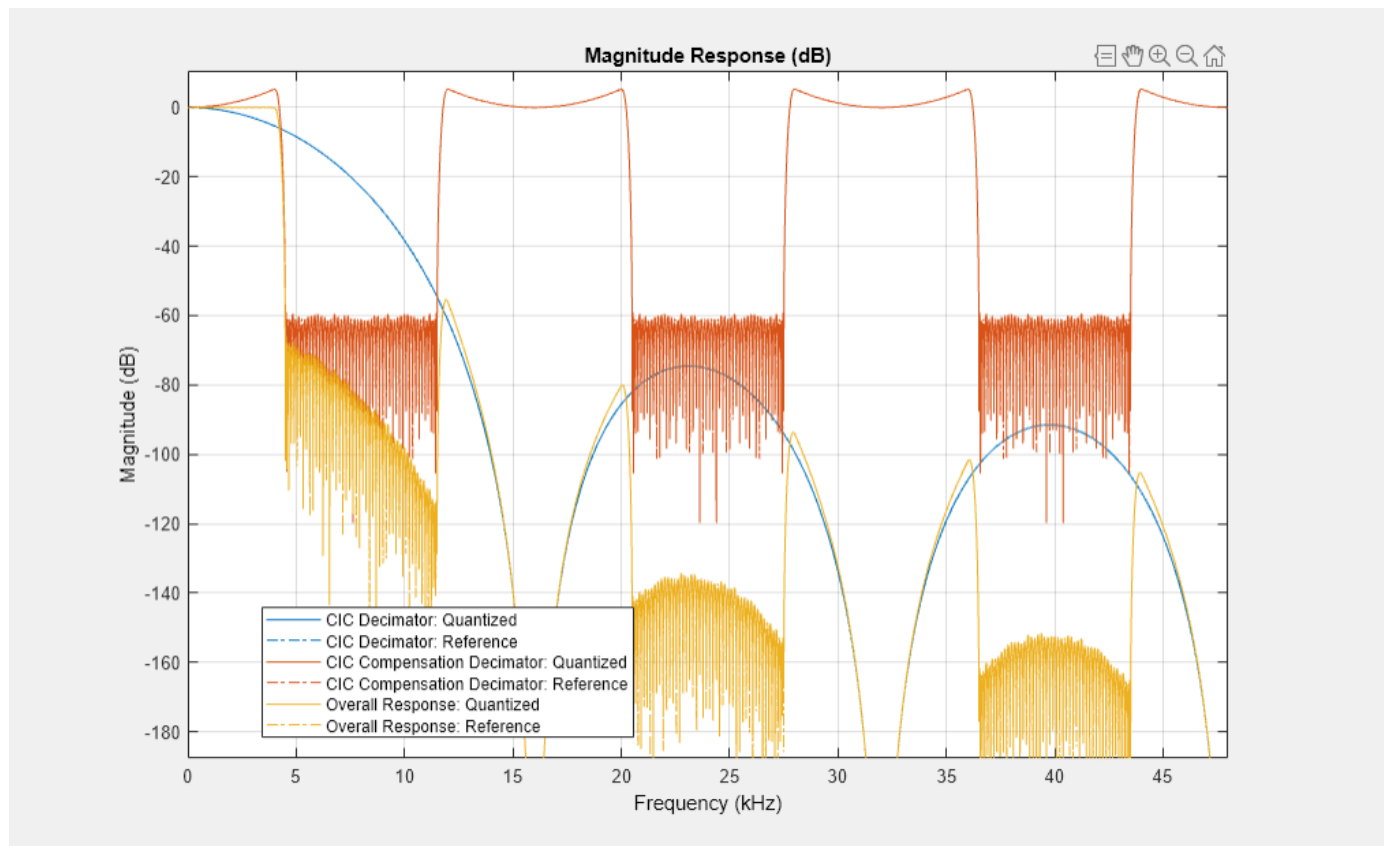
```

Visualize the frequency response of the cascade of cascades.

```

f = fvtool(decimcasc,ciccomp,...
    filtchain,'Fs',[fs*6,fs,fs*6],...
    'Arithmetic','fixed');
legend(f,'CIC Decimator',...
    'CIC Compensation Decimator',...
    'Overall Response');

```



### Generate Code to Filter Using Cascade

Design a two-stage decimator with a 100-Hz transition width, a 2-kHz sampling frequency, and 60-dB attenuation in the stopband. The decimator needs to downsample by a factor of 4.

```
filtCasc = designMultistageDecimator(4,2000,100,60)
```

```
filtCasc =
  dsp.FilterCascade with properties:
    Stage1: [1x1 dsp.FIRDecimator]
    Stage2: [1x1 dsp.FIRDecimator]
    CloneStages: false
```

Verify your design by using `fvtool`.

```
info(filtCasc)

ans =
  'Discrete-Time Filter Cascade
  -----
  Number of stages: 2
  Stage cloning: disabled
```

```
Stage1: dsp.FIRDecimator
-----
Discrete-Time FIR Multirate Filter (real)
-----
Filter Structure : Direct-Form FIR Polyphase Decimator
Decimation Factor : 2
Polyphase Length : 10
Filter Length : 19
Stable : Yes
Linear Phase : Yes (Type 1)

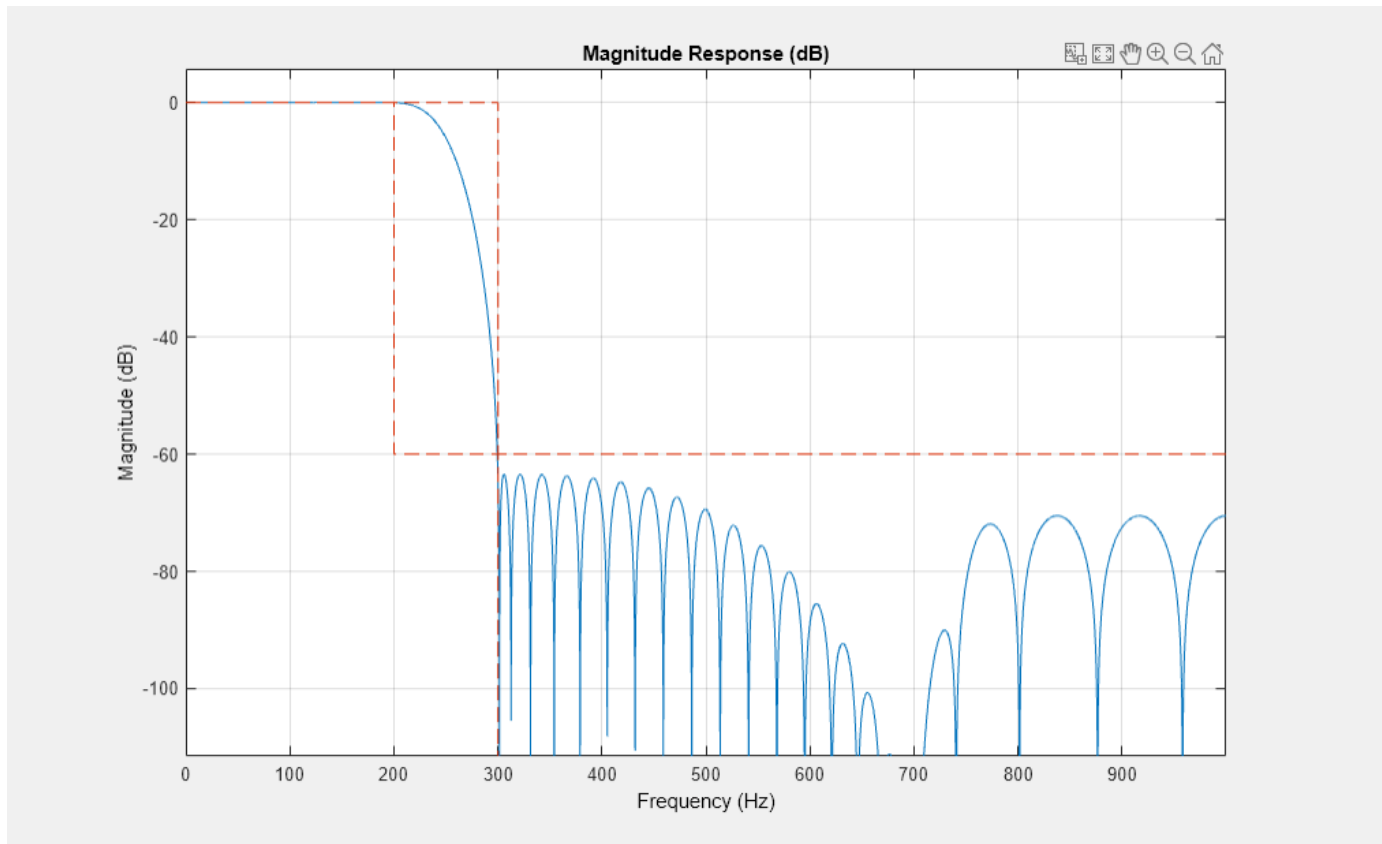
Arithmetic : double

Stage2: dsp.FIRDecimator
-----
Discrete-Time FIR Multirate Filter (real)
-----
Filter Structure : Direct-Form FIR Polyphase Decimator
Decimation Factor : 2
Polyphase Length : 18
Filter Length : 35
Stable : Yes
Linear Phase : Yes (Type 1)

Arithmetic : double

,
```

```
fvtool(filtCasc)
```



Generate code to filter data using this design. You cannot generate C/C++ code from the `dsp.FilterCascade` object directly, but you can generate C/C++ code from the generated function. The function defines the filter stages and calls them in sequence. The function is saved in a file called `myDecimator.m` in the current directory.

```
generateFilteringCode(filtCasc, 'myDecimator');
```

The `myDecimator` function creates a filter cascade and calls each stage object in turn.

```
type myDecimator
```

```
function y = myDecimator(x)
%MYDECIMATOR Construct filter cascade and process each stage

% MATLAB Code
% Generated by MATLAB(R) 9.14 and DSP System Toolbox 9.16.
% Generated on: 03-Mar-2023 11:31:17

% To generate C/C++ code from this function use the codegen command.
% Type 'help codegen' for more information.
%#codegen

%% Construction
persistent filter1 filter2
if isempty(filter1)
    filter1 = dsp.FIRDecimator( ...
        'Numerator', [0.0021878514650437138 0 -0.010189095418136306 0 0.031140395225498142 0 -0.0...
```

```
        filter2 = dsp.FIRDecimator( ...  
            'Numerator', [0.001155501175048853 0 -0.0027482166351234854 0 0.0057681982289523714 0 -0  
end  
  
%% Process  
y1 = filter1( x );  
y = filter2( y1);
```

## Version History

Introduced in R2014b

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- You cannot generate code directly from `dsp.FilterCascade`. If the filters in each stage support code generation, you can generate C/C++ code from the function returned by `generateFilteringCode`.
- See “System Objects in MATLAB Code Generation” (MATLAB Coder).

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

This object supports HDL code generation with the Filter Design HDL Coder product. For workflows and limitations, see “Generate HDL Code for Filter System Objects” (Filter Design HDL Coder).

## See Also

### Objects

`dsp.FIRFilter` | `dsp.Delay`

### Functions

`cascade`

## addStage

**Package:** dsp

Add filter stage to cascade

### Syntax

```
addStage(FC,newFilt)
addStage(FC,newFilt,stageId)
```

### Description

`addStage(FC,newFilt)` adds a filter stage represented by the `newFilt` System object as the final stage of the `dsp.FilterCascade` System object, `FC`. To see which System objects you can add to a filter cascade, use:

```
dsp.FilterCascade.helpSupportedSystemObjects
```

`addStage(FC,newFilt,stageId)` adds `newFilt` at stage position `stageId` of the filter cascade `FC`. All existing filters from `stageId` to the end of the cascade are shifted up in the cascade when `newFilt` is added.

### Examples

#### Add Filter Stage to Filter Cascade

Call `addStage` with only the System object™ arguments,. The function adds the new filter stage as the final stage in the filter cascade.

```
FC = dsp.FilterCascade
```

```
FC =
  dsp.FilterCascade with properties:
    Stage1: [1x1 dsp.FIRFilter]
    CloneStages: true
```

```
addStage(FC,dsp.IIRFilter)
FC
```

```
FC =
  dsp.FilterCascade with properties:
    Stage1: [1x1 dsp.FIRFilter]
    Stage2: [1x1 dsp.IIRFilter]
    CloneStages: true
```



Call `addStage` with an index argument. The function adds the new filter stage at the specified index. This example creates a filter cascade with three stages, and then adds a filter as the second stage of the cascade.

```
FC = dsp.FilterCascade(dsp.CICDecimator,dsp.FIRDecimator,dsp.FIRFilter)
```

```
FC =
  dsp.FilterCascade with properties:
    Stage1: [1x1 dsp.CICDecimator]
    Stage2: [1x1 dsp.FIRDecimator]
    Stage3: [1x1 dsp.FIRFilter]
    CloneStages: true
```

```
addStage(FC,dsp.IIRFilter,2)
```

```
FC
```

```
FC =
  dsp.FilterCascade with properties:
    Stage1: [1x1 dsp.CICDecimator]
    Stage2: [1x1 dsp.IIRFilter]
    Stage3: [1x1 dsp.FIRDecimator]
    Stage4: [1x1 dsp.FIRFilter]
    CloneStages: true
```

## Input Arguments

### **FC — Filter cascade**

`dsp.FilterCascade` System object

Filter cascade, specified as a `dsp.FilterCascade` System object.

### **newFilt — Filter stage to add**

filter System object | delay System object | scalar gain value

Filter stage to add, specified as one of the supported System objects or a scalar gain value. Each stage can be a filter or delay object, or a scalar gain value. To see which System objects you can add to a filter cascade, use:

```
dsp.FilterCascade.helpSupportedSystemObjects
```

### **stageId — Index of filter stage to be added**

positive integer

Index of filter stage to be added, specified as a positive integer. The object adds this stage to the filter cascade. All existing filters from `stageId` to the end of the cascade are shifted up in the cascade when the new filter is added.

## Version History

**Introduced in R2014b**

## **See Also**

### **Objects**

`dsp.FilterCascade`

### **Functions**

`generateFilteringCode` | `getNumStages` | `releaseStages` | `removeStage`

# generateFilteringCode

**Package:** dsp

Generate MATLAB code for a filter cascade

## Syntax

```
generateFilteringCode(FC)
generateFilteringCode(FC, fileName)
```

## Description

`generateFilteringCode(FC)` creates a MATLAB function that contains code to create the stages of a filter cascade, `FC`, and calls each stage in sequence. If the filters in each stage support code generation, you can generate C/C++ code from the function returned by `generateFilteringCode`.

`generateFilteringCode(FC, fileName)` generates code and saves the resulting function to the file specified in `fileName`.

## Examples

### Generate Code to Filter Using Cascade

Design a two-stage decimator with a 100-Hz transition width, a 2-kHz sampling frequency, and 60-dB attenuation in the stopband. The decimator needs to downsample by a factor of 4.

```
filtCasc = designMultistageDecimator(4,2000,100,60)
```

```
filtCasc =
    dsp.FilterCascade with properties:
        Stage1: [1x1 dsp.FIRDecimator]
        Stage2: [1x1 dsp.FIRDecimator]
        CloneStages: false
```

Verify your design by using `fvtool`.

```
info(filtCasc)

ans =
    'Discrete-Time Filter Cascade
    -----
    Number of stages: 2
    Stage cloning: disabled

    Stage1: dsp.FIRDecimator
    -----
    Discrete-Time FIR Multirate Filter (real)
    -----
    Filter Structure    : Direct-Form FIR Polyphase Decimator
```

```
Decimation Factor : 2
Polyphase Length : 10
Filter Length     : 19
Stable           : Yes
Linear Phase     : Yes (Type 1)
```

```
Arithmetic       : double
```

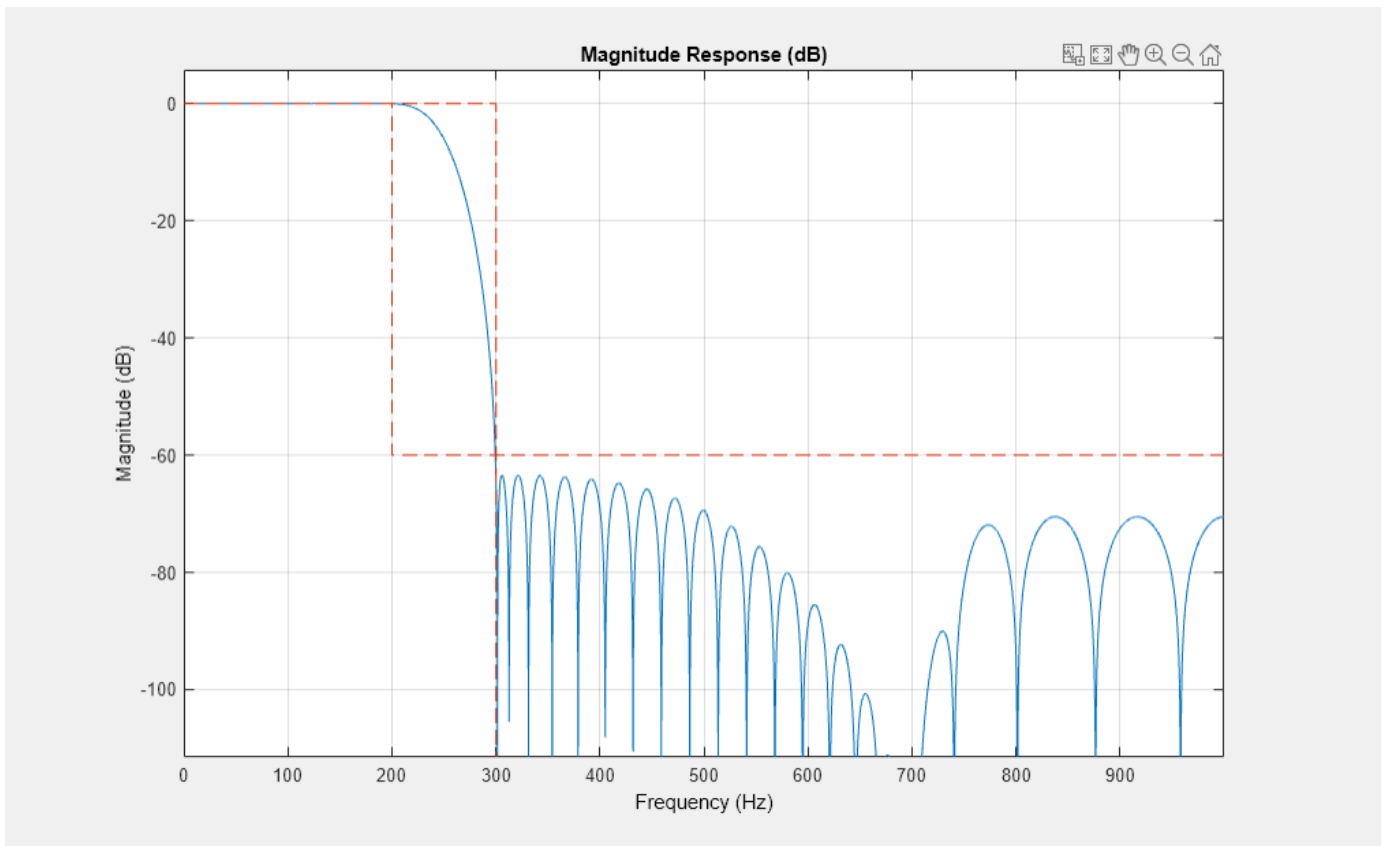
```
Stage2: dsp.FIRDecimator
```

```
-----
Discrete-Time FIR Multirate Filter (real)
-----
```

```
Filter Structure : Direct-Form FIR Polyphase Decimator
Decimation Factor : 2
Polyphase Length : 18
Filter Length     : 35
Stable           : Yes
Linear Phase     : Yes (Type 1)
```

```
Arithmetic       : double
```

```
fvtool(filtCasc)
```



Generate code to filter data using this design. You cannot generate C/C++ code from the `dsp.FilterCascade` object directly, but you can generate C/C++ code from the generated function. The function defines the filter stages and calls them in sequence. The function is saved in a file called `myDecimator.m` in the current directory.

```
generateFilteringCode(filtCasc, 'myDecimator');
```

The `myDecimator` function creates a filter cascade and calls each stage object in turn.

```
type myDecimator

function y = myDecimator(x)
%MYDECIMATOR Construct filter cascade and process each stage

% MATLAB Code
% Generated by MATLAB(R) 9.14 and DSP System Toolbox 9.16.
% Generated on: 03-Mar-2023 11:31:17

% To generate C/C++ code from this function use the codegen command.
% Type 'help codegen' for more information.
%#codegen

%% Construction
persistent filter1 filter2
if isempty(filter1)
    filter1 = dsp.FIRDecimator( ...
        'Numerator', [0.0021878514650437138 0 -0.010189095418136306 0 0.031140395225498142 0 -0.
    filter2 = dsp.FIRDecimator( ...
        'Numerator', [0.001155501175048853 0 -0.0027482166351234854 0 0.0057681982289523714 0 -0
end

%% Process
y1 = filter1( x );
y = filter2( y1);
```

## Input Arguments

### FC — Filter cascade

`dsp.FilterCascade` System object

Filter cascade, specified as a `dsp.FilterCascade` System object.

### fileName — File name

character vector | string scalar

File name where the generated function is saved, specified as a character vector or string scalar.

Data Types: `char` | `string`

## Version History

Introduced in R2014b

## **See Also**

### **Objects**

`dsp.FilterCascade`

### **Functions**

`addStage` | `getNumStages` | `releaseStages` | `removeStage`

# getNumStages

**Package:** dsp

Get number of stages in filter cascade

## Syntax

```
getNumStages(FC)
```

## Description

getNumStages(FC) returns the number of stages in the dsp.FilterCascade object, FC.

## Examples

### Find Number of Stages

This example shows how to query the number of stages in a filter cascade.

Create a filter cascade with two stages and call getNumStages on the cascade object.

```
FC = cascade(dsp.FIRFilter,dsp.IIRFilter)

FC =
  dsp.FilterCascade with properties:
    Stage1: [1x1 dsp.FIRFilter]
    Stage2: [1x1 dsp.IIRFilter]
    CloneStages: true

FCstages = getNumStages(FC)

FCstages = 2
```

## Input Arguments

### FC — Filter cascade

dsp.FilterCascade System object

Filter cascade, specified as a dsp.FilterCascade System object.

## Version History

**Introduced in R2014b**

## **See Also**

### **Objects**

`dsp.FilterCascade`

### **Functions**

`addStage` | `generateFilteringCode` | `releaseStages` | `removeStage`



# releaseStages

**Package:** dsp

Release locked state of all stages in cascade

## Syntax

```
releaseStages(FC)
```

## Description

releaseStages(FC) calls the release function of each individual stage in the dsp.FilterCascade System object FC.

For instance, if a dsp.FilterCascade object consists of a dsp.FIRFilter and a dsp.FIRInterpolator object, the releaseStages function calls the:

- release function of the dsp.FIRFilter object
- release function of the dsp.FIRInterpolator object

## Examples

### Release Stages of a Filter Cascade

Create and release stages of a filter cascade.

```
firfilt = dsp.FIRFilter;  
y = firfilt(randn);  
FC = dsp.FilterCascade(dsp.FIRInterpolator, firfilt);  
isLocked(FC.Stage2)
```

```
ans = logical  
     1
```

```
releaseStages(FC);  
isLocked(FC.Stage2)
```

```
ans = logical  
     0
```

## Input Arguments

### FC — Filter cascade

dsp.FilterCascade System object

Filter cascade, specified as a dsp.FilterCascade System object.

## **Version History**

**Introduced in R2014b**

### **See Also**

#### **Objects**

`dsp.FilterCascade`

#### **Functions**

`addStage` | `generateFilteringCode` | `getNumStages` | `removeStage`

# removeStage

**Package:** dsp

Remove stage from filter cascade

## Syntax

```
removeStage(FC)
removeStage(FC, stageId)
```

## Description

`removeStage(FC)` removes the final stage of the `dsp.FilterCascade` System object `FC`.

`removeStage(FC, stageId)` removes the stage from stage position `stageId` of the filter cascade `FC`. All existing filter stages from `stageId` to the end of the cascade are shifted down in the cascade when the stage is removed.

## Examples

### Remove Filter Stage

Call `removeStage` with no arguments other than the filter cascade System object™. The function removes the last stage in the filter cascade.

```
FC2= dsp.FilterCascade(dsp.FIRFilter,...
    dsp.IIRFilter)
```

```
FC2 =
    dsp.FilterCascade with properties:
```

```
    Stage1: [1x1 dsp.FIRFilter]
    Stage2: [1x1 dsp.IIRFilter]
    CloneStages: true
```

```
removeStage(FC2);
FC2
```

```
FC2 =
    dsp.FilterCascade with properties:
```

```
    Stage1: [1x1 dsp.FIRFilter]
    CloneStages: true
```

To remove a specific stage of the cascade, specify the index for that stage as an input argument. This example creates a filter cascade that has four stages, and then removes the third stage.

```
FC4 = cascade(dsp.FIRInterpolator,...
    dsp.FIRInterpolator,...
```

```
    dsp.FIRDecimator, ...
    dsp.FIRDecimator)

FC4 =
    dsp.FilterCascade with properties:

        Stage1: [1x1 dsp.FIRInterpolator]
        Stage2: [1x1 dsp.FIRInterpolator]
        Stage3: [1x1 dsp.FIRDecimator]
        Stage4: [1x1 dsp.FIRDecimator]
    CloneStages: true

removeStage(FC4,3);
FC4

FC4 =
    dsp.FilterCascade with properties:

        Stage1: [1x1 dsp.FIRInterpolator]
        Stage2: [1x1 dsp.FIRInterpolator]
        Stage3: [1x1 dsp.FIRDecimator]
    CloneStages: true
```

## Input Arguments

### FC — Filter cascade

`dsp.FilterCascade` System object

Filter cascade, specified as a `dsp.FilterCascade` System object.

### stageId — Index of filter stage to be removed

positive integer

Index of filter stage to be removed, specified as a positive integer. The function removes this stage from the filter cascade. All existing filters from `stageId` to the end of the cascade are shifted up in the cascade when the filter is removed.

## Version History

Introduced in R2014b

## See Also

### Objects

`dsp.FilterCascade`

### Functions

`addStage` | `generateFilteringCode` | `getNumStages` | `releaseStages`

# dsp.FilteredXLMSFilter

**Package:** dsp

Filtered XLMS filter

## Description

The `dsp.FilteredXLMSFilter` System object computes output, error and coefficients using filtered-x least mean square FIR adaptive filter.

To implement the adaptive FIR filter object:

- 1 Create the `dsp.FilteredXLMSFilter` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
fxlms = dsp.FilteredXLMSFilter
fxlms = dsp.FilteredXLMSFilter(len)
fxlms = dsp.FilteredXLMSFilter(Name,Value)
```

### Description

`fxlms = dsp.FilteredXLMSFilter` returns a filtered-x least mean square FIR adaptive filter System object, `fxlms`. This System object is used to compute the filtered output and the filter error for a given input and desired signal.

`fxlms = dsp.FilteredXLMSFilter(len)` returns a `FilteredXLMSFilter` System object, `fxlms`, with the `Length` property set to `len`.

`fxlms = dsp.FilteredXLMSFilter(Name,Value)` returns a `FilteredXLMSFilter` System object, `fxlms`, with each specified property set to the specified value. Enclose each property name in single quotes. Unspecified properties have default values.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

**Length — Length of filter coefficients vector**

10 (default) | positive integer

Specify the length of the FIR filter coefficients vector as a positive integer value. This property is nontunable.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**StepSize — Adaptation step size**

0.1 (default) | positive scalar

Specify the adaptation step size factor as a positive numeric scalar.

**Tunable:** Yes

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**LeakageFactor — Adaptation leakage factor**

1 (default) | scalar

Specify the leakage factor used in a leaky adaptive filter as a numeric value between 0 and 1, both inclusive. When the value is less than 1, the System object implements a leaky adaptive algorithm. The default value is 1, providing no leakage in the adapting method.

**Tunable:** Yes

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical

**SecondaryPathCoefficients — Coefficients of the secondary path filter model**

[0.0051 -3.2506e-18 -0.0419 1.3210e-17 0.2885 0.4968 0.2885 1.3210e-17 -0.0419 -3.2506e-18 0.0051] (default) | vector

Specify the coefficients of the secondary path filter model as a numeric vector. The secondary path connects the output actuator and the error sensor. The default value is a vector that represents the coefficients of a 10th-order FIR lowpass filter.

**Tunable:** Yes

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**SecondaryPathEstimate — Estimate of secondary path filter model**

[0.0051 -3.2506e-18 -0.0419 1.3210e-17 0.2885 0.4968 0.2885 1.3210e-17 -0.0419 -3.2506e-18 0.0051] (default) | vector

Specify the estimate of the secondary path filter model as a numeric vector. The secondary path connects the output actuator and the error sensor. The default value equals to the SecondaryPathCoefficients property value. This property is not tunable.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**InitialCoefficients — Initial coefficients of filter**

0 (default) | scalar | vector

Specify the initial values of the FIR adaptive filter coefficients as a scalar or a vector of length equal to the value of the Length property.

**Tunable:** Yes

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### LockCoefficients — Locked status of coefficient updates

`false` (default) | `true`

Specify whether to lock the filter coefficient values. By default, the value of this property is `false`, and the object continuously updates the filter coefficients. If this property is set to `true`, the filter coefficients do not update and their values remain the same.

**Tunable:** Yes

## Usage

## Syntax

```
[y,err] = fxlms(x,d)
```

### Description

`[y,err] = fxlms(x,d)` filters the input `x`, using `d` as the desired signal, and returns the filtered output `y` and the filter error `err`. The System object estimates the filter weights needed to minimize the error between the output signal and the desired signal. You can access these coefficients by accessing the `Coefficients` property of the object. This can be done only after calling the object. For example, to access the optimized coefficients of the `fxlms` filter, call `fxlms.Coefficients` after you pass the input and desired signal to the object.

### Input Arguments

#### **x** — Data input

scalar | column vector

The signal to be filtered by the filtered XLMS filter. The input, `x`, and the desired signal, `d`, must have the same size and data type.

The input, `x` can be a variable-size signal. You can change the number of elements in the column vector even when the object is locked. The System object locks when you call the object to run its algorithm.

Data Types: `single` | `double`

#### **d** — Desired signal

scalar | column vector

The filtered XLMS filter adapts its coefficients to minimize the error, `err`, and converge the input signal `x` to the desired signal `d` as closely as possible.

The input, `x`, and the desired signal, `d`, must have the same size and data type.

The desired signal, `d`, can be a variable-size signal. You can change the number of elements in the column vector even when the object is locked. The System object locks when you call the object to run its algorithm.

Data Types: `single` | `double`

Complex Number Support: Yes

## Output Arguments

### **y** — Filtered output

scalar | column vector

Filtered output, returned as a scalar or a column vector. The object adapts its filter coefficients to converge the input signal  $x$  to match the desired signal  $d$ . The filter outputs the converged signal.

Data Types: `single` | `double`

### **err** — Difference between output and desired signal

scalar | column vector

Difference between the output signal  $y$  and the desired signal  $d$ , returned as a scalar or a column vector. The objective of the filtered XLMS filter is to minimize this error. The object adapts its coefficients to converge towards optimal filter coefficients that produce an output signal that matches closely with the desired signal. To access the filtered XLMS filter coefficients, call `fxlms.Coefficients` after you pass the input and desired signal to the object.

Data Types: `single` | `double`

Complex Number Support: Yes

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Specific to `dsp.FilteredXLMSFilter`

`msesim` Estimated mean squared error for adaptive filters

## Common to All System Objects

`step` Run System object algorithm

`release` Release resources and allow changes to System object property values and input characteristics

`reset` Reset internal states of System object

## Examples

### Active Noise Control of Random Noise Signal

Generate noise, create FIR primary path system model, generate observation noise, filter the primary path system model output with added noise, and create FIR secondary path system model.

```
x = randn(1000,1);  
g = fir1(47,0.4);  
n = 0.1*randn(1000,1);  
d = filter(g,1,x) + n;  
b = fir1(31,0.5);
```

Use the `dsp.FilteredXLMSFilter` System object™ to compute the filtered output and the filter error for the input and the signal to be canceled.



```

mu = 0.008;
fxlms = dsp.FilteredXLMSFilter(32, 'StepSize', mu, 'LeakageFactor', ...
    1, 'SecondaryPathCoefficients', b);
[y,e] = fxlms(x,d);

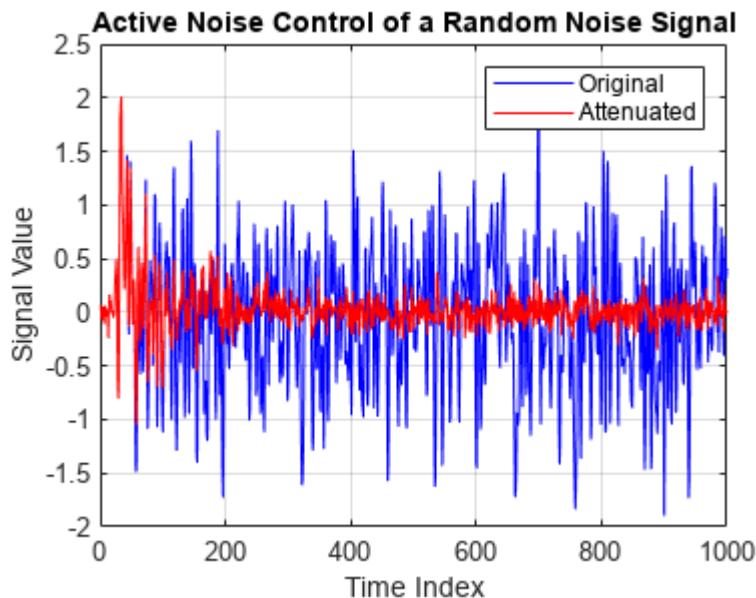
```

Plot the results.

```

plot(1:1000,d,'b',1:1000,e,'r');
title('Active Noise Control of a Random Noise Signal');
legend('Original','Attenuated');
xlabel('Time Index'); ylabel('Signal Value'); grid on;

```



### System Identification of FIR Filter Using Filtered XLMS Filter

Identify an unknown system by performing active noise control using a filtered-x LMS algorithm. The objective of the adaptive filter is to minimize the error signal between the output of the adaptive filter and the output of the unknown system (or the system to be identified). Once the error signal is minimal, the unknown system converges to the adaptive filter.

#### Initialization

Create a `dsp.FIRFilter` System object that represents the system to be identified. Pass the signal, `x`, to the FIR filter. The output of the unknown system is the desired signal, `d`, which is the sum of the output of the unknown system (FIR filter) and an additive noise signal, `n`.

```

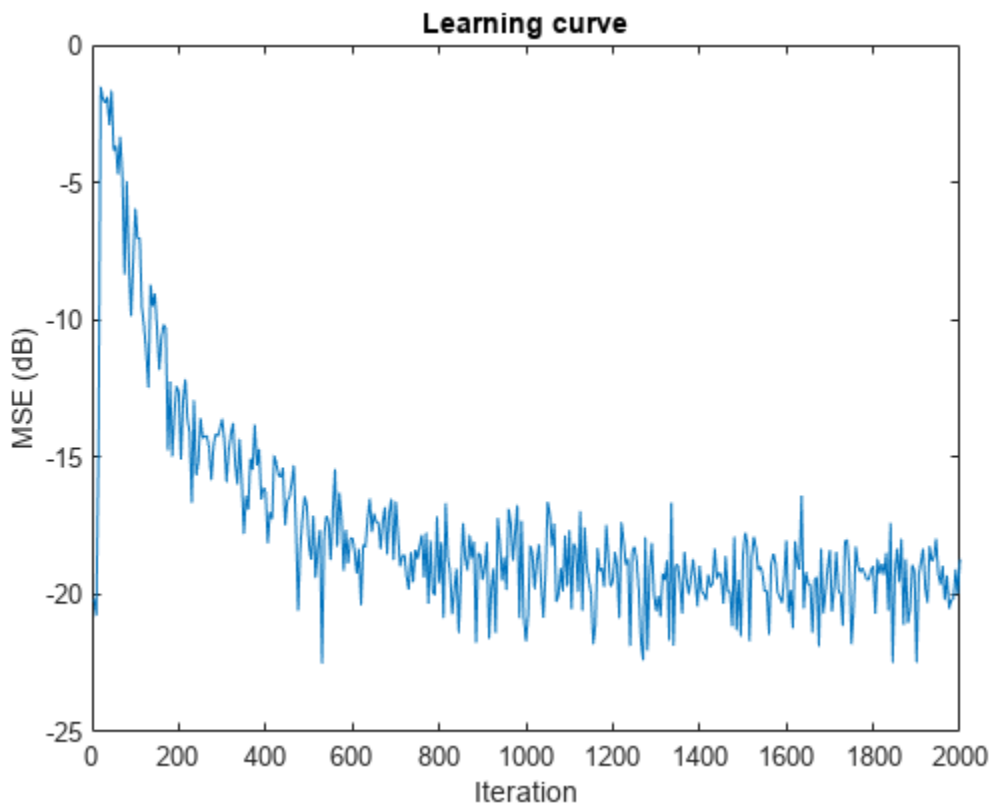
num = fir1(31,0.5);
fir = dsp.FIRFilter('Numerator',num);
iir = dsp.IIRFilter('Numerator',sqrt(0.75),...
    'Denominator',[1 -0.5]);
x = iir(sign(randn(2000,25)));
n = 0.1*randn(size(x));
d = fir(x) + n;

```

## Adaptive Filter

Create a `dsp.FilteredXLMSFilter` System object to create an adaptive filter that uses the filtered-x LMS algorithm. Set the length of the adaptive filter to 32 taps, step size to 0.008, and the decimation factor for analysis and simulation to 5. The variable `simmse` represents the error between the output of the unknown system, `d`, and the output of the adaptive filter.

```
l = 32;
mu = 0.008;
m = 5;
fxlms = dsp.FilteredXLMSFilter(l,'StepSize',mu);
[simmse,meanWsim,Wsim,traceKsim] = msesim(fxlms,x,d,m);
plot(m*(1:length(simmse)),10*log10(simmse))
xlabel('Iteration')
ylabel('MSE (dB)')
% Plot the learning curve for filtered-x LMS filter
% used in system identification
title('Learning curve')
```



With each iteration of adaptation, the value of `simmse` decreases to a minimal value, indicating that the unknown system has converged to the adaptive filter.

## Version History

Introduced in R2013b

## References

- [1] Kuo, S.M. and Morgan, D.R. *Active Noise Control Systems: Algorithms and DSP Implementations*. New York: John Wiley & Sons, 1996.
- [2] Widrow, B. and Stearns, S.D. *Adaptive Signal Processing*. Upper Saddle River, N.J: Prentice Hall, 1985.

## See Also

### Objects

`dsp.LMSFilter` | `dsp.RLSFilter` | `dsp.AffineProjectionFilter` |  
`dsp.AdaptiveLatticeFilter` | `dsp.FrequencyDomainAdaptiveFilter` | `dsp.FIRFilter`

## `dsp.FIRDecimator`

**Package:** `dsp`

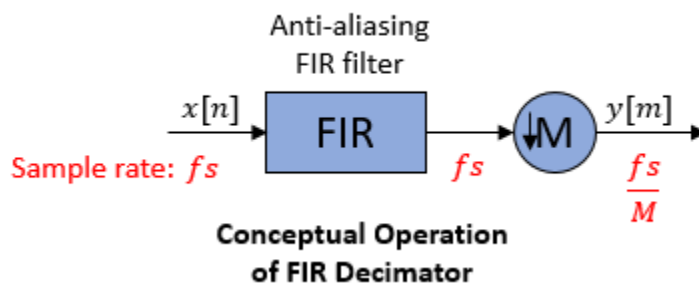
Perform polyphase FIR decimation

### Description

The `dsp.FIRDecimator` System object performs an efficient polyphase decimation using an integer downsampling factor  $M$  along the first dimension.

Conceptually, the FIR decimator (as shown in the schematic) consists of an anti-aliasing FIR filter followed by a downsampler.

The FIR filter filters the data in each channel of the input using a direct-form FIR filter. The FIR filter coefficients can be specified through the `Numerator` property, or can be automatically designed by the object using the `designMultirateFIR` function. The `designMultirateFIR` function designs an anti-aliasing FIR filter. The downsampler that follows the FIR filter downsamples each channel of filtered data by taking every  $M$ -th sample and discarding the  $M - 1$  samples that follow.  $M$  is the value of the decimation factor that you specify. The resulting discrete-time signal has a sample rate that is  $1/M$  times the original sample rate.



Note that the actual object algorithm implements a direct-form FIR polyphase structure, an efficient equivalent of the combined system depicted in the diagram. For more details, see "Algorithms" on page 4-500.

To resample vector or matrix inputs along the first dimension:

- 1 Create the `dsp.FIRDecimator` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

Under specific conditions, this System object also supports SIMD code generation. For details, see "Code Generation" on page 4-503.

## Creation

### Syntax

```

firdecim = dsp.FIRDecimator
firdecim = dsp.FIRDecimator(M)
firdecim = dsp.FIRDecimator(M,'Auto')
firdecim = dsp.FIRDecimator(M,num)
firdecim = dsp.FIRDecimator( ____,Name,Value)
firdecim = dsp.FIRDecimator(M,'legacy')

```

### Description

`firdecim = dsp.FIRDecimator` returns an FIR decimator object with a decimation factor of 2. The object designs the FIR filter coefficients using the `designMultirateFIR(1,2)` function.

`firdecim = dsp.FIRDecimator(M)` returns an FIR decimator with the integer-valued `DecimationFactor` property set to `M`. The object designs its filter coefficients based on the decimation factor `M` that you specify while creating the object, using the `designMultirateFIR(1,M)` function. The designed filter corresponds to a lowpass with a cutoff at  $\pi/M$  in radial frequency units.

`firdecim = dsp.FIRDecimator(M,'Auto')` returns an FIR decimator with the `NumeratorSource` property set to `'Auto'`. In this mode, every time there is an update in the decimation factor, the object redesigns the filter using `designMultirateFIR(1,M)`.

`firdecim = dsp.FIRDecimator(M,num)` returns an FIR decimator with the `DecimationFactor` property set to `M` and the `Numerator` property set to `num`.

`firdecim = dsp.FIRDecimator( ____,Name,Value)` returns an FIR decimator object with each specified property set to the specified value. Enclose each property name in quotes. You can use this syntax with any previous input argument combinations.

`firdecim = dsp.FIRDecimator(M,'legacy')` returns an FIR decimator where the filter coefficients are designed using `fir1(35,0.4)`. The designed filter has a cutoff frequency of  $0.4\pi$  radians/sample.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

### Main Properties

#### DecimationFactor — Decimation factor

2 (default) | positive integer

Decimation factor  $M$ , specified as a positive integer. The FIR decimator reduces the sampling rate of the input by this factor. The number of input rows must be a multiple of the decimation factor.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### NumeratorSource — FIR filter coefficient source

'Property' (default) | 'Input port' | 'Auto'

FIR filter coefficient source, specified as one of the following:

- 'Property' -- The numerator coefficients are specified through the `Numerator` property.
- 'Input port' -- The numerator coefficients are specified as an input to the object algorithm.
- 'Auto' -- The numerator coefficients are designed automatically using the `designMultirateFIR(1,M)` function.

### Numerator — FIR filter coefficients

`designMultirateFIR(1,M)` (default) | row vector

Numerator coefficients of the FIR filter, specified as a row vector in powers of  $z^{-1}$ . The following equation defines the system function for a filter of length  $N+1$ :

$$H(z) = \sum_{l=0}^N b_l z^{-l}$$

The vector  $\mathbf{b} = [b_0, b_1, \dots, b_N]$  represents the vector of filter coefficients.

To prevent aliasing as a result of downsampling, the filter transfer function should have a normalized cutoff frequency no greater than  $1/M$ . To design an effective anti-aliasing filter, use the `designMultirateFIR` function. For an example, see "Decimate Sum of Sine Waves" on page 4-493.

### Dependencies

This property is visible only when you set `NumeratorSource` to 'Property'.

When `NumeratorSource` is set to 'Auto', the numerator coefficients are automatically redesigned using `designMultirateFIR(1,M)`. To access the filter coefficients in the automatic design mode, type `objName.Numerator` in the MATLAB command prompt.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### Structure — Filter structure

'Direct form' (default) | 'Direct form transposed'

Specify the implementation of the FIR filter as either `Direct form` or `Direct form transposed`.

### Code Generation Properties

#### AllowArbitraryInputLength — Allow arbitrary input length in generated code

`false` (default) | `true`

Allow arbitrary frame length for fixed-size input signals in the generated code, specified as `true` or `false`. When you specify:

- `true` -- The input frame length does not have to be a multiple of the decimation factor. The output of the object in the generated code is a variable-size array.

- `false` -- The input frame length must be a multiple of the decimation factor.

When you specify variable-size signals, the input frame length can be arbitrary and the object ignores this property in the generated code. When you run this object in MATLAB, the object supports arbitrary input frame lengths for fixed-size and variable-size signals and this property does not affect the object behavior.

Data Types: `logical`

### Fixed-Point Properties

#### FullPrecisionOverride — Full-precision override for fixed-point arithmetic

`true` (default) | `false`

Flag to use full-precision rules for fixed-point arithmetic, specified as one of the following:

- `true` -- The object computes all internal arithmetic and output data types using the full-precision rules. These rules provide the most accurate fixed-point numerics. In this mode, other fixed-point properties do not apply. No quantization occurs within the object. Bits are added, as needed, to ensure that no roundoff or overflow occurs.
- `false` -- Fixed-point data types are controlled through individual fixed-point property settings.

For more information, see “Full Precision for Fixed-Point System Objects” and “Set System Object Fixed-Point Properties”.

#### RoundingMethod — Rounding method for fixed-point operations

'Floor' (default) | 'Ceiling' | 'Convergent' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'

Rounding method for fixed-point operations. For more details, see rounding mode.

#### Dependencies

This property is not visible and has no effect on the numerical results when the following conditions are met:

- `FullPrecisionOverride` set to `true`.
- `FullPrecisionOverride` set to `false`, `ProductDataType` set to 'Full precision', `AccumulatorDataType` set to 'Full precision', and `OutputDataType` set to 'Same as accumulator'.

Under these conditions, the object operates in full precision mode.

#### OverflowAction — Overflow action for fixed-point operations

'Wrap' (default) | 'Saturate'

Overflow action for fixed-point operations, specified as one of the following:

- 'Wrap' -- The object wraps the result of its fixed-point operations.
- 'Saturate' -- The object saturates the result of its fixed-point operations.

For more details on overflow actions, see overflow mode for fixed-point operations.

#### Dependencies

This property is not visible and has no effect on the numerical results when the following conditions are met:

- FullPrecisionOverride set to true.
- FullPrecisionOverride set to false, OutputDataType set to 'Same as accumulator', ProductDataType set to 'Full precision', and AccumulatorDataType set to 'Full precision'

Under these conditions, the object operates in full precision mode.

### **CoefficientsDataType — Data type of FIR filter coefficients**

Same word length as input (default) | Custom

Data type of the FIR filter coefficients, specified as:

- Same word length as input -- The word length of the coefficients is the same as that of the input. The fraction length is computed to give the best possible precision.
- Custom -- The coefficients data type is specified as a custom numeric type through the CustomCoefficientsDataType property.

### **CustomCoefficientsDataType — Word and fraction lengths of coefficients data type**

numerictype([],16,15) (default) | custom numeric type

Word and fraction lengths of the coefficients data type, specified as an autosigned numeric type with a word length of 16 and a fraction length of 15.

#### **Dependencies**

This property applies when you set the CoefficientsDataType property to Custom.

### **ProductDataType — Data type of product output**

'Full precision' (default) | 'Custom' | 'Same as input'

Data type of the product output in this object, specified as one of the following:

- 'Full precision' -- The product output data type has full precision.
- 'Same as input' -- The object specifies the product output data type to be the same as that of the input data type.
- 'Custom' -- The product output data type is specified as a custom numeric type through the CustomProductDataType property.

For more information on the product output data type, see “Multiplication Data Types”.

#### **Dependencies**

This property applies when you set FullPrecisionOverride to false.

### **CustomProductDataType — Word and fraction lengths of product data type**

numerictype([],32,30) (default) | custom numeric type

Word and fraction lengths of the product data type, specified as an autosigned numeric type with a word length of 32 and a fraction length of 30.

#### **Dependencies**

This property applies only when you set FullPrecisionOverride to false and ProductDataType to 'Custom'.



**AccumulatorDataType — Data type of accumulation operation**

'Full precision' (default) | 'Same as input' | 'Same as product' | 'Custom'

Data type of an accumulation operation in this object, specified as one of the following:

- 'Full precision' -- The accumulation operation has full precision.
- 'Same as product' -- The object specifies the accumulator data type to be the same as that of the product output data type.
- 'Same as input' -- The object specifies the accumulator data type to be the same as that of the input data type.
- 'Custom' -- The accumulator data type is specified as a custom numeric type through the CustomAccumulatorDataType property.

**Dependencies**

This property applies when you set FullPrecisionOverride to false.

**CustomAccumulatorDataType — Word and fraction lengths of accumulator data type**

numericType([],32,30) (default) | custom numeric type

Word and fraction lengths of the accumulator data type, specified as an autosigned numeric type with a word length of 32 and a fraction length of 30.

**Dependencies**

This property applies only when you set FullPrecisionOverride to false and AccumulatorDataType to 'Custom'.

**OutputDataType — Data type of object output**

'Same as accumulator' (default) | 'Same as input' | 'Same as product' | 'Custom'

Data type of the object output, specified as one of the following:

- 'Same as accumulator' -- The output data type is the same as that of the accumulator output data type.
- 'Same as input' -- The output data type is the same as that of the input data type.
- 'Same as product' -- The output data type is the same as that of the product output data type.
- 'Custom' -- The output data type is specified as a custom numeric type through the CustomOutputDataType property.

**Dependencies**

This property applies when you set FullPrecisionOverride to false.

**CustomOutputDataType — Word and fraction lengths of output data type**

numericType([],16,15) (default) | custom numeric type

Word and fraction lengths of the output data type, specified as an autosigned numeric type with a word length of 16 and a fraction length of 15.

**Dependencies**

This property applies only when you set FullPrecisionOverride to false and OutputDataType to 'Custom'.

## Usage

### Syntax

```
y = firdecim(x)
y = firdecim(x,num)
```

### Description

`y = firdecim(x)` outputs the filtered and downsampled values, `y`, of the input signal, `x`.

`y = firdecim(x,num)` uses the FIR filter, `num`, to decimate the input signal. This configuration is valid only when the 'NumeratorSource' property is set to 'Input port'.

### Input Arguments

#### **x** — Data input

column vector | matrix

Data input, specified as a column vector or a matrix of size  $P$ -by- $Q$ . The columns in the input signal represent  $Q$  independent channels.

Under most conditions, the number of inputs rows  $P$  can be arbitrary and does not have to be a multiple of the `DecimationFactor` property. See this table for details.

| Input Signal  | When you Run Object in MATLAB                | When you Generate Code Using MATLAB Coder   |
|---------------|--|---|
| Fixed-size    | Object supports arbitrary input frame length | Object supports arbitrary input frame length when you set <code>AllowArbitraryInputLength</code> to <code>true</code> while generating code |
| Variable-size | Object supports arbitrary input frame length | Object supports arbitrary input frame length  |

Variable-size signals change in frame length once you lock the object while the fixed-size signals remain constant. When the object does not support arbitrary frame length, the input frame length must be a multiple of the `DecimationFactor` property.

This object does not support complex unsigned fixed-point inputs.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

#### **num** — FIR filter coefficients

row vector

FIR filter coefficients, specified as a row vector.

### Dependencies

This input is accepted only when the 'NumeratorSource' property is set to 'Input port'.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

## Output Arguments

### y — FIR decimator output

column vector | matrix

FIR decimator output, returned as a column vector or a matrix. When the input is of size  $P$ -by- $Q$ , and  $P$  is not a multiple of the decimation factor  $M$ , the output signal has an upper bound size of  $\text{ceil}(P/M)$ -by- $Q$ . If  $P$  is a multiple of the decimation factor, then the output is of size  $(P/M)$ -by- $Q$ .

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fi

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Specific to dsp.FIRDecimator

|             |   |
|-------------|---|
| freqz       | Frequency response of discrete-time filter System object                      |
| fvtool      | Visualize frequency response of DSP filters                                   |
| info        | Information about filter System object  |
| cost        | Estimate cost of implementing filter System object                            |
| polyphase   | Polyphase decomposition of multirate filter                                   |
| generatehdl | Generate HDL code for quantized DSP filter (requires Filter Design HDL Coder) |
| impz        | Impulse response of discrete-time filter System object                        |
| coeffs      | Returns the filter System object coefficients in a structure                  |

## Common to All System Objects

|         |  |
|---------|--|
| step    | Run System object algorithm  |
| release | Release resources and allow changes to System object property values and input characteristics |
| reset   | Reset internal states of System object   |

## Examples

### Decimate Sum of Sine Waves

Decimate a sum of sine waves by a factor of 2 and by a factor of 4.

Start with a cosine wave that has an angular frequency of  $\frac{\pi}{4}$  radians/sample.

```
x = cos(pi/4*(0:95)');
```

### Design Default Filter

Create a `dsp.FIRDecimator` object. To prevent aliasing, the object uses an anti-aliasing lowpass filter before downsampling. By default, the anti-aliasing lowpass filter is designed using the `designMultirateFIR` function. The function designs the filter based on the decimation factor that

you specify, and stores the coefficients in the Numerator property. For a decimation factor of 2, the object designs the coefficients using `designMultirateFIR(1,2)`.

```
firdecim = dsp.FIRDecimator(2)
```

```
firdecim =  
dsp.FIRDecimator with properties:
```

```
Main
```

```
DecimationFactor: 2
```

```
NumeratorSource: 'Property'
```

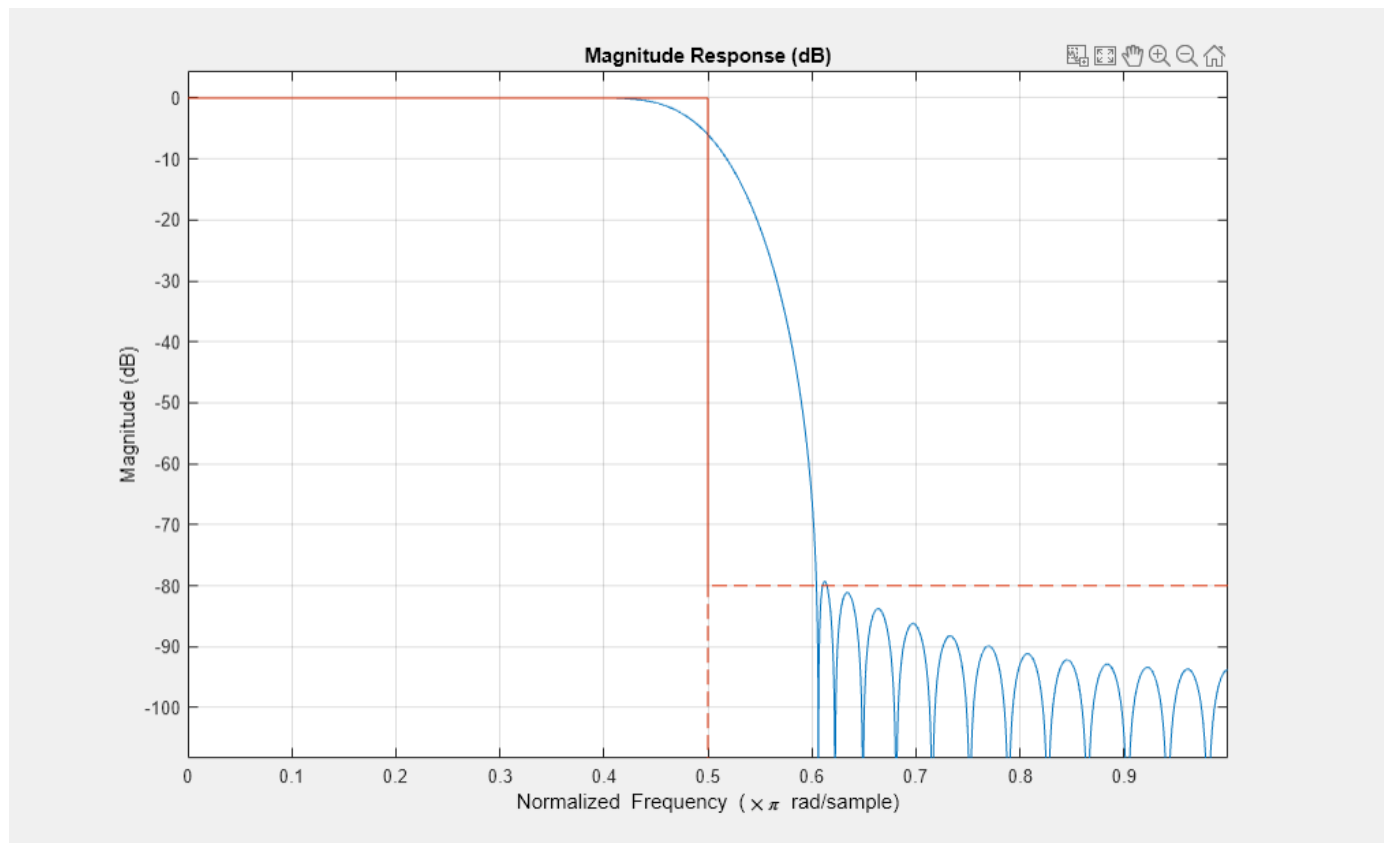
```
Numerator: [0 -1.0054e-04 0 3.8704e-04 0 -0.0010 0 0.0022 0 -0.0043 0 0.0077 0 -0.012
```

```
Structure: 'Direct form'
```

```
Show all properties
```

Visualize the filter response using `fvtool`. The designed filter meets the ideal filter constraints that are marked in red. The cutoff frequency is approximately half the spectrum.

```
fvtool(firdecim)
```



### Decimate by 2

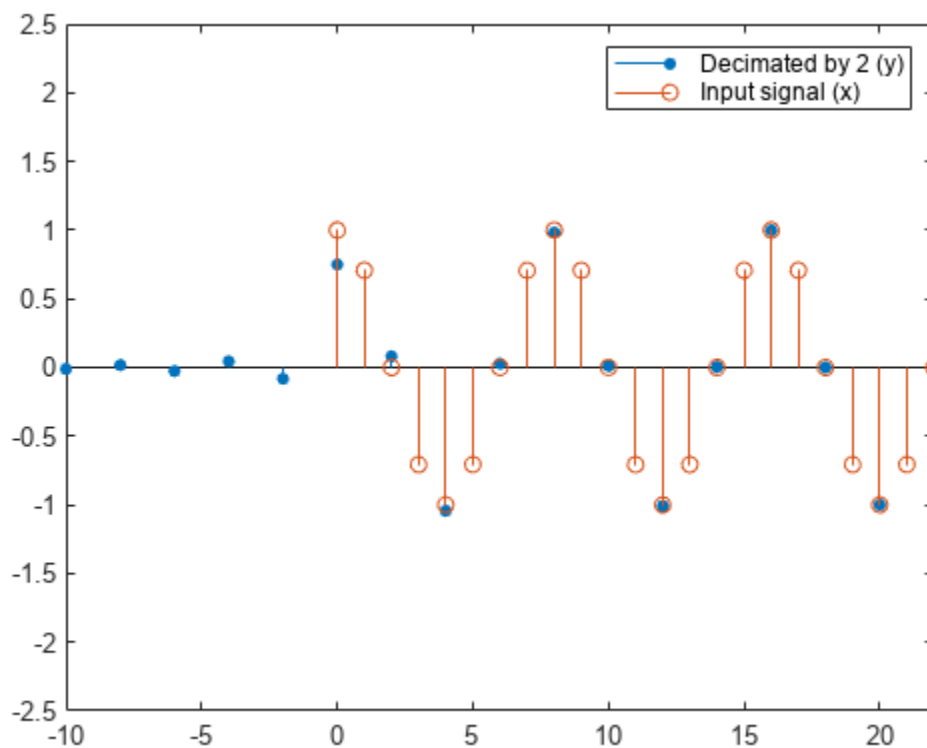
Decimate the cosine signal by a factor of 2.

```
y = firdecim(x);
```

Plot the original and the decimated signals. In order to plot the two signals on the same plot, you must account for the output delay of the FIR decimator and the scaling introduced by the filter. Use the `outputDelay` function to compute the delay value introduced by the decimator. Shift the output by this delay value.

Visualize the input and the resampled signals. After a short transition, the output converges to a cosine of frequency  $\frac{\pi}{2}$  as expected, which is twice the frequency of the input signal  $\frac{\pi}{4}$ . Due to the decimation factor of 2, the output samples coincide with every other input sample.

```
[delay,FsOut] = outputDelay(firdecim);
nx = (0:length(x)-1);
ty = (0:length(y)-1)/FsOut-delay;
stem(ty,y,'filled',MarkerSize=4); hold on;
stem(nx,x); hold off;
xlim([-10,22])
ylim([-2.5 2.5])
legend('Decimated by 2 (y)','Input signal (x)');
```



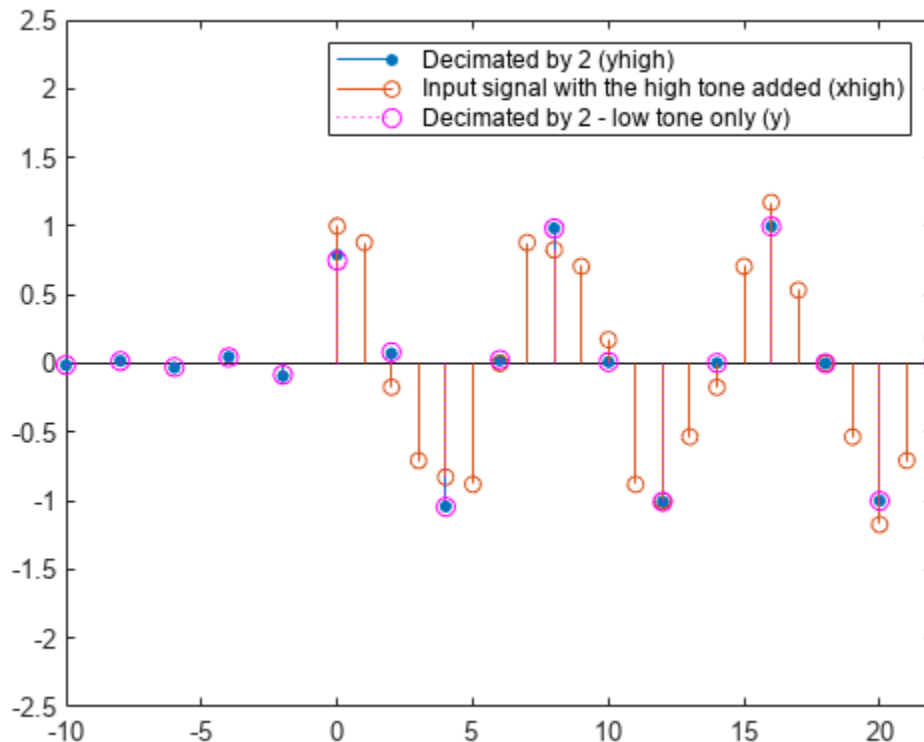
### Add a High Frequency Component to Input and Decimate

Add another frequency component to the input signal, a sine with an angular frequency of  $\frac{2\pi}{3}$  radians/sample. Since  $\omega = \frac{2\pi}{3}$  is above the FIR lowpass cutoff,  $\frac{\pi}{2}$ , the frequency  $\frac{2\pi}{3}$  radians/sample is filtered out from the signal.

```
xhigh = x + 0.2*sin(2*pi/3*(0:95));
release(firdecim)
yhigh = firdecim(xhigh);
```

Plot the input signal, decimated signal, and the output of the low frequency component. The decimated signal `yhigh` has the high frequency component filtered out. `yhigh` is almost identical to the output of the low frequency component `y`.

```
stem(ty,yhigh,'filled',MarkerSize=4); hold on;
stem(nx,xhigh);
stem(ty,y,'m',MarkerSize=7);
hold off;
xlim([-10,22])
ylim([-2.5 2.5])
legend('Decimated by 2 (yhigh)',...
       'Input signal with the high tone added (xhigh)',...
       'Decimated by 2 - low tone only (y)');
```



### Decimate by 4 in Automatic Filter Design Mode

Now decimate by a factor of 4. In order for the filter design to be updated automatically based on the new decimation factor, set the `NumeratorSource` property to `'Auto'`. Alternately, you can pass `'Auto'` as the keyword while creating the object. The object then operates in the automatic filter design mode. Every time there is a change in the decimation factor, the object updates the filter design.

```

release(firdecim)
firdecim.NumeratorSource = 'Auto';
firdecim.DecimationFactor = 4

firdecim =
    dsp.FIRDecimator with properties:

    Main
        DecimationFactor: 4
        NumeratorSource: 'Auto'
        Structure: 'Direct form'

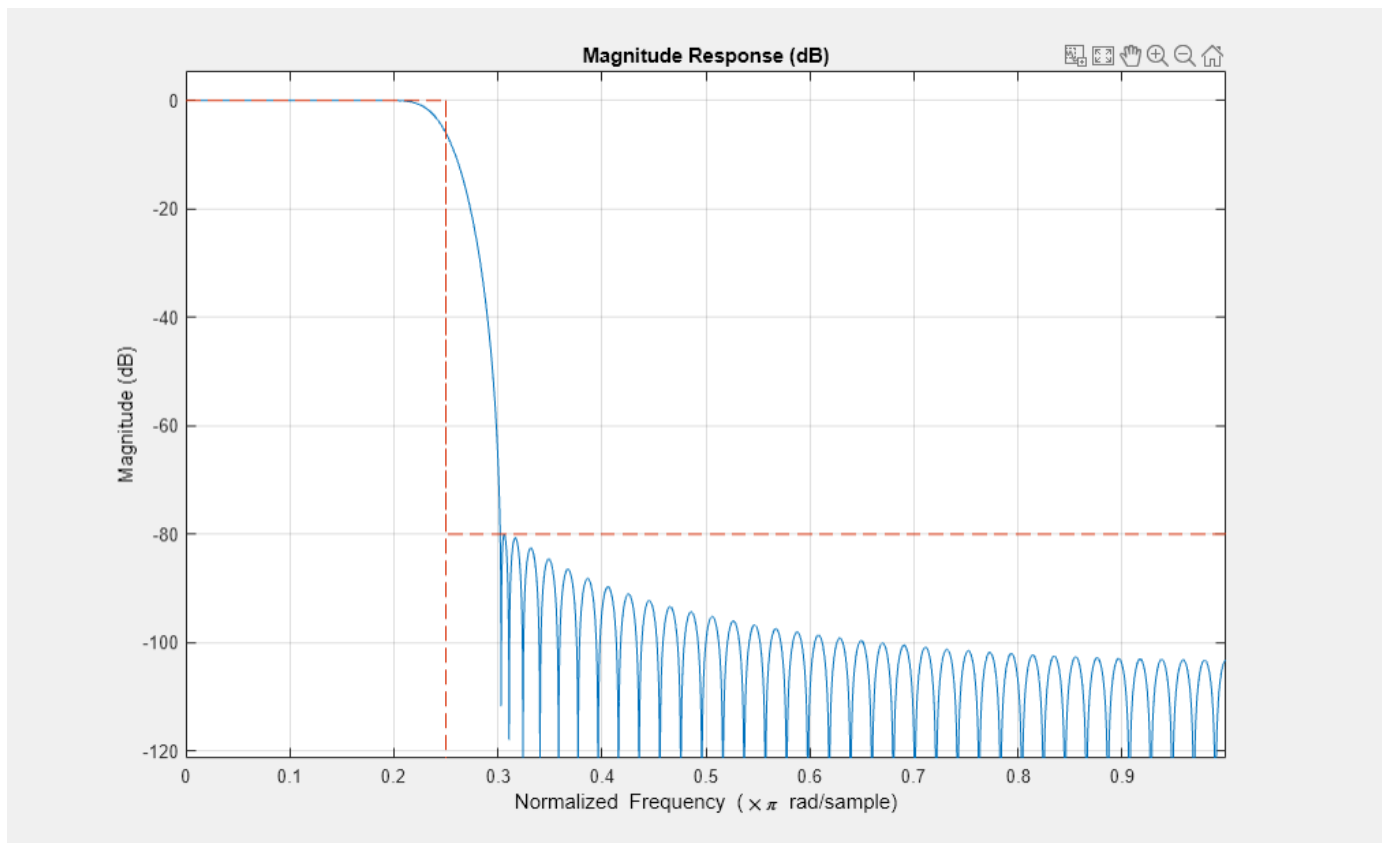
    Show all properties

```

To access the filter coefficients in the automatic mode, type `firdecim.Numerator` in the MATLAB command prompt.

The designed filter occupies a narrower passband that is approximately a quarter of the spectrum.

```
fvtool(firdecim)
```



Decimate the cosine signal by a factor of 4. After a short transition, the output converges to a cosine of frequency  $\pi$  as expected, which is four times the lower frequency component of the input signal  $\frac{\pi}{4}$ . This time, the amplitude of the output is half the amplitude of the input since the gain of the FIR at

$\omega = \frac{\pi}{4}$  is exactly  $\frac{1}{2}$ . The high frequency component  $\frac{2\pi}{3}$  diminishes by the lowpass FIR whose cutoff frequency is  $\frac{\pi}{4}$ .

```
yAuto = firdecim(xhigh);
```

Plot the input signal with the high frequency component added, low frequency component scaled by 1/2, and the decimated signal. Recalculate the output delay and the output sample rate since the decimation factor has changed.

```
[delay,FsOut] = outputDelay(firdecim);
```

```
tyAuto = (0:length(yAuto)-1)/FsOut-delay;
stem(tyAuto,yAuto,'filled',MarkerSize=4); hold on;
stem(nx,xhigh);
stem(nx,x/2,'m:',MarkerSize=7); hold off;
```

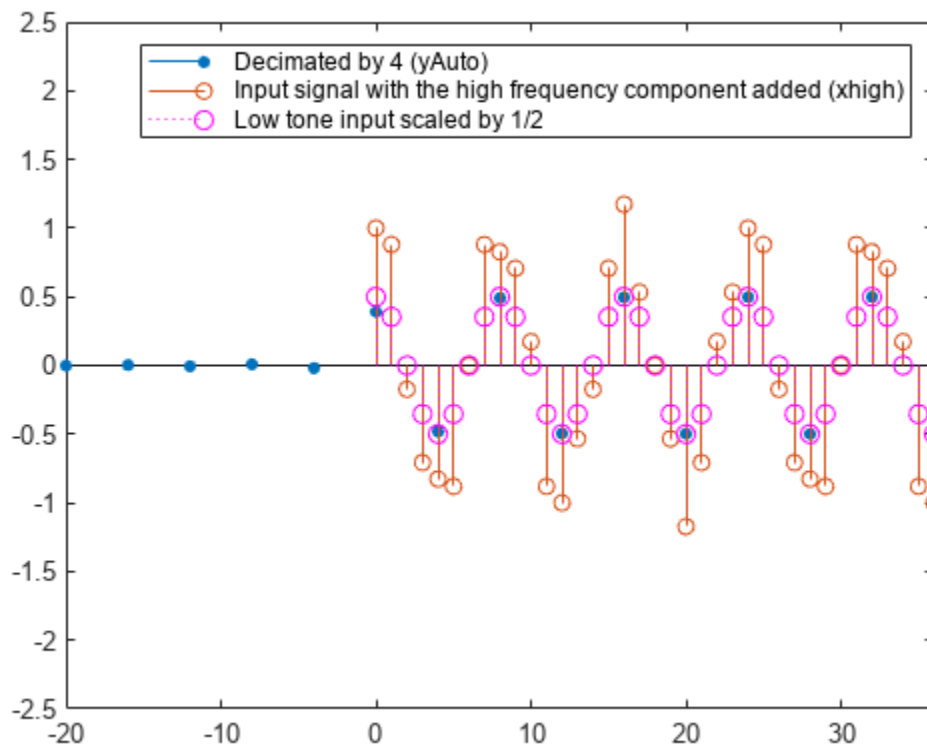
```
xlim([-20,36])
```

```
ylim([-2.5 2.5])
```

```
legend('Decimated by 4 (yAuto)',...
```

```
'Input signal with the high frequency component added (xhigh)',...
```

```
'Low tone input scaled by 1/2');
```





## Reduce Sample Rate of Audio Signal

Reduce the sample rate of an audio signal by a factor of 2 and play the decimated signal using the `audioDeviceWriter` object.

**Note:** The `audioDeviceWriter` System object™ is not supported in MATLAB Online.

Create a `dsp.AudioFileReader` object. The default audio file read by the object has a sample rate of 22050 Hz.

```
afr = dsp.AudioFileReader('OutputDataType',...
    'single');
```

Create a `dsp.FIRDecimator` object and specify the decimation factor to be 2. The object designs the filter using `designMultirateFIR(1,2)` and stores the coefficients in the `Numerator` property of the object.

```
firdecim = dsp.FIRDecimator(2)
```

```
firdecim =
    dsp.FIRDecimator with properties:
```

```
    Main
```

```
    DecimationFactor: 2
```

```
    NumeratorSource: 'Property'
```

```
        Numerator: [0 -1.0054e-04 0 3.8704e-04 0 -0.0010 0 0.0022 0 -0.0043 0 0.0077 0 -0.012
```

```
        Structure: 'Direct form'
```

```
Show all properties
```

Create an `audioDeviceWriter` object. Specify the sample rate to be 22050/2.

```
adw = audioDeviceWriter(22050/2)
```

```
adw =
```

```
    audioDeviceWriter with properties:
```

```
        Driver: 'DirectSound'
```

```
        Device: 'Default'
```

```
        SampleRate: 11025
```

```
Show all properties
```

Read the audio signal using the file reader object, decimate the signal by a factor of 2, and play the decimated signal.

```
while ~isDone(afr)
    frame = afr();
    y = firdecim(frame);
    adw(y);
end

release(afr);
pause(0.5);
release(adw);
```

## More About

### Polyphase Subfilters

A polyphase implementation of an FIR decimator *splits* the lowpass FIR filter impulse response into  $M$  different subfilters, where  $M$  is the downsampling or decimation factor. For more details on the polyphase implementation, see “Algorithms” on page 4-500.

Let  $h(n)$  denote the FIR filter impulse response of length  $N+1$  and  $x(n)$  the input signal. Decimating the filter output by a factor of  $M$  is equivalent to the downsampled convolution:

$$y(n) = \sum_{l=0}^N h(l)x(nM - l)$$

The key to the efficiency of polyphase filtering is that specific input values are only multiplied by select values of the impulse response in the downsampled convolution. For example, letting  $M = 2$ , the input values  $x(0), x(2), x(4), \dots$  are combined only with the filter coefficients  $h(0), h(2), h(4), \dots$ , and the input values  $x(1), x(3), x(5), \dots$  are combined only with the filter coefficients  $h(1), h(3), h(5), \dots$ . By splitting the filter coefficients into two polyphase subfilters, no unnecessary computations are performed in the convolution. The outputs of the convolutions with the polyphase subfilters are interleaved and summed to yield the filter output.

The following code demonstrates how to construct the two polyphase subfilters for the default order 35 filter.

```
M = 2;
Num = fir1(35,0.4);
FiltLength = length(Num);
Num = flipud(Num(:));

if (rem(FiltLength, M) ~= 0)
    nzeros = M - rem(FiltLength, M);
    Num = [zeros(nzeros,1); Num]; % Appending zeros
end

len = length(Num);
nrows = len / M;
PolyphaseFilt = flipud(reshape(Num, M, nrows).');
```

The columns of `PolyphaseFilt` are subfilters containing the two *phases* of the filter in `Num`. For a general downsampling factor of  $M$ , there are  $M$  phases and therefore  $M$  subfilters.

## Algorithms

The FIR decimation filter is implemented efficiently using a polyphase structure. For more details on polyphase filters, see Polyphase Subfilters on page 4-500.

To derive the polyphase structure, start with the transfer function of the FIR filter:

$$H(z) = b_0 + b_1z^{-1} + \dots + b_Nz^{-N}$$

$N+1$  is the length of the FIR filter.

You can rearrange this equation as follows:

$$\begin{aligned}
 H(z) = & (b_0 + b_M z^{-M} + b_{2M} z^{-2M} + \dots + b_{N-M+1} z^{-(N-M+1)}) + \\
 & z^{-1} (b_1 + b_{M+1} z^{-M} + b_{2M+1} z^{-2M} + \dots + b_{N-M+2} z^{-(N-M+1)}) + \\
 & \vdots \\
 & z^{-(M-1)} (b_{M-1} + b_{2M-1} z^{-M} + b_{3M-1} z^{-2M} + \dots + b_N z^{-(N-M+1)})
 \end{aligned}$$

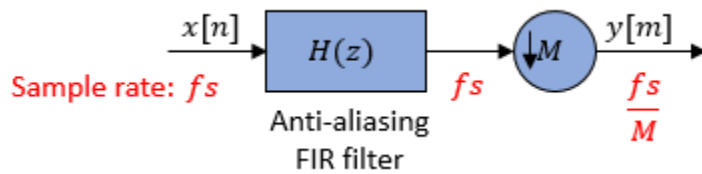
$M$  is the number of polyphase components, and its value equals the decimation factor that you specify.

You can write this equation as:

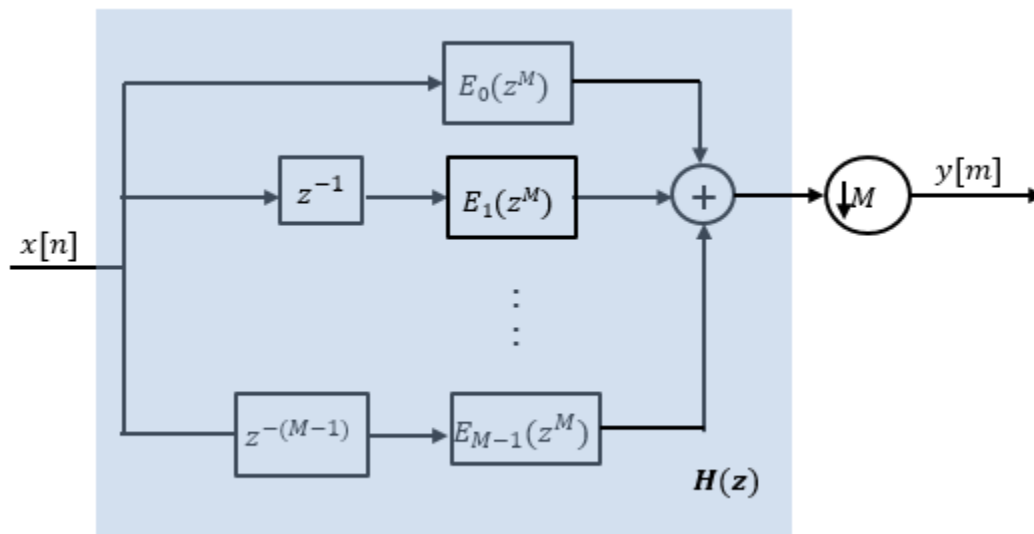
$$H(z) = E_0(z^M) + z^{-1}E_1(z^M) + \dots + z^{-(M-1)}E_{M-1}(z^M)$$

$E_0(z^M), E_1(z^M), \dots, E_{M-1}(z^M)$  are the polyphase components of the FIR filter  $H(z)$ .

Conceptually, the FIR decimation filter contains a lowpass FIR filter followed by a downsampler.



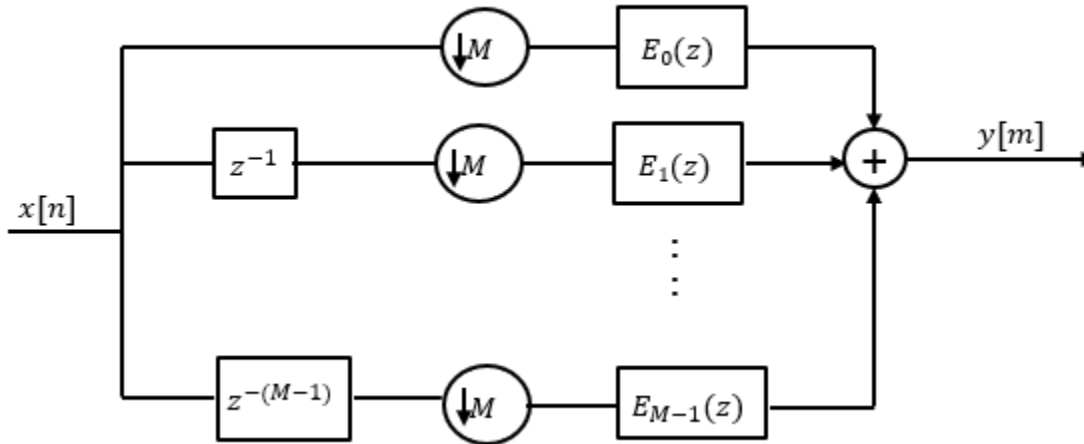
Replace  $H(z)$  with its polyphase representation.



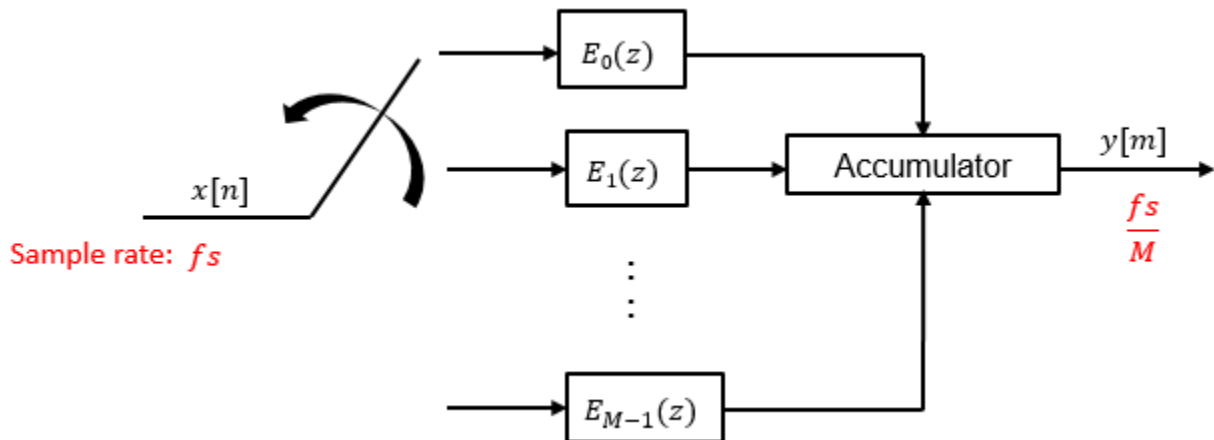
Here is the multirate noble identity for decimation.



Applying the noble identity for decimation moves the downsampling operation to before the filtering operation. This move enables you to filter the signal at a lower rate.



You can replace the delays and the decimation factor at the input with a commutator switch. The switch starts on the first branch 0 and moves in the counterclockwise direction as shown in this diagram. The accumulator at the output receives the processed input samples from each branch of the polyphase structure and accumulates these processed samples until the switch goes to branch 0. When the switch goes to branch 0, the accumulator outputs the accumulated value.



When the first input sample is delivered, the switch feeds this input to the branch 0 and the decimator computes the first output value. As more input samples come in, the switch moves in the counter clockwise direction through branches  $M-1$ ,  $M-2$ , and all the way up to branch 0, delivering one sample at a time to each branch. When the switch comes to branch 0, the decimator outputs the next set of output values. This process continues as data keeps coming in. Every time the switch comes to the branch 0, the decimator outputs  $y[m]$ . The decimator effectively outputs one sample for every  $M$  samples it receives. Hence the sample rate at the output of the FIR decimation filter is  $f_s/M$ .

## Version History

Introduced in R2012a

### R2022b: Support for arbitrary input frame length

Starting in R2022b, this object supports an input signal with an arbitrary frame length. so the input frame length does not have to be a multiple of the decimation factor.

When you generate code, to support arbitrary frame length for fixed-size signals, you must set the `AllowArbitraryInputLength` property to `true` while generating code.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

The `dsp.FIRDecimator` System object supports SIMD code generation using Intel AVX2 technology under these conditions:

- Filter structure is set to 'Direct form'.
- Input signal is real-valued with real filter coefficients.
- Input signal is complex-valued with real or complex filter coefficients.
- Input signal has a data type of `single` or `double`.

The SIMD technology significantly improves the performance of the generated code.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

This object supports HDL code generation with the Filter Design HDL Coder product. For workflows and limitations, see “Generate HDL Code for Filter System Objects” (Filter Design HDL Coder).

## See Also

### Functions

`freqz` | `fvtool` | `info` | `cost` | `polyphase` | `generatehdl` | `impz` | `coeffs`

### Objects

`dsp.FIRInterpolator` | `dsp.FIRRateConverter`

### Blocks

FIR Decimation

### Topics

“Analysis Methods for Filter System Objects” on page 3-2

## **dsp.FIRFilter**

**Package:** dsp

Static or time-varying FIR filter

### **Description**

The `dsp.FIRFilter` System object filters each channel of the input using static or time-varying FIR filter implementations.

To filter each channel of the input:

- 1 Create the `dsp.FIRFilter` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

Under specific conditions, this System object also supports SIMD code generation. For details, see “Code Generation” on page 4-514.

### **Creation**

#### **Syntax**

```
fir = dsp.FIRFilter
fir = dsp.FIRFilter(num)
fir = dsp.FIRFilter(Name,Value)
```

#### **Description**

`fir = dsp.FIRFilter` returns a finite impulse response (FIR) filter object, `fir`, which independently filters each channel of the input over time using a specified FIR filter implementation.

`fir = dsp.FIRFilter(num)` returns an FIR filter System object, `fir`, with the `Numerator` property set to `num`.

`fir = dsp.FIRFilter(Name,Value)` returns an FIR filter System object, `fir`, with each property set to the specified value.

#### **Properties**

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

**Structure – Filter structure**

Direct form (default) | Direct form symmetric | Direct form antisymmetric | Direct form transposed | Lattice MA

Specify the filter structure. You can specify the filter structure as one of Direct form | Direct form symmetric | Direct form antisymmetric | Direct form transposed | Lattice MA.

**NumeratorSource – Source of filter coefficients**

Property (default) | Input port

Specify the source of the filter coefficients as Property or Input port. When you specify Input port, the filter object updates the time-varying filter once every frame.

**Dependencies**

This applies when you set the Structure to Direct form | Direct form symmetric | Direct form antisymmetric | Direct form transposed.

**ReflectionCoefficientsSource – Source of filter coefficients**

Property (default) | Input port

Specify the source of the Lattice filter coefficients as Property or Input port. When you specify Input port, the filter object updates the time-varying filter once every frame.

**Dependencies**

This applies when you set the Structure to Lattice MA.

**Numerator – Numerator coefficients**

[0.5 0.5] (default) | row vector

Specify the filter coefficients as a real or complex numeric row vector.

**Tunable:** Yes

**Dependencies**

This property applies when you set the NumeratorSource property to Property, and the “Structure” on page 4-0 property is set to Direct form, Direct form symmetric, Direct form antisymmetric, or Direct form transposed.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**ReflectionCoefficients – Reflection coefficients of lattice filter structure**

[0.5 0.5] (default) | row vector

Specify the reflection coefficients of a lattice filter as a real or complex numeric row vector.

**Tunable:** Yes

**Dependencies**

This property applies when you set the “Structure” on page 4-0 property to Lattice MA, and the ReflectionCoefficientsSource property to Property.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**InitialConditions — Initial conditions for the FIR filter**`0 (default) | scalar | vector | matrix`

Specify the initial conditions of the filter states. The number of states or delay elements equals the number of reflection coefficients for the lattice structure, or the number of filter coefficients-1 for the other direct form structures.

You can specify the initial conditions as a scalar, vector, or matrix. If you specify a scalar value, the FIR filter object initializes all delay elements in the filter to that value. If you specify a vector whose length equals the number of delay elements in the filter, each vector element specifies a unique initial condition for the corresponding delay element. The object applies the same vector of initial conditions to each channel of the input signal.

If you specify a vector whose length equals the product of the number of input channels and the number of delay elements in the filter, each element specifies a unique initial condition for the corresponding delay element in the corresponding channel.

If you specify a matrix with the same number of rows as the number of delay elements in the filter, and one column for each channel of the input signal, each element specifies a unique initial condition for the corresponding delay element in the corresponding channel.

**Tunable:** Yes

Data Types: `single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64`

**Fixed-Point Properties****FullPrecisionOverride — Full precision override for fixed-point arithmetic**`true (default) | false`

Specify whether to use full precision rules. If you set `FullPrecisionOverride` to `true`, which is the default, the object computes all internal arithmetic and output data types using full precision rules. These rules provide the most accurate fixed-point numerics. It also turns off the display of other fixed-point properties because they do not apply individually. These rules guarantee that no quantization occurs within the object. Bits are added, as needed, to ensure that no roundoff or overflow occurs. If you set `FullPrecisionOverride` to `false`, fixed-point data types are controlled through individual fixed-point property settings. For more information, see “Full Precision for Fixed-Point System Objects”.

**RoundingMethod — Rounding method for fixed-point operations**`Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero`

Specify the rounding method.

**Dependencies**

This property applies only if the object is not in full precision mode.

**OverflowAction — Overflow action for fixed-point operations**`Wrap (default) | Saturate`

Specify the overflow action as `Wrap` or `Saturate`.

**Dependencies**

This property applies only if the object is not in full precision mode.



**CoefficientsDataType — Coefficients word and fraction lengths**

Same word length as input (default) | Custom

Specify the coefficients fixed-point data type as Same word length as input or Custom.

**Dependencies**

This property applies when you set the NumeratorSource property to Property.

**CustomCoefficientsDataType — Custom coefficients word and fraction lengths**

numerictype(true,16,15) (default) | numerictype

Specify the coefficients fixed-point type as a signed or unsigned numerictype object.

**Dependencies**

This property applies when you set the CoefficientsDataType property to Custom.

**ReflectionCoefficientsDataType — Reflection coefficients word and fraction lengths**

Same word length as input (default) | Custom

Specify the reflection coefficients fixed-point data type as Same word length as input or Custom.

**Dependencies**

This property applies when you set the ReflectionCoefficientsSource property to Property.

**CustomReflectionCoefficientsDataType — Custom reflection coefficients word and fraction lengths**

numerictype(true,16,15) (default) | numerictype

Specify the reflection coefficients fixed-point type as a signed or unsigned numerictype object.

**Dependencies**

This property applies when you set the ReflectionCoefficientsDataType property to Custom.

**ProductDataType — Product word and fraction lengths**

Full precision (default) | Same as input | Custom

Specify the product fixed-point data type as Full precision, Same as input, or Custom.

**CustomProductDataType — Custom product word and fraction lengths**

numerictype(true,32,30) (default) | numerictype

Specify the product fixed-point type as a signed or unsigned scaled numerictype object.

**Dependencies**

This property applies when you set the ProductDataType property to Custom.

**AccumulatorDataType — Accumulator word and fraction lengths**

Full precision (default) | Same as input | Same as product | Custom

Specify the accumulator fixed-point data type to Full precision, Same as input, Same as product, or Custom.

**CustomAccumulatorDataType — Custom accumulator word and fraction lengths**`numerictype(true,32,30)` (default) | `numerictype`

Specify the accumulator fixed-point type as a signed or unsigned scaled `numerictype` object.

**Dependencies**

This property applies when you set the `AccumulatorDataType` property to `Custom`.

**StateDataType — State word and fraction lengths**`Same as accumulator` (default) | `Same as input` | `Custom`

Specify the state fixed-point data type as one of `Same as input`, `Same as accumulator`, or `Custom`.

**Dependencies**

This property does not apply to any of the direct form or direct form I filter structures.

**CustomStateDataType — Custom state word and fraction lengths**`numerictype(true,16,15)` (default) | `numerictype`

Specify the state fixed-point type as a signed or unsigned scaled `numerictype` object.

**Dependencies**

This property applies when you set the `StateDataType` property to `Custom`.

**OutputDataType — Output word and fraction lengths**`Same as accumulator` (default) | `Same as input` | `Custom`

Specify the output fixed-point data type as one of `Same as input`, `Same as accumulator`, or `Custom`.

**CustomOutputDataType — Custom output word and fraction lengths**`numerictype(true,16,15)` (default) | `numerictype`

Specify the output fixed-point type as a signed or unsigned scaled `numerictype` object.

**Dependencies**

This property applies when you set the “`OutputDataType`” on page 4-0 property to `Custom`.

**Usage****Syntax**

```
y = fir(x)
y = fir(x,coeff)
```

**Description**

`y = fir(x)` applies an FIR filter to the real or complex input signal `x` to produce the output `y`.

$y = \text{fir}(x, \text{coeff})$  uses the time-varying coefficients, `coeff`, to filter the input signal `x` and produce the output `y`. You can use this option when you set the `NumeratorSource` or `ReflectionCoefficientsSource` property to `Input port`.

## Input Arguments

### **x** — Data input

vector | matrix

Data input, specified as a vector or a matrix. When the input data is of a fixed-point type, it must be signed when the structure is set to `Direct form symmetric` or `Direct form antisymmetric`. The FIR filter object operates on each channel of the input signal independently over successive calls to the object.

This System object supports variable-size input.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fi`

### **coeff** — Filter coefficients

row vector

Time-varying filter coefficients, specified as a row vector. The data and coefficient inputs must have the same data type.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fi`

Complex Number Support: Yes

## Output Arguments

### **y** — Filtered output

vector | matrix

Filtered output, returned as a vector or a matrix. The output has the same size and data type as the input.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fi`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Specific to `dsp.FIRFilter`

|                          |   |
|--------------------------|---|
| <code>freqz</code>       | Frequency response of discrete-time filter System object                      |
| <code>fvtool</code>      | Visualize frequency response of DSP filters                                   |
| <code>impz</code>        | Impulse response of discrete-time filter System object                        |
| <code>info</code>        | Information about filter System object  |
| <code>coeffs</code>      | Returns the filter System object coefficients in a structure                  |
| <code>cost</code>        | Estimate cost of implementing filter System object                            |
| <code>grpdelay</code>    | Group delay response of discrete-time filter System object                    |
| <code>generatehdl</code> | Generate HDL code for quantized DSP filter (requires Filter Design HDL Coder) |

## Common to All System Objects

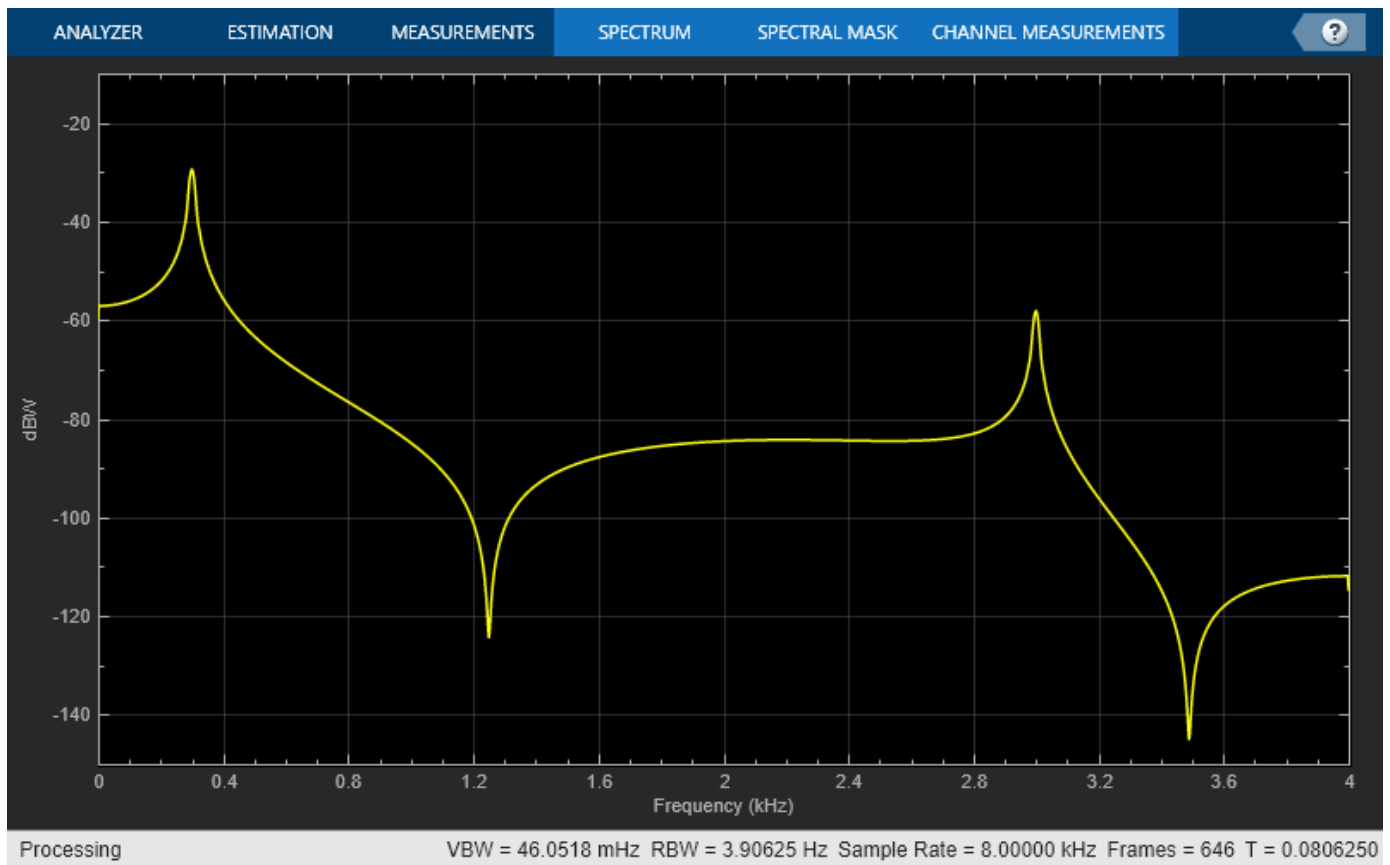
`step` Run System object algorithm  
`release` Release resources and allow changes to System object property values and input characteristics  
`reset` Reset internal states of System object

## Examples

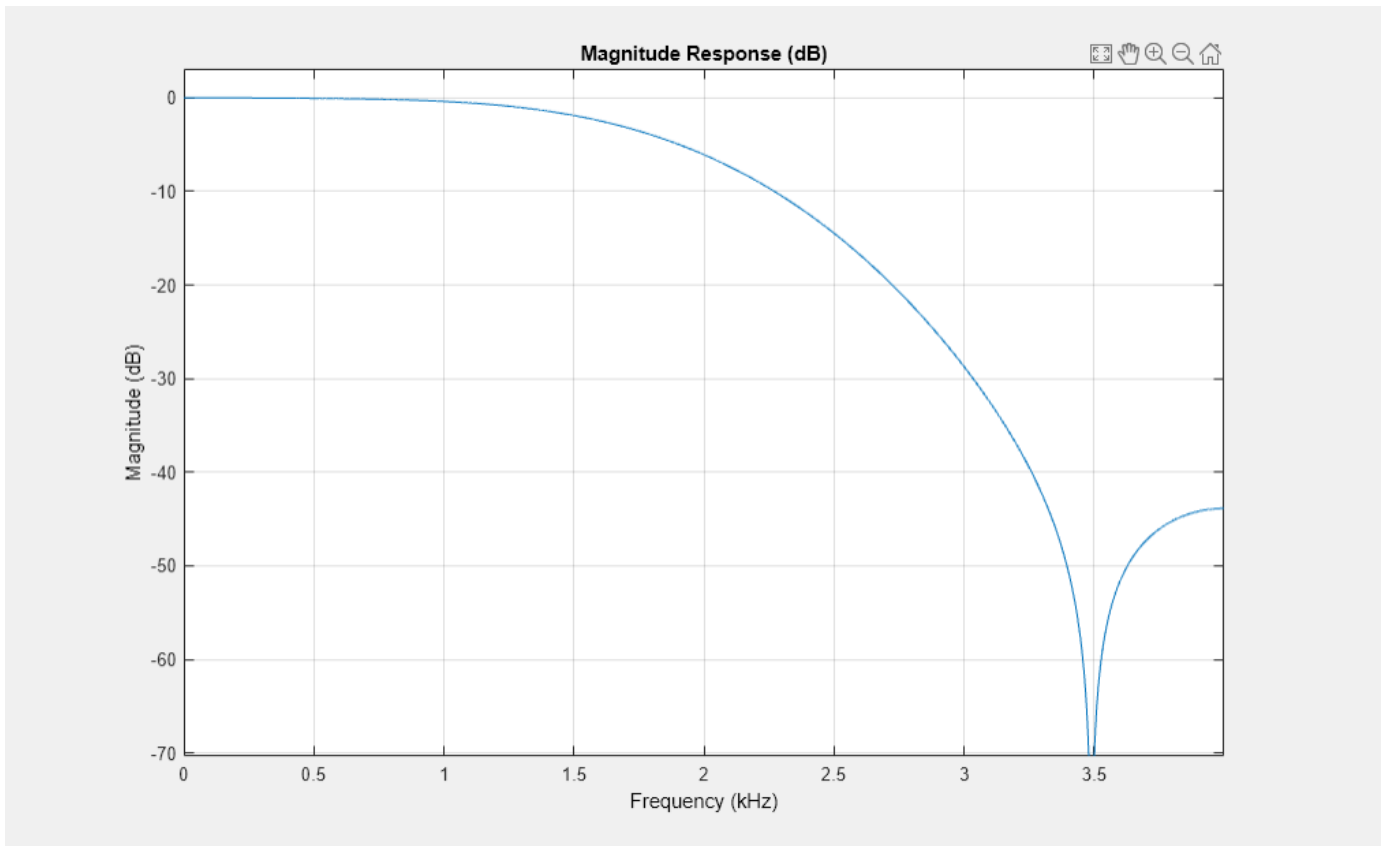
### Lowpass Filter a Sinusoid Signal Using FIRFilter object

Use an FIR filter to apply a low pass filter to a waveform with two sinusoidal components.

```
t = (0:1000)'/8e3;  
xin = sin(2*pi*0.3e3*t)+sin(2*pi*3e3*t);  
  
sr = dsp.SignalSource;  
sr.Signal = xin;  
sink = dsp.SignalSink;  
  
fir = dsp.FIRFilter(fir1(10,0.5));  
  
sa = spectrumAnalyzer(...  
    'SampleRate',8e3,...  
    'Method','welch',...  
    'PlotAsTwoSidedSpectrum',false,...  
    'OverlapPercent',80,...  
    'SpectrumUnits','dBW',...  
    'YLimits',[-150 -10]);  
  
while ~isDone(sr)  
    input = sr();  
    filteredOutput = fir(input);  
    sink(filteredOutput);  
    sa(filteredOutput)  
end
```

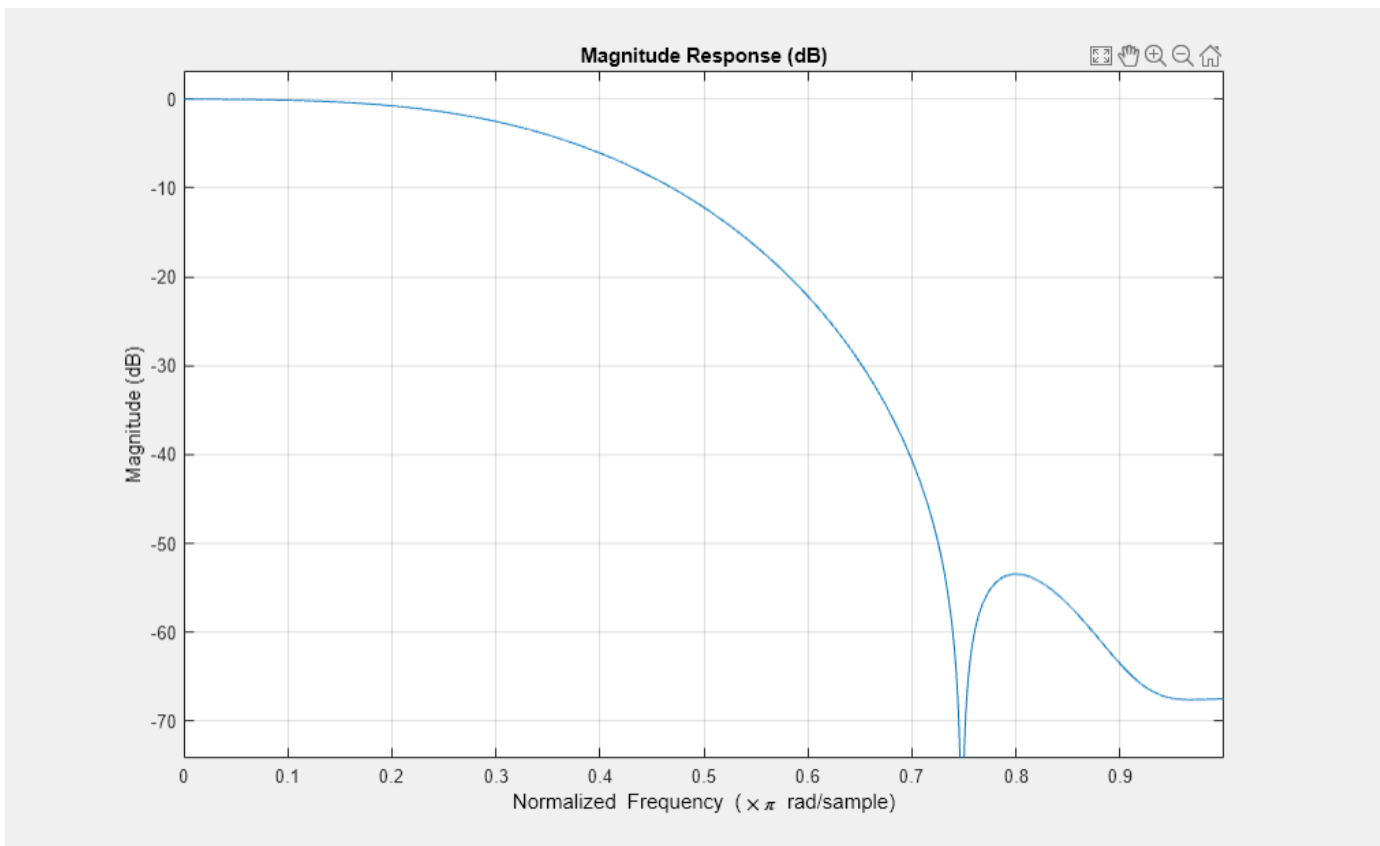


```
filteredResult = sink.Buffer;  
fvtool(fir,'Fs',8000)
```



Design an FIR filter as a System object.

```
N = 10;  
Fc = 0.4;  
B = fir1(N,Fc);  
fir1 = dsp.FIRFilter(B);  
fvtool(fir1)
```



This can also be achieved by using `fdesign` as a constructor and `design` to design the filter.

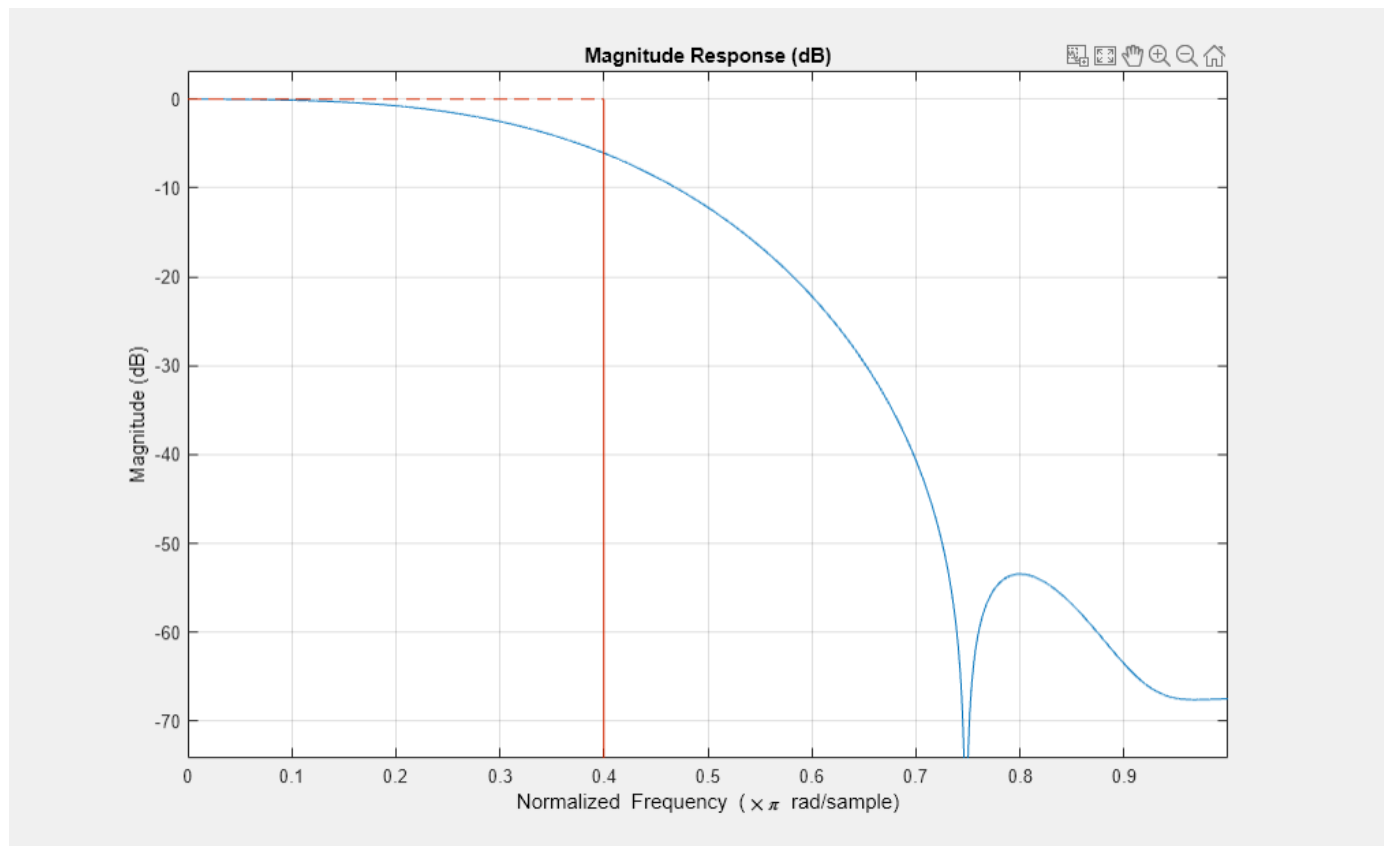
```
N = 10;
Fc = 0.4;
specLowpass = fdesign.lowpass('N,Fc',...
    N,Fc);
fir2 = design(specLowpass,...
    'systemobject',true)

fir2 =
    dsp.FIRFilter with properties:

        Structure: 'Direct form'
        NumeratorSource: 'Property'
        Numerator: [-1.2414e-18 -0.0126 -0.0247 0.0635 0.2748 0.3981 0.2748 0.0635 -0.0247 -
        InitialConditions: 0

    Show all properties

fvtool(fir2);
```



## Algorithms

This object implements the algorithm, inputs, and outputs described on the Discrete FIR Filter block reference page. The object properties correspond to the block parameters.

## Version History

Introduced in R2012a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Only the `Numerator` property is tunable for code generation.
- See “System Objects in MATLAB Code Generation” (MATLAB Coder).

The `dsp.FIRFilter` System object supports SIMD code generation using Intel AVX2 technology under these conditions:



- Filter structure is set to 'Direct form' or 'Direct form transposed'.
- Input signal is real-valued with real filter coefficients.
- Input signal is complex-valued with real or complex filter coefficients.
- Input signal has a data type of `single` or `double`.

The SIMD technology significantly improves the performance of the generated code.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

This object supports HDL code generation with the HDL Coder or Filter Design HDL Coder products. For HDL Coder workflow and limitations, see “HDL Code Generation for System Objects” (HDL Coder). For Filter Design HDL Coder workflow and limitations, see “Generate HDL Code for Filter System Objects” (Filter Design HDL Coder).

## **See Also**

### **Functions**

`freqz` | `fvtool` | `impz` | `info` | `coeffs` | `cost` | `grpdelay` | `generatehdl` | `designFracDelayFIR`

### **Objects**

`dsp.SOSFilter`

### **Blocks**

Discrete FIR Filter

### **Topics**

“FIR Nyquist (L-th band) Filter Design”

“Analysis Methods for Filter System Objects” on page 3-2

# dsp.FIRHalfbandDecimator

**Package:** dsp

Halfband decimator

## Description

The `dsp.FIRHalfbandDecimator` System object performs an efficient polyphase decimation of the input signal by a factor of two. You can use `dsp.FIRHalfbandDecimator` to implement the analysis portion of a two-band filter bank to filter a signal into lowpass and highpass subbands.

`dsp.FIRHalfbandDecimator` uses an FIR equiripple design or a Kaiser window design to construct the halfband filters and a polyphase implementation to filter the input.

To filter and downsample your data:

- 1 Create the `dsp.FIRHalfbandDecimator` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
firhalfbanddecim = dsp.FIRHalfbandDecimator  
firhalfbanddecim = dsp.FIRHalfbandDecimator(Name=Value)
```

### Description

`firhalfbanddecim = dsp.FIRHalfbandDecimator` returns a halfband decimator, `firhalfbanddecim`, with the default settings. Under the default settings, the System object filters and downsamples the input data with a halfband frequency of 11025 Hz, a transition width of 4.1 kHz, and a stopband attenuation of 80 dB. The design method is set to "Auto".

`firhalfbanddecim = dsp.FIRHalfbandDecimator(Name=Value)` returns a halfband decimator, with additional properties specified by one or more `Name=Value` pair arguments.

Example: `firhalfbanddecim = dsp.FIRHalfbandDecimator(Specification="Filter order and stopband attenuation")` creates an FIR halfband decimator object with filter order set to 52 and stopband attenuation set to 80 dB.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see *System Design in MATLAB Using System Objects*.

### Specification — Filter design parameters

"Transition width and stopband attenuation" (default) | "Filter order and stopband attenuation" | "Filter order and transition width" | "Coefficients"

Filter design parameters, specified as a character vector. When you set `Specification` to one of the following, you choose two of the three available design parameters to design the FIR Halfband filter.

- "Transition width and stopband attenuation" -- Transition width and stopband attenuation are the design parameters.
- "Filter order and stopband attenuation" -- Filter order and stopband attenuation are the design parameters.
- "Filter order and transition width" -- Filter order and transition width are the design parameters.

The filter is designed using the optimal equiripple filter design method or the kaiser-window-based design method.

When you set `Specification` to "Coefficients", you specify the halfband filter coefficients directly through the `Numerator` property.

### FilterOrder — Filter order

52 (default) | even positive integer

Filter order, specified as an even positive integer.

#### Dependencies

This property applies when you set `Specification` to either "Filter order and stopband attenuation" or "Filter order and transition width".

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### StopbandAttenuation — Stopband attenuation

80 (default) | positive real scalar

Stopband attenuation in dB, specified as a positive real scalar.

#### Dependencies

This property applies when you set `Specification` to either "Filter order and stopband attenuation" or "Transition width and stopband attenuation".

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### TransitionWidth — Transition width

4100 (default) | positive real scalar

Transition width in Hz, specified as a positive real scalar. The value of the transition width in Hz must be less than 1/2 the input sample rate.

#### Dependencies

This property applies when you set `Specification` to either "Transition width and stopband attenuation" or "Filter order and transition width".

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **Numerator — FIR halfband filter coefficients**

`firhalfband('minororder',0.407,1e-4)` (default) | row vector

FIR halfband filter coefficients, specified as a row vector. The coefficients must comply with the FIR halfband impulse response format. For details on this format, see “Halfband Filters” on page 4-528 and “FIR Halfband Filter Design”. If half the order of the filter,  $(\text{length}(\text{Numerator}) - 1)/2$ , is even, every other coefficient starting from the first coefficient must be a zero except for the center coefficient which must be a 0.5. If half the order of the filter is odd, the sequence of alternating zeros with a 0.5 at the center starts at the second coefficient.

#### **Dependencies**

This property applies when you set `Specification` to “Coefficients”.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **DesignMethod — Filter design method**

“Auto” (default) | “Equiripple” | “Kaiser”

Specify the filter design method as one of the following:

- “Auto” -- The algorithm automatically chooses the filter design method depending on the filter design parameters. The algorithm uses the equiripple or the Kaiser window method to design the filter.

If the design constraints are very tight, such as very high stopband attenuation or very narrow transition width, then the algorithm automatically chooses the Kaiser method, as this method is optimal for designing filters with very tight specifications. However, if the design constraints are not tight, then the algorithm chooses the equiripple method.

When you set the `DesignMethod` property to “Auto”, you can determine the method used by the algorithm by examining the passband and stopband ripple characteristics of the designed filter. If the object used the equiripple method, the passband and stopband ripples of the designed filter have a constant amplitude in the frequency response. If the filter design method the object chooses in the “Auto” mode is not suitable for your application, manually specify the `DesignMethod` as “Equiripple” or “Kaiser”.

- “Equiripple” -- The algorithm uses the equiripple method.
- “Kaiser” -- The algorithm uses the Kaiser window method.

For more details on these two methods, see “Algorithms” on page 4-529.

#### **Dependencies**

To enable this property, set `Specification` to “Transition width and stopband attenuation”, “Filter order and stopband attenuation”, or “Filter order and transition width”.

Data Types: `char` | `string`

### **SampleRate — Input sample rate**

44100 (default) | positive real scalar

Input sample rate in Hz, specified as a positive real scalar. The input sample rate defaults to 44100 Hz. If you specify transition width as one of your filter design parameters, the transition width cannot exceed 1/2 the input sample rate.

Data Types: `single` | `double`

### Fixed-Point Properties

#### CoefficientsDataType — Word and fraction lengths of coefficients

`numericity(1,16)` (default) | `numericity` object

Word and fraction lengths of coefficients, specified as a signed or unsigned `numericity` object. The default, `numericity(1,16)` corresponds to a signed numeric type object with 16-bit coefficients and a fraction length determined based on the coefficient values, to give the best possible precision.

This property is not tunable.

Word length of the output is same as the word length of the input. Fraction length of the output is computed such that the entire dynamic range of the output can be represented without overflow. For details on how the fraction length of the output is computed, see “Fixed-Point Precision Rules for Avoiding Overflow in FIR Filters”.

#### RoundingMethod — Rounding method for output fixed-point operations

"Floor" (default) | "Ceiling" | "Convergent" | "Nearest" | "Round" | "Simplest" | "Zero"

Rounding method for output fixed-point operations, specified as a character vector. For more information on the rounding modes, see “Precision and Range”.

## Usage

### Syntax

```
y_low = firhalfbanddecim(x)
[y_low,y_high] = firhalfbanddecim(x)
```

### Description

`y_low = firhalfbanddecim(x)` filters the input signal `x` using the FIR halfband filter, `firhalfbanddecim`, and downsamples the output by a factor of 2.

`[y_low,y_high] = firhalfbanddecim(x)` computes the `y_low` and `y_high`, of the analysis filter bank, `firhalfbanddecim` for input `x`. A  $K_i$ -by- $N$  input matrix is treated as  $N$  independent channels. The System object generates two power-complementary output signals by adding and subtracting the two polyphase branch outputs respectively. `y_low` and `y_high` are of the same size ( $K_o$ -by- $N$ ) and data type.  $K_o = K_i/2$ , where 2 is the decimation factor.

### Input Arguments

#### **x** — Data input

column vector | matrix

Data input, specified as a column vector or a matrix. If the input signal is a matrix, each column of the matrix is treated as an independent channel. The number of rows in the input signal must be a multiple of 2.

This object supports variable-size input signal.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

### Output Arguments

#### **y<sub>low</sub>** — Lowpass subband of decimator output

column vector | matrix

Lowpass subband of decimator output, returned as a column vector or a matrix. The output, `ylow` is a lowpass halfband filtered and downsampled version of the input `x`. Due to the halfband nature of the filter, the downsampling factor is always 2.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

#### **y<sub>high</sub>** — Highpass subband of decimator output

column vector | matrix

Highpass subband of decimator output, returned as a column vector or a matrix. The output, `yhigh` is a highpass halfband filtered and downsampled version of the input `x`. Due to the halfband nature of the filter, the downsampling factor is always 2.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

### Specific to `dsp.FIRHalfbandDecimator`

|                        |  |
|------------------------|--|
| <code>freqz</code>     | Frequency response of discrete-time filter System object     |
| <code>fvtool</code>    | Visualize frequency response of DSP filters                  |
| <code>info</code>      | Information about filter System object                       |
| <code>cost</code>      | Estimate cost of implementing filter System object           |
| <code>coeffs</code>    | Returns the filter System object coefficients in a structure |
| <code>polyphase</code> | Polyphase decomposition of multirate filter                  |

### Common to All System Objects

|                      |  |
|----------------------|--|
| <code>step</code>    | Run System object algorithm  |
| <code>release</code> | Release resources and allow changes to System object property values and input characteristics |
| <code>reset</code>   | Reset internal states of System object   |

## Examples

### Impulse and Frequency Response of Halfband Decimation Filter

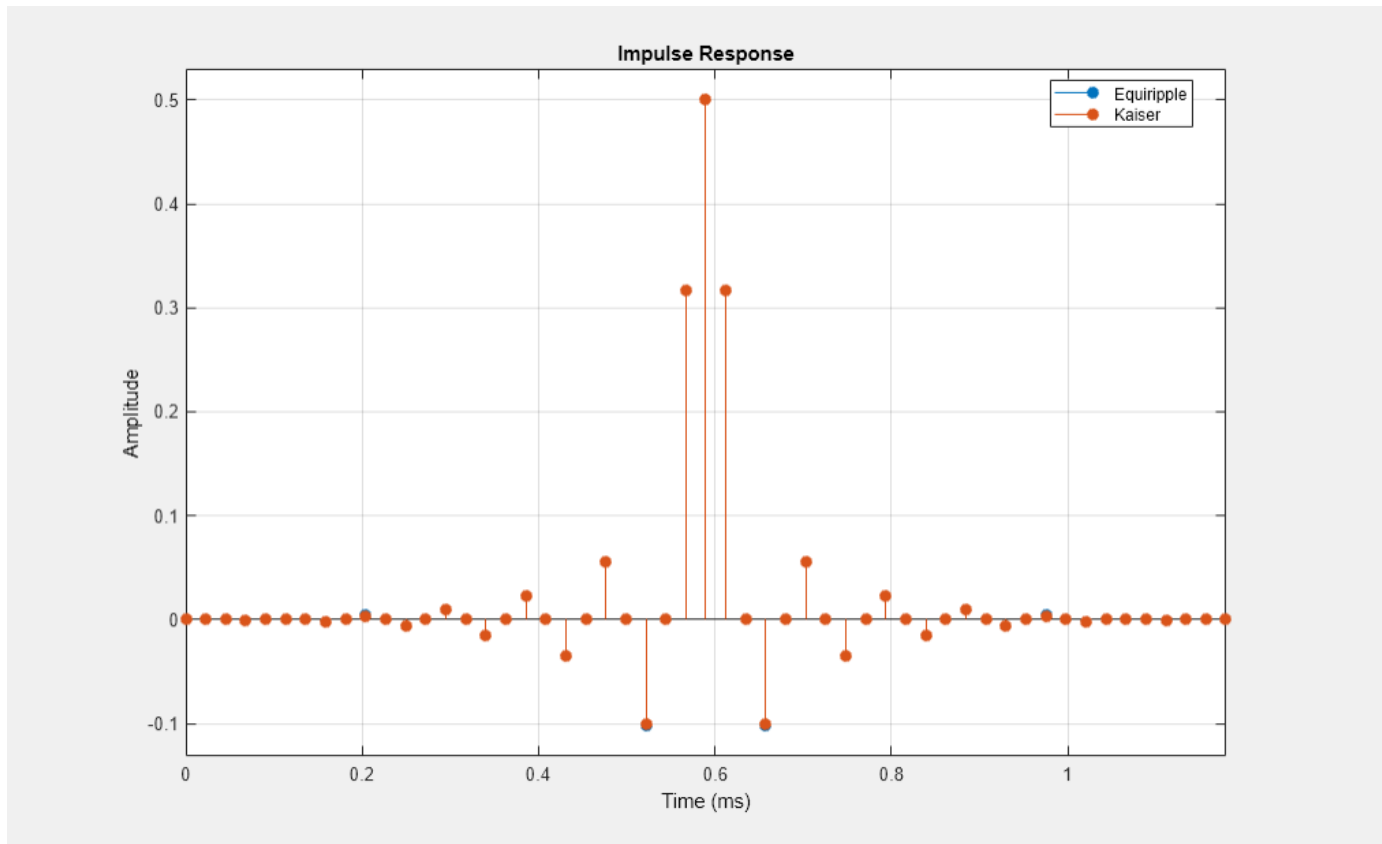
Create two lowpass halfband decimation filters for data sampled at 44.1 kHz. The design method in the first filter is set to "Equiripple" and in the second filter is set to "Kaiser".

The output data rate is 1/2 the input sample rate, or 22.05 kHz. Specify the filter order to be 52 with a transition width of 4.1 kHz.

```
Fs = 44.1e3;
filterspec = "Filter order and transition width";
Order = 52;
TW = 4.1e3;
firhalfbanddecimEquiripple = dsp.FIRHalfbandDecimator(...
    Specification=filterspec,...
    FilterOrder=Order,...
    TransitionWidth=TW,...
    DesignMethod="Equiripple",...
    SampleRate=Fs);
firhalfbanddecimKaiser = dsp.FIRHalfbandDecimator(...
    Specification=filterspec,...
    FilterOrder=Order,...
    TransitionWidth=TW,...
    DesignMethod="Kaiser",...
    SampleRate=Fs);
```

Plot the impulse response of both the filters. The zeroth-order coefficient is delayed 26 samples, which is equal to the group delay of the filter. This yields a causal halfband filter.

```
hfvt = fvtool(firhalfbanddecimEquiripple,firhalfbanddecimKaiser,...
    Analysis="impulse");
legend(hfvt,{'Equiripple','Kaiser'})
```



Plot the magnitude and phase response.

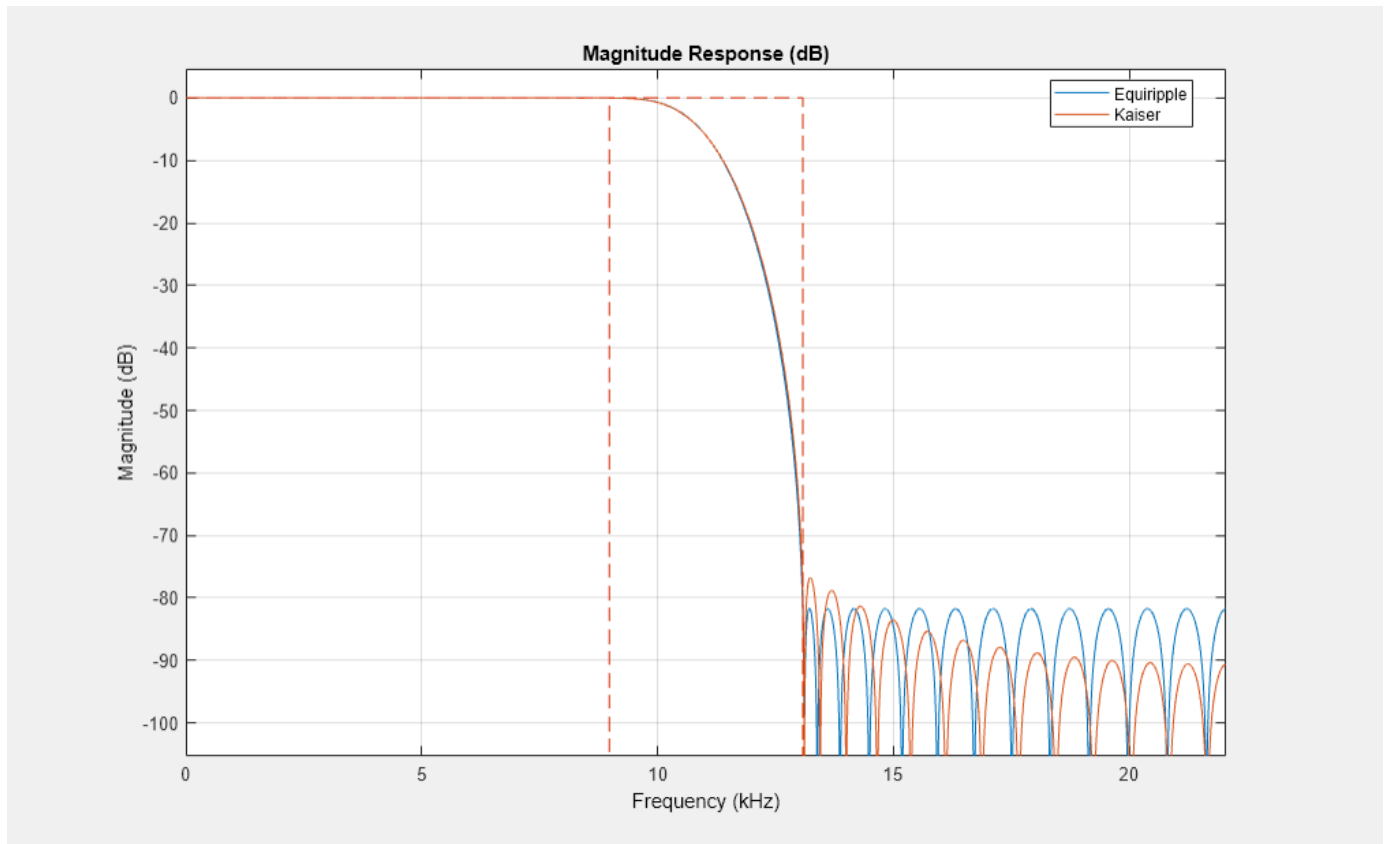
If the filter specifications are tight, say a very high filter order with a very narrow transition width, the filter designed using the "Kaiser" method converges more effectively.

```

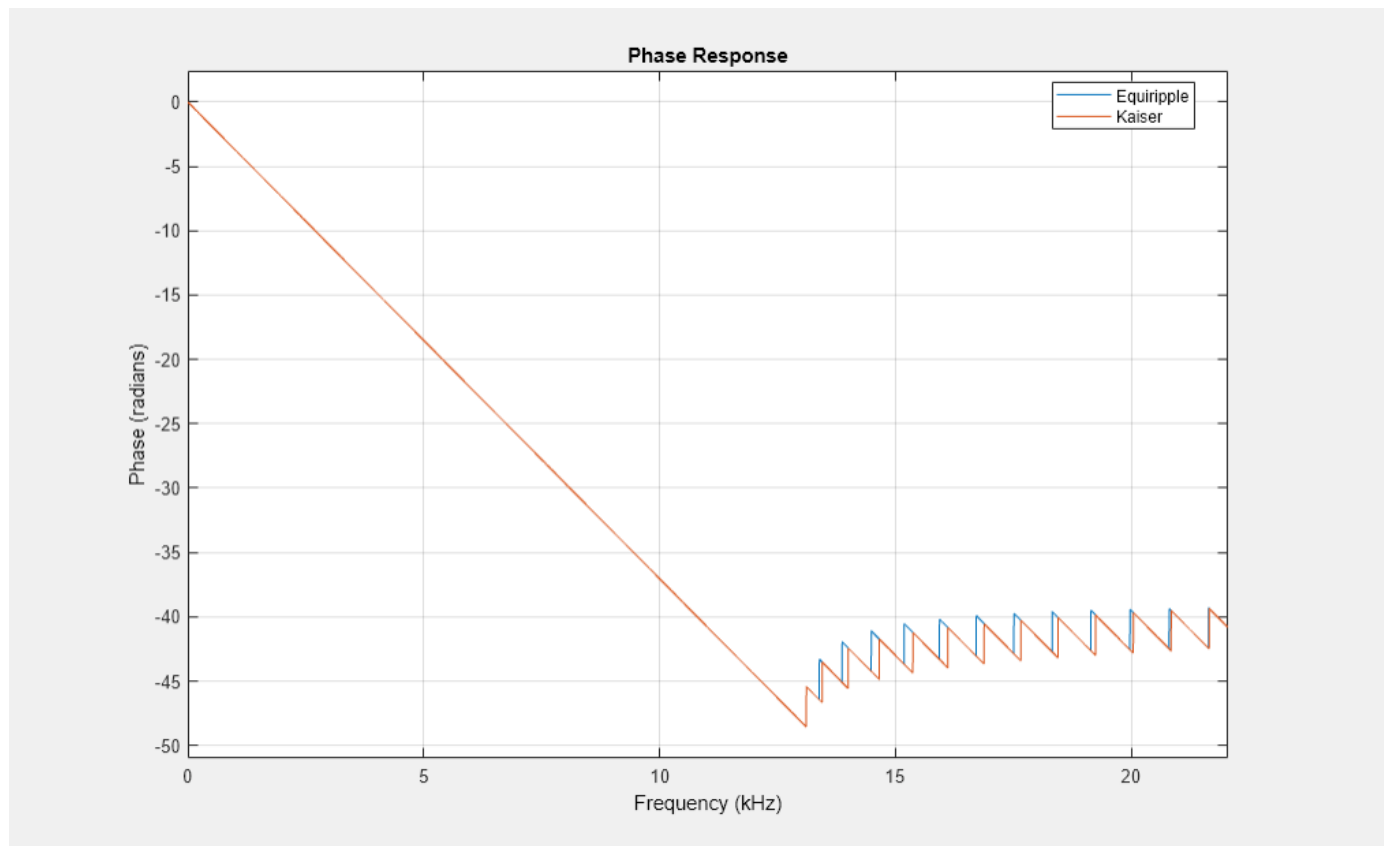
hvftMag = fvtool(firhalfbanddecimEquiripple, firhalfbanddecimKaiser, ...
    Analysis="Magnitude");
legend(hvftMag, {'Equiripple', 'Kaiser'})

```





```
hvftPhase = fvtool(firhalfbanddecimEqui, firhalfbanddecimKaiser, ...  
    Analysis="Phase");  
legend(hvftPhase, {'Equiripple', 'Kaiser'})
```



### Extract Low Frequency Subband From Speech

Use a halfband analysis filter bank and interpolation filter to extract the low frequency subband from a speech signal.

**Note:** The `audioDeviceWriter` System object™ is not supported in MATLAB Online.

Set up the audio file reader, the analysis filter bank, audio device writer, and interpolation filter. The sample rate of the audio data is 22050 Hz. The order of the halfband filter is 52, with a transition width of 2 kHz.

Set the design method to "Auto". This mode chooses one of the filter design methods, equiripple or Kaiser, based on the design parameters of the filter.

```
afr = dsp.AudioFileReader(...
    "speech_dft.mp3",...
    SamplesPerFrame=1024);

filtSpec = "Filter order and transition width";
Order = 52;
TW = 2000;

firhalfbanddecim = dsp.FIRHalfbandDecimator(...
    Specification=filtSpec,...
```

```

FilterOrder=Order,...
TransitionWidth=TW,...
DesignMethod="Auto",...
SampleRate=afr.SampleRate);

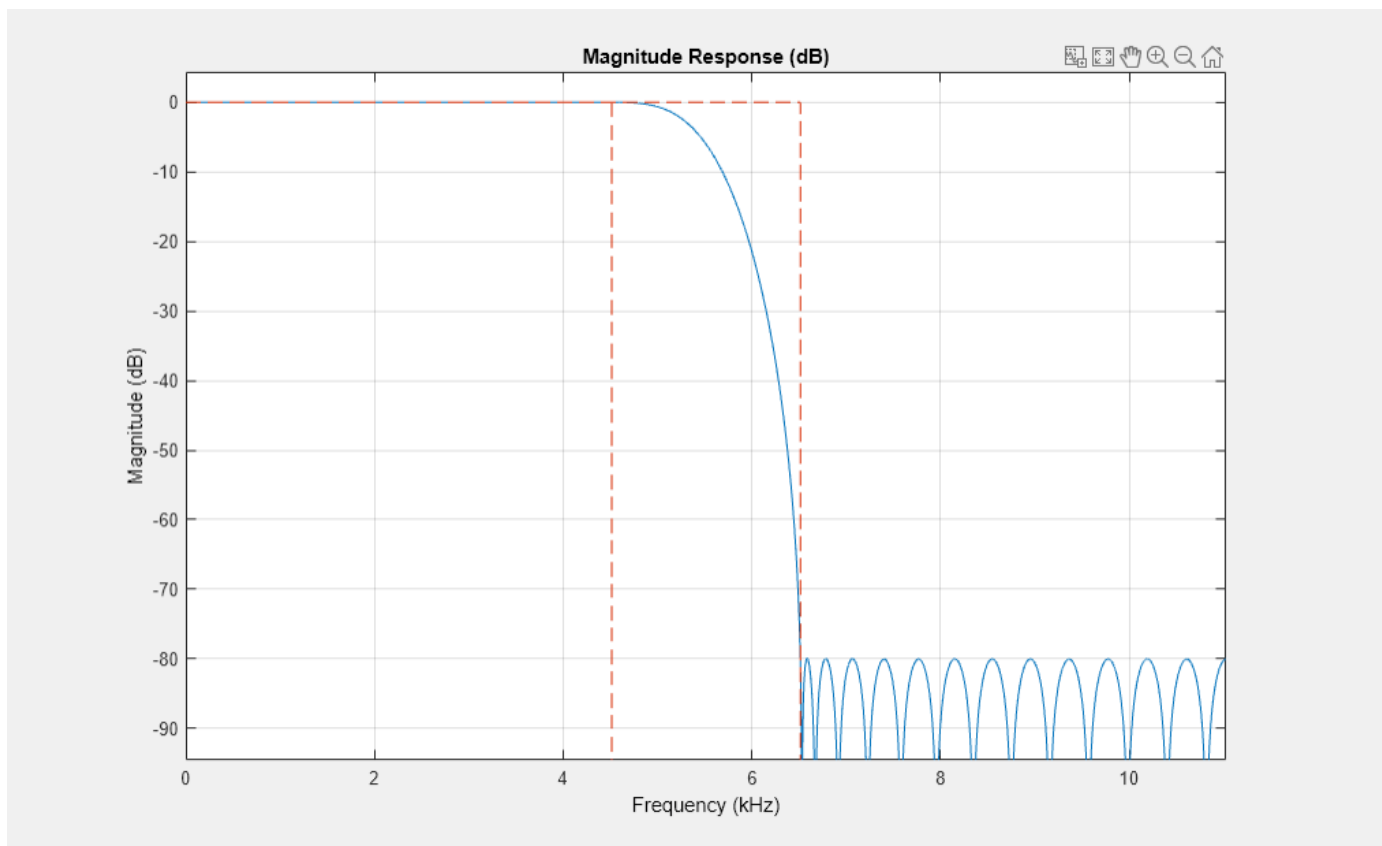
firhalfbandinterp = dsp.FIRHalfbandInterpolator(...
    Specification=filtSpec,...
    FilterOrder=Order,...
    TransitionWidth=TW,...
    DesignMethod="Auto",...
    SampleRate=afr.SampleRate/2);

adw = audioDeviceWriter(SampleRate=afr.SampleRate);

```

View the magnitude response of the halfband filter.

```
fvtool(firhalfbanddecim)
```



Read the speech signal from the audio file in frames of 1024 samples. Filter the speech signal into lowpass and highpass subbands with a halfband frequency of 5512.5 Hz. Reconstruct a lowpass approximation of the speech signal by interpolating the lowpass subband. Play the filtered output.

```

while ~isDone(afr)
    audioframe = afr();
    xlo = firhalfbanddecim(audioframe);
    ylow = firhalfbandinterp(xlo);
    adw(ylow);
end

```

Wait until the audio file is played to the end, then close the input file and release the audio output resource.

```
release(afr);  
release(adw);
```

### Two-Channel Filter Bank

Use a halfband decimator and interpolator to implement a two-channel filter bank. This example uses an audio file input and shows that the power spectrum of the filter bank output does not differ significantly from the input.

**Note:** The `audioDeviceWriter` System object™ is not supported in MATLAB Online.

Set up the audio file reader and device writer. Construct the FIR halfband decimator and interpolator. Finally, set up the spectrum analyzer to display the power spectra of the filter-bank input and output.

```
AF = dsp.AudioFileReader("speech_dft.mp3", SamplesPerFrame=1024);  
AP = audioDeviceWriter(SampleRate=AF.SampleRate);
```

```
filterspec = "Filter order and transition width";  
Order = 52;  
TW = 2000;
```

```
firhalfbanddecim = dsp.FIRHalfbandDecimator(...  
    Specification=filterspec, FilterOrder=Order, ...  
    TransitionWidth=TW, DesignMethod="Auto", ...  
    SampleRate=AF.SampleRate);
```

```
firhalfbandinterp = dsp.FIRHalfbandInterpolator(...  
    Specification=filterspec, FilterOrder=Order, ...  
    TransitionWidth=TW, SampleRate=AF.SampleRate/2, ...  
    DesignMethod="Auto", ...  
    FilterBankInputPort=true);
```

```
SpecAna = spectrumAnalyzer(SampleRate=AF.SampleRate, ...  
    PlotAsTwoSidedSpectrum=false, ...  
    ShowLegend=true, ...  
    ChannelNames={'Input signal', 'Filtered output signal'});
```

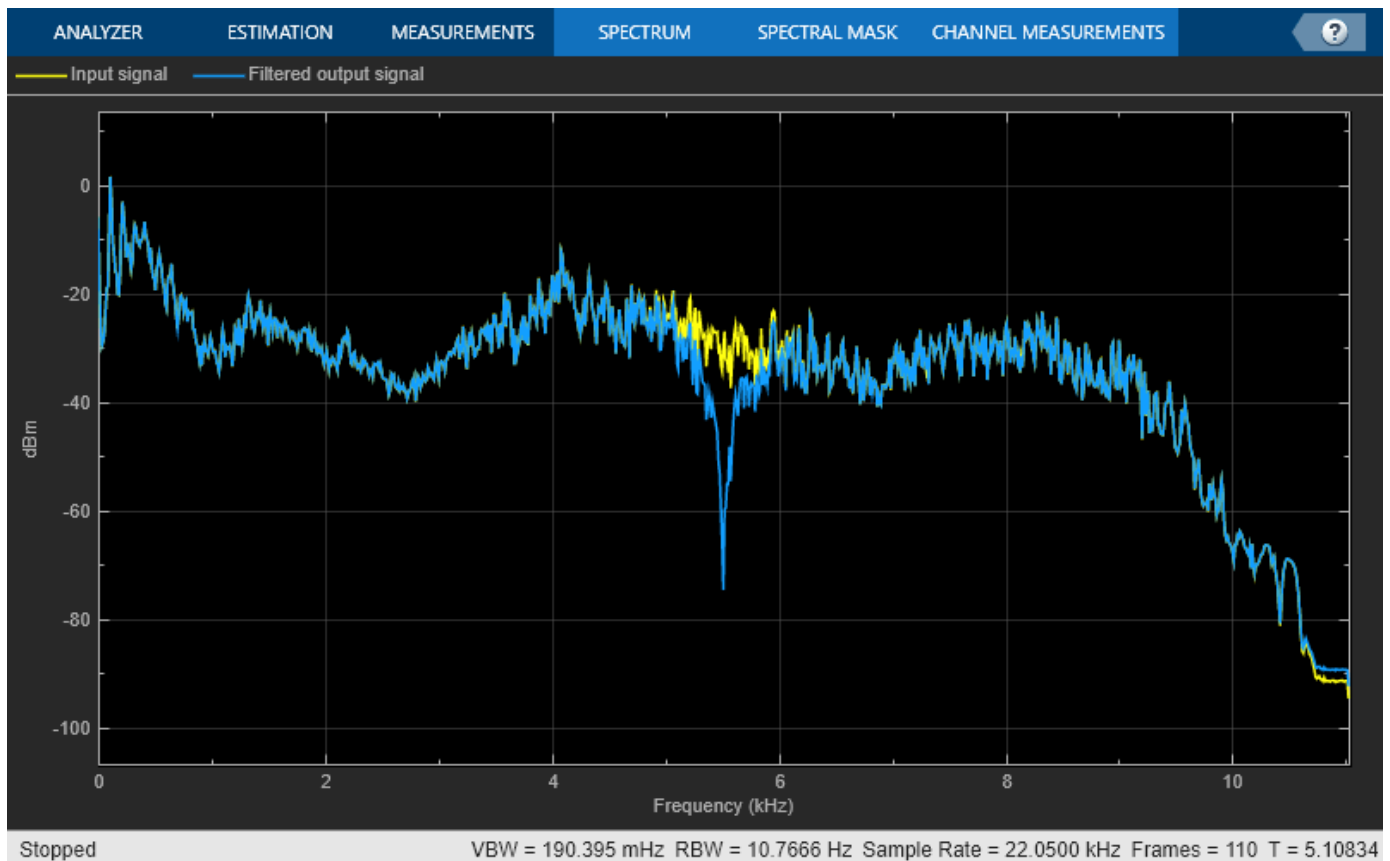
Read the audio 1024 samples at a time. Filter the input to obtain the lowpass and highpass subband signals decimated by a factor of two. This is the analysis filter bank. Use the halfband interpolator as the synthesis filter bank. Display the running power spectrum of the audio input and the output of the synthesis filter bank. Play the output.

```
while ~isDone(AF)  
    audioInput = AF();  
    [xlo, xhigh] = firhalfbanddecim(audioInput);  
    audioOutput = firhalfbandinterp(xlo, xhigh);  
    spectrumInput = [audioInput audioOutput];  
    SpecAna(spectrumInput);  
    AP(audioOutput);
```

```
end
```

```
release(AF);
```

```
release(AP);
release(SpecAna);
```



### Filter Input into Lowpass and Highpass Subbands Using FIR Halfband Decimator

Create a halfband decimator for data sampled at 44.1 kHz. Use a minimum-order design with a transition width of 2 kHz and a stopband attenuation of 60 dB.

Set the design method to "Auto". This mode chooses one of the filter design methods, equiripple or Kaiser, based on the specified filter design parameters.

```
hfirhalfbanddecim = dsp.FIRHalfbandDecimator(...
    Specification="Transition width and stopband attenuation",...
    TransitionWidth=2000,...
    StopbandAttenuation=60,...
    DesignMethod="Auto",...
    SampleRate=44.1e3);
```

Filter a two-channel input into low and highpass subbands.

```
x = randn(1024,2);
[ylow,yhigh] = hfirhalfbanddecim(x);
```

## More About

### Halfband Filters

An ideal lowpass halfband filter is given by

$$h(n) = \frac{1}{2\pi} \int_{-\pi/2}^{\pi/2} e^{j\omega n} d\omega = \frac{\sin(\frac{\pi}{2}n)}{\pi n}.$$

An ideal filter is not realizable because the impulse response is noncausal and not absolutely summable. However, the impulse response of an ideal lowpass filter possesses some important properties that are required in a realizable approximation. The impulse response of an ideal lowpass halfband filter is:

- Equal to 0 for all even-indexed samples.
- Equal to 1/2 at  $n=0$  as shown by L'Hôpital's rule on the continuous-valued equivalent of the discrete-time impulse response

The ideal highpass halfband filter is given by

$$g(n) = \frac{1}{2\pi} \int_{-\pi}^{-\pi/2} e^{j\omega n} d\omega + \frac{1}{2\pi} \int_{\pi/2}^{\pi} e^{j\omega n} d\omega.$$

Evaluating the preceding integral gives the following impulse response

$$g(n) = \frac{\sin(\pi n)}{\pi n} - \frac{\sin(\frac{\pi}{2}n)}{\pi n}.$$

The impulse response of an ideal highpass halfband filter is:

- Equal to 0 for all even-indexed samples
- Equal to 1/2 at  $n=0$

The FIR halfband decimator uses a causal FIR approximation to the ideal halfband response, which is based on minimizing the  $\ell^\infty$  norm of the error (minimax). See "Algorithms" on page 4-529 for more information.

### Kaiser Window

The coefficients of a Kaiser window are computed from this equation:

$$w(n) = \frac{I_0\left(\beta \sqrt{1 - \left(\frac{n - N/2}{N/2}\right)^2}\right)}{I_0(\beta)}, \quad 0 \leq n \leq N,$$

where  $I_0$  is the zeroth-order modified Bessel function of the first kind.

To obtain a Kaiser window that represents an FIR filter with stopband attenuation of  $\alpha$  dB, use this  $\beta$ .

$$\beta = \begin{cases} 0.1102(\alpha - 8.7), & \alpha > 50 \\ 0.5842(\alpha - 21)^{0.4} + 0.07886(\alpha - 21), & 50 \geq \alpha \geq 21 \\ 0, & \alpha < 21 \end{cases}$$

The filter order  $n$  is given by:

$$n = \frac{\alpha - 7.95}{2.285(\Delta\omega)}$$

where  $\Delta\omega$  is the transition width.

## Algorithms

### Filter Design Method

The FIR halfband decimator algorithm uses the equiripple or the Kaiser window method to design the FIR halfband filter. When the design constraints are tight, such as very high stopband attenuation or very narrow transition width, use the Kaiser window method. When the design constraints are not tight, use the equiripple method. If you are not sure of which method to use, set the design method to `Auto`. In this mode, the algorithm automatically chooses a design method that optimally meets the specified filter constraints.

### Halfband Equiripple Design

In the equiripple method, the algorithm uses a minimax (minimize the maximum error) FIR design to design a fullband linear phase filter with the desired specifications. The algorithm upsamples a fullband filter to replace the even-indexed samples of the filter with zeros and creates a halfband filter. It then sets the filter tap corresponding to the group delay of the filter in samples to  $1/2$ . This yields a causal linear-phase FIR filter approximation to the ideal halfband filter defined in “Halfband Filters” on page 4-528. See [1] for a description of this filter design method using the Remez exchange algorithm. Since you can design a filter using this approximation method with a constant ripple both in the passband and stopband, the filter is also known as the equiripple filter.

### Kaiser Window Design

In the Kaiser window method, the algorithm first truncates the ideal halfband filter defined in “Halfband Filters” on page 4-528, then it applies a Kaiser window defined in “Kaiser Window” on page 4-528. This yields a causal linear-phase FIR filter approximation to the ideal halfband filter.

For more information on designing FIR halfband filters, see “FIR Halfband Filter Design”.

### Polyphase Implementation with Halfband Filters

The FIR halfband decimator uses an efficient polyphase implementation for halfband filters when you filter the input signal. The chief advantage of the polyphase implementation is that you can downsample the signal prior to filtering. This allows you to filter at the lower sampling rate.

Splitting a filter’s impulse response  $h(n)$  into two polyphase components results in an even polyphase component with  $z$ -transform of

$$H_0(z) = \sum_n h(2n)z^{-n},$$

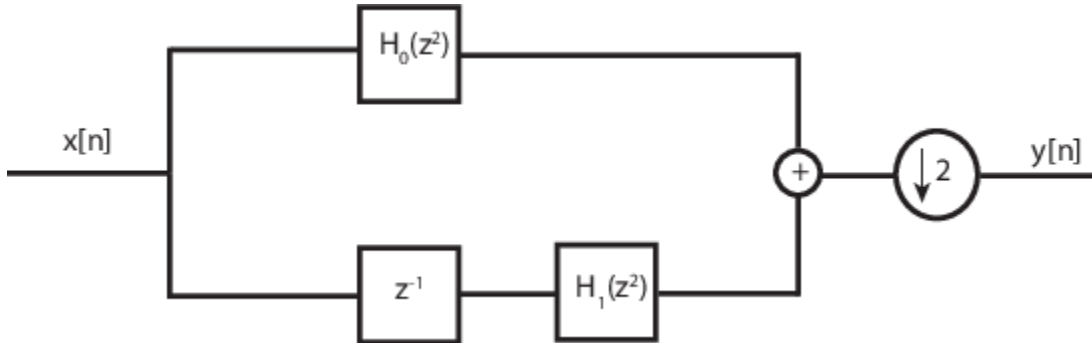
and an odd polyphase component with  $z$ -transform of

$$H_1(z) = \sum_n h(2n + 1)z^{-n}.$$

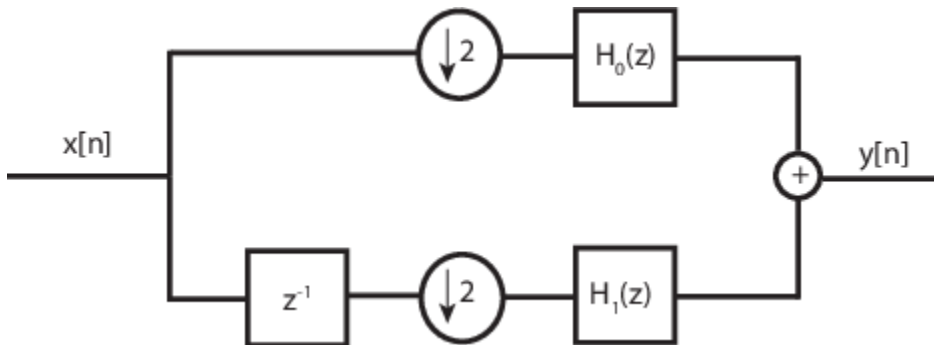
The  $z$ -transform of the filter can be written in terms of the even and odd polyphase components as

$$H(z) = H_0(z^2) + z^{-1}H_1(z^2).$$

You can represent filtering the input signal and then downsampling it by 2 using this figure.

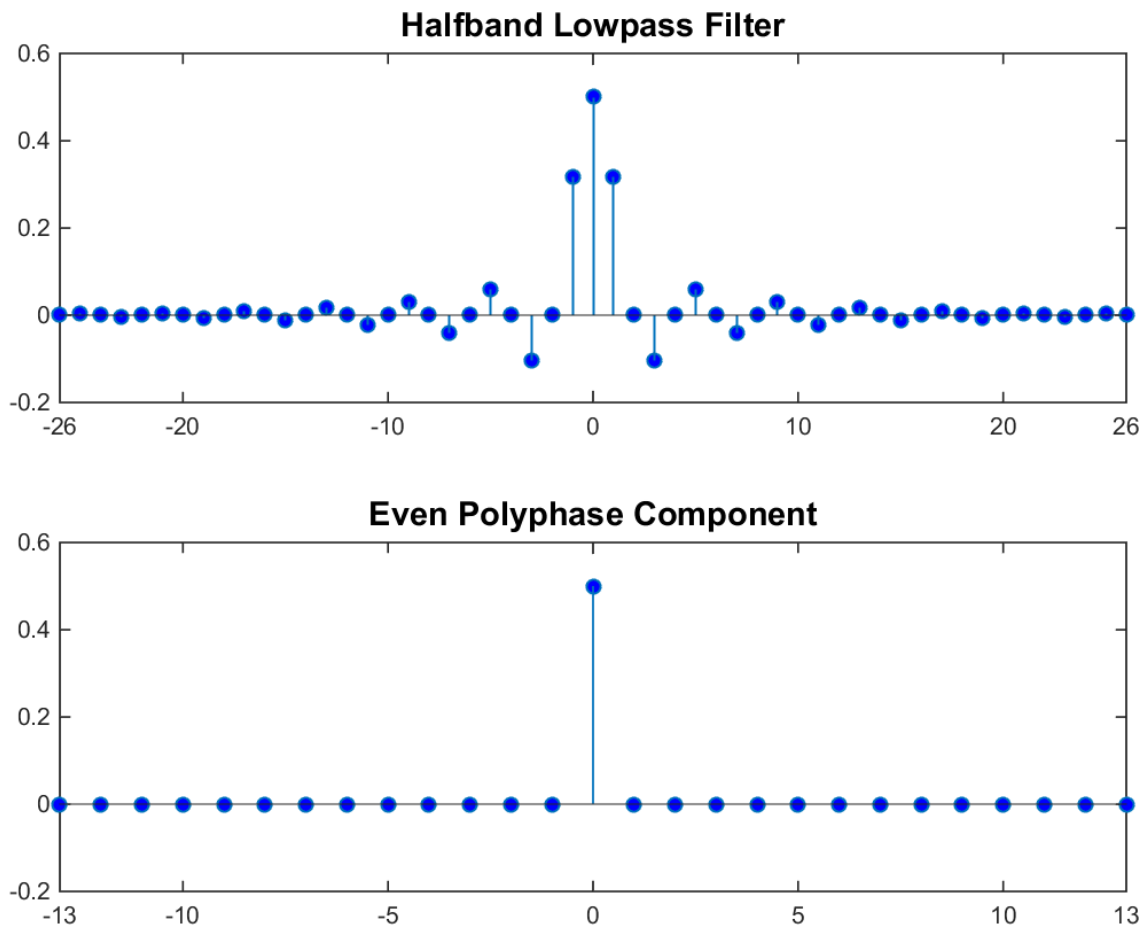


Using the multirate noble identity for downsampling, you can move the downsampling operation before the filtering. This allows you to filter at the lower rate.



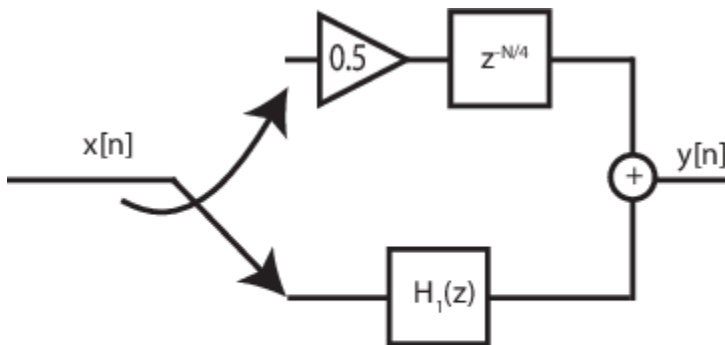
For a halfband filter, the only nonzero coefficient in the even polyphase component is the coefficient corresponding to  $z^0$ . Implementing the halfband filter as a causal FIR filter shifts the nonzero coefficient to approximately  $z^{-N/4}$  where  $N$  is the number of filter taps. This process is illustrated in the following figure.





The top plot shows a halfband filter of order 52. The bottom plot shows the even polyphase component. Both of these filters are noncausal. Delaying the even polyphase component by 13 samples creates a causal FIR filter.

To efficiently implement the halfband decimator, the algorithm replaces the delay block and downsampling operator with a commutator switch. This is illustrated in the following figure where one polyphase component is replaced by a gain and delay.



The commutator switch takes input samples from a single branch and supplies every other sample to one of the two polyphase components for filtering. This halves the sampling rate of the input signal. Which polyphase component reduces to a simple delay depends on whether the half order of the filter is even or odd. This is because the delay required to make the even polyphase component causal can be odd or even depending on the filter half order.

To confirm this behavior, run the following code in the MATLAB command prompt and inspect the polyphase components of the following filters.

```
filterspec = "Filter order and stopband attenuation";
halfOrderEven = dsp.FIRHalfbandDecimator(Specification=filterspec,...
    FilterOrder=64,StopbandAttenuation=80,DesignMethod="Auto");
halfOrderOdd = dsp.FIRHalfbandDecimator(Specification=filterspec,...
    FilterOrder=54,StopbandAttenuation=80,DesignMethod="Auto");
polyphase(halfOrderEven)
polyphase(halfOrderOdd)
```

To summarize, the FIR halfband decimator:

- Decimates the input prior to filtering and filters the even and odd polyphase components of the input separately with the even and odd polyphase components of the filter.
- Exploits the fact that one filter polyphase component is a simple delay for a halfband filter.

## Version History

Introduced in R2014b

## References

- [1] Harris, F.J. *Multirate Signal Processing for Communication Systems*, Prentice Hall, 2004, pp. 208–209.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

This object supports code generation for ARM Cortex-M and ARM Cortex-A processors.

## See Also

### Functions

freqz | fvtool | info | cost | coeffs | polyphase

### Objects

dsp.FIRHalfbandInterpolator | dsp.IIRHalfbandDecimator |  
dsp.DyadicAnalysisFilterBank | dsp.Channelizer

### Blocks

FIR Halfband Decimator | FIR Halfband Interpolator | IIR Halfband Decimator | Dyadic Analysis Filter Bank

### Topics

“FIR Halfband Filter Design”

“Analysis Methods for Filter System Objects” on page 3-2

# **dsp.FIRHalfbandInterpolator**

**Package:** dsp

Halfband interpolator

## **Description**

The `dsp.FIRHalfbandInterpolator` System object performs efficient polyphase interpolation of the input signal using an upsampling factor of two. You can use `dsp.FIRHalfbandInterpolator` to implement the synthesis portion of a two-band filter bank to synthesize a signal from lowpass and highpass subbands. `dsp.FIRHalfbandInterpolator` uses an FIR equiripple design or a Kaiser window design to construct the halfband filters and a polyphase implementation to filter the input. For more details, see “Algorithms” on page 4-547.

To upsample and interpolate your data:

- 1 Create the `dsp.FIRHalfbandInterpolator` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## **Creation**

### **Syntax**

```
firhalfbandinterp = dsp.FIRHalfbandInterpolator  
firhalfbandinterp = dsp.FIRHalfbandInterpolator(Name=Value)
```

### **Description**

`firhalfbandinterp = dsp.FIRHalfbandInterpolator` returns a FIR halfband interpolation filter, `firhalfbandinterp`, with the default settings. Under the default settings, the System object upsamples and interpolates the input data using a halfband frequency of 11025 Hz, a transition width of 4.1 kHz, and a stopband attenuation of 80 dB. The design method is set to "Auto".

`firhalfbandinterp = dsp.FIRHalfbandInterpolator(Name=Value)` returns a halfband interpolator, with additional properties specified by one or more `Name=Value` pair arguments.

Example: `firhalfbandinterp = dsp.FIRHalfbandInterpolator(Specification="Filter order and stopband attenuation")` creates an FIR halfband interpolator object with filter order set to 52 and stopband attenuation set to 80 dB.

## **Properties**

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see *System Design in MATLAB Using System Objects*.

### Specification — Filter design parameters

"Transition width and stopband attenuation" (default) | "Filter order and stopband attenuation" | "Filter order and transition width" | "Coefficients"

Filter design parameters, specified as a character vector. When you set `Specification` to one of the following, you choose two of the three available design parameters to design the FIR Halfband filter.

- "Transition width and stopband attenuation" -- Transition width and stopband attenuation are the design parameters.
- "Filter order and stopband attenuation" -- Filter order and stopband attenuation are the design parameters.
- "Filter order and transition width" -- Filter order and transition width are the design parameters.

The filter is designed using optimal equiripple filter design method.

When you set `Specification` to "Coefficients", you specify the halfband filter coefficients directly through the `Numerator` property.

### FilterOrder — Filter order

52 (default) | even positive integer

Filter order, specified as an even positive integer.

#### Dependencies

This property applies when you set `Specification` to either "Filter order and stopband attenuation" or "Filter order and transition width".

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### StopbandAttenuation — Stopband attenuation

80 (default) | positive real scalar

Stopband attenuation in dB, specified as a positive real scalar.

#### Dependencies

This property applies when you set `Specification` to either "Filter order and stopband attenuation" or "Transition width and stopband attenuation".

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### TransitionWidth — Transition width

4100 (default) | positive real scalar

Transition width in Hz, specified as a positive real scalar. The value of the transition width in Hz must be less than 1/2 the input sample rate.

#### Dependencies

This property applies when you set `Specification` to either "Transition width and stopband attenuation" or "Filter order and transition width".

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**Numerator — FIR halfband filter coefficients**

`2*firhalfband("minorder",0.407,1e-4)` (default) | row vector

FIR halfband filter coefficients, specified as a row vector. The coefficients must comply with the FIR halfband impulse response format. For details on this format, see “Halfband Filters” on page 4-546 and “FIR Halfband Filter Design”. If half the order of the filter,  $(\text{length}(\text{Numerator}) - 1)/2$  is even, every other coefficient starting from the first coefficient must be a zero except for the center coefficient which must be a 1.0. If half the order of the filter is odd, the sequence of alternating zeros with a 1.0 at the center starts at the second coefficient.

**Dependencies**

This property applies when you set `Specification` to “Coefficients”.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**DesignMethod — Filter design method**

“Auto” (default) | “Equiripple” | “Kaiser”

Specify the filter design method as one of the following:

- “Auto” -- The algorithm automatically chooses the filter design method depending on the filter design parameters. The algorithm uses the equiripple or the Kaiser window method to design the filter.

If the design constraints are very tight, such as very high stopband attenuation or very narrow transition width, then the algorithm automatically chooses the Kaiser method, as this method is optimal for designing filters with very tight specifications. However, if the design constraints are not tight, then the algorithm chooses the equiripple method.

When you set the `DesignMethod` property to “Auto”, you can determine the method used by the algorithm by examining the passband and stopband ripple characteristics of the designed filter. If the object used the equiripple method, the passband and stopband ripples of the designed filter have a constant amplitude in the frequency response. If the filter design method the object chooses in the “Auto” mode is not suitable for your application, manually specify the `DesignMethod` as “Equiripple” or “Kaiser”.

- “Equiripple” -- The algorithm uses the equiripple method.
- “Kaiser” -- The algorithm uses the Kaiser window method.

For more details on these two methods, see “Algorithms” on page 4-547.

**Dependencies**

To enable this property, set `Specification` to “Transition width and stopband attenuation”, “Filter order and stopband attenuation”, or “Filter order and transition width”.

Data Types: `char` | `string`

**SampleRate — Input sample rate**

44100 (default) | positive real scalar

Input sample rate in Hz, specified as a positive real scalar. The input sample rate defaults to 44100 Hz. If you specify a transition width as one of your filter design parameters, the transition width cannot exceed 1/2 the input sample rate.

Data Types: `single` | `double`

### **FilterBankInputPort — Synthesis filter bank**

`false` (default) | `true`

Synthesis filter bank, specified as either `false` or `true`. If this property is `false`, `dsp.FIRHalfbandInterpolator` is an interpolation filter for a single vector- or matrix-valued input when you call the algorithm. If this property is `true`, `dsp.FIRHalfbandInterpolator` is a synthesis filter bank and the algorithm accepts two inputs, the lowpass and highpass subbands to synthesize.

### **Fixed-Point Properties**

#### **CoefficientsDataType — Word and fraction lengths of coefficients**

`numericType(1,16)` (default) | `numericType` object

Word and fraction lengths of coefficients, specified as a signed or unsigned `numericType` object. The default, `numericType(1,16)` corresponds to a signed numeric type object with 16-bit coefficients and a fraction length determined based on the coefficient values, to give the best possible precision.

This property is not tunable.

Word length of the output is same as the word length of the input. Fraction length of the output is computed such that the entire dynamic range of the output can be represented without overflow. For details on how the fraction length of the output is computed, see “Fixed-Point Precision Rules for Avoiding Overflow in FIR Filters”.

#### **RoundingMethod — Rounding method for output fixed-point operations**

`"Floor"` (default) | `"Ceiling"` | `"Convergent"` | `"Nearest"` | `"Round"` | `"Simplest"` | `"Zero"`

Rounding method for output fixed-point operations, specified as a character vector. For more information on the rounding modes, see “Precision and Range”.

## **Usage**

### **Syntax**

```
y = firhalfbandinterp(x1)
y = firhalfbandinterp(x1,x2)
```

### **Description**

`y = firhalfbandinterp(x1)` upsamples by two and interpolates the input signal `x1` using the FIR halfband interpolator, `firhalfbandinterp`.

`y = firhalfbandinterp(x1,x2)` implements a halfband synthesis filter bank for the inputs `x1` and `x2`. `x1` is the lowpass output of a halfband analysis filter bank and `x2` is the highpass output of a halfband analysis filter bank. `dsp.FIRHalfbandInterpolator` implements a synthesis filter bank only when the `FilterBankInputPort` property is set to `true`.

## Input Arguments

### **x1 — Data input**

column vector | matrix

Data input to the FIR halfband interpolator, specified as a column vector or a matrix. This signal is the lowpass output of a halfband analysis filter bank. If the input signal is a matrix, each column of the matrix is treated as an independent channel.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

### **x2 — Second data input**

column vector | matrix

Second data input to the synthesis filter bank, specified as a column vector or a matrix. This signal is the highpass output of a halfband analysis filter bank. If the input signal is a matrix, each column of the matrix is treated as an independent channel.

The size, data type, and complexity of both the inputs must be the same.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

## Output Arguments

### **y — Output of interpolator**

column vector | matrix

Output of the interpolator, returned as a column vector or a matrix. The number of rows in the interpolator output is twice the number of rows in the input signal.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Specific to `dsp.FIRHalfbandInterpolator`

|                        |  |
|------------------------|--|
| <code>freqz</code>     | Frequency response of discrete-time filter System object     |
| <code>fvtool</code>    | Visualize frequency response of DSP filters                  |
| <code>info</code>      | Information about filter System object                       |
| <code>cost</code>      | Estimate cost of implementing filter System object           |
| <code>coeffs</code>    | Returns the filter System object coefficients in a structure |
| <code>polyphase</code> | Polyphase decomposition of multirate filter                  |

## Common to All System Objects

|                   |                             |
|-------------------|-----------------------------|
| <code>step</code> | Run System object algorithm |
|-------------------|-----------------------------|



release Release resources and allow changes to System object property values and input characteristics  
 reset Reset internal states of System object

## Examples

### Impulse and Frequency Response of Halfband Interpolation Filter

Create two lowpass halfband interpolation filters for upsampling data to 44.1 kHz. The design method in the first filter is set to "Equiripple" and in the second filter is set to "Kaiser".

Specify a filter order of 52 and a transition width of 4.1 kHz.

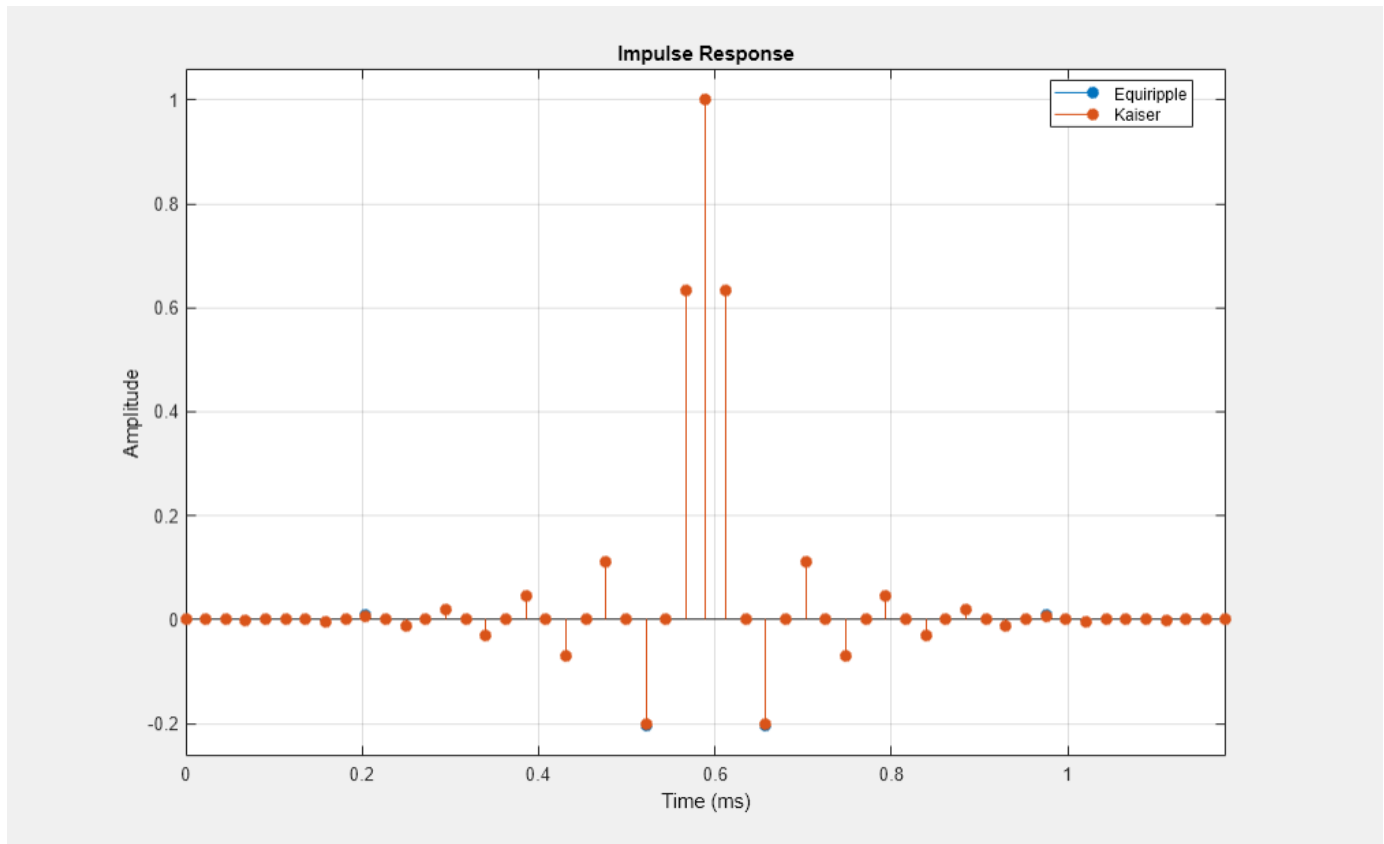
```
Fs = 44.1e3;
InputSampleRate = Fs/2;
Order = 52;
TW = 4.1e3;
filterspec = "Filter order and transition width";

firhalfbandinterpEqui = dsp.FIRHalfbandInterpolator(...
    Specification=filterspec,FilterOrder=Order,...
    TransitionWidth=TW,DesignMethod="Equiripple",...
    SampleRate=InputSampleRate);

firhalfbandinterpKaiser = dsp.FIRHalfbandInterpolator(...
    Specification=filterspec,FilterOrder=Order,...
    TransitionWidth=TW,DesignMethod="Kaiser",...
    SampleRate=InputSampleRate);
```

Plot the impulse response of both the filters. The zeroth-order coefficient is delayed 26 samples, which is equal to the group delay of the filter. This yields a causal halfband filter.

```
hfvt = fvtool(firhalfbandinterpEqui,firhalfbandinterpKaiser,...
    Analysis="Impulse");
legend(hfvt,{'Equiripple','Kaiser'})
```



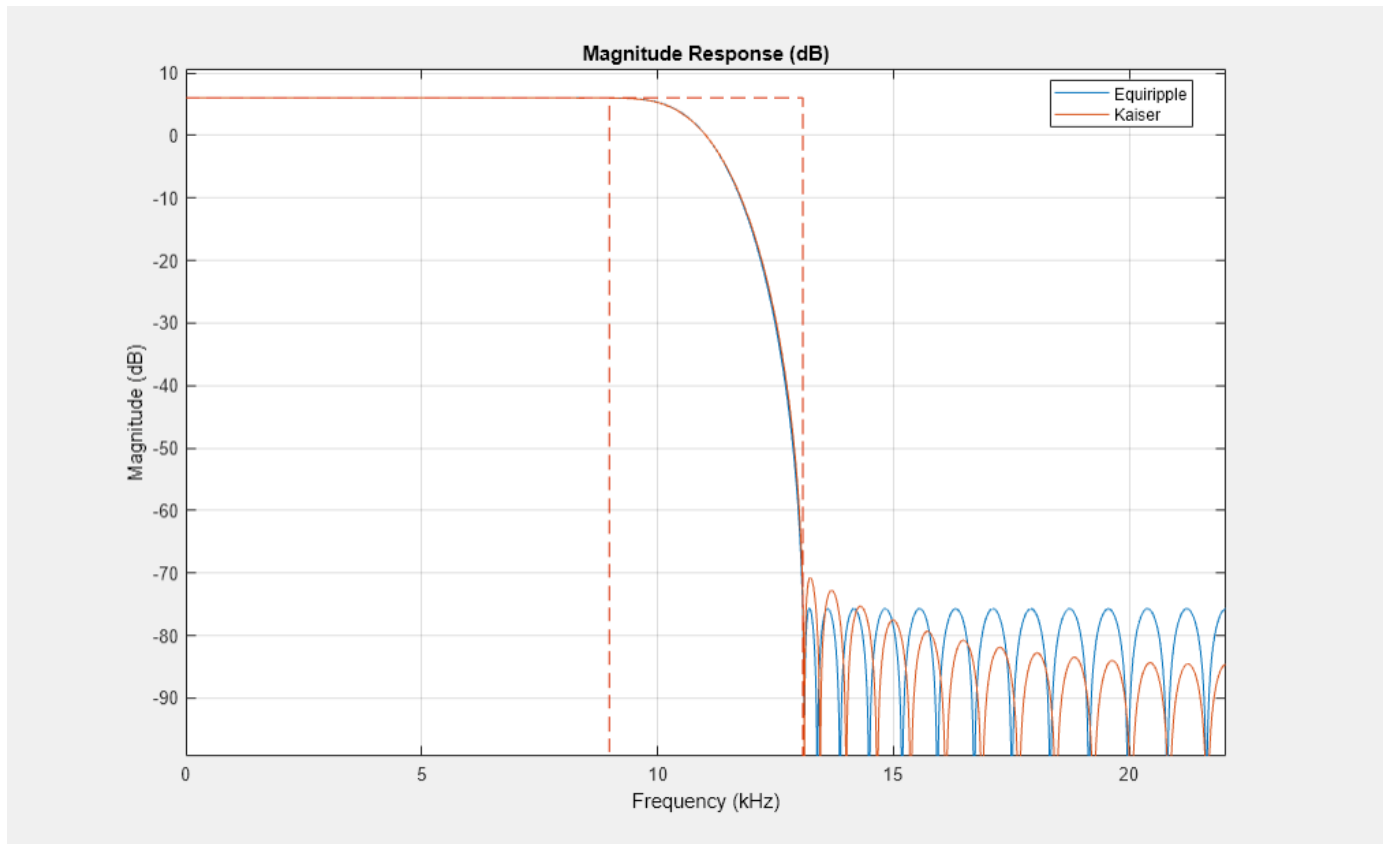
Plot the magnitude and phase response.

If the filter specifications are tight, say a very high filter order with a very narrow transition width, the filter designed using the "Kaiser" method converges more effectively.

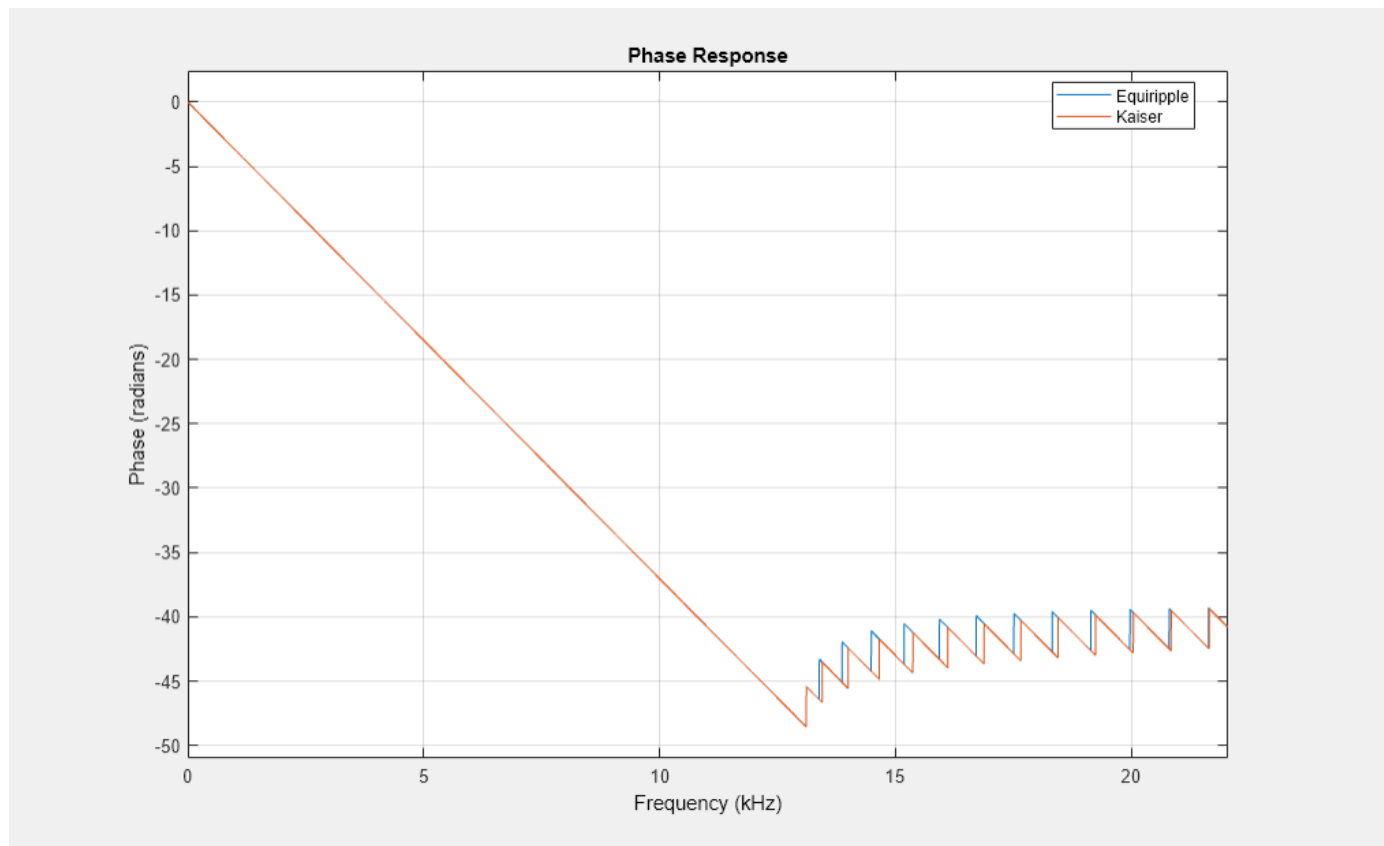
```

hvftMag = fvtool(firhalfbandinterpEqui,firhalfbandinterpKaiser,...
    Analysis="Magnitude");
legend(hvftMag,{'Equiripple','Kaiser'})

```



```
hvftPhase = fvtool(firhalfbandinterpEqui, firhalfbandinterpKaiser, ...  
    Analysis="Phase");  
legend(hvftPhase, {'Equiripple', 'Kaiser'})
```



### Extract Low Frequency Subband From Speech

Use a halfband analysis filter bank and interpolation filter to extract the low frequency subband from a speech signal.

**Note:** The `audioDeviceWriter` System object™ is not supported in MATLAB Online.

Set up the audio file reader, the analysis filter bank, audio device writer, and interpolation filter. The sample rate of the audio data is 22050 Hz. The order of the halfband filter is 52, with a transition width of 2 kHz.

Set the design method to "Auto". This mode chooses one of the filter design methods, equiripple or Kaiser, based on the design parameters of the filter.

```

afr = dsp.AudioFileReader(...
    "speech_dft.mp3",...
    SamplesPerFrame=1024);

filtSpec = "Filter order and transition width";
Order = 52;
TW = 2000;

firhalfbanddecim = dsp.FIRHalfbandDecimator(...
    Specification=filtSpec,...

```

```

    FilterOrder=Order,...
    TransitionWidth=TW,...
    DesignMethod="Auto",...
    SampleRate=afr.SampleRate);

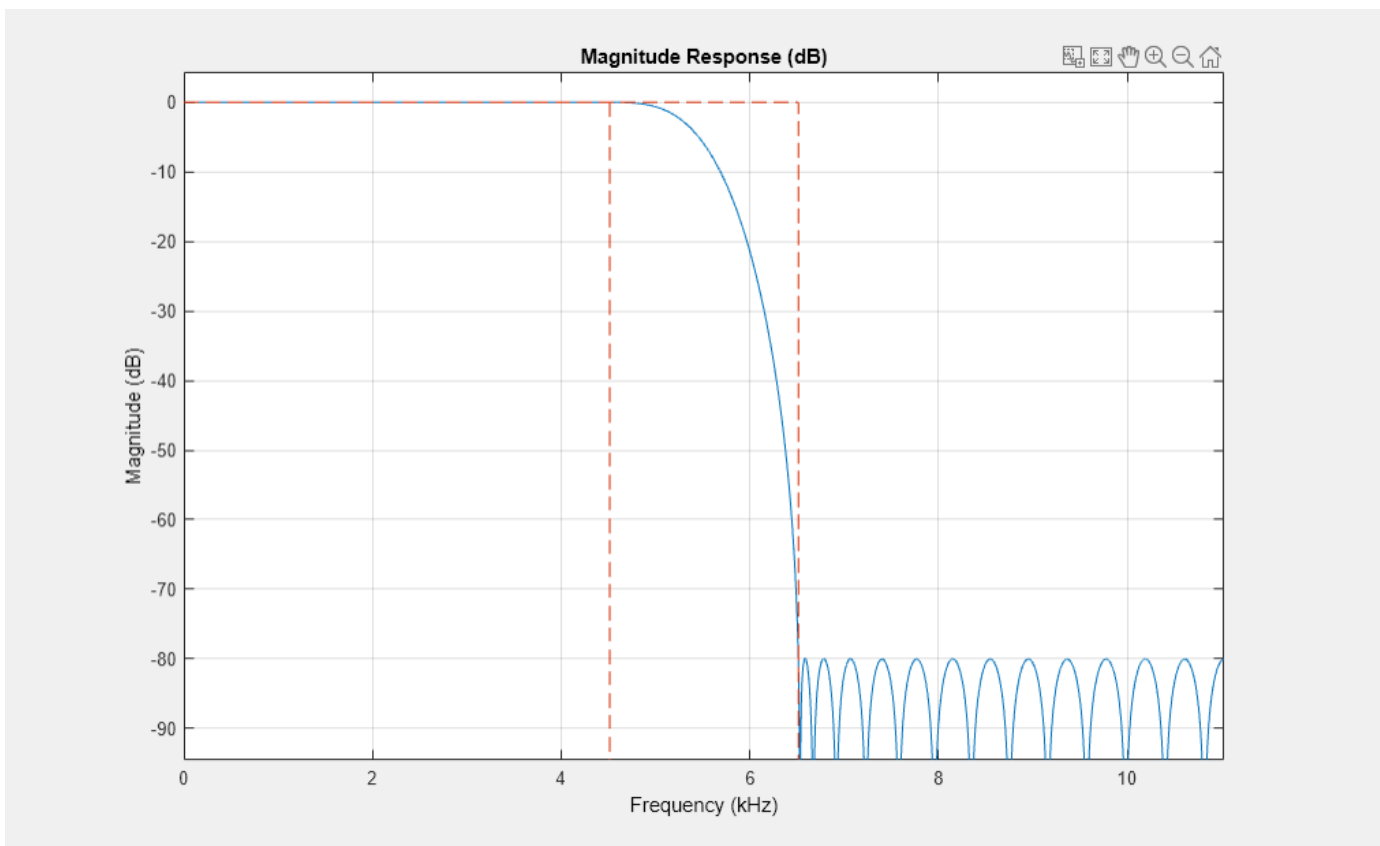
firhalfbandinterp = dsp.FIRHalfbandInterpolator(...
    Specification=filtSpec,...
    FilterOrder=Order,...
    TransitionWidth=TW,...
    DesignMethod="Auto",...
    SampleRate=afr.SampleRate/2);

adw = audioDeviceWriter(SampleRate=afr.SampleRate);

```

View the magnitude response of the halfband filter.

```
fvtool(firhalfbanddecim)
```



Read the speech signal from the audio file in frames of 1024 samples. Filter the speech signal into lowpass and highpass subbands with a halfband frequency of 5512.5 Hz. Reconstruct a lowpass approximation of the speech signal by interpolating the lowpass subband. Play the filtered output.

```

while ~isDone(afr)
    audioframe = afr();
    xlo = firhalfbanddecim(audioframe);
    ylow = firhalfbandinterp(xlo);
    adw(ylow);
end

```

Wait until the audio file is played to the end, then close the input file and release the audio output resource.

```
release(afr);  
release(adw);
```

### Two-Channel Filter Bank

Use a halfband decimator and interpolator to implement a two-channel filter bank. This example uses an audio file input and shows that the power spectrum of the filter bank output does not differ significantly from the input.

**Note:** The `audioDeviceWriter` System object™ is not supported in MATLAB Online.

Set up the audio file reader and device writer. Construct the FIR halfband decimator and interpolator. Finally, set up the spectrum analyzer to display the power spectra of the filter-bank input and output.

```
AF = dsp.AudioFileReader("speech_dft.mp3", SamplesPerFrame=1024);  
AP = audioDeviceWriter(SampleRate=AF.SampleRate);
```

```
filterspec = "Filter order and transition width";  
Order = 52;  
TW = 2000;
```

```
firhalfbanddecim = dsp.FIRHalfbandDecimator(...  
    Specification=filterspec, FilterOrder=Order, ...  
    TransitionWidth=TW, DesignMethod="Auto", ...  
    SampleRate=AF.SampleRate);
```

```
firhalfbandinterp = dsp.FIRHalfbandInterpolator(...  
    Specification=filterspec, FilterOrder=Order, ...  
    TransitionWidth=TW, SampleRate=AF.SampleRate/2, ...  
    DesignMethod="Auto", ...  
    FilterBankInputPort=true);
```

```
SpecAna = spectrumAnalyzer(SampleRate=AF.SampleRate, ...  
    PlotAsTwoSidedSpectrum=false, ...  
    ShowLegend=true, ...  
    ChannelNames={'Input signal', 'Filtered output signal'});
```

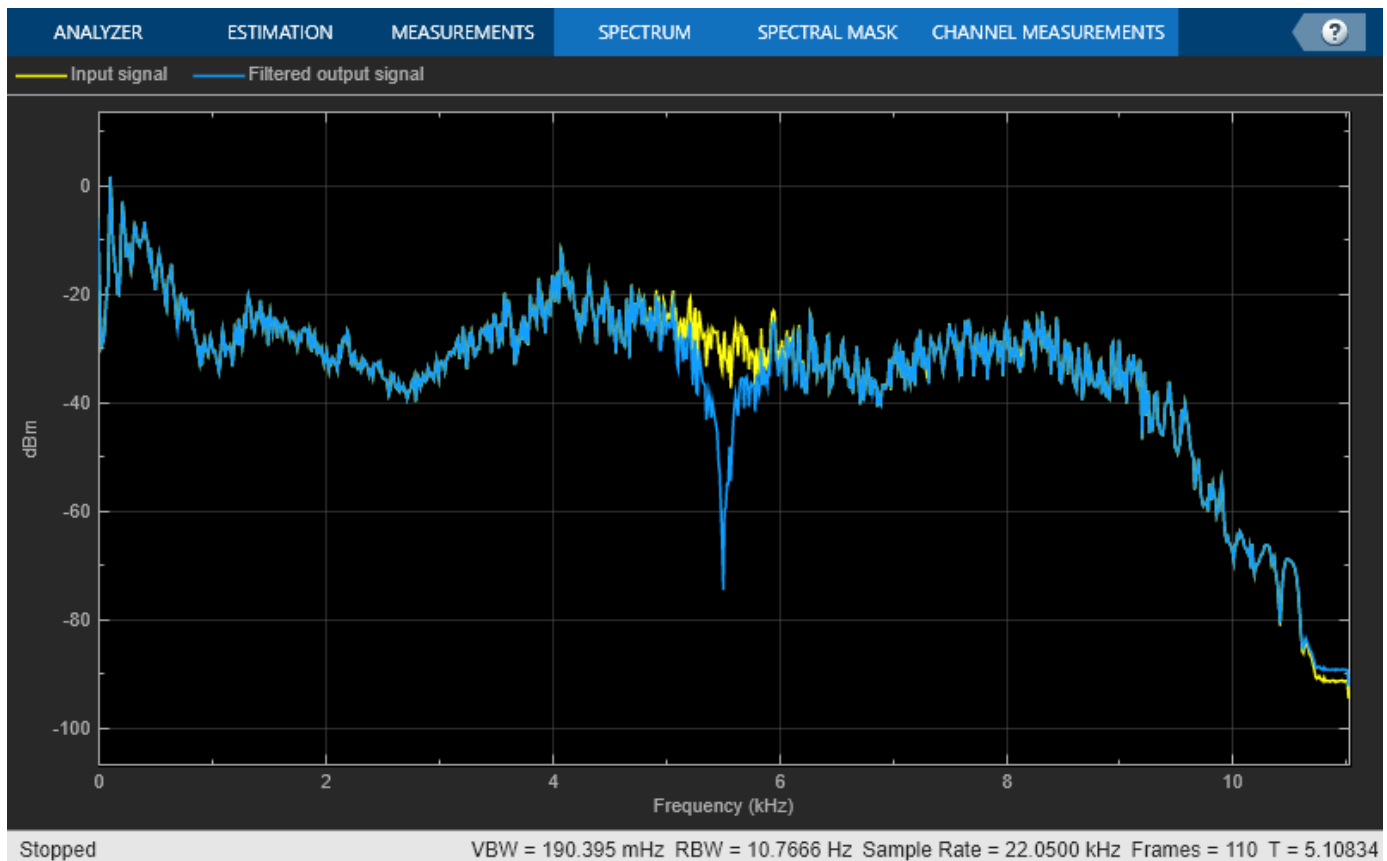
Read the audio 1024 samples at a time. Filter the input to obtain the lowpass and highpass subband signals decimated by a factor of two. This is the analysis filter bank. Use the halfband interpolator as the synthesis filter bank. Display the running power spectrum of the audio input and the output of the synthesis filter bank. Play the output.

```
while ~isDone(AF)  
    audioInput = AF();  
    [xlo, xhigh] = firhalfbanddecim(audioInput);  
    audioOutput = firhalfbandinterp(xlo, xhigh);  
    spectrumInput = [audioInput audioOutput];  
    SpecAna(spectrumInput);  
    AP(audioOutput);
```

```
end
```

```
release(AF);
```

```
release(AP);
release(SpecAna);
```



### Upsample and Interpolate Multichannel Input Using FIR Halfband Interpolator

Create a half-band interpolation filter for data sampled at 44.1 kHz. The filter order is 52 with a transition width of 4.1 kHz. Use the filter to upsample and interpolate a multichannel input.

Set the design method to "Auto". This mode chooses one of the filter design methods, equiripple or Kaiser, based on the specified filter design parameters.

```
Fs = 44.1e3;
filterspec = "Filter order and transition width";
Order = 52;
TW = 4.1e3;
firhalfbandinterp = dsp.FIRHalfbandInterpolator(...
    Specification=filterspec,...
    FilterOrder=Order,...
    TransitionWidth=TW,...
    DesignMethod="Auto",...
    SampleRate=Fs);

x = randn(1024,4);
y = firhalfbandinterp(x);
```

## More About

### Halfband Filters

An ideal lowpass halfband filter is given by

$$h(n) = \frac{1}{2\pi} \int_{-\pi/2}^{\pi/2} e^{j\omega n} d\omega = \frac{\sin(\frac{\pi}{2}n)}{\pi n}.$$

An ideal filter is not realizable because the impulse response is noncausal and not absolutely summable. However, the impulse response of an ideal lowpass filter possesses some important properties that are required in a realizable approximation. The impulse response of an ideal lowpass halfband filter is:

- Equal to 0 for all even-indexed samples
- Equal to 1/2 at  $n=0$  as shown by L'Hôpital's rule on the continuous-valued equivalent of the discrete-time impulse response

The ideal highpass halfband filter is given by

$$g(n) = \frac{1}{2\pi} \int_{-\pi}^{-\pi/2} e^{j\omega n} d\omega + \frac{1}{2\pi} \int_{\pi/2}^{\pi} e^{j\omega n} d\omega.$$

Evaluating the preceding integral gives the following impulse response

$$g(n) = \frac{\sin(\pi n)}{\pi n} - \frac{\sin(\frac{\pi}{2}n)}{\pi n}.$$

The impulse response of an ideal highpass halfband filter is:

- Equal to 0 for all even-indexed samples
- Equal to 1/2 at  $n=0$

The FIR halfband interpolator uses a causal FIR approximation to the ideal halfband response, which is based on minimizing the  $\ell^\infty$  norm of the error (minimax). See "Algorithms" on page 4-547 for more information.

### Kaiser Window

The coefficients of a Kaiser window are computed from this equation:

$$w(n) = \frac{I_0\left(\beta\sqrt{1 - \left(\frac{n - N/2}{N/2}\right)^2}\right)}{I_0(\beta)}, \quad 0 \leq n \leq N,$$

where  $I_0$  is the zeroth-order modified Bessel function of the first kind.

To obtain a Kaiser window that represents an FIR filter with stopband attenuation of  $\alpha$  dB, use this  $\beta$ .

$$\beta = \begin{cases} 0.1102(\alpha - 8.7), & \alpha > 50 \\ 0.5842(\alpha - 21)^{0.4} + 0.07886(\alpha - 21), & 50 \geq \alpha \geq 21 \\ 0, & \alpha < 21 \end{cases}$$



The filter order  $n$  is given by:

$$n = \frac{\alpha - 7.95}{2.285(\Delta\omega)}$$

where  $\Delta\omega$  is the transition width.

## Algorithms

### Filter Design Method

The FIR halfband interpolator algorithm uses the equiripple or the Kaiser window method to design the FIR halfband filter. When the design constraints are tight, such as very high stopband attenuation or very narrow transition width, use the Kaiser window method. When the design constraints are not tight, use the equiripple method. If you are not sure of which method to use, set the design method to `Auto`. In this mode, the algorithm automatically chooses a design method that optimally meets the specified filter constraints.

### Halfband Equiripple Design

In the equiripple method, the algorithm uses a minimax (minimize the maximum error) FIR design to design a fullband linear phase filter with the desired specifications. The algorithm upsamples a fullband filter to replace the even-indexed samples of the filter with zeros and creates a halfband filter. It then sets the filter tap corresponding to the group delay of the filter in samples to  $1/2$ . This yields a causal linear-phase FIR filter approximation to the ideal halfband filter defined in “Halfband Filters” on page 4-546. See [1] for a description of this filter design method using the Remez exchange algorithm. Since you can design a filter using this approximation method with a constant ripple both in the passband and stopband, the filter is also known as the equiripple filter.

### Kaiser Window Design

In the Kaiser window method, the algorithm first truncates the ideal halfband filter defined in “Halfband Filters” on page 4-546, then it applies a Kaiser window defined in “Kaiser Window” on page 4-546. This yields a causal linear-phase FIR filter approximation to the ideal halfband filter.

The coefficients of the designed halfband interpolation filter are scaled by the interpolation factor, two, to preserve the output power of the signal.

For more information on designing FIR halfband filters, see “FIR Halfband Filter Design”.

### Polyphase Implementation with Halfband Filters

The FIR halfband interpolator uses an efficient polyphase implementation for halfband filters when you filter the input signal. You can use a polyphase implementation to move the upsampling operation after filtering. This allows you to filter at the lower sampling rate.

Splitting a filter’s impulse response  $h(n)$  into two polyphase components results in an even polyphase component with  $z$ -transform of

$$H_0(z) = \sum_n h(2n)z^{-n},$$

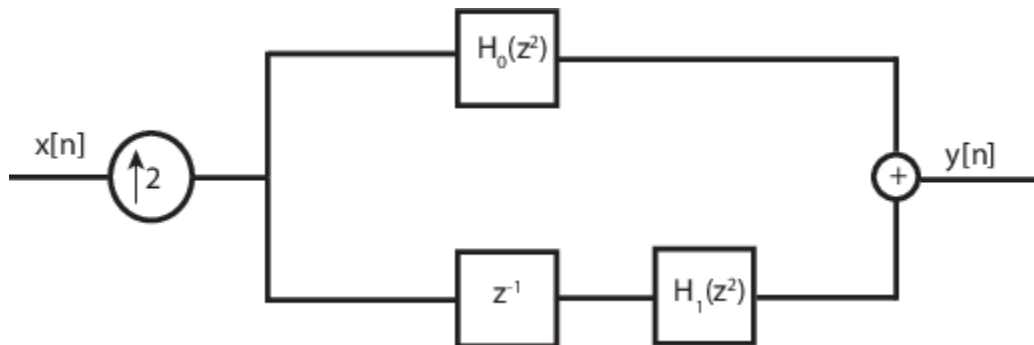
and an odd polyphase component with  $z$ -transform of

$$H_1(z) = \sum_n h(2n+1)z^{-n}.$$

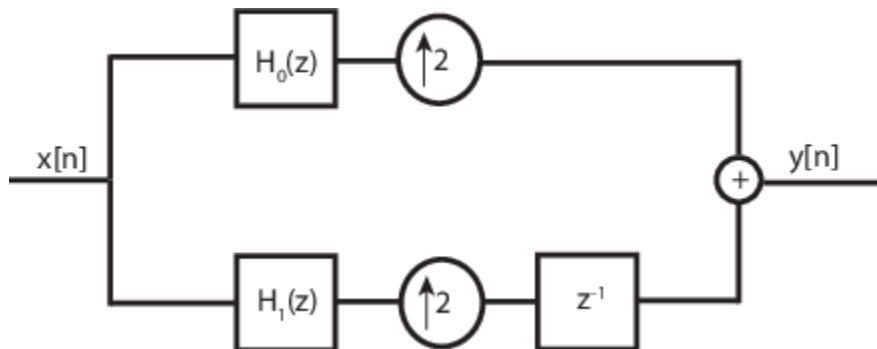
The  $z$ -transform of the filter can be written in terms of the even and odd polyphase components as

$$H(z) = H_0(z^2) + z^{-1}H_1(z^2).$$

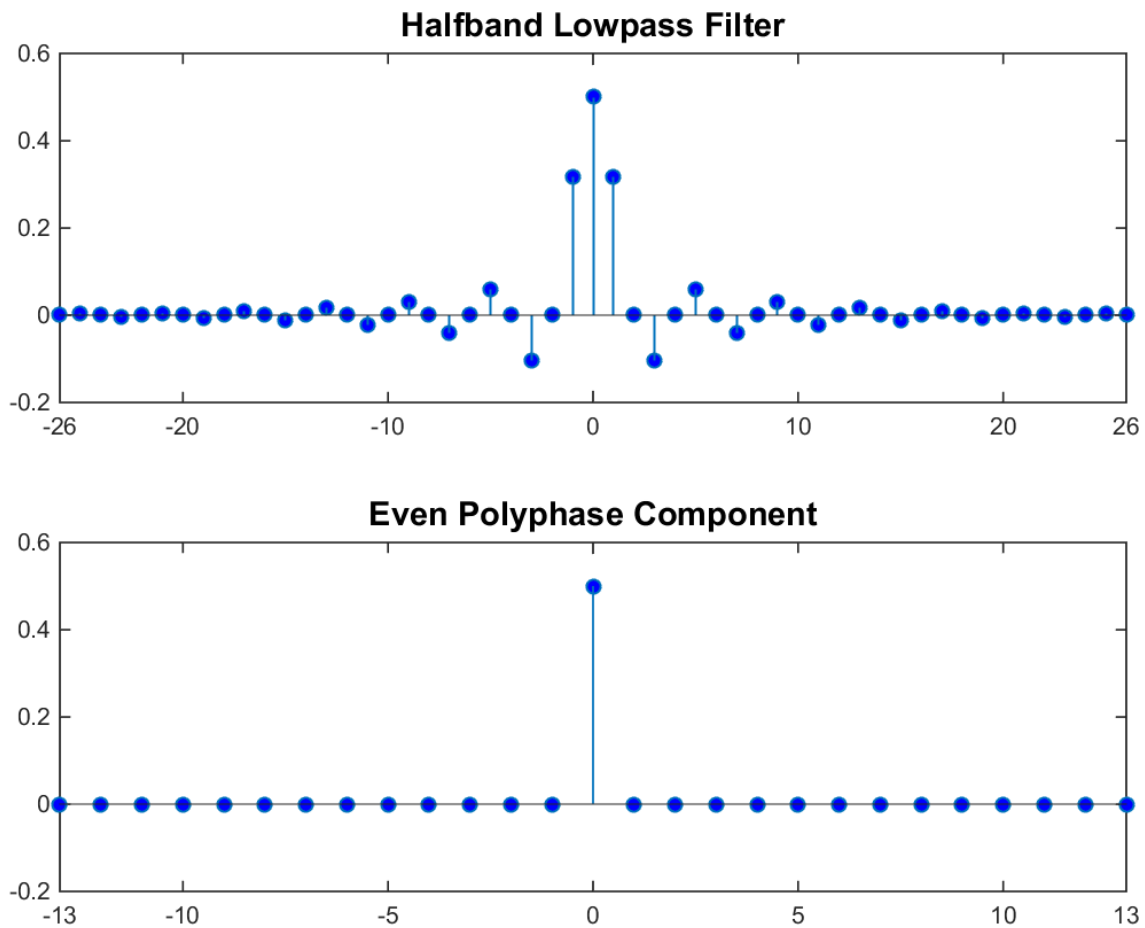
You can represent the upsampling by 2 and then filtering the signal using this figure.



Using the multirate noble identity for upsampling, you can move the upsampling operation after the filtering. This enables you to filter at the lower rate.

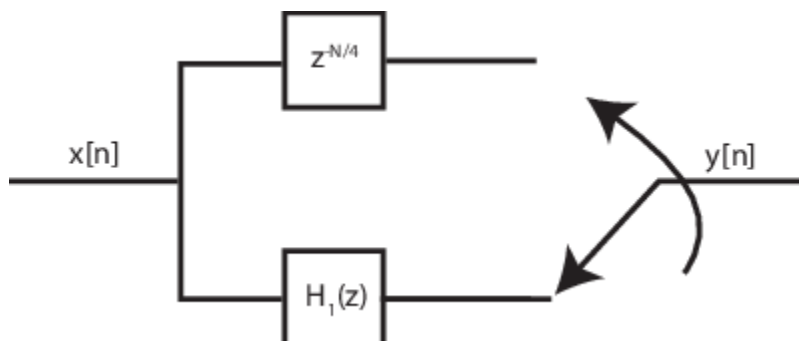


For a halfband filter, the only nonzero coefficient in the even polyphase component is the coefficient corresponding to  $z^0$ . Implementing the halfband filter as a causal FIR filter shifts the nonzero coefficient to approximately  $z^{-N/4}$ , where  $N$  is the number of filter taps. This process is shown in the following figure.



The top plot shows a halfband filter of order 52. The bottom plot shows the even polyphase component. Both filters are noncausal. Delaying the even polyphase component by 13 samples creates a causal FIR filter.

To efficiently implement the halfband interpolator, the algorithm replaces the upsampling operator, delay block, and adder with a commutator switch. This is shown in the following figure, where one polyphase component is replaced by a delay.



The commutator switch takes input samples from the two branches alternately, one sample at a time. This doubles the sampling rate of the input signal. The polyphase component that reduces to a simple delay depends on whether the half order of the filter is even or odd. This is because the delay required to make the even polyphase component causal can be odd or even, depending on the filter half order.

To confirm this behavior, run the following code in the MATLAB command prompt and inspect the polyphase components of the following filters.

```
filterspec = "Filter order and stopband attenuation";  
halfOrderEven = dsp.FIRHalfbandInterpolator(Specification=filterspec,...  
    FilterOrder=64,StopbandAttenuation=80);  
halfOrderOdd = dsp.FIRHalfbandInterpolator(Specification=filterspec,...  
    FilterOrder=54,StopbandAttenuation=80);  
polyphase(halfOrderEven)  
polyphase(halfOrderOdd)
```

One of the polyphase components has a single nonzero coefficient indicating that it is a simple delay. To preserve the output power of the signal, the coefficients are scaled by the interpolation factor, two. To see this scaling, compare the polyphase components of a halfband interpolator with the coefficients of a halfband decimator.

```
hfirinterp = dsp.FIRHalfbandInterpolator;  
hfirdecim = dsp.FIRHalfbandDecimator;  
polyphase(hfirdecim)  
polyphase(hfirinterp)
```

To summarize, the FIR halfband interpolator:

- Filters the input before upsampling with the even and odd polyphase components of the filter.
- Exploits the fact that one filter polyphase component is a simple delay for a halfband filter.

## Version History

**Introduced in R2014b**

## References

- [1] Harris, F.J. *Multirate Signal Processing for Communication Systems*, Prentice Hall, 2004, pp. 208-209.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

This object supports code generation for ARM Cortex-M and ARM Cortex-A processors. To learn more about ARM Cortex code generation, see “Code Generation for ARM Cortex-M and ARM Cortex-A Processors”.

This object also supports SIMD code generation using Intel AVX2 technology when the input signal has a data type of `single` or `double`.

The SIMD technology significantly improves the performance of the generated code.

## See Also

### Functions

[freqz](#) | [fvtool](#) | [info](#) | [cost](#) | [coeffs](#) | [polyphase](#)

### Objects

[dsp.FIRHalfbandDecimator](#) | [dsp.IIRHalfbandInterpolator](#) |  
[dsp.DyadicSynthesisFilterBank](#) | [dsp.ChannelSynthesizer](#)

### Blocks

[FIR Halfband Interpolator](#) | [FIR Halfband Decimator](#) | [IIR Halfband Interpolator](#) | [Dyadic Synthesis Filter Bank](#)

### Topics

[“FIR Halfband Filter Design”](#)

[“Analysis Methods for Filter System Objects”](#) on page 3-2

## dsp.FIRInterpolator

**Package:** dsp

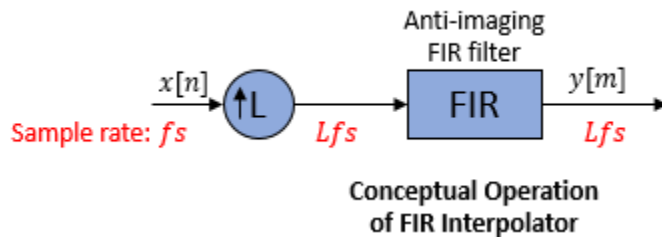
Perform polyphase FIR interpolation

### Description

The `dsp.FIRInterpolator` System object performs an efficient polyphase interpolation using an integer upsampling factor  $L$  along the first dimension.

Conceptually, the FIR interpolator (as shown in the schematic) consists of an upsampler followed by an FIR anti-imaging filter, which is usually an approximation of an ideal band-limited interpolation filter. The coefficients of the anti-imaging filter can be specified through the `Numerator` property, or can be automatically designed by the object using the `designMultirateFIR` function.

The upsampler upsamples each channel of the input to a higher rate by inserting  $L-1$  zeros between samples. The FIR filter that follows filters each channel of the upsampled data. The resulting discrete-time signal has a sample rate that is  $L$  times the original sample rate.



Note that the actual object algorithm implements a direct-form FIR polyphase structure, an efficient equivalent of the combined system depicted in the diagram. For more details, see "Algorithms" on page 4-566.

To upsample an input:

- 1 Create the `dsp.FIRInterpolator` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

Under specific conditions, this System object also supports SIMD code generation. For details, see "Code Generation" on page 4-569.

## Creation

### Syntax

```
firinterp = dsp.FIRInterpolator
firinterp = dsp.FIRInterpolator(L)
```

```

firinterp = dsp.FIRInterpolator(L, 'Auto')
firinterp = dsp.FIRInterpolator(L,num)
firinterp = dsp.FIRInterpolator(L,method)
firinterp = dsp.FIRInterpolator( ____,Name,Value)
firinterp = dsp.FIRInterpolator(L, 'legacy')

```

## Description

`firinterp = dsp.FIRInterpolator` returns an FIR interpolator with an interpolation factor of 3. The object designs the FIR filter coefficients using the `designMultirateFIR(3,1)` function.

`firinterp = dsp.FIRInterpolator(L)` returns an FIR interpolator with the `InterpolationFactor` property set to `L`. The object designs its filter coefficients based on the interpolation factor `L` that you specify while creating the object using the `designMultirateFIR(L,1)` function. The designed filter corresponds to a lowpass with a cutoff at  $\pi/L$  in radial frequency units.

`firinterp = dsp.FIRInterpolator(L, 'Auto')` returns an FIR interpolator with the `NumeratorSource` property set to `'Auto'`. In this mode, every time there is an update in the interpolation factor, the object redesigns the filter using the design method specified in `DesignMethod`.

`firinterp = dsp.FIRInterpolator(L,num)` returns an FIR interpolator with the `InterpolationFactor` property set to `L` and the `Numerator` property set to `num`.

`firinterp = dsp.FIRInterpolator(L,method)` returns an FIR interpolator with the `InterpolationFactor` property set to `L` and the `DesignMethod` property set to `method`. When you pass the design method as an input, the `NumeratorSource` property is automatically set to `'Auto'`.

`firinterp = dsp.FIRInterpolator( ____,Name,Value)` returns an FIR interpolator object with each specified property set to the specified value. Enclose each property name in quotes. You can use this syntax with any previous input argument combinations.

`firinterp = dsp.FIRInterpolator(L, 'legacy')` returns an FIR interpolator where the filter coefficients are designed using `fir1(15,0.25)`. The designed filter has a cutoff frequency of  $0.25\pi$  radians/sample.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

### InterpolationFactor — Interpolation factor

3 (default) | positive integer

Specify the integer factor,  $L$ , by which to increase the sampling rate of the input signal. The polyphase implementation uses  $L$  polyphase subfilters to compute convolutions at the lower sample rate. The FIR interpolator delays and interleaves these lower-rate convolutions to obtain the higher-rate output.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **NumeratorSource — FIR filter coefficient source**

'Property' (default) | 'Input port' | 'Auto'

FIR filter coefficient source, specified as either:

- 'Property' -- The numerator coefficients are specified through the `Numerator` property.
- 'Input port' -- The numerator coefficients are specified as an input to the object algorithm.
- 'Auto' -- The numerator coefficients are designed automatically using the design method specified in `DesignMethod`.

### **Numerator — FIR filter coefficients**

`designMultirateFIR(L,1)` (default) | row vector

Numerator coefficients of the anti-imaging FIR filter, specified as a row vector in powers of  $z^{-1}$ . The following equation defines the system function for a filter of length  $N+1$ :

$$H(z) = \sum_{n=0}^N b(n)z^{-n}$$

The vector  $\mathbf{b} = [b(0), b(1), \dots, b(N)]$  represents the vector of filter coefficients.

To act as an effective anti-imaging filter, the coefficients usually correspond to a lowpass filter with a normalized cutoff frequency no greater than the reciprocal of the `InterpolationFactor`. Use `designMultirateFIR` to design such a filter. More generally, any complex bandpass filter can be used. For an example, see “Double the Sample Rate Using FIR Interpolator” on page 4-565.

The filter coefficients are scaled by the value of the `InterpolationFactor` property before filtering the signal. To form the  $L$  polyphase subfilters, `Numerator` is appended with zeros if necessary.

### **Dependencies**

This property is visible only when you set `NumeratorSource` to 'Property'.

When `NumeratorSource` is set to 'Auto', the numerator coefficients are automatically redesigned using the design method specified in `DesignMethod`. To access the filter coefficients in the automatic design mode, type `objName.Numerator` in the MATLAB command prompt.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **DesignMethod — Auto design method**

'Kaiser' (default) | 'ZOH' | 'Linear'

Design method of the FIR filter coefficients, specified as one of the following:

- 'Kaiser' -- Kaiser method. Approximate anti-imaging lowpass filter using the `designMultirateFIR` function.
- 'ZOH' -- Zero order hold method. Hold the input sequence values.
- 'Linear' -- Linear interpolation method.

### **Dependencies**

This property is visible only when you set the `NumeratorSource` property to 'Auto', or if you pass the 'auto' keyword as an input while creating the object.



## Fixed-Point Properties

### FullPrecisionOverride — Full-precision override for fixed-point arithmetic

true (default) | false

Flag to use full-precision rules for fixed-point arithmetic, specified as one of the following:

- `true` -- The object computes all internal arithmetic and output data types using the full-precision rules. These rules provide the most accurate fixed-point numerics. In this mode, other fixed-point properties do not apply. No quantization occurs within the object. Bits are added, as needed, to ensure that no roundoff or overflow occurs.
- `false` -- Fixed-point data types are controlled through individual fixed-point property settings.

For more information, see “Full Precision for Fixed-Point System Objects” and “Set System Object Fixed-Point Properties”.

### RoundingMethod — Rounding method for fixed-point operations

'Floor' (default) | 'Ceiling' | 'Convergent' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'

Rounding method for fixed-point operations. For more details, see rounding mode.

#### Dependencies

This property is not visible and has no effect on the numerical results when the following conditions are met:

- `FullPrecisionOverride` set to `true`.
- `FullPrecisionOverride` set to `false`, `ProductDataType` set to `'Full precision'`, `AccumulatorDataType` set to `'Full precision'`, and `OutputDataType` set to `'Same as accumulator'`.

Under these conditions, the object operates in full precision mode.

### OverflowAction — Overflow action for fixed-point operations

'Wrap' (default) | 'Saturate'

Overflow action for fixed-point operations, specified as one of the following:

- `'Wrap'` -- The object wraps the result of its fixed-point operations.
- `'Saturate'` -- The object saturates the result of its fixed-point operations.

For more details on overflow actions, see overflow mode for fixed-point operations.

#### Dependencies

This property is not visible and has no effect on the numerical results when the following conditions are met:

- `FullPrecisionOverride` set to `true`.
- `FullPrecisionOverride` set to `false`, `OutputDataType` set to `'Same as accumulator'`, `ProductDataType` set to `'Full precision'`, and `AccumulatorDataType` set to `'Full precision'`.

Under these conditions, the object operates in full precision mode.

**CoefficientsDataType — Data type of FIR filter coefficients**

Same word length as input (default) | Custom

Data type of the FIR filter coefficients, specified as:

- `Same word length as input` -- The word length of the coefficients is the same as that of the input. The fraction length is computed to give the best possible precision.
- `Custom` -- The coefficients data type is specified as a custom numeric type through the `CustomCoefficientsDataType` property.

**CustomCoefficientsDataType — Word and fraction lengths of coefficients data type**

`numerictype([],16,15)` (default) | custom numeric type

Word and fraction lengths of the coefficients data type, specified as an autosigned `numerictype` with a word length of 16 and a fraction length of 15.

**Dependencies**

This property applies when you set the `CoefficientsDataType` property to `Custom`.

**ProductDataType — Data type of product output**

'Full precision' (default) | 'Custom' | 'Same as input'

Data type of the product output in this object, specified as one of the following:

- `'Full precision'` -- The product output data type has full precision.
- `'Same as input'` -- The object specifies the product output data type to be the same as that of the input data type.
- `'Custom'` -- The product output data type is specified as a custom numeric type through the `CustomProductDataType` property.

For more information on the product output data type, see “Multiplication Data Types”.

**Dependencies**

This property applies when you set `FullPrecisionOverride` to `false`.

**CustomProductDataType — Word and fraction lengths of product data type**

`numerictype([],32,30)` (default) | custom numeric type

Word and fraction lengths of the product data type, specified as an autosigned numeric type with a word length of 32 and a fraction length of 30.

**Dependencies**

This property applies only when you set `FullPrecisionOverride` to `false` and `ProductDataType` to `'Custom'`.

**AccumulatorDataType — Data type of accumulation operation**

'Full precision' (default) | 'Same as input' | 'Same as product' | 'Custom'

Data type of an accumulation operation in this object, specified as one of the following:

- `'Full precision'` -- The accumulation operation has full precision.
- `'Same as product'` -- The object specifies the accumulator data type to be the same as that of the product output data type.

- 'Same as input' -- The object specifies the accumulator data type to be the same as that of the input data type.
- 'Custom' -- The accumulator data type is specified as a custom numeric type through the CustomAccumulatorDataType property.

#### Dependencies

This property applies when you set FullPrecisionOverride to false.

#### CustomAccumulatorDataType — Word and fraction lengths of accumulator data type

numericType([],32,30) (default) | custom numeric type

Word and fraction lengths of the accumulator data type, specified as an autosigned numeric type with a word length of 32 and a fraction length of 30.

#### Dependencies

This property applies only when you set FullPrecisionOverride to false and AccumulatorDataType to 'Custom'.

#### OutputDataType — Data type of object output

'Same as accumulator' (default) | 'Same as input' | 'Same as product' | 'Custom'

Data type of the object output, specified as one of the following:

- 'Same as accumulator' -- The output data type is the same as that of the accumulator output data type.
- 'Same as input' -- The output data type is the same as that of the input data type.
- 'Same as product' -- The output data type is the same as that of the product output data type.
- 'Custom' -- The output data type is specified as a custom numeric type through the CustomOutputDataType property.

#### Dependencies

This property applies when you set FullPrecisionOverride to false.

#### CustomOutputDataType — Word and fraction lengths of output data type

numericType([],16,15) (default) | custom numeric type

Word and fraction lengths of the output data type, specified as an autosigned numeric type with a word length of 16 and a fraction length of 15.

#### Dependencies

This property applies only when you set FullPrecisionOverride to false and OutputDataType to 'Custom'.

## Usage

### Syntax

```
y = firinterp(x)
y = firinterp(x,num)
```

### Description

`y = firinterp(x)` interpolates the input signal `x` along the first dimension, and outputs the upsampled and filtered values, `y`.

`y = firinterp(x,num)` uses the FIR filter, `num`, to interpolate the input signal. This configuration is valid only when the 'NumeratorSource' property is set to 'Input port'.

### Input Arguments

#### **x** — Data input

vector | matrix

Data input, specified as a vector or a matrix. A  $P$ -by- $Q$  input matrix is treated as  $Q$  independent channels, and the System object interpolates each channel over the first dimension and generates a  $P*L$ -by- $Q$  output matrix, where  $L$  is the interpolation factor.

This object supports variable-size input and does not support complex unsigned fixed-point inputs.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fi

#### **num** — FIR filter coefficients

row vector

FIR filter coefficients, specified as a row vector.

### Dependencies

This input is accepted only when the 'NumeratorSource' property is set to 'Input port'.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fi

### Output Arguments

#### **y** — FIR interpolator output

vector | matrix

FIR interpolator output, returned as a vector or a matrix of size  $P*L$ -by- $Q$ , where  $L$  is the interpolation factor.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fi

### Object Functions

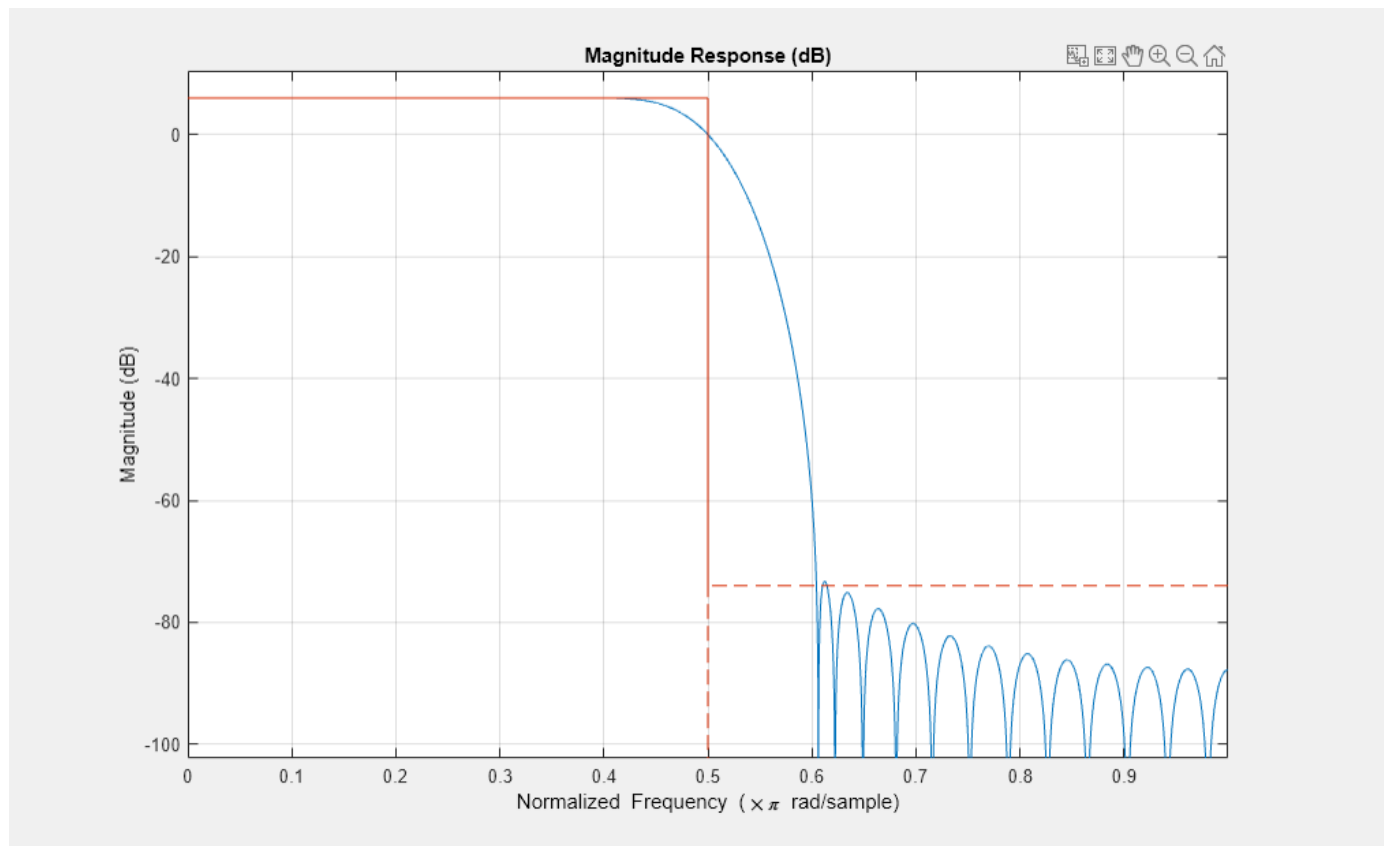
To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

### Specific to dsp.FIRInterpolator

`freqz`            Frequency response of discrete-time filter System object  
`fvtool`          Visualize frequency response of DSP filters





### Interpolate by 2

Interpolate the cosine signal by a factor of 2.

```
y = firinterp(x);
```

Plot the original and interpolated signals. In order to plot the two signals on the same plot, you must account for the output delay of the FIR interpolator and the scaling introduced by the filter. Use the `outputDelay` function to compute the delay value introduced by the interpolator. Shift the output by this delay value.

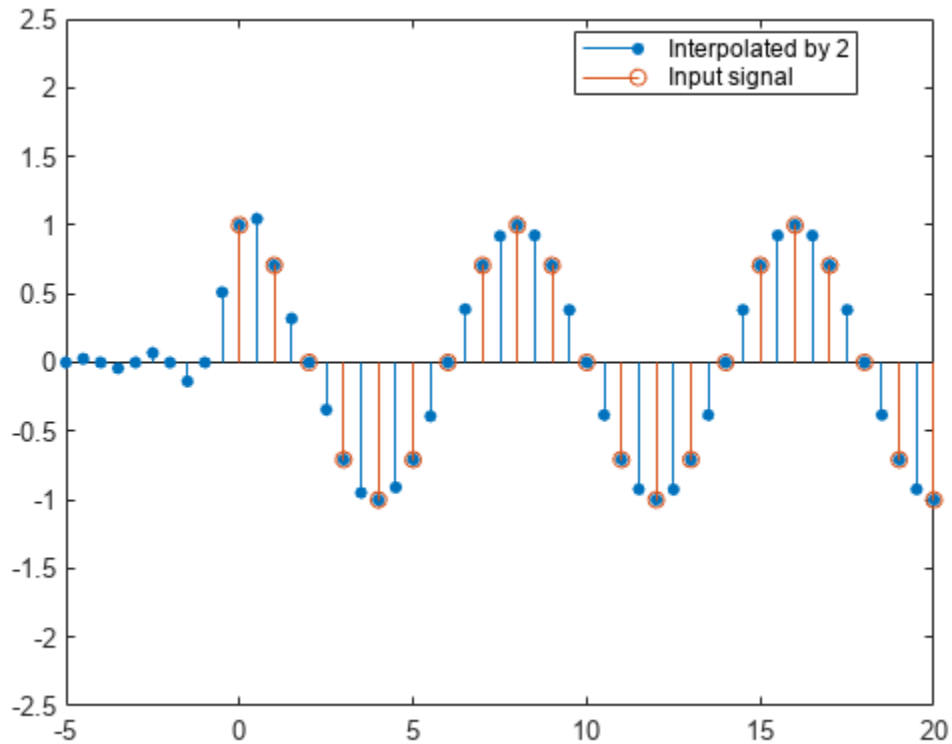
Visualize the input and the resampled signals. The input and output values coincide every other sample, due to the interpolation factor of 2.

```
[delay,FsOut] = outputDelay(firinterp)
```

```
delay = 12
```

```
FsOut = 2
```

```
nx = (0:length(x)-1);
ty = (0:length(y)-1)/FsOut-delay;
stem(ty,y,'filled',MarkerSize=4); hold on;
stem(nx,x); hold off;
xlim([-5,20])
ylim([-2.5 2.5])
legend('Interpolated by 2','Input signal','Location','best');
```



### Interpolate by 4 in Automatic Filter Design Mode

Now interpolate by a factor of 4. In order for the filter design to be updated automatically based on the new interpolation factor, set the `NumeratorSource` property to `'Auto'`. Alternately, you can pass the keyword `'Auto'` as an input while creating the object. The object then operates in the automatic filter design mode. Every time there is a change in the interpolation factor, the object updates the filter design.

```
release(firinterp)
firinterp.NumeratorSource = 'Auto';
firinterp.InterpolationFactor = 4

firinterp =
  dsp.FIRInterpolator with properties:

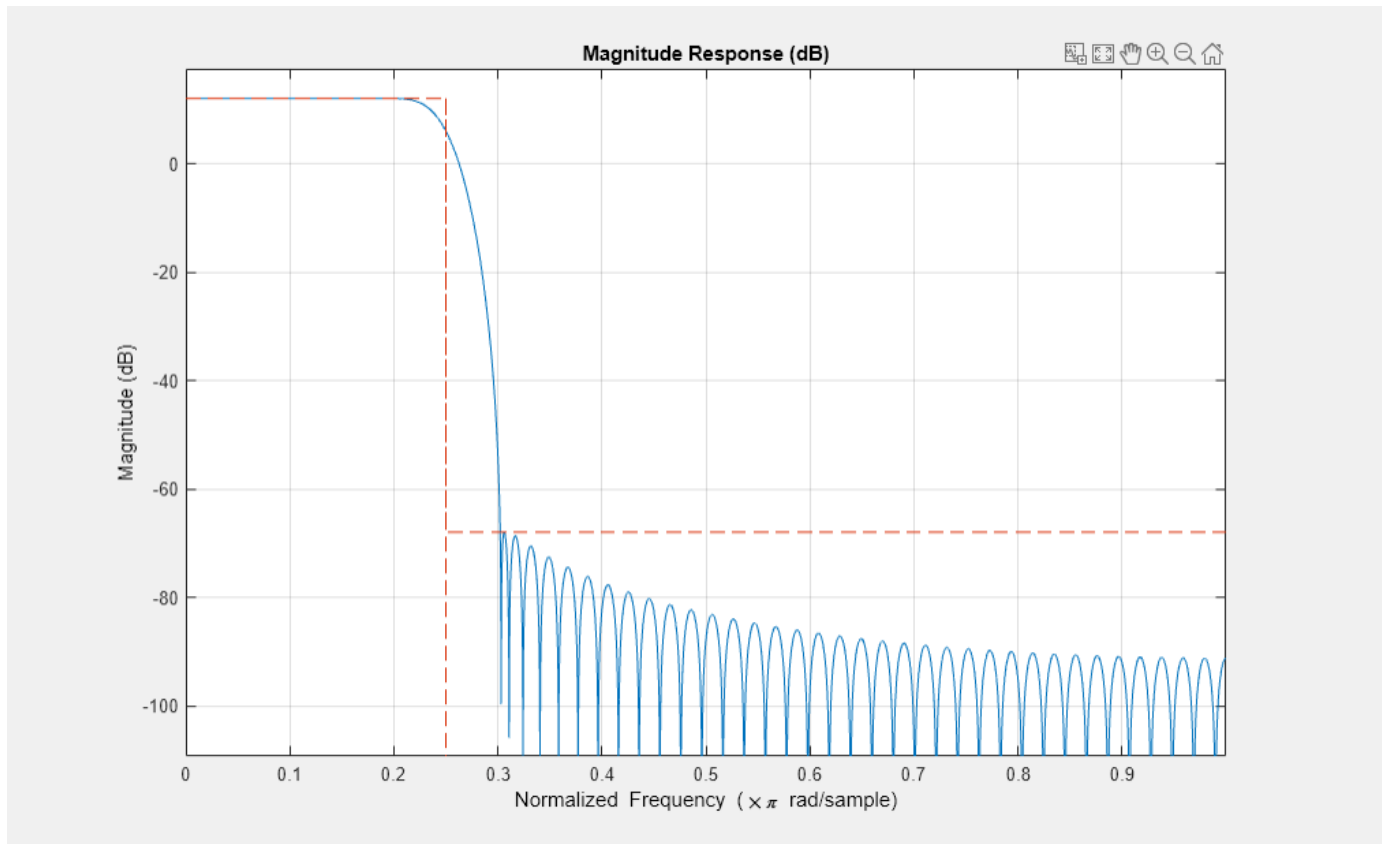
    InterpolationFactor: 4
    NumeratorSource: 'Auto'
    DesignMethod: 'Kaiser'

Show all properties
```

To access the filter coefficients in the automatic filter design mode, type `firinterp.Numerator` in the MATLAB command prompt.

The designed filter occupies a narrower passband that is approximately a quarter of the spectrum.

```
fvtool(firinterp)
```



Interpolate the cosine signal by a factor of 4.

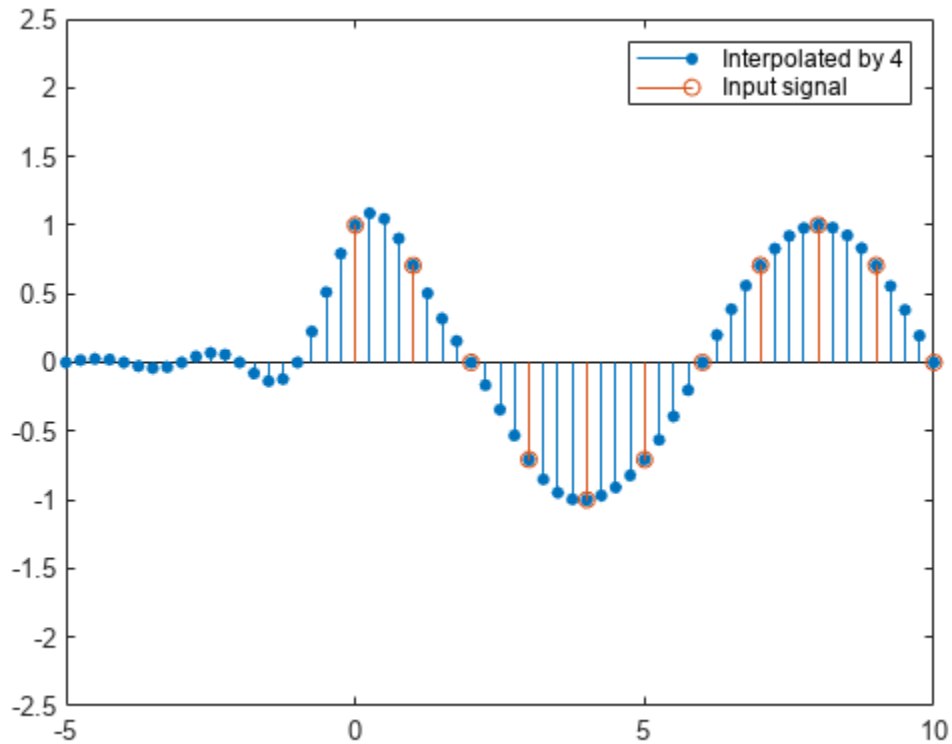
```
yAuto = firinterp(x);
```

Plot the original and resampled signals. Recalculate the delay and the output sample rate values since the interpolation factor has changed. The input and output values coincide every 4 output samples, owing to the interpolation factor of 4.

```
[delay,FsOut] = outputDelay(firinterp);
```

```
nx = (0:length(x)-1);
tyAuto = (0:length(yAuto)-1)/FsOut-delay;
stem(tyAuto,yAuto,'filled',MarkerSize=4); hold on;
stem(nx,x); hold off;
xlim([-5,10])
ylim([-2.5 2.5])
legend('Interpolated by 4','Input signal');
```





### Specify Signal Interpolation Model

In the automatic design mode, you can also specify the underlying D/A signal interpolation model through the `DesignMethod` property.

### Set `DesignMethod` to 'linear'

If you set the `DesignMethod` to 'linear', the object uses the linear interpolation model.

```
release(firinterp)
firinterp.DesignMethod = 'linear'

firinterp =
  dsp.FIRInterpolator with properties:

    InterpolationFactor: 4
    NumeratorSource: 'Auto'
    DesignMethod: 'Linear'

Show all properties
```

Interpolate the signal using the linear interpolation model.

```
ylinear = firinterp(x);
```

Plot the original and the linearly interpolated signal.

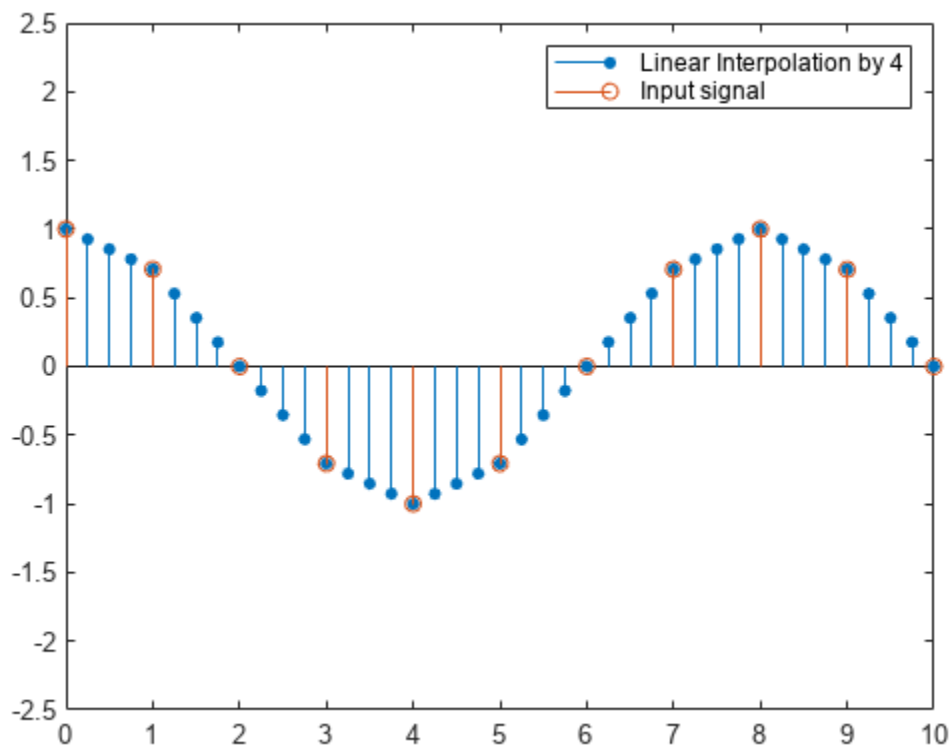
```

[delay,FsOut] = outputDelay(firinterp);
nx = (0:length(x)-1);

% Calculate output times for vector ylinear in input units
tylinear = (0:length(ylinear)-1)/FsOut-delay;

stem(tylinear,ylinear,'filled',MarkerSize=4); hold on;
stem(nx,x);
hold off;
xlim([0,10])
ylim([-2.5 2.5])
legend('Linear Interpolation by 4','Input signal');

```



### Set DesignMethod to 'ZOH' and Change InterpolationFactor to 5

If you set the DesignMethod to 'ZOH', the object uses the zero order hold method. Change the interpolation factor to 5.

```

release(firinterp)
firinterp.DesignMethod = 'ZOH';
firinterp.InterpolationFactor = 5

firinterp =
    dsp.FIRInterpolator with properties:
        InterpolationFactor: 5
        NumeratorSource: 'Auto'
        DesignMethod: 'ZOH'

```

Show all properties

Interpolate the signal using the zero order hold method.

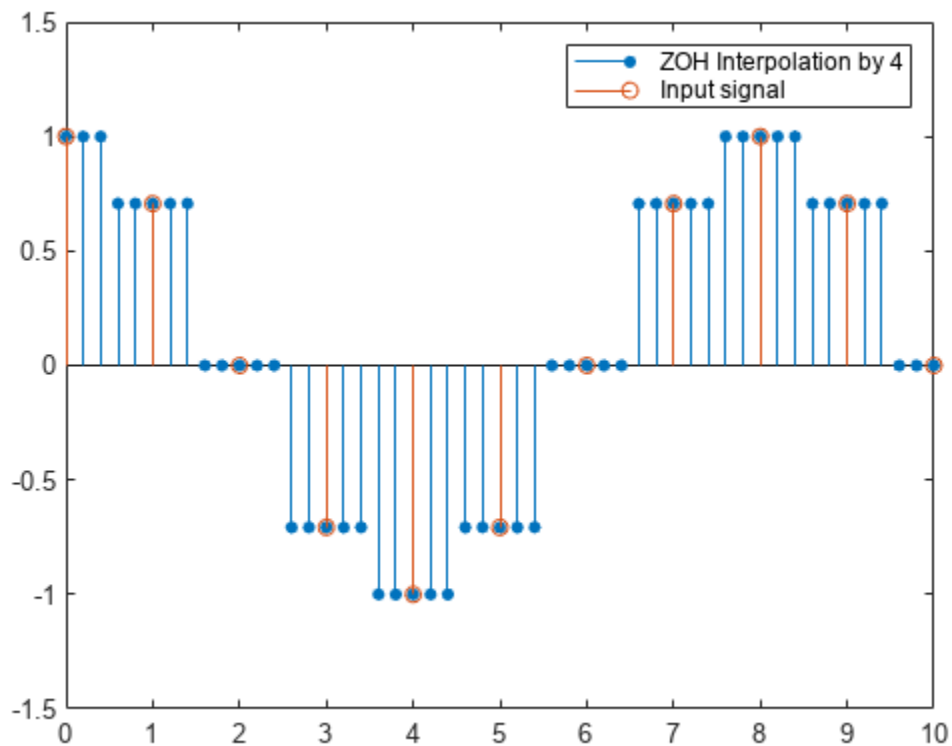
```
yzoh = firinterp(x);
```

Plot the original and ZOH interpolated signal.

```
[delay,FsOut] = outputDelay(firinterp);
nx = (0:length(x)-1);
```

```
% Calculate output times for vector yzoh in input units
tyzoh = (0:length(yzoh)-1)/FsOut-delay;
```

```
stem(tyzoh,yzoh,'filled',MarkerSize=4); hold on;
stem(nx,x); hold off;
xlim([0,10])
ylim([-1.5 1.5])
legend('ZOH Interpolation by 4','Input signal');
```



### Double the Sample Rate Using FIR Interpolator

Double the sample rate of an audio signal and play the interpolated signal using the `audioDeviceWriter` object.



$$H(z) = b_0 + b_1z^{-1} + \dots + b_Nz^{-N}$$

$N+1$  is the length of the FIR filter.

You can rearrange this equation as follows:

$$H(z) = (b_0 + b_Lz^{-L} + b_{2L}z^{-2L} + \dots + b_{N-L+1}z^{-(N-L+1)}) + z^{-1}(b_1 + b_{L+1}z^{-L} + b_{2L+1}z^{-2L} + \dots + b_{N-L+2}z^{-(N-L+1)}) + \dots + z^{-(L-1)}(b_{L-1} + b_{2L-1}z^{-L} + b_{3L-1}z^{-2L} + \dots + b_Nz^{-(N-L+1)})$$

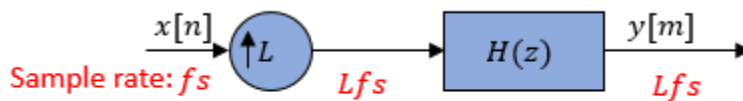
$L$  is the number of polyphase components, and its value equals the interpolation factor that you specify.

You can write this equation as:

$$H(z) = E_0(z^L) + z^{-1}E_1(z^L) + \dots + z^{-(L-1)}E_{L-1}(z^L)$$

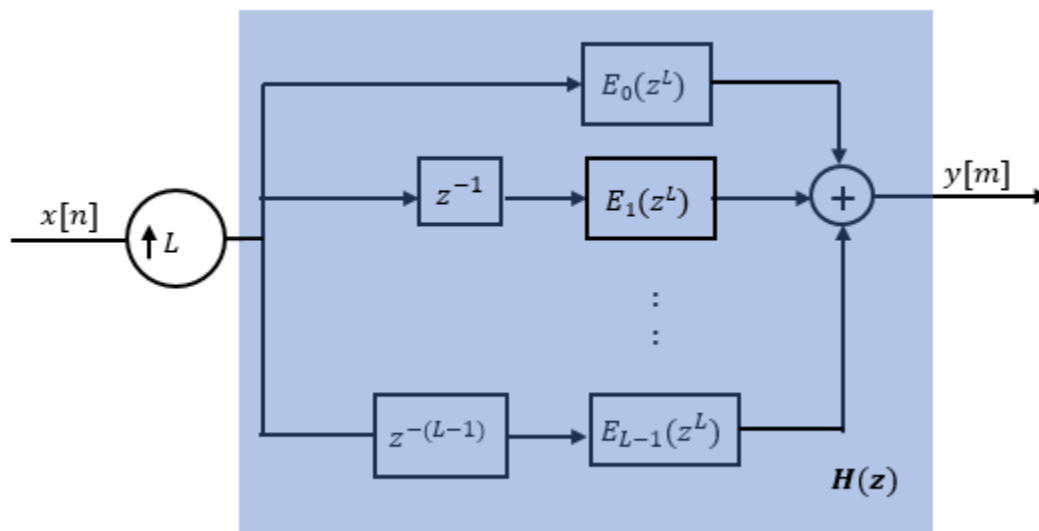
$E_0(z^L), E_1(z^L), \dots, E_{L-1}(z^L)$  are polyphase components of the FIR filter  $H(z)$ .

Conceptually, the FIR interpolation filter contains an upsampler followed by an FIR lowpass filter  $H(z)$ .

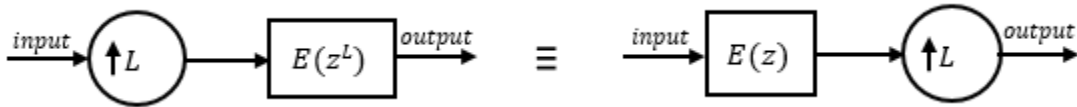


**Conceptual Operation  
of FIR Interpolator**

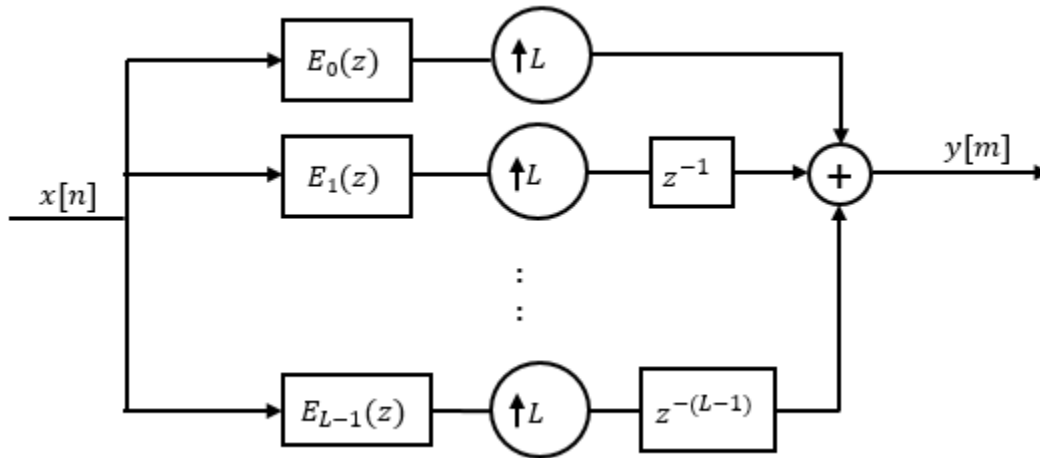
Replace  $H(z)$  with its polyphase representation.



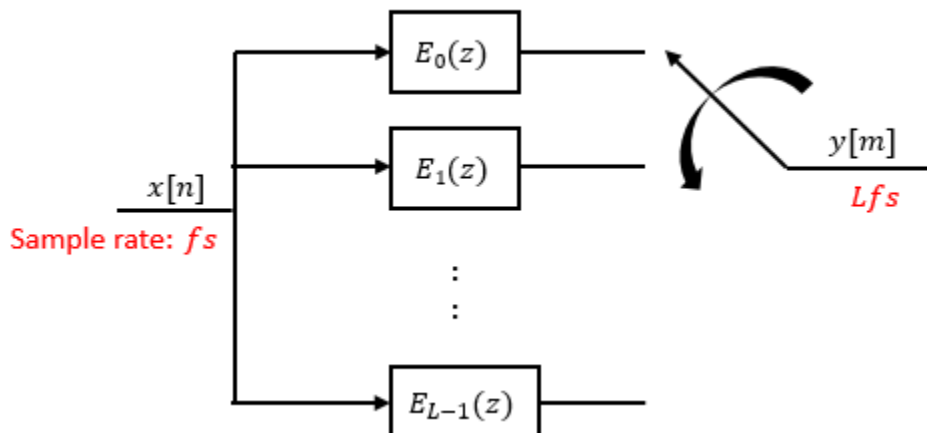
Here is the multirate noble identity for interpolation.



Applying the noble identity for interpolation moves the upsampling operation to after the filtering operation. This move enables you to filter the signal at a lower rate.



You can replace the upsampling operator, delay block, and adder with a commutator switch. The switch starts on the first branch 0 and moves in the counterclockwise direction, each time receiving one sample from each branch. The interpolator effectively outputs  $L$  samples for every one input sample it receives. Hence the sample rate at the output of the FIR interpolation filter is  $Lf_s$ .



## Version History

Introduced in R2012a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

The `dsp.FIRInterpolator` System object supports SIMD code generation using Intel AVX2 technology under these conditions:

- Input signal is real-valued with real filter coefficients.
- Input signal is complex-valued with real or complex filter coefficients.
- Input signal has a data type of `single` or `double`.

The SIMD technology significantly improves the performance of the generated code.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

This object supports HDL code generation with the Filter Design HDL Coder product. For workflows and limitations, see “Generate HDL Code for Filter System Objects” (Filter Design HDL Coder).

## See Also

### Functions

`freqz` | `fvtool` | `info` | `cost` | `polyphase` | `generatehdl` | `impz` | `coeffs`

### Objects

`dsp.FIRDecimator` | `dsp.FIRRateConverter`

### Blocks

FIR Interpolation

### Topics

“Analysis Methods for Filter System Objects” on page 3-2

## dsp.FIRRateConverter

**Package:** dsp

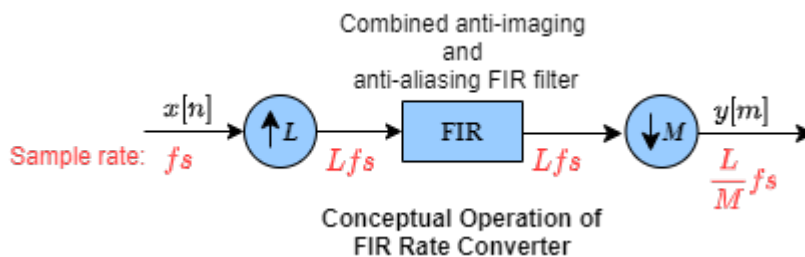
Perform polyphase FIR sample rate conversion

### Description

The `dsp.FIRRateConverter` System object performs an efficient polyphase sample rate conversion using a rational factor  $L/M$  along the first dimension.

The FIR rate converter cascades an interpolator with a decimator. Conceptually, the rate converter (as shown in the schematic) consists of an upsampler, followed by a combined anti-imaging and anti-aliasing FIR filter, followed by a downsampler. The coefficients of the anti-imaging and anti-aliasing FIR filter can be specified through the `Numerator` property, or can be automatically designed by the object using the `designMultirateFIR` function. For an example, see “Resample Signal Using FIR Rate Converter” on page 4-576.

The upsampler increases the sample rate of the signal by a factor  $L$  and the downsampler reduces the sample rate of the signal by a factor  $M$ . Use upsampling and downsampling factors that are relatively prime or coprime. The resulting discrete-time signal has a sample rate that is  $L/M$  times the original sample rate.



Note that the actual object algorithm implements a polyphase structure, an efficient equivalent of the combined system depicted in the diagram. For more details, see “Algorithms” on page 4-581.

To perform sampling rate conversion:

- 1 Create the `dsp.FIRRateConverter` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

### Creation

#### Syntax

```
firrc = dsp.FIRRateConverter
firrc = dsp.FIRRateConverter(L,M)
firrc = dsp.FIRRateConverter(L,M,'Auto')
```



```

firrc = dsp.FIRRateConverter(L,M,num)
firrc = dsp.FIRRateConverter(L,M,method)
firrc = dsp.FIRRateConverter( ____,Name,Value)
firrc = dsp.FIRRateConverter(L,M,'legacy')

```

## Description

`firrc = dsp.FIRRateConverter` returns an FIR sample rate converter object with an interpolation factor of 3 and a decimation factor of 2. The object designs the FIR filter coefficients using the `designMultirateFIR(3,2)` function.

`firrc = dsp.FIRRateConverter(L,M)` returns an FIR sample rate converter with the integer-valued `InterpolationFactor` property set to `L` and the `DecimationFactor` property set to `M`. The object designs its filter coefficients based on the rate conversion factors that you specify while creating the object, using the `designMultirateFIR(L,M)` function. The designed filter corresponds to a lowpass filter with normalized cutoff frequency no greater than  $\min(\pi/L, \pi/M)$  in radial frequency units.

`firrc = dsp.FIRRateConverter(L,M,'Auto')` returns an FIR sample rate converter where the `NumeratorSource` property is set to `'Auto'`. In this mode, every time there is an update in the rate conversion factors, the object redesigns the filter using the design method specified in `DesignMethod`.

`firrc = dsp.FIRRateConverter(L,M,num)` returns an FIR sample rate converter with the `Numerator` property set to `num`.

`firrc = dsp.FIRRateConverter(L,M,method)` returns an FIR sample rate converter with the `DesignMethod` property set to `method`. When you pass the design method as an input, the `NumeratorSource` property is automatically set to `'Auto'`.

`firrc = dsp.FIRRateConverter( ____,Name,Value)` returns an FIR sample rate converter with each specified property set to the specified value. Enclose each property name in quotes. You can use this syntax with any previous input argument combinations.

Example: `firrc = dsp.FIRRateConverter('FullPrecisionOverride','false')` enables the fixed-point data types to be controlled through the individual fixed-point property settings.

`firrc = dsp.FIRRateConverter(L,M,'legacy')` returns an FIR sample rate converter where the filter coefficients are designed using `firpm(70,[0 0.28 0.32 1],[1 1 0 0])`. The designed filter has a cutoff frequency of  $\pi/3$  radians/sample.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

### Main Properties

#### InterpolationFactor — Interpolation factor

3 (default) | positive integer

Interpolation factor  $L$ , specified as a positive integer.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

#### **DecimationFactor — Decimation factor**

2 (default) | positive integer

Decimation factor  $M$ , specified as a positive integer.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

#### **NumeratorSource — FIR filter coefficient source**

'Property' (default) | 'Auto'

FIR filter coefficient source, specified as one of the following:

- 'Property' -- The numerator coefficients are specified through the `Numerator` property.
- 'Auto' -- The numerator coefficients are designed automatically using the design method specified in `DesignMethod`.

#### **Numerator — FIR filter coefficients**

`designMultirateFIR(L,M)` (default) | row vector

Numerator coefficients of the FIR filter, specified as a row vector in powers of  $z^{-1}$ .

The following equation defines the transfer function of the filter:

$$H(z) = \sum_{n=0}^N b(n)z^{-n}$$

The vector  $\mathbf{b} = [b(0), b(1), \dots, b(N)]$  represents the vector of filter coefficients.

Use a lowpass filter with normalized cutoff frequency no greater than  $\min(1/L, 1/M)$ , where  $L$  is the interpolation factor and  $M$  is the decimation factor. The gain of the lowpass filter should equal  $L$ , the interpolation factor. Choose the length of the filter to be a multiple of the interpolation factor. To design such a filter, use the `designMultirateFIR` function. For an example, see "Resample Signal Using FIR Rate Converter" on page 4-576. All initial filter states are zero.

#### **Dependencies**

This property is visible only when you set `NumeratorSource` to 'Property'.

When `NumeratorSource` is set to 'Auto', the numerator coefficients are automatically redesigned using the design method specified in `DesignMethod`. To access the filter coefficients in the automatic design mode, type `objName.Numerator` in the MATLAB command prompt.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

#### **DesignMethod — Auto design method**

'Kaiser' (default) | 'ZOH' | 'Linear'

Design method of the FIR filter coefficients, specified as one of the following:

- 'Kaiser' -- Kaiser method. Approximate combined anti-aliasing and anti-imaging lowpass filter using the `designMultirateFIR` function.

- 'ZOH' -- Zero order hold method. Hold the input sequence values.
- 'Linear' -- Linear interpolation method.

### Dependencies

This property is visible only when you set the `NumeratorSource` property to 'Auto', or if you pass the 'auto' keyword as an input while creating the object.

### Code Generation Properties

#### AllowArbitraryInputLength — Allow arbitrary input length in generated code

false (default) | true

Allow arbitrary frame length for fixed-size input signals in the generated code, specified as true or false. When you specify:

- true -- The input frame length does not have to be a multiple of the decimation factor. The output of the object in the generated code is a variable-size array.
- false -- The input frame length must be a multiple of the decimation factor.

When you specify variable-size signals, the input frame length can be arbitrary and the object ignores this property in the generated code. When you run this object in MATLAB, the object supports arbitrary input frame lengths for fixed-size and variable-size signals and this property does not affect the object behavior.

Data Types: logical

### Fixed-Point Properties

#### FullPrecisionOverride — Full precision override for fixed-point arithmetic

true (default) | false

Specify whether to use full precision rules. If you set `FullPrecisionOverride` to true, which is the default, the object computes all internal arithmetic and output data types using full precision rules. These rules provide the most accurate fixed-point numerics. It also turns off the display of other fixed-point properties because they do not apply individually. These rules guarantee that no quantization occurs within the object. Bits are added, as needed, to ensure that no roundoff or overflow occurs. If you set `FullPrecisionOverride` to false, fixed-point data types are controlled through individual fixed-point property settings. For more information, see “Full Precision for Fixed-Point System Objects”.

#### RoundingMethod — Rounding method for fixed-point operations

Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Specify the rounding method as one of | Ceiling | Convergent | Floor | Nearest | Round | Simplest | Zero |.

### Dependencies

This property applies only if the object is not in full precision mode.

#### OverflowAction — Overflow action for fixed-point operations

Wrap (default) | Saturate

Specify the overflow action as one of | Wrap | Saturate |.

**Dependencies**

This property applies only if the object is not in full precision mode.

**CoefficientsDataType — Word and fraction lengths of filter coefficients**

Same word length as input (default) | Custom

Specify the filter coefficient fixed-point data type as one of | Same word length as input | Custom |.

**CustomCoefficientsDataType — Word and fraction lengths of filter coefficients**

numerictype([],16,15) (default) | numerictype

Specify the filter coefficient fixed-point type as a numerictype object with a Signedness of Auto.

**Dependencies**

This property applies only when the CoefficientsDataType property is Custom.

**ProductDataType — Product word and fraction lengths**

Full precision (default) | Same as input | Custom

Specify the product fixed-point data type as one of | Full precision | Same as input | Custom |.

**CustomProductDataType — Product word and fraction lengths**

numerictype([],32,30) (default) | numerictype

Specify the product fixed-point type as a scaled numerictype object with a Signedness of Auto.

**Dependencies**

This property applies only when the ProductDataType property is Custom.

**AccumulatorDataType — Accumulator word and fraction lengths**

Full precision (default) | Same as product | Same as input | Custom

Specify the accumulator fixed-point data type as one of | Full precision | Same as product | Same as input | Custom |.

**CustomAccumulatorDataType — Accumulator word and fraction lengths**

numerictype([],32,30) (default) | numerictype

Specify the accumulator fixed-point type as a scaled numerictype object with a Signedness of Auto.

**Dependencies**

This property applies only when the AccumulatorDataType property is Custom.

**OutputDataType — Output word and fraction lengths**

Same as accumulator (default) | Same as product | Same as input | Custom

Specify the output fixed-point data type as one of | Same as accumulator | Same as product | Same as input | Custom |.

**CustomOutputDataType — Output word and fraction lengths**

numerictype([],16,15) (default) | numerictype

Specify the output fixed-point type as a scaled numeric type object with a Signedness of Auto.

### Dependencies

This property applies only when the OutputDataType property is Custom.

## Usage

### Syntax

```
y = firrc(x)
```

### Description

`y = firrc(x)` resamples the input `x` and returns the resampled signal `y`.

### Input Arguments

#### **x** — Data input

column vector | matrix

Data input, specified as a column vector or a matrix of size  $P$ -by- $Q$ . The columns in the input signal represent  $Q$  independent channels.

Under most conditions, the number of inputs rows  $P$  can be arbitrary and does not have to be a multiple of the DecimationFactor property. See this table for details.

| Input Signal  | When you Run Object in MATLAB                | When you Generate Code Using MATLAB Coder   |
|---------------|--|---|
| Fixed-size    | Object supports arbitrary input frame length | Object supports arbitrary input frame length when you set AllowArbitraryInputLength to true while generating code |
| Variable-size | Object supports arbitrary input frame length | Object supports arbitrary input frame length  |

Variable-size signals change in frame length once you lock the object while the fixed-size signals remain constant. When the object does not support arbitrary frame length, the input frame length must be a multiple of the DecimationFactor property.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fi

### Output Arguments

#### **y** — Resampled output

column vector | matrix

Resampled output, returned as a column vector or a matrix. When the input is of size  $P$ -by- $Q$ , and  $P$  is not a multiple of the decimation factor  $M$ , the output signal has an upper bound size of  $\text{ceil}(LP/M)$ -by- $Q$ . If  $P$  is a multiple of the decimation factor, then the output is of size  $(LP/M)$ -by- $Q$ .

The number of rows in the output signal is given by  $LP/M$ , where  $P$  is the number of input rows,  $L$  is the interpolation factor, and  $M$  is the decimation factor.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

### Specific to `dsp.FIRRateConverter`

|                          |   |
|--------------------------|---|
| <code>freqz</code>       | Frequency response of discrete-time filter System object                      |
| <code>fvtool</code>      | Visualize frequency response of DSP filters                                   |
| <code>info</code>        | Information about filter System object  |
| <code>cost</code>        | Estimate cost of implementing filter System object                            |
| <code>coeffs</code>      | Returns the filter System object coefficients in a structure                  |
| <code>polyphase</code>   | Polyphase decomposition of multirate filter                                   |
| <code>generatehdl</code> | Generate HDL code for quantized DSP filter (requires Filter Design HDL Coder) |

### Common to All System Objects

|                      |  |
|----------------------|--|
| <code>step</code>    | Run System object algorithm  |
| <code>release</code> | Release resources and allow changes to System object property values and input characteristics |
| <code>reset</code>   | Reset internal states of System object   |

## Examples

### Resample Signal Using FIR Rate Converter

Resample a 100 Hz sine wave signal by a factor of 3/2.

Create a `dsp.SineWave` object which generates a sinusoidal signal with 54 samples per frame, contains a tone at 100 Hz, and has a sample rate of 1250 Hz.

```
sine = dsp.SineWave(1,100, 'SampleRate',1250, 'SamplesPerFrame',54);
```

```
% Grab a frame
```

```
x = sine();
```

```
% Calculate input time vector
```

```
tx = (0:length(x)-1)/sine.SampleRate;
```

### Design Default Filter

Create a `dsp.FIRRateConverter` object. The object uses a combined anti-imaging and anti-aliasing FIR filter. By default, this filter is designed using the `designMultirateFIR` function. The function designs the filter based on the rate conversion factors that you specify, and stores the coefficients in the `Numerator` property. For an interpolation factor of 3 and a decimation factor of 2, the object designs the coefficients using `designMultirateFIR(3,2)`.

```
firrc = dsp.FIRRateConverter(3,2);
```

### Resample by a Factor of 3/2

Resample the signal by a factor of 3/2.

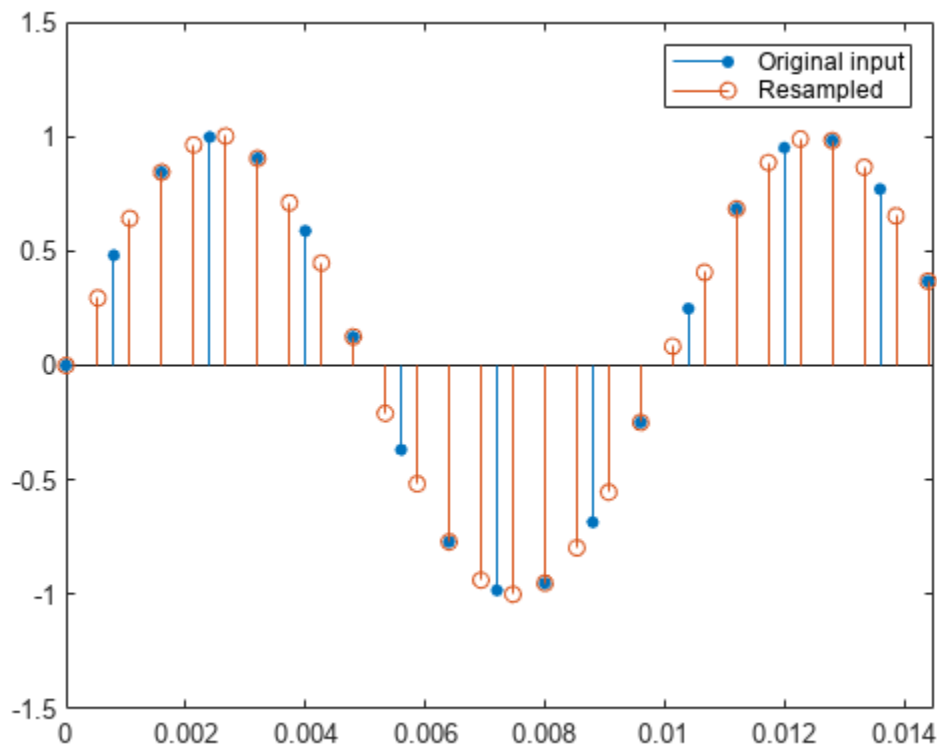
```
y = firrc(x);
```

Plot the original and resampled signals. In order to plot the two signals on the same plot, you must account for the output delay of the FIR rate converter and the scaling introduced by the filter. Use the `outputDelay` function to compute the delay value introduced by the rate converter. Shift the output by this delay value.

Visualize the input and the resampled signals. The input and output values coincide every 3 output samples, and every other input sample, owing to the conversion rate of 3/2.

```
FsIn = sine.SampleRate;
[delay,FsOut] = outputDelay(firrc,FsIn=FsIn);
% Calculate output times for vector y in input units
ty = (0:length(y)-1)/FsOut-delay;

stem(tx,x,'filled',MarkerSize=4)
hold on
stem(ty,y)
hold off
xlim([0.0 0.0145])
ylim([-1.5 1.5])
legend('Original input','Resampled')
```



### Resample by a Factor of 5/3 in Automatic Filter Design Mode

Now change the interpolation factor to 5 and the decimation factor to 3. In order for the filter design to be updated automatically based on the new rate conversion factors, set the `NumeratorSource` property to `'Auto'`. Alternately, you can pass the keyword `'auto'` as an input while creating the object. The object then operates in the automatic filter design mode. Every time there is a change in the rate conversion factors, the object updates the filter design accordingly.

```
release(firrc)
firrc.NumeratorSource = 'Auto';
firrc.InterpolationFactor = 5;
firrc.DecimationFactor = 3

firrc =
    dsp.FIRRateConverter with properties:

    Main
        InterpolationFactor: 5
        DecimationFactor: 3
        NumeratorSource: 'Auto'
        DesignMethod: 'Kaiser'

    Show all properties
```

To access the filter coefficients in the automatic filter design mode, type `firrc.Numerator` in the MATLAB command prompt.

Resample the signal with the updated rate conversion values.

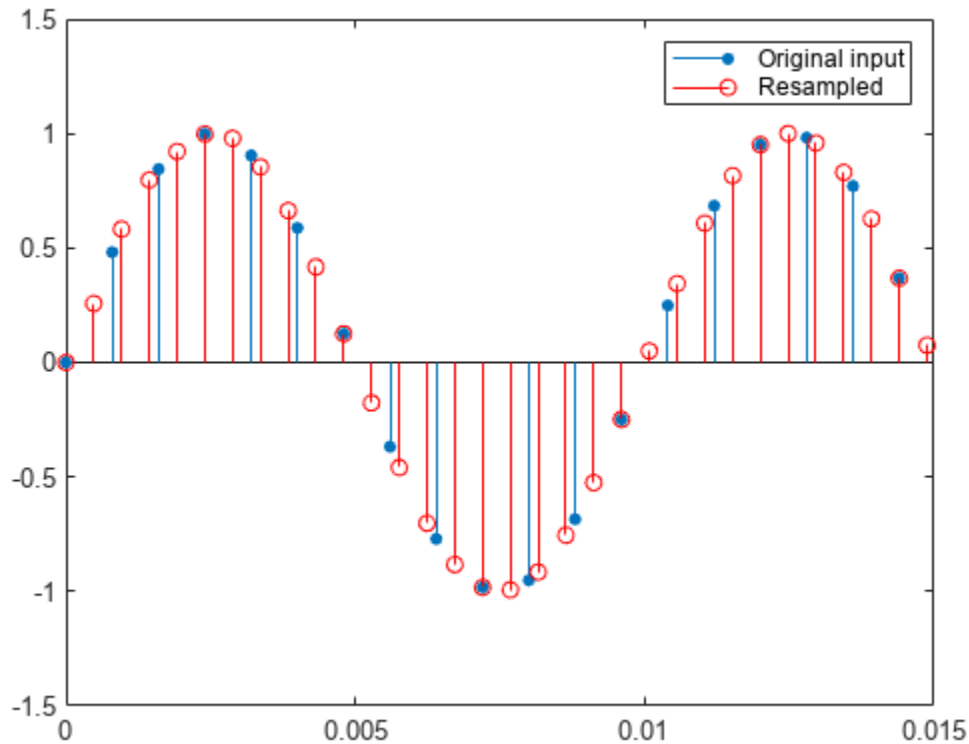
```
yAuto = firrc(x);
```

Plot the original and resampled signals. Recalculate the output delay and the output sample rate since the rate conversion factors have changed. Note that the input and output coincide every 3 input samples, and every 5 output samples, owing to the 5/3 conversion factor.

```
[delay,FsOut] = outputDelay(firrc,FsIn=FsIn);
% Calculate output times for vector yAuto in input units
tyAuto = (0:length(yAuto)-1)/FsOut-delay;

stem(tx,x,'filled',MarkerSize=4)
hold on
stem(tyAuto,yAuto,'r')
hold off
xlim([0.0 0.015])
ylim([-1.5 1.5])
legend('Original input','Resampled')
```





### Specify Signal Interpolation Model

In the automatic design mode, you can also specify the underlying D/A signal interpolation model through the `DesignMethod` property.

Set `DesignMethod` to `'linear'` and change the interpolation factor to 11.

```
release(firrc)
firrc.DesignMethod = 'linear';
firrc.InterpolationFactor = 11;
```

Resample the signal using the linear interpolation model.

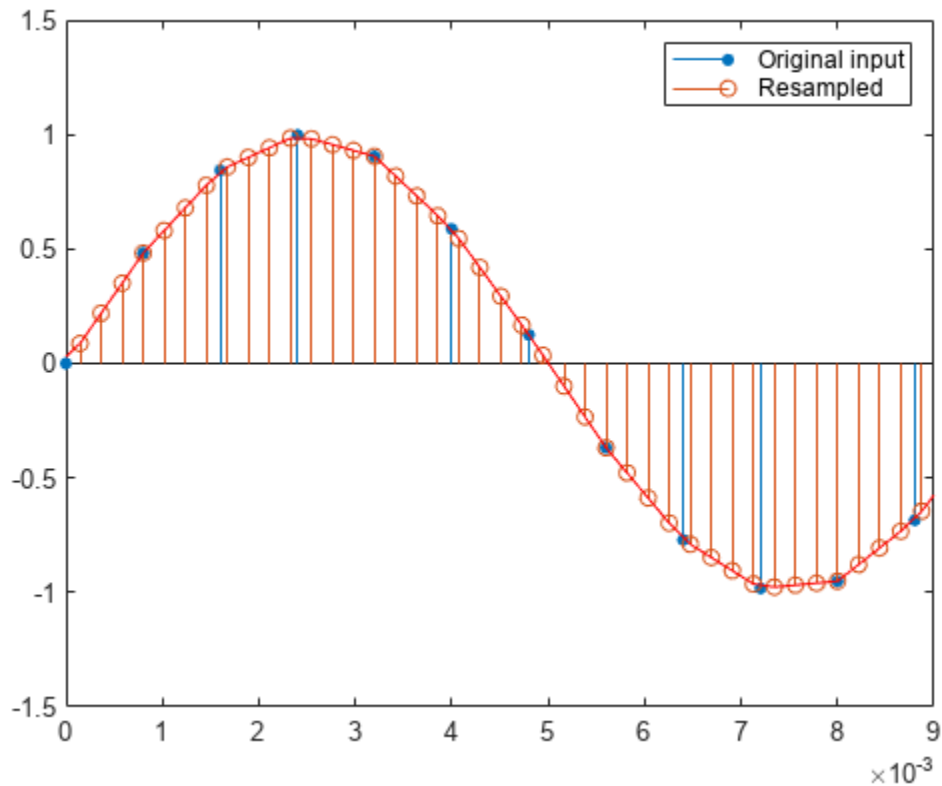
```
yLinear = firrc(x);
```

Plot the original and resampled signals. The output samples lie on a piecewise-linear curve. Note that the input and output coincide every three input samples and every 11 output samples as expected from the ratio 11/3.

```
[delay,FsOut] = outputDelay(firrc,FsIn=FsIn);
% Calculate output times for vector yLinear in input units
tyLinear = (0:length(yLinear)-1)/FsOut-delay;
```

```
stem(tx,x,'filled',MarkerSize=4)
hold on
stem(tyLinear,yLinear)
plot(tyLinear,yLinear,Color=[1 0 0 0.3])
hold off
```

```
xlim([0.0 0.009])
ylim([-1.5 1.5])
legend('Original input','Resampled')
```



### Resample and Play Audio Signal

Resample an audio signal from 48 kHz to 44 kHz and play the resampled signal using the `audioDeviceWriter` object.

**Note:** The `audioDeviceWriter` System object™ is not supported in MATLAB Online.

Create a `dsp.AudioFileReader` object. The object reads an audio file that has a sample rate of 48 kHz.

```
L = 11;
M = 12;
afr = dsp.AudioFileReader('audio48kHz.wav', ...
    'OutputDataType','single', ...
    'SamplesPerFrame',4*M);
```

Create a `dsp.FIRRateConverter` object with an interpolation factor of  $L = 11$  and a decimation factor of  $M = 12$  (the co-prime representation of the ratio  $44k/48k$ ), a reasonable approximation of the standard 44.1 kHz rate. The object designs the filter using `designMultirateFIR(11,12)` and stores the coefficients in the `Numerator` property of the object.

```
firrc = dsp.FIRRateConverter(L,M)
```

```
firrc =
  dsp.FIRRateConverter with properties:

  Main
    InterpolationFactor: 11
    DecimationFactor: 12
    NumeratorSource: 'Property'
    Numerator: [0 2.3076e-05 5.4790e-05 9.3620e-05 1.3665e-04 1.7959e-04 2.1699e-04 2.4...
```

Show all properties

Create an `audioDeviceWriter` object. Specify the sample rate to be 44100 Hz.

```
adw = audioDeviceWriter(44100);
```

Read the audio file, convert the sample rate of the audio signal, and play the resampled audio.

```
while ~isDone(afr)
    audio1 = afr();
    audio2 = firrc(audio1);
    adw(audio2);
end
release(afr)
release(adw)
```

## Algorithms

The FIR rate converter is implemented efficiently using a polyphase structure.

To derive the polyphase structure, start with the transfer function of the FIR filter: This FIR filter is a combined anti-imaging and anti-aliasing filter.

$$H(z) = b_0 + b_1z^{-1} + \dots + b_Nz^{-N}$$

$N+1$  is the length of the FIR filter.

You can rearrange this equation as follows:

$$H(z) = \begin{aligned} & (b_0 + b_Lz^{-L} + b_{2L}z^{-2L} + \dots + b_{N-L+1}z^{-(N-L+1)}) + \\ & z^{-1}(b_1 + b_{L+1}z^{-L} + b_{2L+1}z^{-2L} + \dots + b_{N-L+2}z^{-(N-L+1)}) + \\ & \quad \vdots \\ & z^{-(L-1)}(b_{L-1} + b_{2L-1}z^{-L} + b_{3L-1}z^{-2L} + \dots + b_Nz^{-(N-L+1)}) \end{aligned}$$

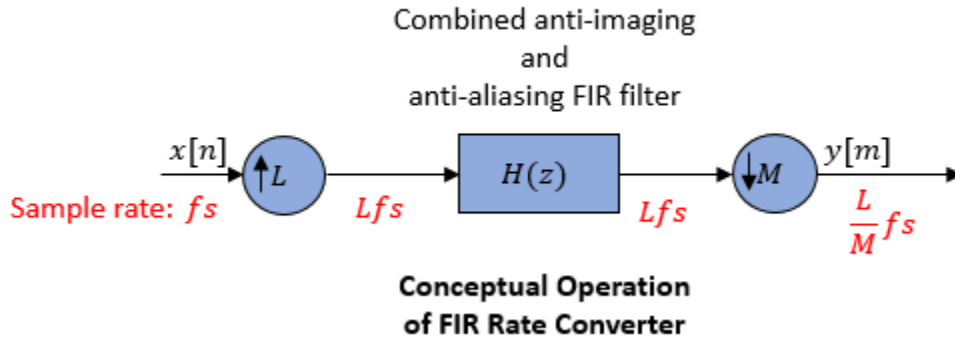
$L$  is the number of polyphase components, and its value equals the interpolation factor that you specify.

You can write this equation as:

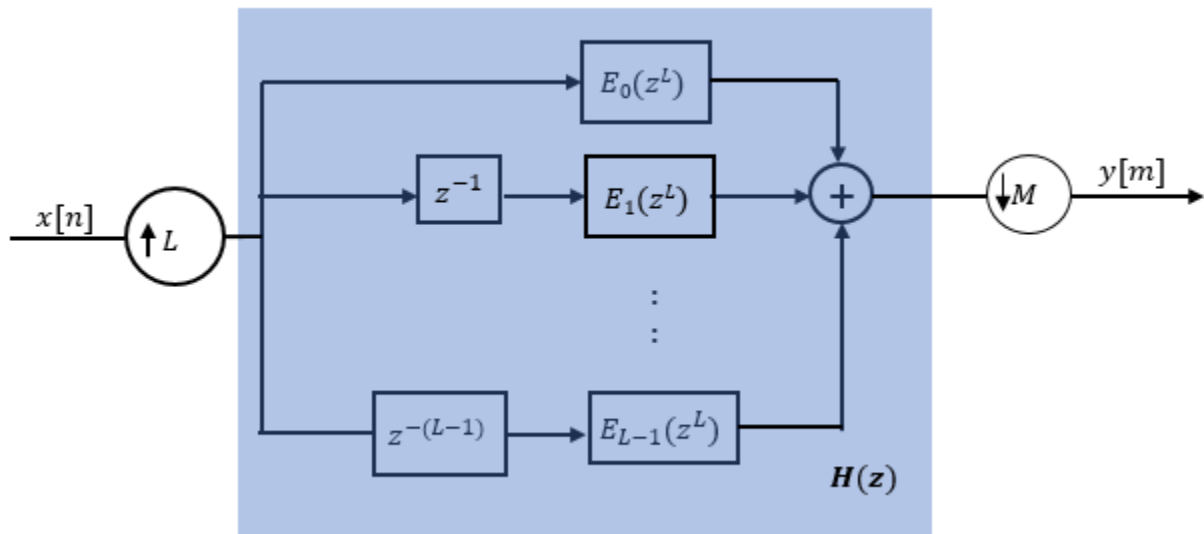
$$H(z) = E_0(z^L) + z^{-1}E_1(z^L) + \dots + z^{-(L-1)}E_{L-1}(z^L)$$

$E_0(z^L), E_1(z^L), \dots, E_{L-1}(z^L)$  are polyphase components of the FIR filter  $H(z)$ .

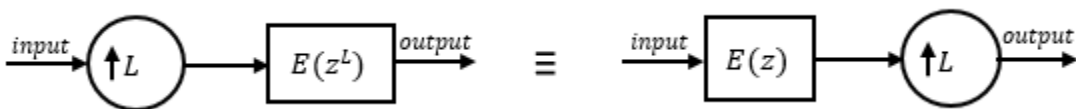
Conceptually, the FIR rate converter contains an upsampler, followed by a combined anti-imaging, anti-aliasing FIR filter  $H(z)$ , which is followed by a downsampler.



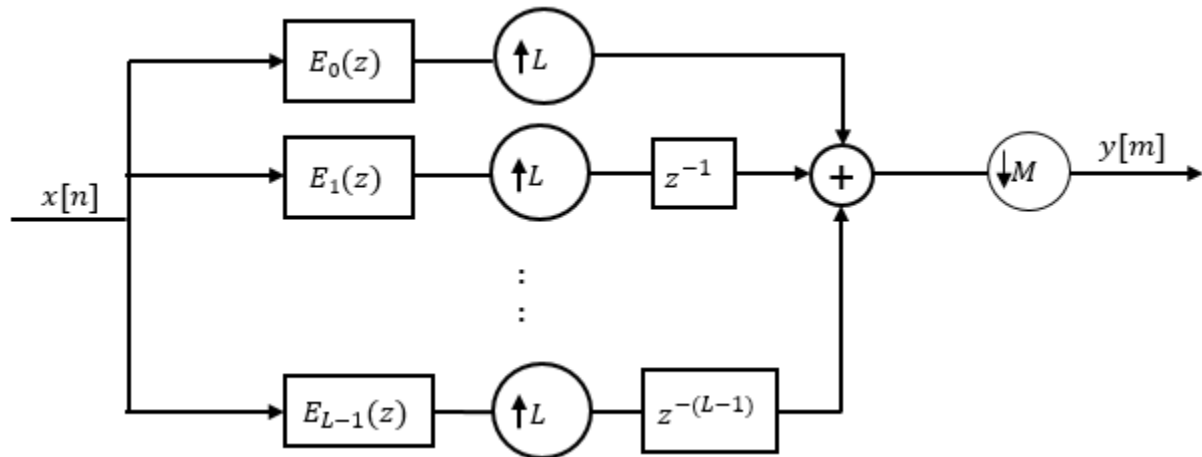
Replace  $H(z)$  with its polyphase representation.



Here is the multirate noble identity for interpolation.



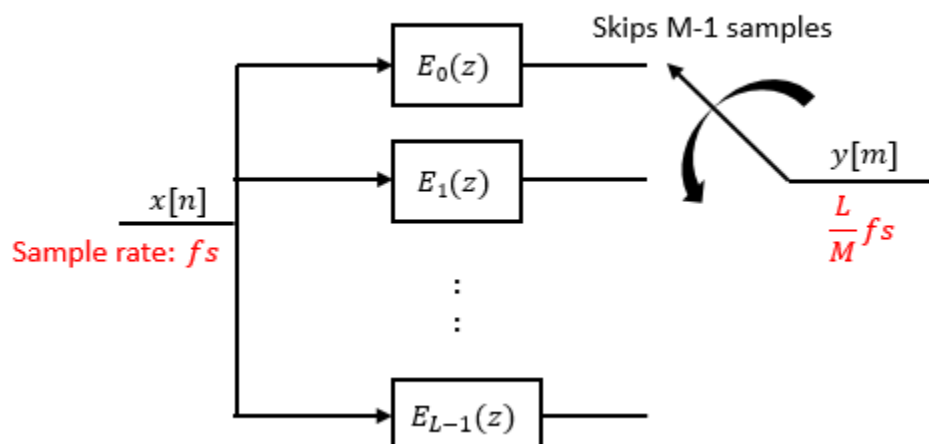
Applying the noble identity for interpolation moves the upsampling operation to after the filtering operation. This move enables you to filter the signal at a lower rate.



You can replace the upsampling operator, delay block, and the adder with a commutator switch. To account for the downsampler that follows, the switch moves in steps of size  $M$ . The switch receives the first sample from branch 0 and moves in the counter clockwise direction, each time skipping  $M-1$  branches.

As an example, consider a rate converter with  $L$  set to 5 and  $M$  set to 3. The polyphase components are  $E_0(z)$ ,  $E_1(z)$ ,  $E_2(z)$ ,  $E_3(z)$ , and  $E_4(z)$ . The switch starts on the first branch 0, skips branches 1 and 2, receives the next sample from branch 3, then skips branches 4 and 0, receives the next sample from branch 2, and so on. The sequence of branches from which the switch receives the data sample is [0, 3, 1, 4, 2, 0, 3, 1, ...].

The rate converter implements the  $L/M$  conversion by first applying the interpolation factor  $L$  to the incoming data, and using the commutator switch at the end to receive only 1 in  $M$  samples, effectively accounting for the downsampling factor  $M$ . Hence, the sample rate at the output of the FIR rate converter is  $Lfs/M$ .



## Version History

Introduced in R2012a

### **R2022b: Support for arbitrary input frame length**

Starting in R2022b, this object supports an input signal with an arbitrary frame length, so the input frame length does not have to be a multiple of the decimation factor.

When you generate code, to support arbitrary frame length for fixed-size signals, you must set the `AllowArbitraryInputLength` property to `true` while generating code.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

This object supports HDL code generation with the Filter Design HDL Coder product. For workflows and limitations, see “Generate HDL Code for Filter System Objects” (Filter Design HDL Coder).

## **See Also**

### **Functions**

`freqz` | `fvtool` | `info` | `cost` | `coeffs` | `polyphase` | `generatehdl`

### **Objects**

`dsp.FIRInterpolator` | `dsp.FIRDecimator`

### **Blocks**

FIR Rate Conversion

### **Topics**

“Analysis Methods for Filter System Objects” on page 3-2

# dsp.FrequencyDomainAdaptiveFilter

**Package:** dsp

Compute output, error, and coefficients using frequency-domain FIR adaptive filter

## Description

The `dsp.FrequencyDomainAdaptiveFilter` System object implements an adaptive finite impulse response (FIR) filter in the frequency domain using the fast block least mean squares (LMS) algorithm. The “Length” on page 4-0 and the “BlockLength” on page 4-0 properties specify the filter length and the block length values the algorithm uses. The “FFTCoefficients” on page 4-0 property contains the discrete Fourier transform of the current filter coefficients. The object offers the constrained and unconstrained versions of the algorithm with partitioned and non-partitioned modes. For details, see “Algorithms” on page 4-598.

To filter a signal using frequency-domain FIR adaptive filter:

- 1 Create the `dsp.FrequencyDomainAdaptiveFilter` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
fdaf = dsp.FrequencyDomainAdaptiveFilter
fdaf = dsp.FrequencyDomainAdaptiveFilter(len)
fdaf = dsp.FrequencyDomainAdaptiveFilter( ___,Name,Value)
```

### Description

`fdaf = dsp.FrequencyDomainAdaptiveFilter` returns a frequency domain FIR adaptive filter System object, `fdaf`. This System object is used to compute the filtered output and the filter error for a given input and desired signal.

`fdaf = dsp.FrequencyDomainAdaptiveFilter(len)` returns a frequency domain FIR adaptive filter object with the `Length` property set to `len`.

`fdaf = dsp.FrequencyDomainAdaptiveFilter( ___,Name,Value)` returns a frequency domain FIR adaptive filter object with each specified property set to the specified value. Enclose each property name in quotes. You can use this syntax with any previous input argument combinations.

Example: `fdaf = dsp.FrequencyDomainAdaptiveFilter('Length',32,'StepSize',0.1)` models a frequency-domain adaptive filter with a length of 32 taps and a step size of 0.1.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see *System Design in MATLAB Using System Objects*.

### Method — Method to calculate filter coefficients

'Constrained FDAF' (default) | 'Unconstrained FDAF' | 'Partitioned constrained FDAF' | 'Partitioned unconstrained FDAF'

Method used to calculate the filter coefficients, specified as:

- 'Constrained FDAF' -- Imposes a gradient constraint on the filter tap weights.
- 'Unconstrained FDAF' -- No gradient constraint is imposed on the filter tap weights.
- 'Partitioned constrained FDAF' -- Partitions the impulse response of the filter to reduce latency.
- 'Partitioned unconstrained FDAF' -- Partitions the impulse response of the filter to reduce latency. No gradient constraint is imposed on the filter tap weights.

For more details, see “Algorithms” on page 4-598.

### Length — Length of filter coefficients vector

32 (default) | positive, integer-valued scalar

Length of the FIR filter coefficients vector, specified as a positive, integer-valued scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### BlockLength — Block length for coefficient updates

32 (default) | positive, integer-valued scalar

Block length for the coefficient updates, specified as a positive, integer-valued scalar. The adaptive filter processes the input data and the desired signal as a block of samples of length set by this property. For details on how this data is processed by the filter, see “Algorithms” on page 4-598. The length of the input vector must be divisible by the `BlockLength` property value. The default value of the `BlockLength` property is set to the value of the `Length` property.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### StepSize — Adaptation step size

1 (default) | real scalar in the range (0,1]

Adaptation step size factor, specified as a real scalar in the range (0,1]. Using a small step size ensures a small steady-state error. However, a small step size decreases the resulting convergence speed of the adaptive filter. Increasing the step size improves the convergence speed, at the cost of increased steady-state mean squared error. When the step size value is 1, the algorithm provides the optimal tradeoff between the convergence speed and the steady-state mean squared error.

**Tunable:** Yes



Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical

### **LeakageFactor — Adaptation leakage factor**

1 (default) | scalar in the range [0,1]

Leakage factor used when implementing a leaky adaptive filter, specified as a scalar numeric value in the range [0,1]. When the value is less than 1, the System object implements a leaky adaptive algorithm. When the value is 1, the object provides no leakage in the adapting method.

**Tunable:** Yes

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical

### **AveragingFactor — Averaging factor for signal power**

0.9 (default) | real scalar in the range (0,1]

Averaging factor used to compute the exponentially windowed FFT input signal powers for the coefficient updates, specified as a real scalar in the range (0,1].

**Tunable:** Yes

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical

### **Offset — Offset for normalization terms**

0 (default) | nonnegative real scalar

Offset for the normalization terms in the coefficient updates, specified as a nonnegative real scalar value. This property value is used to avoid division by zero or division by very small numbers if any of the FFT input signal powers become very small.

**Tunable:** Yes

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### **InitialPower — Initial FFT input signal power**

1 (default) | positive numeric scalar

Initial common value of all of the FFT input signal powers, specified as a positive numeric scalar.

If you change this value once the object is locked, the change takes effect only after you reset the object.

**Tunable:** Yes

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### **InitialCoefficients — Initial time-domain coefficients of filter**

0 (default) | scalar | vector

Initial time-domain coefficients of the adaptive filter, specified as a scalar or a vector of length equal to the `Length` property value. The adaptive filter object uses these coefficients to compute the initial frequency-domain filter coefficients.

If you change this value once the object is locked, the change takes effect only after you reset the object.

**Tunable:** Yes

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **LockCoefficients — Locked status of coefficient updates**

`false` (default) | `true`

Locked status of the coefficient updates, specified as:

- `false` -- The object continuously updates the filter coefficients.
- `true` -- The filter coefficients do not get updated and their values remain at the current value.

**Tunable:** Yes

Data Types: `logical`

### **FFTCoefficients — Current FFT coefficients of filter**

`[]` (default) | row vector

This property is read-only.

Current discrete Fourier transform of the filter coefficients, returned as a row vector. For 'Constrained FDAF' and 'Unconstrained FDAF' algorithms, the length of this vector is equal to the sum of the `Length` value and the `BlockLength` value. This property is initialized to the FFT values of the `InitialCoefficients` property. To get the discrete Fourier transform of the filter coefficients, call the object, and access the `FFTCoefficients` property of the object.

Data Types: `single` | `double`

Complex Number Support: Yes

## **Usage**

### **Syntax**

```
[y,err] = fdaf(x,d)
```

### **Description**

`[y,err] = fdaf(x,d)` filters the input signal, `x`, using `d` as the desired signal, and returns the filtered output in `y` and the filter error in `err`. The System object estimates the filter weights needed to minimize the error between the output signal and the desired signal. The FFT of these filter weights can be obtained by accessing the `FFTCoefficients` property after calling the object algorithm.

### **Input Arguments**

#### **x — Data input**

column vector

The signal to be filtered by the frequency-domain FIR adaptive filter. The input, `x`, and the desired signal, `d`, must have the same size and data type. The length of the input vector must be divisible by the “`BlockLength`” on page 4-0 property value.

The input,  $x$ , can be a variable-size signal as long as the frame length is a multiple of the `BlockLength`. You can change the number of elements in the column vector even when the object is locked. The System object locks when you call the object to run its algorithm.

Data Types: `single` | `double`

### **d — Desired signal**

column vector

The frequency domain adaptive filter adapts its filter weights to minimize the error, `err`, and converge the input signal,  $x$ , to the desired signal,  $d$ , as closely as possible.

The input signal and the desired signal must have the same size and data type. The length of the desired signal vector must be divisible by the `BlockLength` property value.

The input signal can be a variable-size signal as long as the frame length is a multiple of the `BlockLength`. You can change the number of elements in the column vector even when the object is locked. The System object locks when you call the object to run its algorithm.

Data Types: `single` | `double`

### **Output Arguments**

#### **y — Filtered output**

column vector

Filtered output, returned as a column vector. The object adapts its filter weights to converge the input signal,  $x$ , to match the desired signal,  $d$ . The filter outputs the converged signal.

Data Types: `single` | `double`

#### **err — Difference between output and desired signal**

column vector

Difference between the output signal and the desired signal, returned as a column vector. The objective of the adaptive filter is to minimize this error. The object adapts its weights to converge towards optimal filter weights which produce an output signal that matches the desired signal as closely as possible. For more details on how `err` is computed, see [2].

Data Types: `single` | `double`

Complex Number Support: Yes

## **Object Functions**

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## **Common to All System Objects**

|                      |  |
|----------------------|--|
| <code>step</code>    | Run System object algorithm  |
| <code>release</code> | Release resources and allow changes to System object property values and input characteristics |
| <code>reset</code>   | Reset internal states of System object   |

## Examples

### QPSK Adaptive Equalization With FIR Filter

Transmit a quadrature phase shift keying (QPSK) signal across a noisy transmission channel. Minimize the noise in the received signal using a frequency-domain adaptive filter.

The QPSK signal,  $s$ , is transmitted across a noisy channel. The numerator and the denominator coefficients of the channel are contained in the vectors  $b$  and  $a$ , respectively. The received signal,  $r$ , obtained at the end of the transmission channel contains the transmitted QPSK signal and the noise added to the channel,  $n$ . The adaptive filter is used to extract the QPSK signal from the received noisy input. The desired signal,  $d$ , is the delayed version of the QPSK signal.

```
D = 16;
b = exp(1i*pi/4)*[-0.7 1];
a = [1 -0.7];
ntr = 1024;
s = sign(randn(1,ntr+D)) + 1i*sign(randn(1,ntr+D));
n = 0.1*(randn(1,ntr+D) + 1i*randn(1,ntr+D));
r = filter(b,a,s) + n;
x = r(1+D:ntr+D);
d = s(1:ntr);
```

Create a `dsp.FrequencyDomainAdaptiveFilter` object to model a frequency-domain adaptive filter of length 32 taps and a step size of 0.1. The adaptive filter accepts the delayed version of the received signal and the desired signal as inputs. The output of the adaptive filter is compared to the desired signal. The error between the two signals represents the noise added to the transmission channel. The adaptive filter updates its coefficients until this error becomes minimal. To get the discrete Fourier transform of the filter coefficients, call the `fdaf` object, and access the `FFTCoefficients` property of this object.

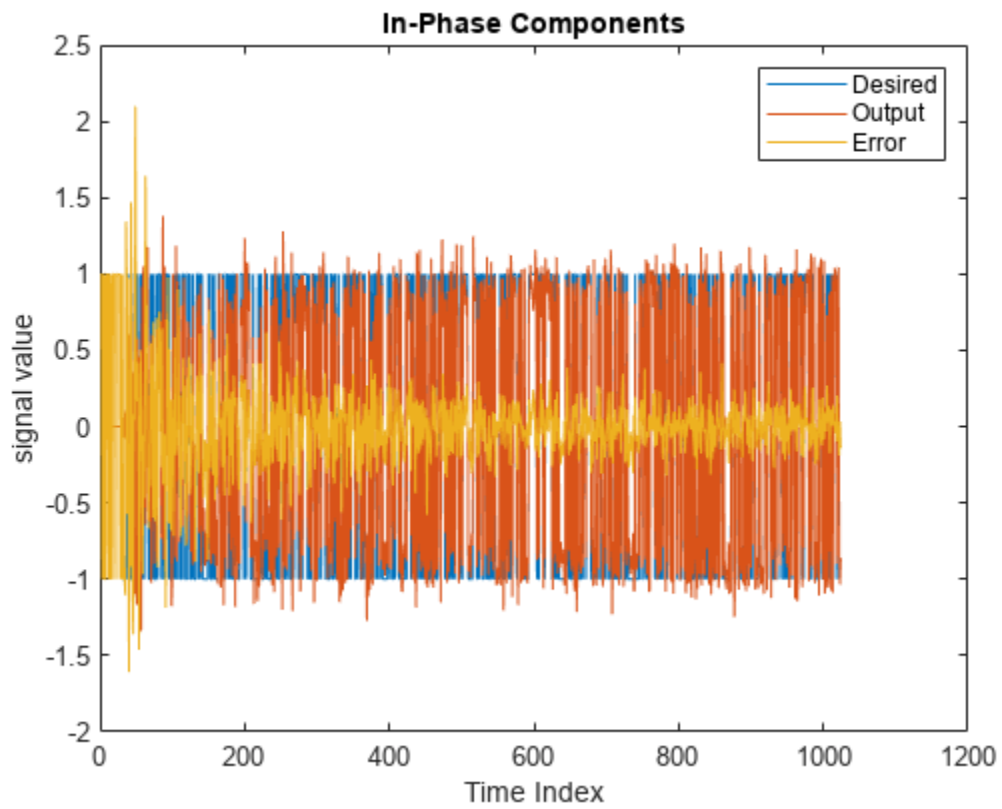
```
mu = 0.1;
fdaf = dsp.FrequencyDomainAdaptiveFilter('Length',32,'StepSize',mu);
[y,e] = fdaf(x,d);
fftCoeffs = fdaf.FFTCoefficients

fftCoeffs = 1x64 complex

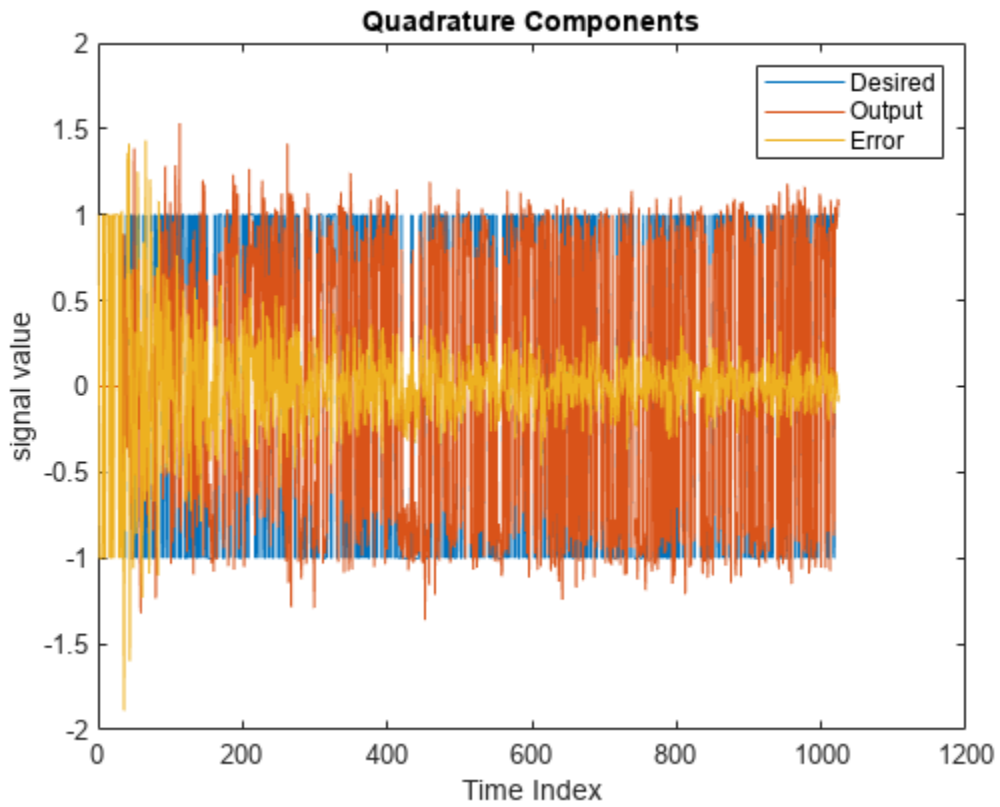
    0.6802 - 0.6847i  -0.2485 - 0.9427i  -0.9675 - 0.2123i  -0.5605 + 0.8002i   0.5748 + 0.7593i
```

Plot the In-Phase and the Quadrature components of the desired, output, and the error signals.

```
plot(1:ntr,real([d;y;e]))
legend('Desired','Output','Error')
title('In-Phase Components')
xlabel('Time Index'); ylabel('signal value')
```

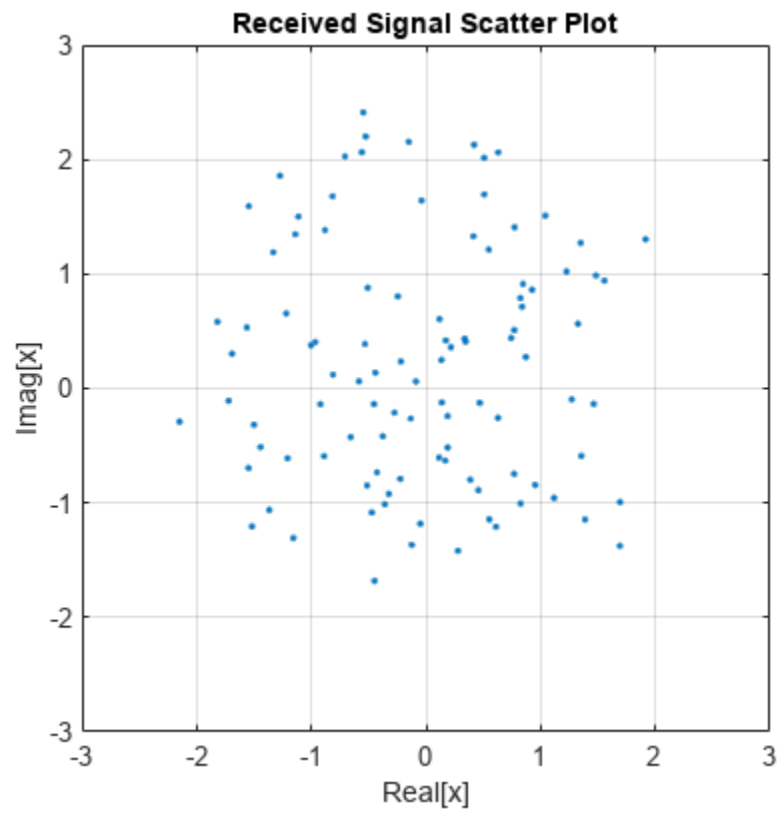


```
plot(1:ntr,imag([d;y;e]))  
legend('Desired','Output','Error')  
title('Quadrature Components')  
xlabel('Time Index')  
ylabel('signal value')
```

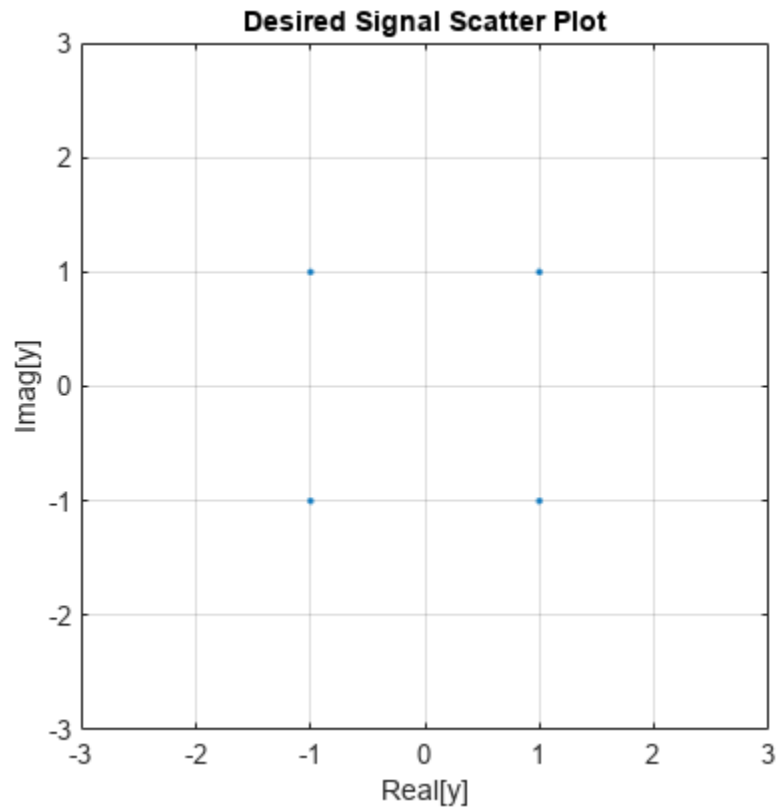


Create scatter plots of the received signal and the desired signal.

```
plot(x(ntr-100:ntr),'.')
axis([-3 3 -3 3])
title('Received Signal Scatter Plot')
axis('square')
xlabel('Real[x]')
ylabel('Imag[x]')
grid on
```



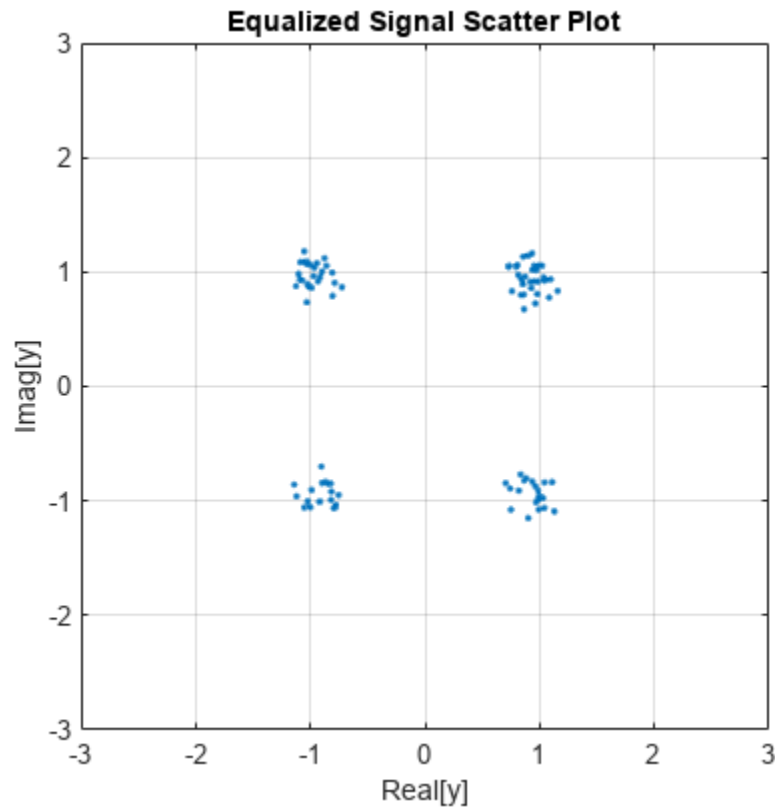
```
plot(d(ntr-100:ntr),'.')  
axis([-3 3 -3 3])  
title('Desired Signal Scatter Plot')  
axis('square')  
xlabel('Real[y]')  
ylabel('Imag[y]')  
grid on
```



The adaptive filter equalizes the received signal to eliminate noise. Plot the scatter plot of the equalized signal.

```
plot(y(ntr-100:ntr),'.')  
axis([-3 3 -3 3])  
title('Equalized Signal Scatter Plot')  
axis('square')  
xlabel('Real[y]')  
ylabel('Imag[y]')  
grid on
```





### Estimate Coefficients of long FIR Filter using Frequency-Domain Adaptive Filter

Use a frequency-domain adaptive filter to estimate the coefficients of a long FIR filter. The FIR filter models the impulse response of a room. Use the partitioned mode in the frequency-domain adaptive filter to reduce filter latency.

**Note:** This example runs only in R2018a or later.

#### Initialization

Generate a long FIR impulse response of 8192 samples and assign the impulse response to a `dsp.FIRFilter` object, `room`. This object models the impulse response of a room. Create a `dsp.FrequencyDomainAdaptiveFilter` filter, `lmsfilt`, in partitioned constrained mode. Set the length of the filter to one-fourth the length of the impulse response of the room. Set the block length of the filter to 128 samples. Set the step size to 0.025, initial power to 0.01, averaging factor to 0.98, offset to 1, and the leakage factor to 1. Initialize a `dsp.ArrayPlot` object to view the filter coefficients. Initialize a `timescope` object to show the mean-squared error between the filter output and the desired signal.

```
fs = 16e3;
m = 8192;
[b,a] = cheby2(4,20,[0.1 0.7]);
impulseResponseGenerator = dsp.IIRFilter(...
    'Numerator', [zeros(1,6) b], ...
```

```

    'Denominator', a);
roomImpulseResponse = impulseResponseGenerator( ...
    (log(0.99*rand(1,m)+0.01).*sign(randn(1,m)).*exp(-0.002*(1:m)))');
roomImpulseResponse = roomImpulseResponse/norm(roomImpulseResponse);
room = dsp.FIRFilter('Numerator',roomImpulseResponse');

lmsfilt = dsp.FrequencyDomainAdaptiveFilter(...
    'Method','Partitioned constrained FDAF',...
    'Length',m/4, ...
    'BlockLength',128,...
    'StepSize',0.025, ...
    'InitialPower',0.01, ...
    'AveragingFactor',0.98,...
    'Offset',1,...
    'LeakageFactor',1);

FrameSize = lmsfilt.BlockLength; NIter = 2000;

AP = dsp.ArrayPlot('YLimits',[-0.2 .2],'ShowLegend',true, ...
    'Position',[0 0 560 420],'ChannelNames', ...
    {'Actual Coefficients','Estimated Coefficients'});

TS = timescope('SampleRate',fs,'TimeSpanSource','property',...
    'TimeSpan',FrameSize*NIter/fs,...
    'TimeUnits','Seconds',...
    'YLimits',[-50 0],'Title','Learning curve',...
    'YLabel','dB', ...
    'BufferLength',FrameSize*NIter,...
    'ShowGrid',true);

signalmean = dsp.MovingAverage('SpecifyWindowLength',false);

```

### Streaming

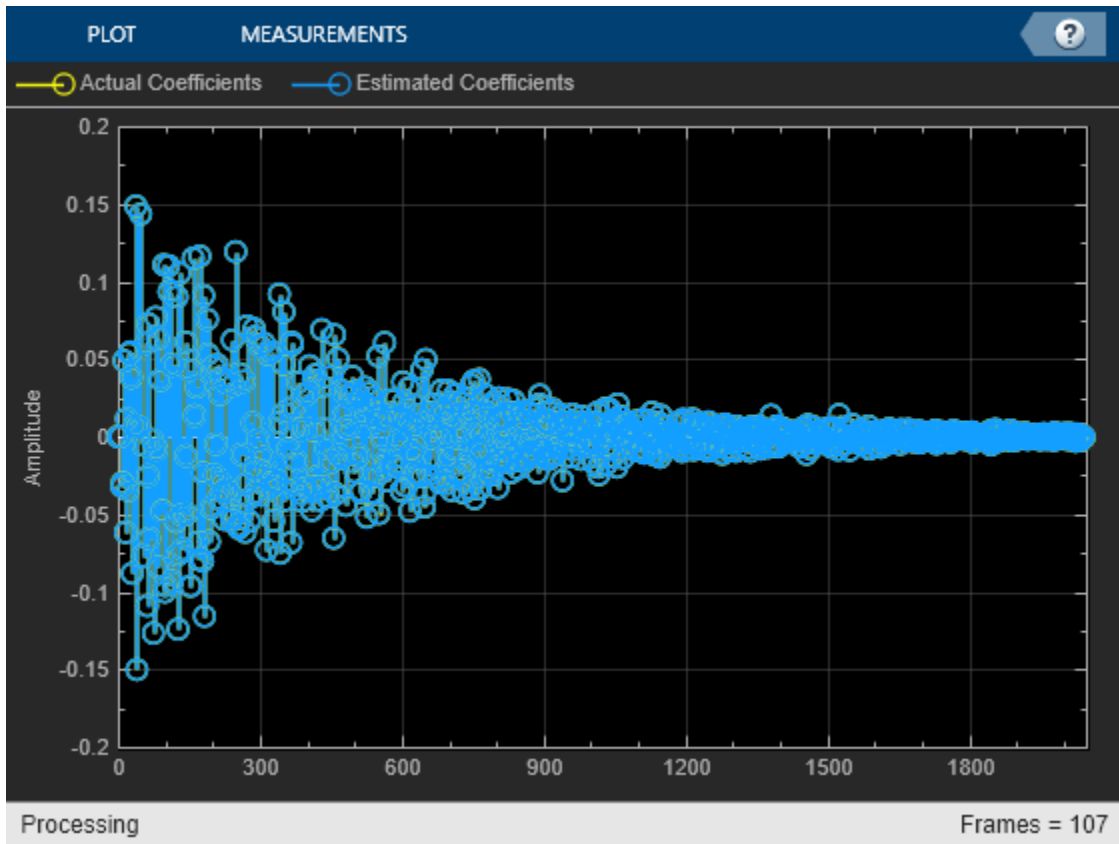
Generate a random input signal using the `randn` function. The frame size of the input matches the block length of the adaptive filter. The desired signal is the sum of the output of the FIR filter (`room`) and a white Gaussian noise signal. Pass the input signal and the desired signal to the adaptive filter. Compute the adaptive filter output and the error between the output and the desired signal.

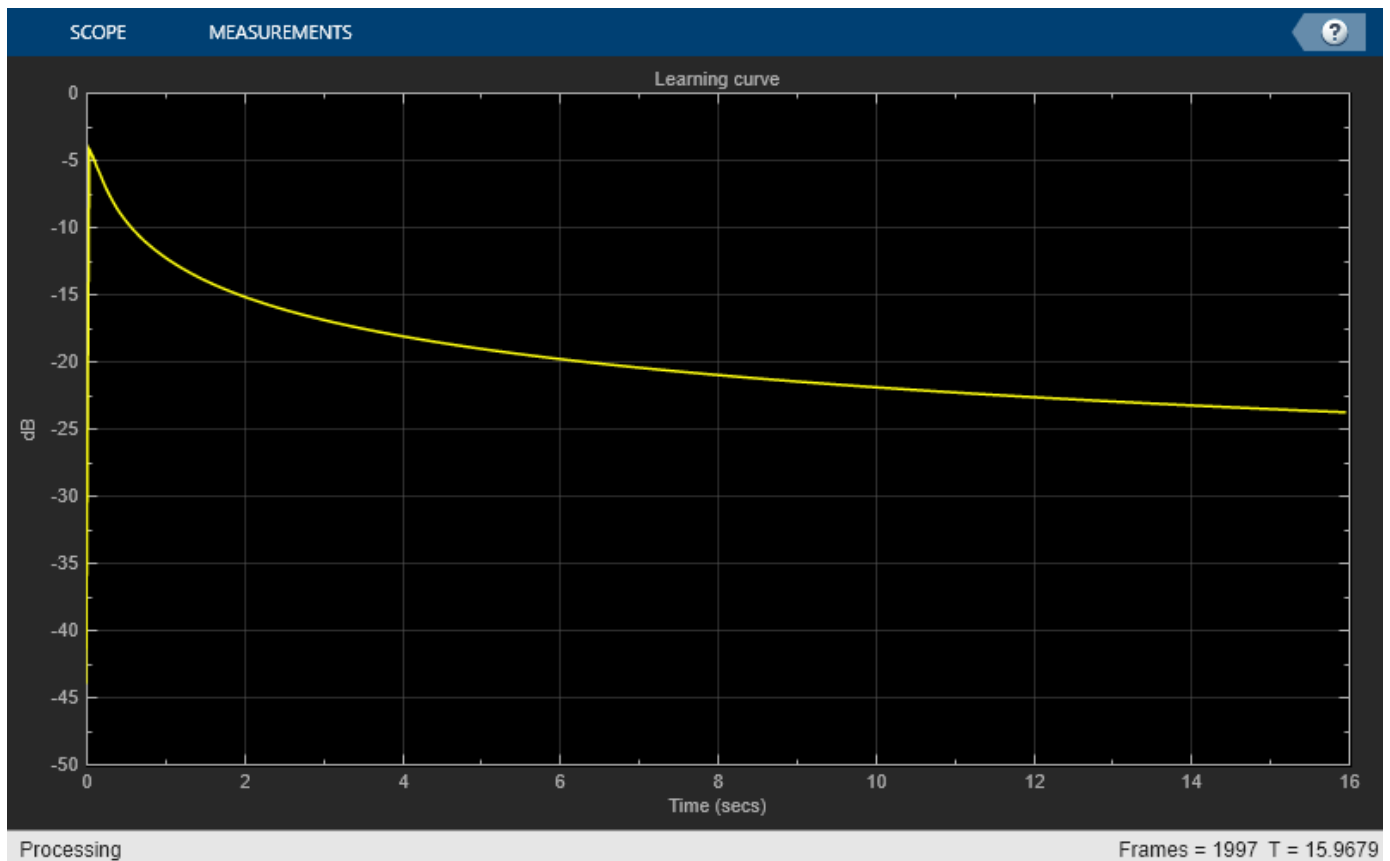
Estimate the time-domain coefficients of the adaptive filter by taking the IFFT of the frequency-domain coefficients vector returned by the `lmsfilt.FFTCoefficients` property. Compare the estimated coefficients with the actual coefficients assigned to the FIR filter (`room`). Once the adaptive filter has converged its output to the desired signal, and minimized the error signal, the estimated coefficients match closely with the actual coefficients. This means that the adaptive filter has successfully adapted itself to model the impulse response of the FIR filter (`room`).

```

for k = 1:NIter
    x = randn(FrameSize,1);
    d = room(x) + 0.01*randn(FrameSize,1);
    [y,e] = lmsfilt(x,d);
    FFTCoeffs = lmsfilt.FFTCoefficients;
    w = ifft(FFTCoeffs,[],2,'symmetric');
    w = w(:,1:FrameSize) + w(:,FrameSize+1:end);
    w = reshape(w.',1,m/4);
    AP([roomImpulseResponse(1:m/4),w.']);
    TS(10*log10(signalmean(abs(e).^2)));
end

```





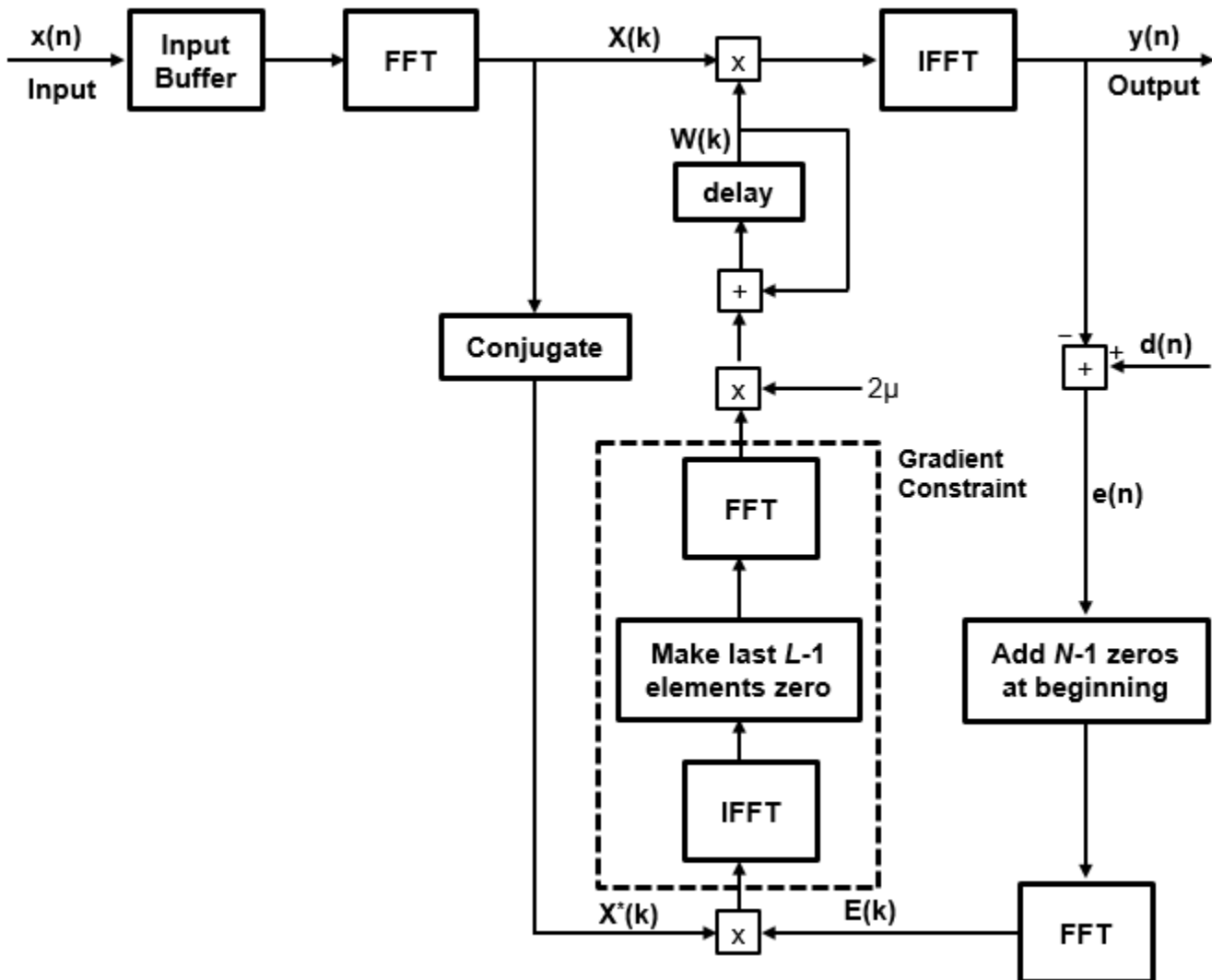
As the filter adapts with time, you can see in the time scope that the mean-squared error becomes minimal. Simultaneously, the estimated coefficients match the actual coefficients closely in the array plot.

## Algorithms

Frequency-domain adaptive filtering consists of three steps - filtering, error estimation, and tap-weight adaptation. This algorithm implements FIR filtering in the frequency domain using the overlap-save or overlap-add method. For more implementation details of these two methods, see the "Algorithms" on page 4-611 section in the `dsp.FrequencyDomainFIRFilter` object page. The error estimation and the tap-weight adaptation are implemented using the fast block LMS algorithm (FBLMS).

### Fast Block LMS Algorithm

The frequency-domain adaptive filter processes input data and the desired signal data as a block of samples using the fast block LMS (FBLMS) algorithm. Here is the block diagram of the frequency-domain adaptive filter using the FBLMS algorithm. The frequency-domain FIR filter in this diagram uses the overlap-save method.



where:

- $N$  -- Filter length
- $L$  -- Block length
- $\mu$  -- Step size parameter
- $x(n)$  -- Input signal
- $X(k)$  -- Transformed input signal in the frequency domain
- $d(n)$  -- Desired signal
- $e(n)$  -- Error between the desired signal and the filter output
- $E(k)$  -- Transformed error signal in the frequency domain
- $W(k)$  -- Tap-weights vector in the frequency domain

For more details on how the error is estimated and the tap-weights are adapted, see [2].

## Constrained and Unconstrained FBLMS Algorithms

The previous diagram is the constrained version. If you remove the gradient constraint portion of the algorithm, you have the unconstrained FBLMS implementation. For details on the convergence behavior of both constrained and unconstrained variations, see [2].

## Partitioned FBLMS Algorithm

The latency of the filter roughly equals the length of the FIR numerator. If the impulse response of the filter is very long, the latency becomes significantly large. The partitioned FBLMS algorithm reduces latency by partitioning the impulse response. The nonpartitioned frequency-domain FIR filtering is faster than the time-domain filtering for long impulse responses, at the cost of increased latency. To mitigate the latency and make the frequency domain filtering even more efficient, the algorithm partitions the impulse response into multiple short blocks and performs overlap-save or overlap-add on each block. The results of the different blocks are then combined to obtain the final output. The latency of this approach is of the order of the block length, rather than the entire impulse response length. This reduced latency comes at the cost of additional computation. For more details on the implementation, see [2].

## Version History

Introduced in R2013b

## References

- [1] Shynk, J.J. "Frequency-Domain and Multirate Adaptive Filtering." *IEEE Signal Processing Magazine*. Vol. 9, Number 1, 1992, pp. 14-37.
- [2] Farhang-Boroujeny, B., *Adaptive Filters: Theory and Applications*, Chichester, England, Wiley, 1998.
- [3] Stockham, T. G., Jr. "High Speed Convolution and Correlation." *Proceedings of the 1966 Spring Joint Computer Conference, AFIPS*, Vol. 28, 1966, pp. 229-233.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- "System Objects in MATLAB Code Generation" (MATLAB Coder).

## See Also

### Objects

`dsp.LMSFilter` | `dsp.RLSFilter` | `dsp.AffineProjectionFilter` |  
`dsp.AdaptiveLatticeFilter` | `dsp.FilteredXLMSFilter` | `dsp.FIRFilter` |  
`dsp.FastTransversalFilter` | `dsp.FrequencyDomainFIRFilter`

**Blocks**

LMS Filter | RLS Filter | LMS Update | Kalman Filter | Fast Block LMS Filter | Block LMS Filter |  
Frequency-Domain FIR Filter | Frequency-Domain Adaptive Filter

**Topics**

“Overview of Adaptive Filters and Applications”  
“Variable-Size Signal Support DSP System Objects”

# **dsp.FrequencyDomainFIRFilter**

**Package:** dsp

Filter input signal in frequency domain

## **Description**

The `dsp.FrequencyDomainFIRFilter` System object implements frequency-domain, fast Fourier transform (FFT)-based filtering to filter a streaming input signal. In the time domain, the filtering operation involves a convolution between the input and the impulse response of the finite impulse response (FIR) filter. In the frequency domain, the filtering operation involves the multiplication of the Fourier transform of the input and the Fourier transform of the impulse response. The frequency-domain filtering is efficient when the impulse response is very long. You can specify the filter coefficients directly in the frequency domain by setting `NumeratorDomain` to 'Frequency'.

This object uses the overlap-save and overlap-add methods to perform the frequency-domain filtering. For filters with a long impulse response length, the latency inherent to these two methods can be significant. To mitigate this latency, the `dsp.FrequencyDomainFIRFilter` object partitions the impulse response into shorter blocks and implements the overlap-save and overlap-add methods on these shorter blocks. To partition the impulse response, set the "PartitionForReducedLatency" on page 4-0 property to `true`. For more details on these two methods and on reducing latency through impulse response partitioning, see "Algorithms" on page 4-611.

To filter the input signal in the frequency domain:

- 1 Create the `dsp.FrequencyDomainFIRFilter` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## **Creation**

### **Syntax**

```
fdf = dsp.FrequencyDomainFIRFilter
fdf = dsp.FrequencyDomainFIRFilter(num)
fdf = dsp.FrequencyDomainFIRFilter(Name,Value)
```

### **Description**

`fdf = dsp.FrequencyDomainFIRFilter` creates a frequency domain FIR filter System object that filters each channel of the input signal independently over time in the frequency domain using the overlap-save or overlap-add method.

`fdf = dsp.FrequencyDomainFIRFilter(num)` creates a frequency domain FIR filter object with the "Numerator" on page 4-0 property set to `num`.

Example: `dsp.FrequencyDomainFIRFilter(fir1(400,2 * 2000 / 8000));`



`fdf = dsp.FrequencyDomainFIRFilter(Name, Value)` creates a frequency domain FIR filter System object with each specified property set to the specified value. Enclose each property name in single quotes. You can use this syntax with any previous input argument combinations.

Example: `dsp.FrequencyDomainFIRFilter('Method', 'Overlap-add');`

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

### Method — Frequency-domain filter method

'Overlap-save' (default) | 'Overlap-add'

Frequency-domain filter method, specified as either 'Overlap-save' or 'Overlap-add'. For more details on these two methods, see “Algorithms” on page 4-611.

### NumeratorDomain — Numerator domain

'Time' (default) | 'Frequency'

Domain of the filter coefficients, specified as one of the following:

- 'Time' -- Specify the time-domain filter numerator in the Numerator property.
- 'Frequency' -- Specify the filter's frequency response in the FrequencyResponse property.

### Numerator — FIR filter coefficients

`fir1(100, 0.3)` (default) | row vector

FIR filter coefficients, specified as a row vector.

**Tunable:** Yes

#### Dependencies

This property applies when NumeratorDomain is set to 'Time'.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`  
Complex Number Support: Yes

### FrequencyResponse — Filter frequency response

`fft(fir1(100, 0.3), 202)` (default) | row vector | matrix

Frequency response of the filter, specified as a row vector or a matrix. When `PartitionForReducedLatency` is `false`, `FrequencyResponse` must be a row vector. The FFT length is equal to the length of the `FrequencyResponse` vector. When `PartitionForReducedLatency` is `true`, `FrequencyResponse` must be a  $2P$ -by- $N$  matrix, where  $P$  is the partition size, and  $N$  is the number of partitions.

**Tunable:** Yes

**Dependencies**

This property applies when `NumeratorDomain` is set to 'Frequency'.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`  
Complex Number Support: Yes

**NumeratorLength — Time-domain numerator length**

101 (default) | positive integer-valued scalar

Time-domain numerator length, specified as a positive integer-valued scalar.

**Dependencies**

This property applies when `NumeratorDomain` is set to 'Frequency' and `PartitionForReducedLatency` is set to `false`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**FilterIsReal — Flag to specify if filter is real**

`true` (default) | `false`

Flag to specify if the filter coefficients are all real, specified as `true` or `false`.

**Dependencies**

This property applies when `NumeratorDomain` is set to 'Frequency'.

**PartitionForReducedLatency — Flag to partition numerator to reduce latency**

`false` (default) | `true`

Flag to partition numerator to reduce latency, specified as one of the following:

- `false` -- The filter uses the traditional overlap-save or overlap-add method. The latency in this case is “`FFTLength`” on page 4-0 - `length(“Numerator” on page 4-0 ) + 1`.
- `true` -- In this mode, the object partitions the numerator into segments of length specified by the “`PartitionLength`” on page 4-0 property. The filter performs overlap-save or overlap-add on each partition, and combines the partial results to form the overall output. The latency is now reduced to the partition length.

**FFTLength — FFT length**

[ ] (default) | positive integer

FFT length, specified as a positive integer. The default value of this property, [ ], indicates that the FFT length is equal to twice the numerator length. The FFT length must be greater than or equal to the numerator length.

Example: 64

**Dependencies**

This property applies when you set `NumeratorDomain` property to 'Time' and “`PartitionForReducedLatency`” on page 4-0 property to `false`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**PartitionLength — Numerator partition length**

32 (default) | positive integer

Numerator partition length, specified as a positive integer less than or equal to the length of the numerator.

Example: 40

Example: 60

### Dependencies

This property applies when you set the `NumeratorDomain` property to 'Time' and "PartitionForReducedLatency" on page 4-0 property to `true`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### Latency — Filter latency

102 (default) | positive integer

This property is read-only.

Filter latency in samples, returned as an integer greater than 0. When "PartitionForReducedLatency" on page 4-0 is `false`, the latency is equal to "FFTLength" on page 4-0 - length("Numerator" on page 4-0) + 1. When `PartitionForReducedLatency` is `true`, the latency is equal to the partition length.

Data Types: `uint32`

## Usage

### Syntax

```
fdfOut = fdf(input)
```

### Description

`fdfOut = fdf(input)` filters the input signal and outputs the filtered signal. The object filters each channel of the input signal independently over time in the frequency domain.

### Input Arguments

#### input — Data input

vector | matrix

Data input, specified as a vector or a matrix. This object supports variable-size input signals. That is, you can change the input frame size (number of rows) even after calling the algorithm. However, the number of channels (number of columns) must remain constant.

Example: `randn(164,4)`

Data Types: `single` | `double`

### Output Arguments

#### fdfOut — Filtered output

vector | matrix

Filtered output, returned as a vector or matrix. The size, data type, and complexity of the output match those of the input.

Data Types: `single` | `double`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Specific to `dsp.FrequencyDomainFIRFilter`

`fvtool` Visualize frequency response of DSP filters

## Common to All System Objects

`step` Run System object algorithm  
`release` Release resources and allow changes to System object property values and input characteristics  
`reset` Reset internal states of System object

## Examples

### Frequency Domain Filtering Using Overlap-Add and Overlap-Save

Filter input signal using overlap-add and overlap-save methods, and compare the outputs to the output of a FIR filter.

#### Initialization

Design the FIR lowpass filter coefficients using the `fir1` function. The sampling frequency is 8 kHz, and the cutoff frequency of the filter is 2 kHz. The impulse response has a length of 400.

```
impL = 400;  
Fs = 8000;  
Fcutoff = 2000;  
imp = fir1(impL, 2*Fcutoff/Fs);
```

Create two `dsp.FrequencyDomainFIRFilter` objects and a `dsp.FIRFilter` object. Set the numerator of all the three filters to `imp`. Delay the FIR output by the latency of the frequency-domain filter.

```
fdf0A = dsp.FrequencyDomainFIRFilter(imp, 'Method', 'overlap-add');  
fdf0S = dsp.FrequencyDomainFIRFilter(imp, 'Method', 'overlap-save');  
fir = dsp.FIRFilter('Numerator', imp);  
dly = dsp.Delay('Length', fdf0A.Latency);
```

Create two `dsp.SineWave` objects. The sine waves generated have a sample rate of 8000 Hz, frame size of 256, and frequencies of 100 Hz and 3 kHz, respectively. Create a `timescope` object to view the filtered outputs.

```
frameLen = 256;  
sin_100Hz = dsp.SineWave('Frequency', 100, 'SampleRate', Fs, ...  
    'SamplesPerFrame', frameLen);
```

```

sin_3KHz = dsp.SineWave('Frequency',3e3,'SampleRate',Fs,...
    'SamplesPerFrame',frameLen);
ts = timescope('TimeSpanOvverrunAction','Scroll',...
    'ShowGrid',true,'TimeSpanSource','Property','TimeSpan',5 * frameLen/Fs,...
    'YLimits',[-1.1 1.1],...
    'ShowLegend',true,...
    'SampleRate',Fs,...
    'ChannelNames',{'Overlap-add','Overlap-save','Direct-form FIR'});

```

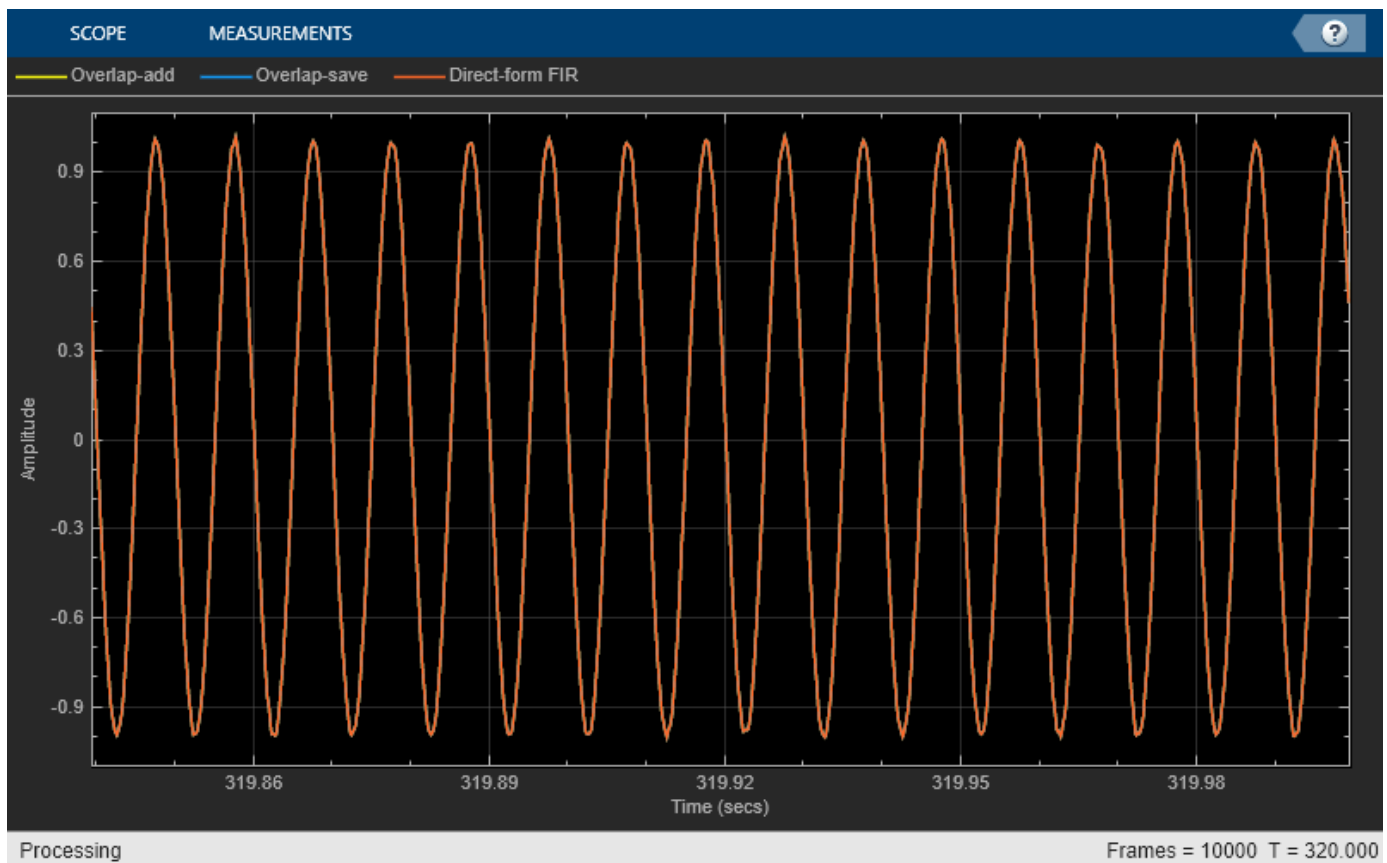
## Streaming

Stream 1e4 frames of noisy input data. Pass this data through the frequency domain filters and the FIR filter. View the filtered outputs in the time scope.

```

numFrames = 1e4;
for idx = 1:numFrames
    x = sin_100Hz() + sin_3KHz() + 0.01*randn(frameLen,1);
    yOA = fdfOA(x);
    yOS = fdfOS(x);
    yFIR = fir(dly(x));
    ts([yOA,yOS,yFIR]);
end

```



The outputs of all the three filters match exactly.

## Reduce Latency Through Partitioned Numerator

Partition the impulse response length of a frequency domain FIR filter. Compare the outputs of the partitioned filter and the original filter.

Design the FIR lowpass filter coefficients using the `fir1` function. The sampling frequency is 8 kHz and the cutoff frequency of the filter is 2 kHz. The impulse response is of length 4000.

```
impL    = 4000;
Fs      = 8000;
Fcutoff = 2000;
imp     = fir1(impL, 2 * Fcutoff / Fs);
```

Create a `dsp.FrequencyDomainFIRFilter` with coefficients set to the `imp` vector. The latency of this filter is given by  $FFTLength - Length(Numerator) + 1$ , which is equal to 4002. By default, FFT length is equal to twice the numerator length. This makes the latency proportional to the impulse response length.

```
fdf0S = dsp.FrequencyDomainFIRFilter(imp, 'Method', 'overlap-save');
fprintf('Frequency domain filter latency is %d samples\n', fdf0S.Latency);
```

```
Frequency domain filter latency is 4002 samples
```

Partition the impulse response to blocks of length 256. The latency after partitioning is proportional to the block length.

```
fdfRL = dsp.FrequencyDomainFIRFilter(imp, 'Method', 'overlap-save', ...
    'PartitionForReducedLatency', true, ...
    'PartitionLength', 256);
fprintf('Frequency domain filter latency is %d samples\n', fdfRL.Latency);
```

```
Frequency domain filter latency is 256 samples
```

Compare the outputs of the two frequency domain filters. The latency of `fdf0S` is 4002, and the latency of `fdfRL` is 256. To compare the two outputs, delay the input to `fdfRL` by 4002 - 256 samples.

```
dly = dsp.Delay('Length', (fdf0S.Latency - fdfRL.Latency));
```

Create two `dsp.SineWave` objects. The sine waves have a sample rate of 8000 Hz, frame size of 256, and frequencies of 100 Hz and 3 kHz, respectively. Create a `timescope` object to view the filtered outputs.

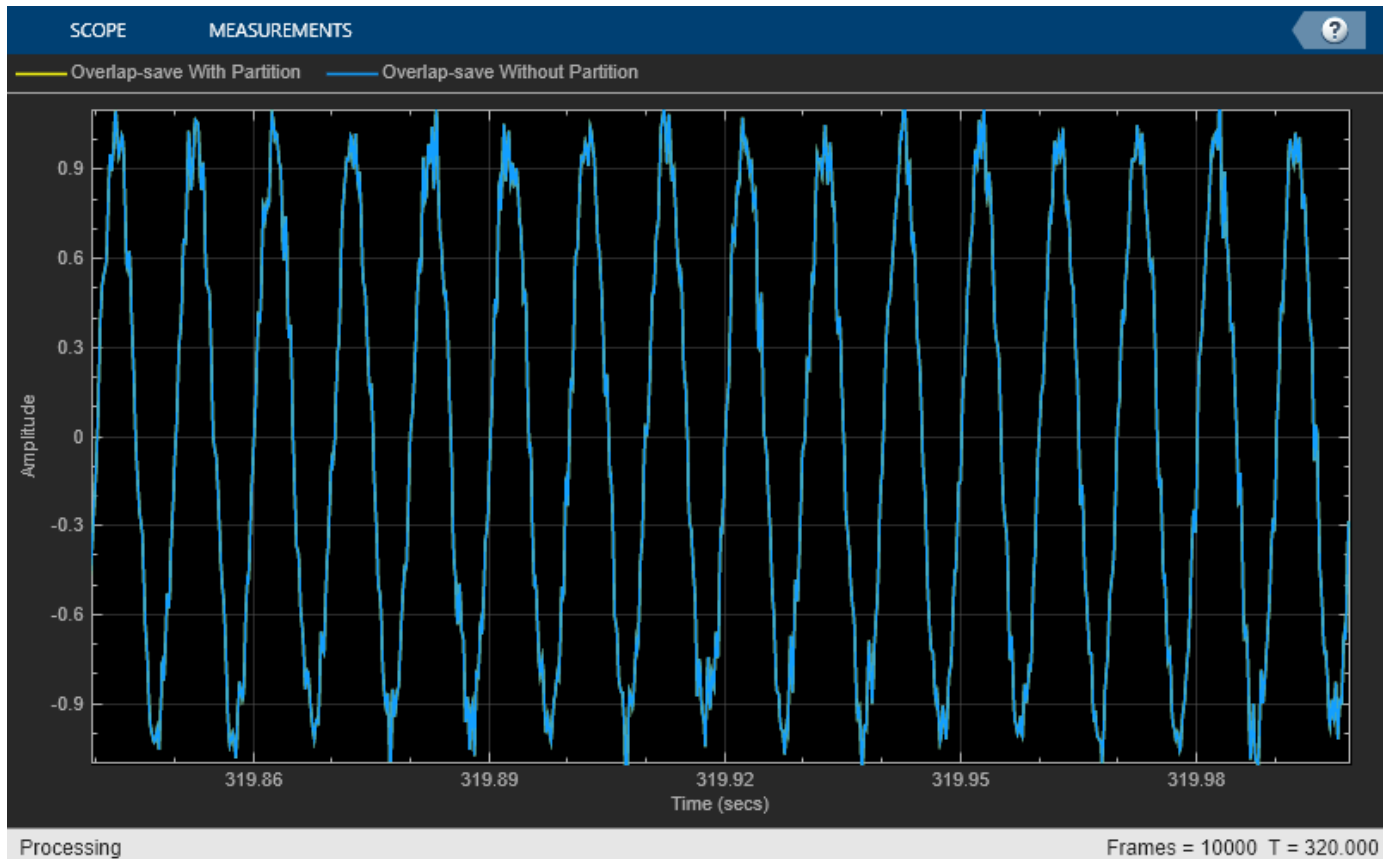
```
frameLen = 256;
sin_100Hz = dsp.SineWave('Frequency', 100, 'SampleRate', Fs, ...
    'SamplesPerFrame', frameLen);
sin_3KHz = dsp.SineWave('Frequency', 3e3, 'SampleRate', Fs, ...
    'SamplesPerFrame', frameLen);
ts = timescope('TimeSpanOvverrunAction', 'Scroll', 'ShowGrid', true, ...
    'TimeSpanSource', 'Property', 'TimeSpan', 5 * frameLen / Fs, ...
    'YLimits', [-1.1 1.1], ...
    'ShowLegend', true, ...
    'SampleRate', Fs, ...
    'ChannelNames', {'Overlap-save With Partition', 'Overlap-save Without Partition'});
```

Stream 1e4 frames of noisy input data. Pass this data through the two frequency domain filters. View the filtered outputs in the time scope.

```

numFrames = 1e4;
for idx = 1:numFrames
    x = sin_100Hz() + sin_3KHz() + .1 * randn(frameLen,1);
    yRL = fdfRL(dly(x));
    yOS = fdfOS(x);
    ts([yRL,yOS]);
end

```



The outputs match exactly.

### Specify Frequency Response of the Frequency-Domain FIR Filter

Specify the numerator coefficients of the frequency-domain FIR filter in the frequency domain. Filter the input signal using the overlap-add method. Compare the frequency-domain FIR filter output to the corresponding time-domain FIR filter output.

#### Initialization

Design the FIR lowpass filter coefficients using the `fir1` function. The sampling frequency is 8 kHz, and the cutoff frequency of the filter is 2 kHz. The time-domain impulse response has a length of 400. Compute the FFT of this impulse response and specify this response as the frequency response of the frequency-domain FIR filter. Set the time-domain numerator length, specified by the `NumeratorLength` property, to the number of elements in the time-domain impulse response.

```

impL    = 400;
Fs      = 8000;
Fcutoff = 2000;
imp     = fir1(impL,2 * Fcutoff / Fs);
H       = fft(imp , 2 * numel(imp));
oa      = dsp.FrequencyDomainFIRFilter('NumeratorDomain','Frequency',...
    'FrequencyResponse', H,...
    'NumeratorLength',numel(imp),...
    'Method','overlap-add');
fprintf('Frequency domain filter latency is %d samples\n',oa.Latency);

```

```
Frequency domain filter latency is 402 samples
```

Create a `dsp.FIRFilter` System object? and specify the numerator as the time-domain coefficients computed using the `fir1` function, `imp`. Delay the FIR output by the latency of the frequency-domain FIR filter.

```

fir = dsp.FIRFilter('Numerator',imp);
dly = dsp.Delay('Length',oa.Latency);

```

Create two `dsp.SineWave` objects. The sine waves generated have a sample rate of 8000 Hz, frame size of 256, and frequencies of 100 Hz and 3 kHz, respectively. Create a `timescope` object to view the filtered outputs.

```

frameLen = 256;

sin_100Hz = dsp.SineWave('Frequency',100,'SampleRate',Fs,...
    'SamplesPerFrame',frameLen);
sin_3KHz  = dsp.SineWave('Frequency',3e3,'SampleRate',Fs,...
    'SamplesPerFrame',frameLen);

ts = timescope('TimeSpanOvrrunAction','Scroll',...
    'ShowGrid',true,'YLimits',[-1.1 1.1],...
    'TimeSpanSource','Property','TimeSpan',5 * frameLen/Fs,...
    'ShowLegend',true,...
    'SampleRate',Fs,...
    'ChannelNames',{'Overlap-add','Direct-form FIR'});

```

### Streaming

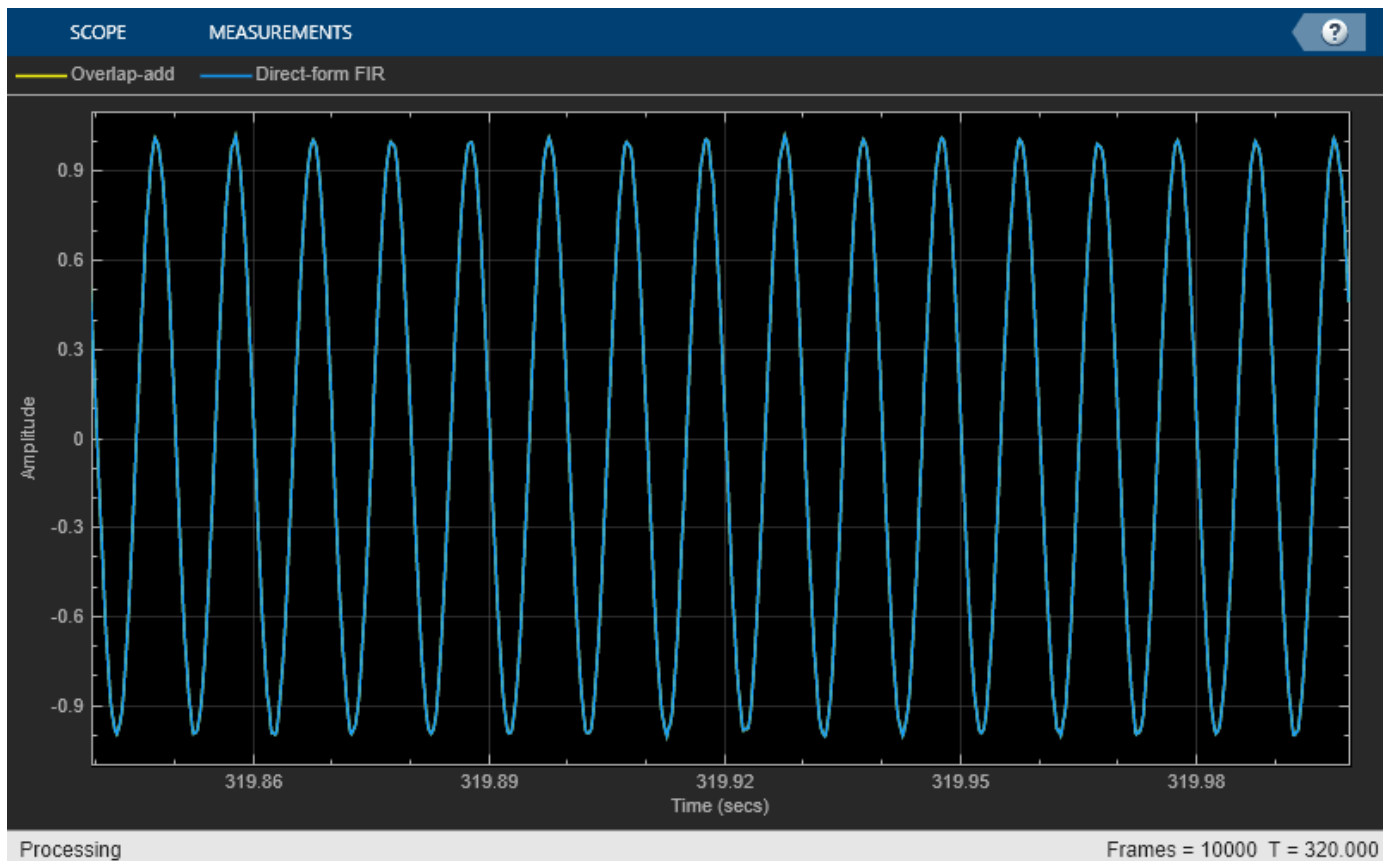
Stream 1e4 frames of noisy input data. Pass this data through the frequency-domain FIR filter and the time-domain FIR filter. View the filtered outputs in the time scope.

```

numFrames = 1e4;
for idx = 1:numFrames
    x = sin_100Hz() + sin_3KHz() + 0.01 * randn(frameLen,1);
    y1 = oa(x);
    y2 = fir(dly(x));
    ts([y1,y2]);
end

```





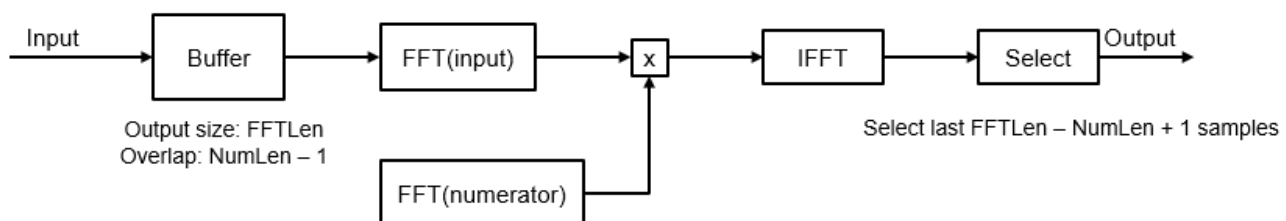
The outputs of both the filters match exactly.

## Algorithms

Overlap-save and overlap-add are the two frequency-domain FFT-based filtering methods this algorithm uses.

### Overlap-Save

The overlap-save method is implemented using the following approach:



The input stream is partitioned into overlapping blocks of size  $FFTLen$ , with an overlap factor of  $NumLen - 1$  samples.  $FFTLen$  is the FFT length and  $NumLen$  is the length of the FIR filter numerator. The FFT of each block of input samples is computed and multiplied with the length- $FFTLen$  FFT of

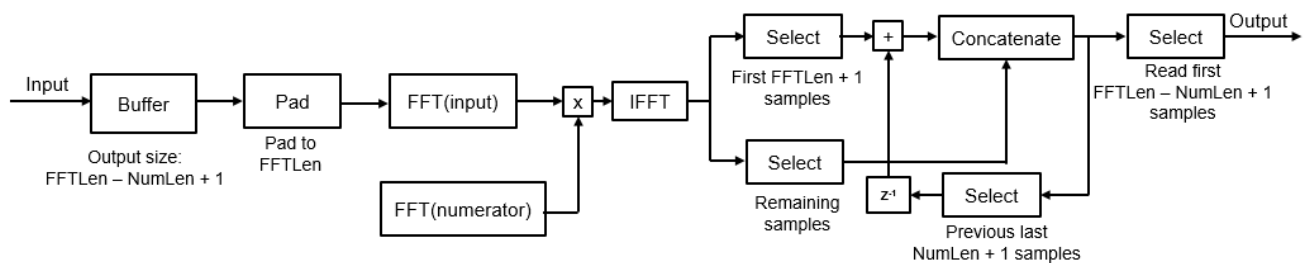
the FIR numerator. The inverse fast Fourier transform (IFFT) of the result is performed, and the last  $FFTLen - NumLen + 1$  samples are saved. The remaining samples are dropped.

The latency of overlap-save is  $FFTLen - NumLen + 1$ . The first  $FFTLen - NumLen + 1$  samples are equal to zero. The filtered value of the first input sample appears as the  $FFTLen - NumLen + 2$  output sample.

Note that the FFT length must be larger than the numerator length, and is typically set to a value much greater than  $NumLen$ .

### Overlap-Add

The overlap-add method is implemented using the following approach:



The input stream is partitioned into blocks of length  $FFLen - NumLen + 1$ , with no overlap between consecutive blocks. Similar to overlap-save, the FFT of the block is computed, and multiplied by the FFT of the FIR numerator. The IFFT of the result is then computed. The first  $NumLen + 1$  samples are modified by adding the values of the last  $NumLen + 1$  samples from the previous computed IFFT.

The latency of overlap-add is  $FFTLen - NumLen + 1$ . The first  $FFTLen - NumLen + 1$  samples are equal to zero. The filtered value of the first input sample appears as the  $FFTLen - NumLen + 2$  output sample.

### Reduce Latency Through Impulse Response Partitioning

With an FFT length that is twice the length of the FIR numerator, the latency roughly equals the length of the FIR numerator. If the impulse response is very long, the latency becomes significantly large. However, frequency domain FIR filtering is still faster than the time-domain filtering. To mitigate the latency and make the frequency domain filtering even more efficient, the algorithm partitions the impulse response into multiple short blocks and performs overlap-save or overlap-add on each block. The results of the different blocks are then combined to obtain the final output. The latency of this approach is of the order of the block length, rather than the entire impulse response length. This reduced latency comes at the cost of additional computation. For more details, see [1].

## Version History

Introduced in R2017b

## References

- [1] Stockham, T. G., Jr. "High Speed Convolution and Correlation." *Proceedings of the 1966 Spring Joint Computer Conference, AFIPS*, Vol 28, 1966, pp. 229-233.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

## See Also

### Functions

fvtool

### Objects

[dsp.FIRFilter](#) | [dsp.Delay](#) | [dsp.VariableBandwidthFIRFilter](#) |  
[dsp.FrequencyDomainAdaptiveFilter](#)

### Blocks

[Frequency-Domain FIR Filter](#) | [Frequency-Domain Adaptive Filter](#) | [Variable Bandwidth FIR Filter](#)

## dsp.HampelFilter

**Package:** dsp

Filter outliers using Hampel identifier

### Description

The `dsp.HampelFilter` System object detects and removes the outliers of the input signal by using the Hampel identifier. The Hampel identifier is a variation of the three-sigma rule of statistics that is robust against outliers. For each sample of the input signal, the object computes the median of a window composed of the current sample and  $\frac{Len-1}{2}$  adjacent samples on each side of current sample. *Len* is the window length you specify through the `WindowLength` property. The object also estimates the standard deviation of each sample about its window median by using the median absolute deviation. If a sample differs from the median by more than the threshold multiplied by the standard deviation, the filter replaces the sample with the median. For more information, see “Algorithms” on page 4-621.

To filter the input signal using a Hampel identifier:

- 1 Create the `dsp.HampelFilter` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

### Creation

#### Syntax

```
hampFilt = dsp.HampelFilter
hampFilt = dsp.HampelFilter(Len)
hampFilt = dsp.HampelFilter(Len, Lim)
hampFilt = dsp.HampelFilter(Name,Value)
```

#### Description

`hampFilt = dsp.HampelFilter` returns a Hampel filter object, `hampFilt`, using the default properties.

`hampFilt = dsp.HampelFilter(Len)` sets the `WindowLength` property to `Len`.

`hampFilt = dsp.HampelFilter(Len, Lim)` sets the `WindowLength` property to `Len` and the `Threshold` property to `Lim`.

Example: `hampFilt = dsp.HampelFilter(11,2);`

`hampFilt = dsp.HampelFilter(Name,Value)` specifies properties using `Name,Value` pairs. Unspecified properties have default values.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see *System Design in MATLAB Using System Objects*.

If a property is listed as *tunable*, then you can change its value even when the object is locked.

### WindowLength — Length of the sliding window

11 (default) | positive odd scalar integer

Length of the sliding window, specified as a positive odd scalar integer. The window of finite length slides over the data, and the object computes the median and median absolute deviation of the data in the window.

Data Types: `single` | `double`

### Threshold — Threshold for outlier detection

3 (default) | positive real scalar

Threshold for outlier detection, specified as a positive real scalar. For information on how this property is used to detect the outlier, see “Algorithms” on page 4-621.

**Tunable:** Yes

Data Types: `single` | `double`

## Usage

### Syntax

```
y = hampFilt(x)
[y,isOutlier] = hampFilt(x)
```

### Description

`y = hampFilt(x)` detects and removes the outliers of each channel of the input signal, `x`, independently over time using the Hampel filter.

`[y,isOutlier] = hampFilt(x)` returns a logical array, `isOutlier`, in which each true element indicates that the corresponding element in the input is an outlier. `isOutlier` is the same size as the input and output vectors.

### Input Arguments

#### **x** — Data input

vector | matrix

Data input, specified as a vector or a matrix. The object accepts multichannel inputs, that is,  $m$ -by- $n$  size inputs, where  $m \geq 1$ , and  $n > 1$ .  $m$  is the number of samples in each frame (channel), and  $n$  is the

number of channels. The object also accepts variable-size inputs. After the object is locked, you can change the size of each input channel, but you cannot change the number of channels.

Data Types: `single` | `double`

### Output Arguments

#### **y** — Filtered data

`vector` | `matrix`

Filtered data, returned as a vector or a matrix.

Data Types: `single` | `double`

#### **isOutlier** — Indicate outlier

`true` | `false`

Logical array whose elements indicate if the corresponding element in the input array is an outlier. If an element in `isOutlier` is `true`, the corresponding element in the input array is an outlier.

Data Types: `logical`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Common to All System Objects

|                      |  |
|----------------------|--|
| <code>step</code>    | Run System object algorithm  |
| <code>release</code> | Release resources and allow changes to System object property values and input characteristics |
| <code>reset</code>   | Reset internal states of System object   |

## Examples

### Remove High-Frequency Noise Using Hampel Filter

Filter high-frequency noise from a noisy sine wave signal using a Hampel filter. Compare the performance of the Hampel filter with a median filter.

#### Initialization

Set up a `dsp.HampelFilter` and a `dsp.MedianFilter` object. These objects use the sliding window method with a window length of 7. Create a time scope for viewing the output.

```
Fs = 1000;  
hampFilt = dsp.HampelFilter(7);  
medFilt = dsp.MedianFilter(7);  
scope = timescope('SampleRate',Fs, ...  
    'TimeSpanOvverrunAction','Scroll', ...  
    'TimeSpanSource','Property',...  
    'TimeSpan',1,'ShowGrid',true, ...
```

```

        'LayoutDimensions',[3 1], ...
        'NumInputPorts',3);
scope.ActiveDisplay = 1;
scope.Title = 'Signal + Noise';
scope.YLimits = [-1 3];
scope.ActiveDisplay = 2;
scope.Title = 'Hampel Filter Output (Window Length = 7)';
scope.YLimits = [-1 1];
scope.ActiveDisplay = 3;
scope.Title = 'Median Filter Output (Window Length = 7)';
scope.YLimits = [-1 1];

```

### Filter the Noisy Sine Wave Using a Window of Length 7

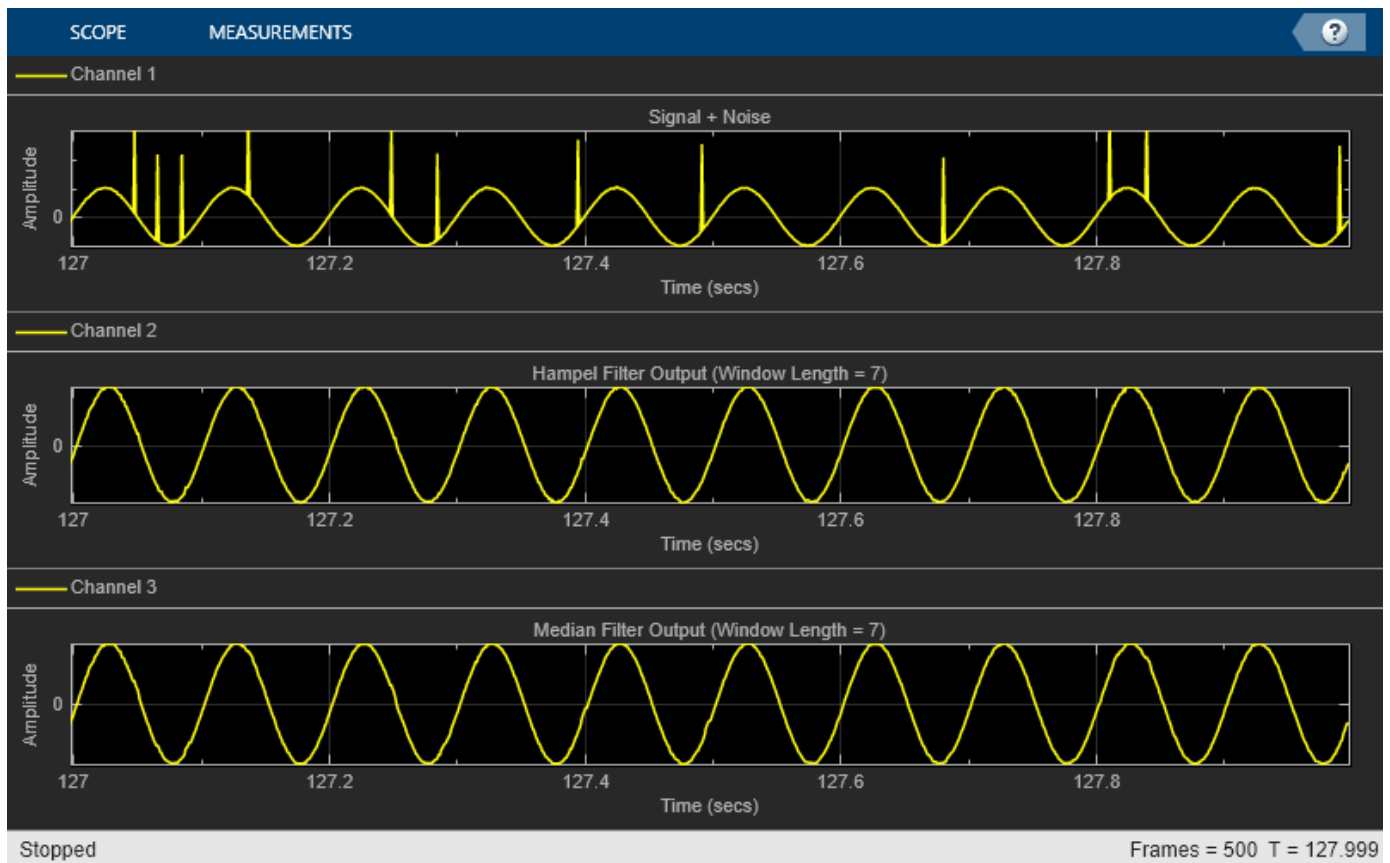
Generate a noisy sine wave signal with a frequency of 10 Hz. Apply the Hampel filter and the median filter object to the signal. View the output on the time scope.

```

FrameLength = 256;
sine = dsp.SineWave('SampleRate',Fs,'Frequency',10,...
    'SamplesPerFrame',FrameLength);

for i = 1:500
    hfn = 3 * (rand(FrameLength,1) < 0.02);
    x = sine() + 1e-2 * randn(FrameLength,1) + hfn;
    y1 = hampFilt(x);
    y2 = medFilt(x);
    scope(x,y1,y2);
end
release(scope)

```



Both filters remove the high-frequency noise. However, when you increase the window length, the Hampel filter is preferred. Unlike the median filter, the Hampel filter preserves the shape of the sine wave even with large window lengths.

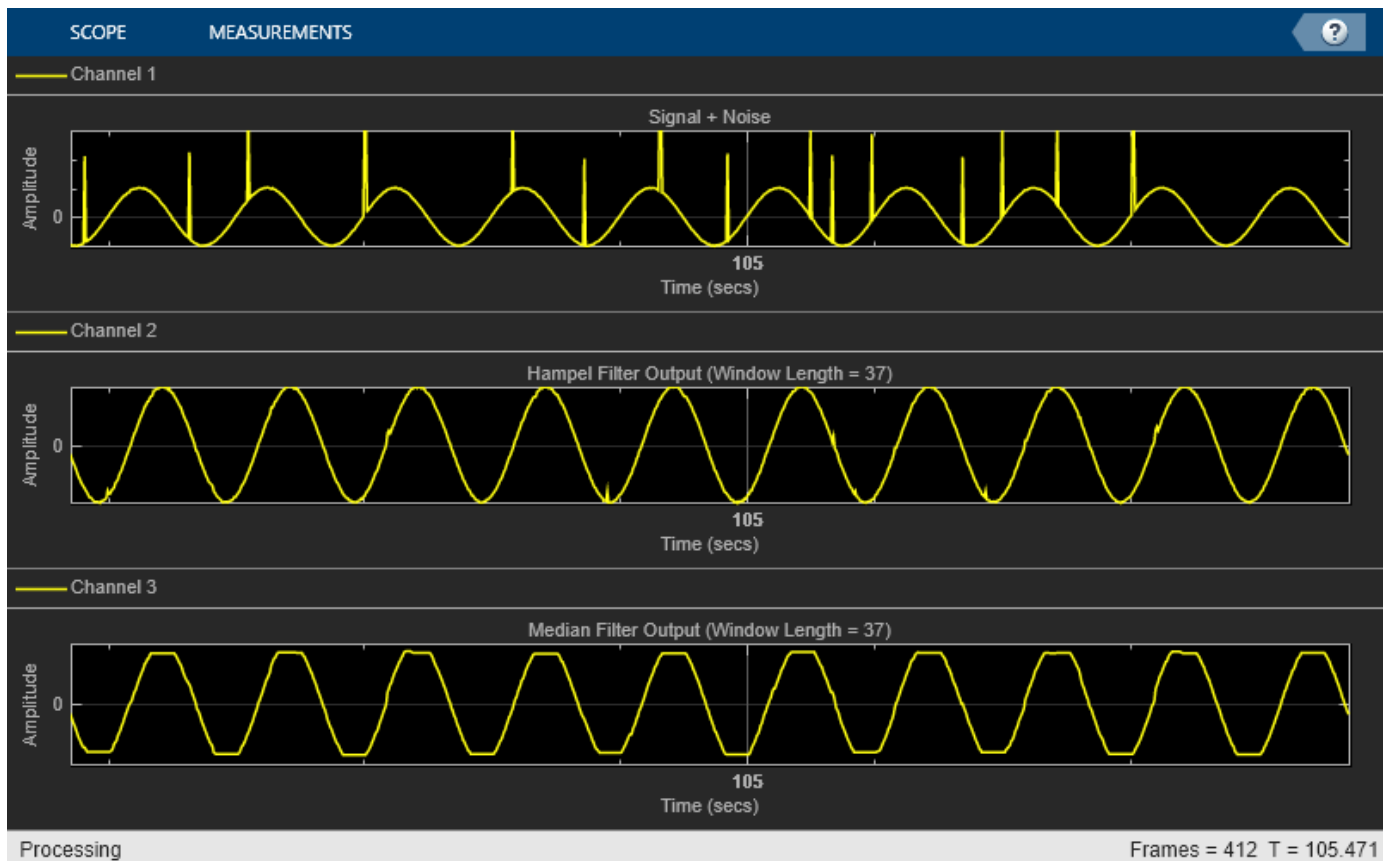
### Filter the Noisy Sine Wave Using a Window of Length 37

Increase the window length of both the filters to 37. Filter the noisy sine wave and view the filtered output on the time scope. To change the window length of the filters, you must release the filter objects at the start of the processing loop.

```
release(hampFilt);
release(medFilt);
hampFilt.WindowLength = 37;
medFilt.WindowLength = 37;
scope.ActiveDisplay = 1;
scope.Title = 'Signal + Noise';
scope.ActiveDisplay = 2;
scope.Title = 'Hampel Filter Output (Window Length = 37)';
scope.ActiveDisplay = 3;
scope.Title = 'Median Filter Output (Window Length = 37)';
for i = 1:500
    hfn = 3 * (rand(FrameLength,1) < 0.02);
    x = sine() + 1e-2 * randn(FrameLength,1) + hfn;
    y1 = hampFilt(x);
    y2 = medFilt(x);
    scope(x,y1,y2);
end
```



```
end
release(scope)
```



The median filter flattens the crests and troughs of the sine wave due to the median operation over a large window of data. The Hampel filter preserves the shape of the signal, in addition to removing the outliers.

### Remove High-Frequency Noise from Gyroscope Data Using Hampel Filter

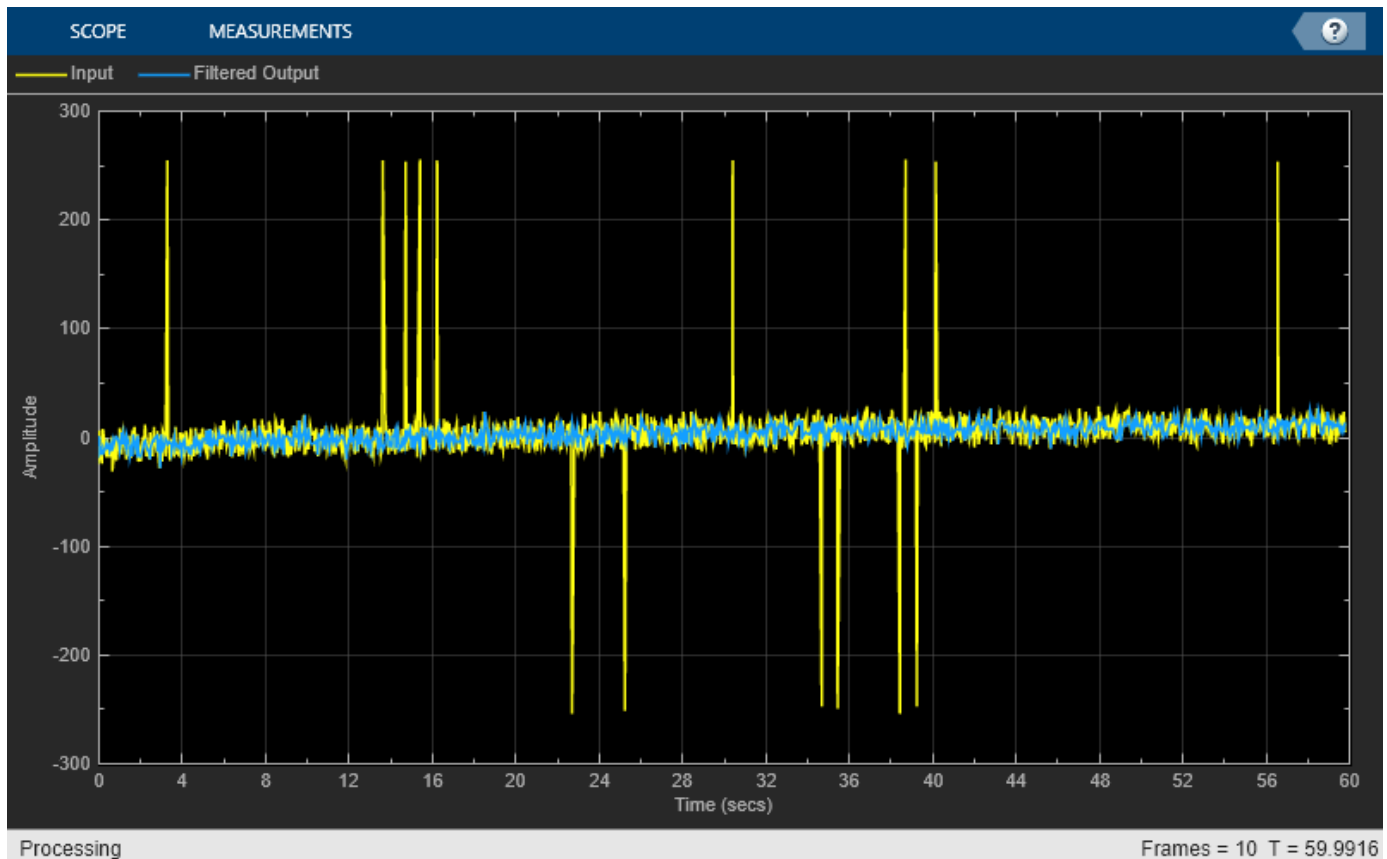
Remove the high-frequency outliers from a streaming signal using the `dsp.HampelFilter` System object™.

Use the `dsp.MatFileReader` System object to read the gyroscope MAT file. The file contains three columns of data, with each column containing 7140 samples. The three columns represent the x-axis, y-axis, and z-axis data from the gyroscope motion sensor. Choose a frame size of 714 samples so that each column of the data contains 10 frames. The `dsp.HampelFilter` System object uses a window length of 11. Create a `timescope` object to view the filtered output.

```
reader = dsp.MatFileReader('SamplesPerFrame',714,'Filename','LSM9DSHampelgyroData73.mat', ...
    'VariableName','data');
hampFilt = dsp.HampelFilter(11);
scope = timescope('NumInputPorts',1,'SampleRate',119,'YLimits',[-300 300], ...
    'ChannelNames',{'Input','Filtered Output'},...
    'TimeSpanSource','Property','TimeSpan',60,'ShowLegend',true);
```

Filter the gyroscope data using the `dsp.HampelFilter` System object. View the filtered z-axis data in the Time Scope.

```
for i = 1:10
    gyroData = reader();
    filteredData = hampFilt(gyroData);
    scope([gyroData(:,3), filteredData(:,3)]);
end
```



The Hampel filter removes all the outliers and preserves the shape of the signal.

## More About

### Hampel Identifier

The Hampel identifier is a variation of the three-sigma rule of statistics that is robust against outliers.

Given a sequence  $x_1, x_2, x_3, \dots, x_n$  and a sliding window of length  $k$ , define point-to-point median and standard-deviation estimates using:

- Local median —  $m_i = \text{median}(x_{i-k}, x_{i-k+1}, x_{i-k+2}, \dots, x_i, \dots, x_{i+k-2}, x_{i+k-1}, x_{i+k})$
- Standard deviation —  $\sigma_i = \kappa \text{median}(|x_{i-k} - m_i|, \dots, |x_{i+k} - m_i|)$ , where  $\kappa = \frac{1}{\sqrt{2} \text{erfc}^{-1}(1/2)} \approx 1.4826$

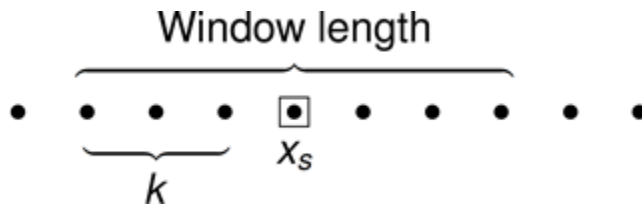
The quantity  $\sigma_i / \kappa$  is known as the median absolute deviation (MAD).

If a sample  $x_i$  is such that

$$|x_i - m_i| > n_\sigma \sigma_i$$

for a given threshold  $n_\sigma$ , then the Hampel identifier declares  $x_i$  an outlier and replaces it with  $m_i$ . If  $n_\sigma$  is 0, then the Hampel filter behaves as a regular median filter.

## Algorithms



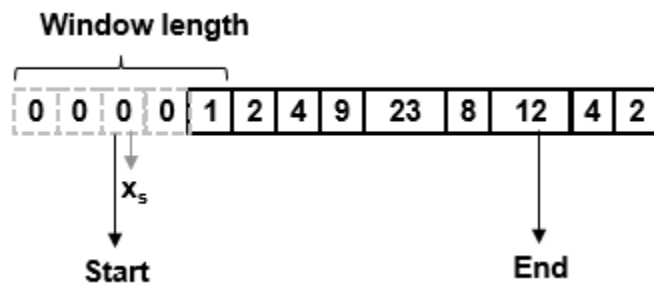
For a given sample of data,  $x_s$ , the algorithm:

- Centers the window of odd length at the current sample.
- Computes the local median,  $m_i$ , and standard deviation,  $\sigma_i$ , over the current window of data.
- Compares the current sample with  $n_\sigma \times \sigma_i$ , where  $n_\sigma$  is the threshold value. If  $|x_s - m_i| > n_\sigma \times \sigma_i$ , the filter identifies the current sample,  $x_s$ , as an outlier and replaces it with the median value,  $m_i$ .

Consider a frame of data that is passed into the Hampel filter.

|   |   |   |   |    |   |    |   |   |
|---|---|---|---|----|---|----|---|---|
| 1 | 2 | 4 | 9 | 23 | 8 | 12 | 4 | 2 |
|---|---|---|---|----|---|----|---|---|

In this example, the Hampel filter slides a window of length 5 ( $Len$ ) over the data. The filter has a threshold value of 2 ( $n_\sigma$ ). To have a complete window at the beginning of the frame, the filter algorithm prepends the frame with  $Len - 1$  zeros. To compute the first sample of the output, the window centers on the  $\left[\frac{Len - 1}{2} + 1\right]^{\text{th}}$  sample in the appended frame, the third zero in this case. The filter computes the median, median absolute deviation, and the standard deviation over the data in the local window.



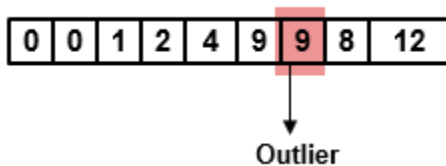
- Current sample:  $x_s = 0$ .
- Window of data:  $win = [0\ 0\ 0\ 0\ 1]$ .

- Local median:  $m_i = \text{median}([0\ 0\ 0\ 0\ 1]) = 0$ .
- Median absolute deviation:  $mad_i = \text{median}(|x_{i-k} - m_i|, \dots, |x_{i+k} - m_i|)$ . For this window of data,  $mad = \text{median}(|0 - 0|, \dots, |1 - 0|) = 0$ .
- Standard deviation:  $\sigma_i = \kappa \times mad_i = 0$ , where  $\kappa = \frac{1}{\sqrt{2}\text{erfc}^{-1}(1/2)} \approx 1.4826$ .
- The current sample,  $x_s = 0$ , does not obey the relation for outlier detection.  
 $[|x_s - m_i| = 0] > [(n_\sigma \times \sigma_i) = 0]$

Therefore, the Hampel filter outputs the current input sample,  $x_s = 0$ .

Repeat this procedure for every succeeding sample until the algorithm centers the window on the  $\left[End - \frac{Len - 1}{2}\right]^{\text{th}}$  sample, marked as End. Because the window centered on the last  $\frac{Len - 1}{2}$  samples cannot be full, these samples are processed with the next frame of input data.

Here is the first output frame the Hampel filter generates:



The seventh sample of the appended input frame, 23, is an outlier. The Hampel filter replaces this sample with the median over the local window [4 9 23 8 12].

## Version History

Introduced in R2017a

## References

- [1] Bodenham, Dean. "Adaptive Filtering and Change Detection for Streaming Data." PH.D. Thesis. Imperial College, London, 2012.
- [2] Liu, Hancong, Sirish Shah, and Wei Jiang. "On-line outlier detection and data cleaning." *Computers and Chemical Engineering*. Vol. 28, March 2004, pp. 1635-1647.
- [3] Suomela, Jukka. Median Filtering Is Equivalent to Sorting, 2014.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

## See Also

### Functions

[hampel](#)

### Objects

[dsp.MedianFilter](#) | [dsp.MovingAverage](#)

### Blocks

[Hampel Filter](#) | [Median Filter](#) | [Moving Average](#)

# dsp.HighpassFilter

**Package:** dsp

FIR or IIR highpass filter

## Description

The `dsp.HighpassFilter` System object independently filters each channel of the input over time using the given design specifications. You can set the `FilterType` property of `dsp.HighpassFilter` to 'FIR' or 'IIR' to implement the object as an FIR or IIR highpass filter.

When the `FilterType` property is set to 'FIR', using this object is an alternative to using the `firceqrip` and `firgr` functions with `dsp.FIRFilter`. The `dsp.HighpassFilter` object condenses the two-step process into one. You can use `measure` to verify that the design meets the prescribed specifications.

To filter each channel of your input:

- 1 Create the `dsp.HighpassFilter` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
HPF = dsp.HighpassFilter  
HPF = dsp.HighpassFilter(Name=Value)
```

### Description

`HPF = dsp.HighpassFilter` returns a minimum order FIR highpass filter, `HPF`, with the default filter settings. Calling the object with the default property settings filters the input data with a stopband frequency of 8 kHz, a passband frequency of 12 kHz, a stopband attenuation of 80 dB, and a passband ripple of 0.1 dB.

`HPF = dsp.HighpassFilter(Name=Value)` returns a highpass filter with additional properties specified by one or more `Name-Value` pair arguments. `Name` is the property name and `Value` is the corresponding value. For example, `StopbandFrequency=8000` sets the stopband frequency specification of the filter to 8000 Hz.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

### FilterType — Filter type

'FIR' (default) | 'IIR'

Filter type, specified as one of these options:

- 'FIR' — The object designs an FIR highpass filter.
- 'IIR' — The object designs an IIR highpass (biquad) filter.

### DesignForMinimumOrder — Flag to design minimum order filter

true (default) | false

Flag to design minimum order filter, specified as:

- true -- The object designs the minimum order filter that meets the filter design specifications.
- false -- The object designs the filter with the order that you specify in the FilterOrder property.

### FilterOrder — Order of the FIR or IIR filter

50 (default) | positive integer

Order of the FIR or IIR filter, specified as a positive integer.

#### Dependencies

To enable this property, set DesignForMinimumOrder to false.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### StopbandFrequency — Filter stopband edge frequency

8000 (default) | real positive scalar

Filter stopband edge frequency, specified as a real positive scalar in Hz or in normalized frequency units.

If you set the NormalizedFrequency property to:

- false -- The value of the stopband edge frequency is in Hz. The value must be less than the passband edge frequency and half the SampleRate property value.
- true -- The value of the stopband edge frequency is in normalized frequency units. The value must be a positive scalar less than the passband edge frequency and less than 1.0.

#### Dependencies

To enable this property, set the DesignForMinimumOrder property to true.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### PassbandFrequency — Filter passband edge frequency

12000 (default) | real positive scalar

Filter passband edge frequency, specified as a real positive scalar in Hz or in normalized frequency units.

If you set the NormalizedFrequency property to:

- `false` -- The value of the passband edge frequency is in Hz. The value must be less than half the `SampleRate` property value and greater than the `StopbandFrequency` property value.
- `true` -- The value of the passband edge frequency is in normalized frequency units. The value must be a positive scalar less than `1.0` and greater than the `StopbandFrequency` property value.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **StopbandAttenuation — Minimum attenuation in the stopband**

`80` (default) | real positive scalar

Minimum attenuation in the stopband, specified as a real positive scalar in dB.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **PassbandRipple — Maximum ripple of filter response in the passband**

`0.1` (default) | real positive scalar

Maximum ripple of filter response in the passband, specified as a real positive scalar in dB.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **NormalizedFrequency — Flag to set frequencies in normalized units**

`false` (default) | `true`

Flag to set frequencies in normalized units, specified as one of these values:

- `true` -- The passband edge and stopband edge frequencies must be in the normalized frequency units and less than `1.0`.
- `false` -- The passband edge and stopband edge frequencies are in Hz. You can specify the input sample rate through the `SampleRate` property.

Data Types: `logical`

### **SampleRate — Input sample rate**

`44100` (default) | positive real scalar

Input sample rate in Hz, specified as a positive real scalar.

#### **Dependency**

To enable this property, set `NormalizedFrequency` to `false`.

Data Types: `single` | `double`

#### **Fixed-Point Properties**

### **RoundingMethod — Rounding method for output fixed-point operations**

'Floor' (default) | 'Ceiling' | 'Convergent' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'

Rounding method for output fixed-point operations, specified as a character vector. For more information on the rounding modes, see "Precision and Range".

### **CoefficientsDataType — Word and fraction lengths of coefficients**

`numericType([],16)` (default) | `numericType` object



Word and fraction lengths of coefficients, specified as a `numericType` object. The default, `numericType(1,16)` corresponds to a signed numeric type object with 16-bit coefficients and a fraction length determined based on the coefficient values, to give the best possible precision.

This property is not tunable.

Word length of the output is same as the word length of the input. Fraction length of the output is computed such that the entire dynamic range of the output can be represented without overflow. For details on how the fraction length of the output is computed, see “Fixed-Point Precision Rules for Avoiding Overflow in FIR Filters”.

## Usage

### Syntax

`y = HPF(x)`

### Description

`y = HPF(x)` highpass filters the input signal, `x`. `y` is a highpass-filtered version of `x`.

### Input Arguments

#### **x** — Noisy data input

vector | matrix

Noisy data input, specified as a vector or a matrix. If the input signal is a matrix, each column of the matrix is treated as an independent channel. The number of rows in the input signal denote the channel length. This object accepts variable-size inputs. After the object is locked, you can change the size of each input channel, but you cannot change the number of channels.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

### Output Arguments

#### **y** — Filtered output

vector | matrix

Filtered output, returned as a vector or a matrix. The output has the same size, data type, and complexity characteristics as the input.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

`release(obj)`

### Specific to `dsp.HighpassFilter`

|                          |   |
|--------------------------|---|
| <code>freqz</code>       | Frequency response of discrete-time filter System object                      |
| <code>fvtool</code>      | Visualize frequency response of DSP filters                                   |
| <code>impz</code>        | Impulse response of discrete-time filter System object                        |
| <code>info</code>        | Information about filter System object  |
| <code>coeffs</code>      | Returns the filter System object coefficients in a structure                  |
| <code>cost</code>        | Estimate cost of implementing filter System object                            |
| <code>grpdelay</code>    | Group delay response of discrete-time filter System object                    |
| <code>generatehdl</code> | Generate HDL code for quantized DSP filter (requires Filter Design HDL Coder) |
| <code>measure</code>     | Measure frequency response characteristics of filter System object            |

### Common to All System Objects

|                      |  |
|----------------------|--|
| <code>step</code>    | Run System object algorithm  |
| <code>release</code> | Release resources and allow changes to System object property values and input characteristics |
| <code>reset</code>   | Reset internal states of System object   |

## Examples

### Impulse and Frequency Response of FIR and IIR Highpass Filters

Create a minimum order FIR highpass filter for data sampled at 44.1 kHz. Specify a passband frequency of 12 kHz, a stopband frequency of 8 kHz, a passband ripple of 0.1 dB, and a stopband attenuation of 80 dB.

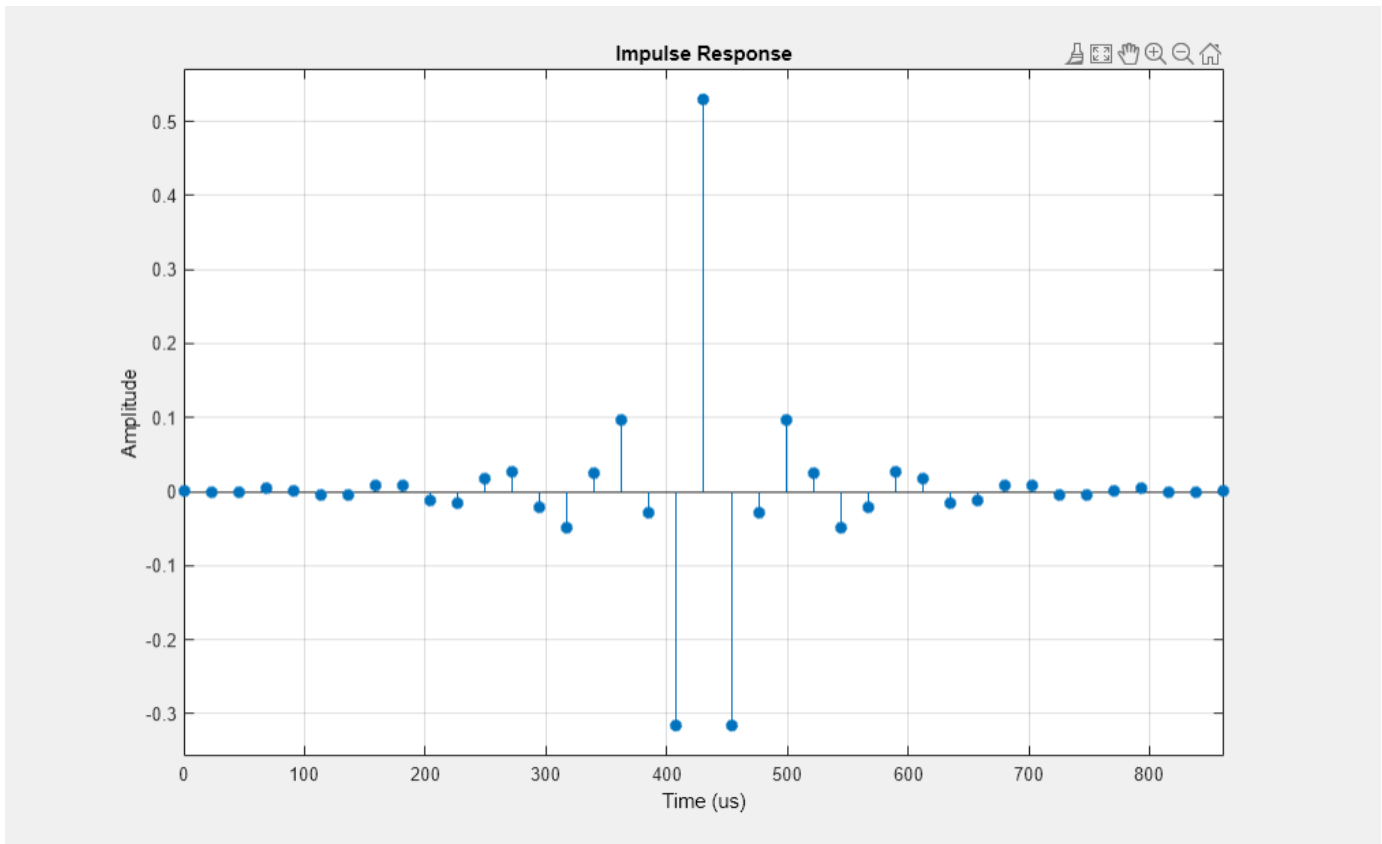
```
Fs = 44.1e3;
filtertype = 'FIR';
Fpass = 12e3;
Fstop = 8e3;
Rp = 0.1;
Astop = 80;
FIRHPF = dsp.HighpassFilter(SampleRate=Fs,...
                           FilterType=filtertype,...
                           PassbandFrequency=Fpass,...
                           StopbandFrequency=Fstop,...
                           PassbandRipple=Rp,...
                           StopbandAttenuation=Astop);
```

Design a minimum order IIR highpass filter with the same properties as the FIR highpass filter. Use `clone` to create a system object with the same properties as the FIR Highpass filter. Change the `FilterType` property of the cloned filter to IIR.

```
IIRHPF = clone(FIRHPF);
IIRHPF.FilterType = 'IIR';
```

Plot the impulse response of the FIR highpass filter. The zeroth order coefficient is delayed by 19 samples, which is equal to the group delay of the filter. The FIR highpass filter is a causal FIR filter

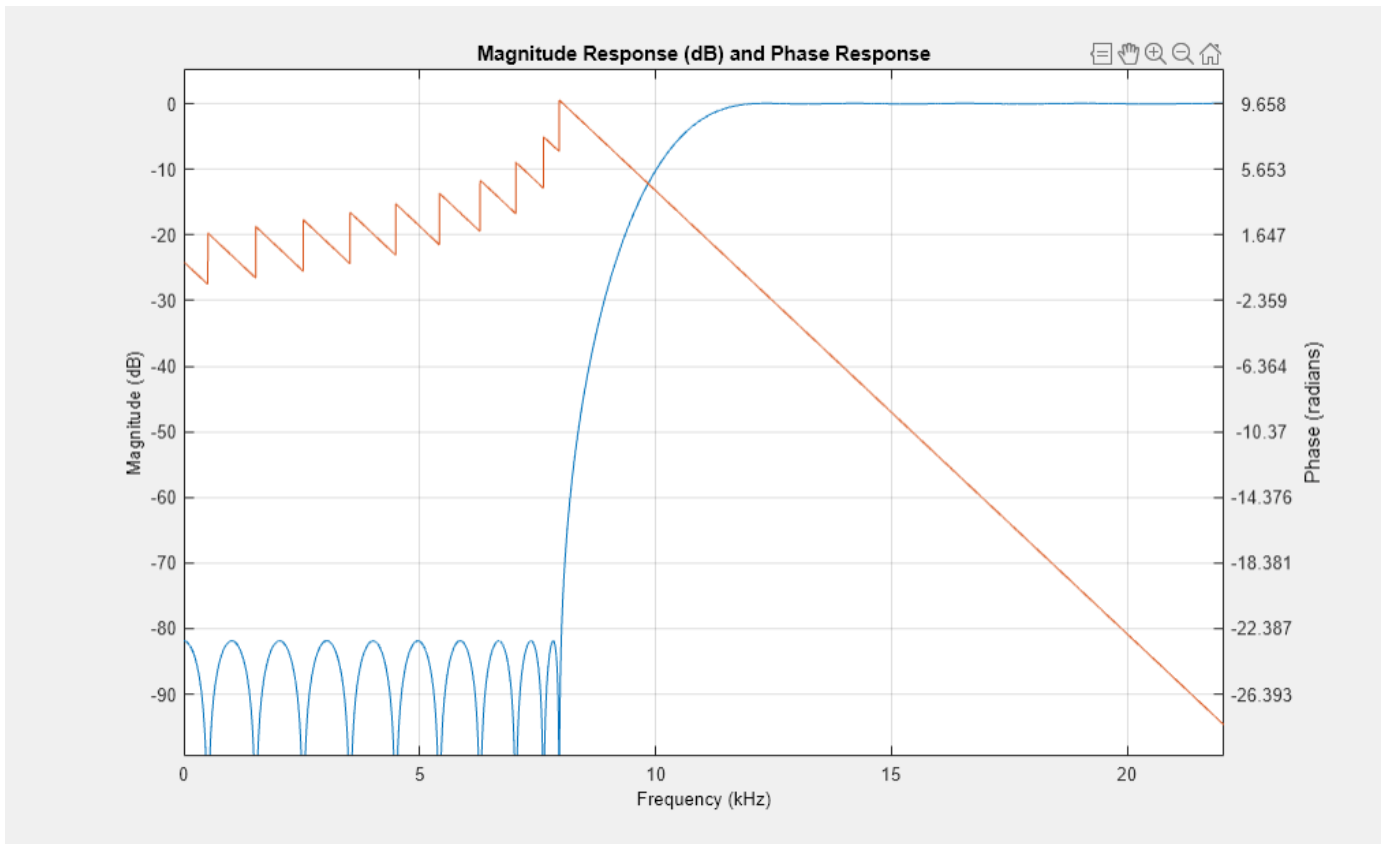
```
fvtool(FIRHPF,Analysis='impulse')
```



Plot the impulse response of the IIR highpass filter.

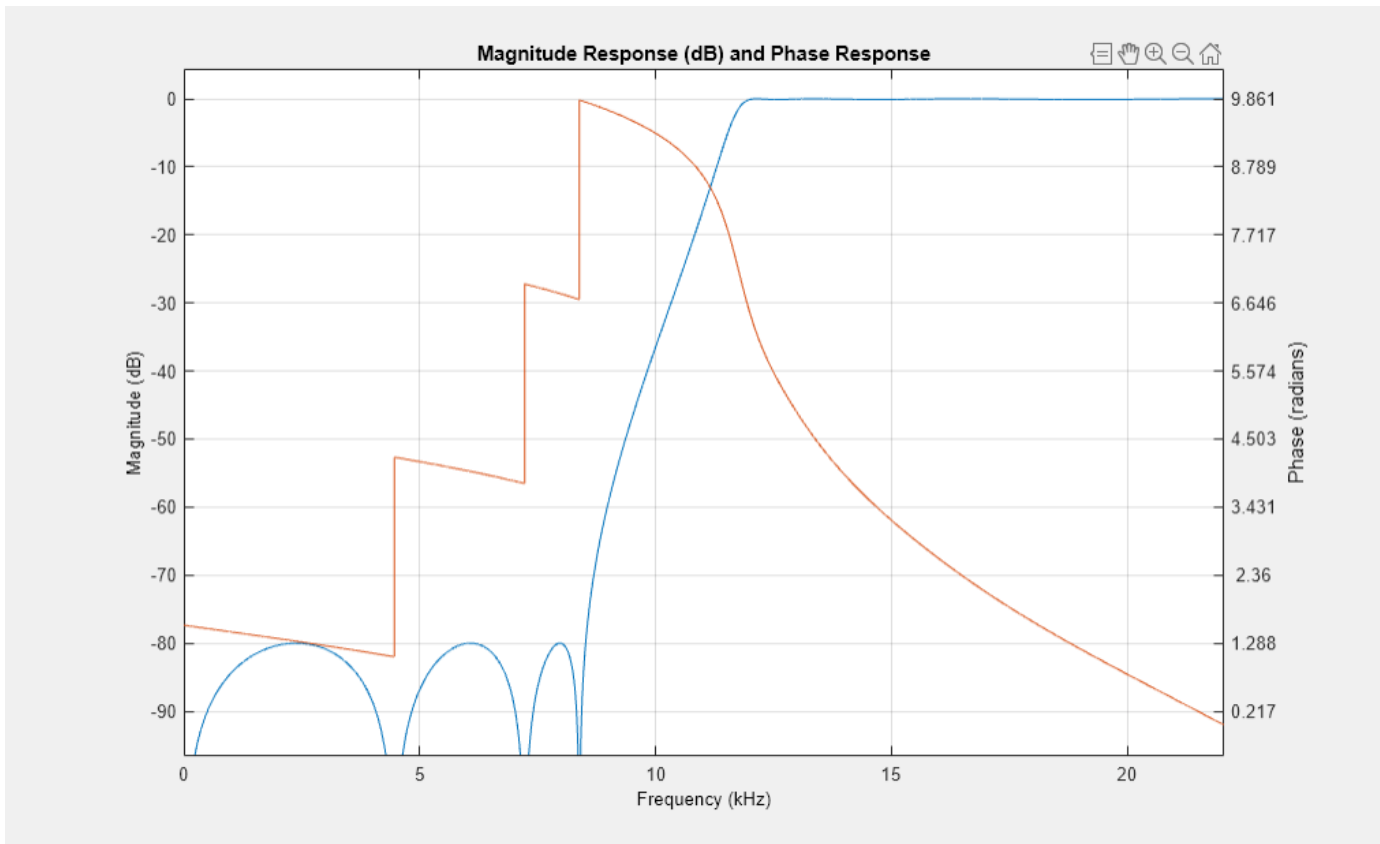
```
fvtool(IIRHPF,Analysis='impulse')
```





Plot the magnitude and phase response of the IIR highpass filter.

```
fvtool(IIRHPF,Analysis='freq')
```



Calculate the cost of implementing the FIR highpass filter.

```
cost(FIRHPF)
```

```
ans = struct with fields:
    NumCoefficients: 39
    NumStates: 38
    MultiplicationsPerInputSample: 39
    AdditionsPerInputSample: 38
```

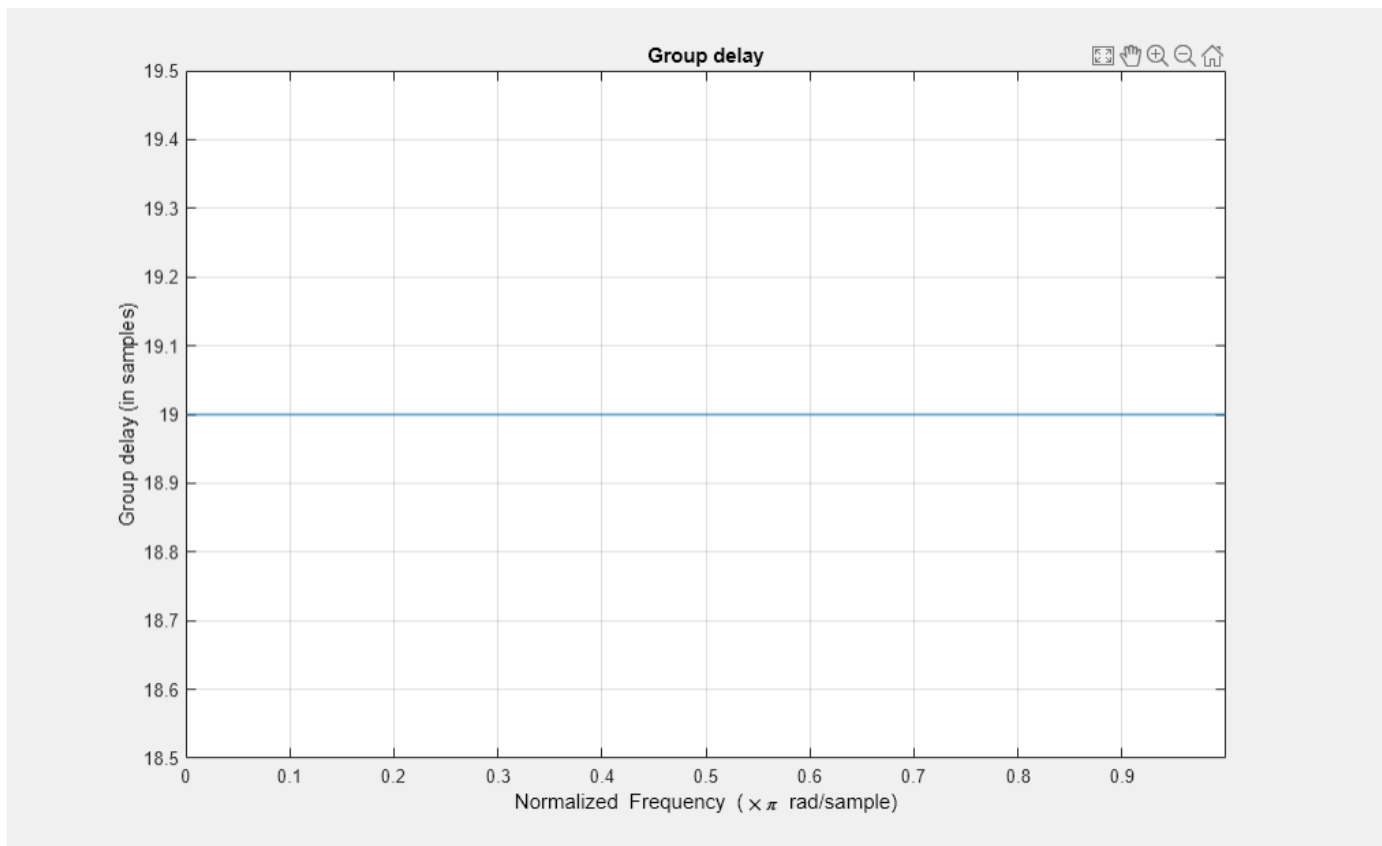
Calculate the cost of implementing the IIR highpass filter. The IIR filter is more efficient to implement than its FIR counterpart.

```
cost(IIRHPF)
```

```
ans = struct with fields:
    NumCoefficients: 18
    NumStates: 14
    MultiplicationsPerInputSample: 18
    AdditionsPerInputSample: 14
```

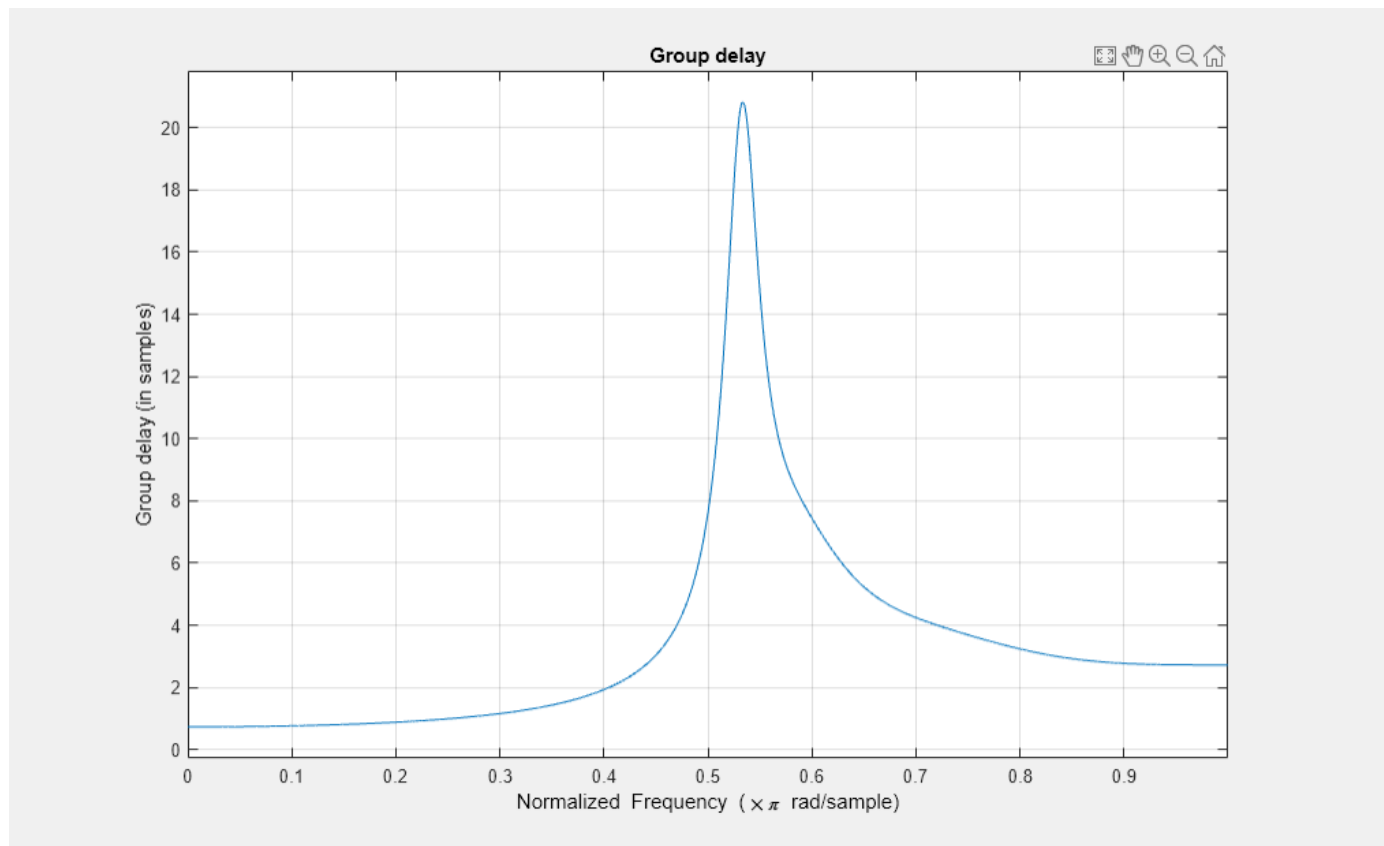
Calculate the group delay of the FIR highpass filter.

```
grpdelay(FIRHPF)
```



Calculate the group delay of the IIR highpass filter. The FIR filter has a constant group delay (linear phase) while its IIR counterpart does not.

```
grpdelay(IIRHPF)
```



### Filter White Gaussian Noise Signal With FIR Highpass Filter

Create a highpass filter using the `dsp.HighpassFilter` object. Setting the `NormalizedFrequency` property to `true` designs the filter with frequency specifications in normalized frequency units.

```
LPF = dsp.HighpassFilter(NormalizedFrequency=true)
```

```
LPF =  
dsp.HighpassFilter with properties:
```

```
    FilterType: 'FIR'  
DesignForMinimumOrder: true  
    StopbandFrequency: 0.3628  
    PassbandFrequency: 0.5442  
    StopbandAttenuation: 80  
    PassbandRipple: 0.1000  
    NormalizedFrequency: true
```

```
Show all properties
```

Create a `spectrumAnalyzer` object to visualize the input and output signal spectra. With a sample rate of  $44.1 \times 10^3$  Hz, the stopband frequency and the passband frequency of the filter translate to 8000 Hz and 12000 Hz, respectively.



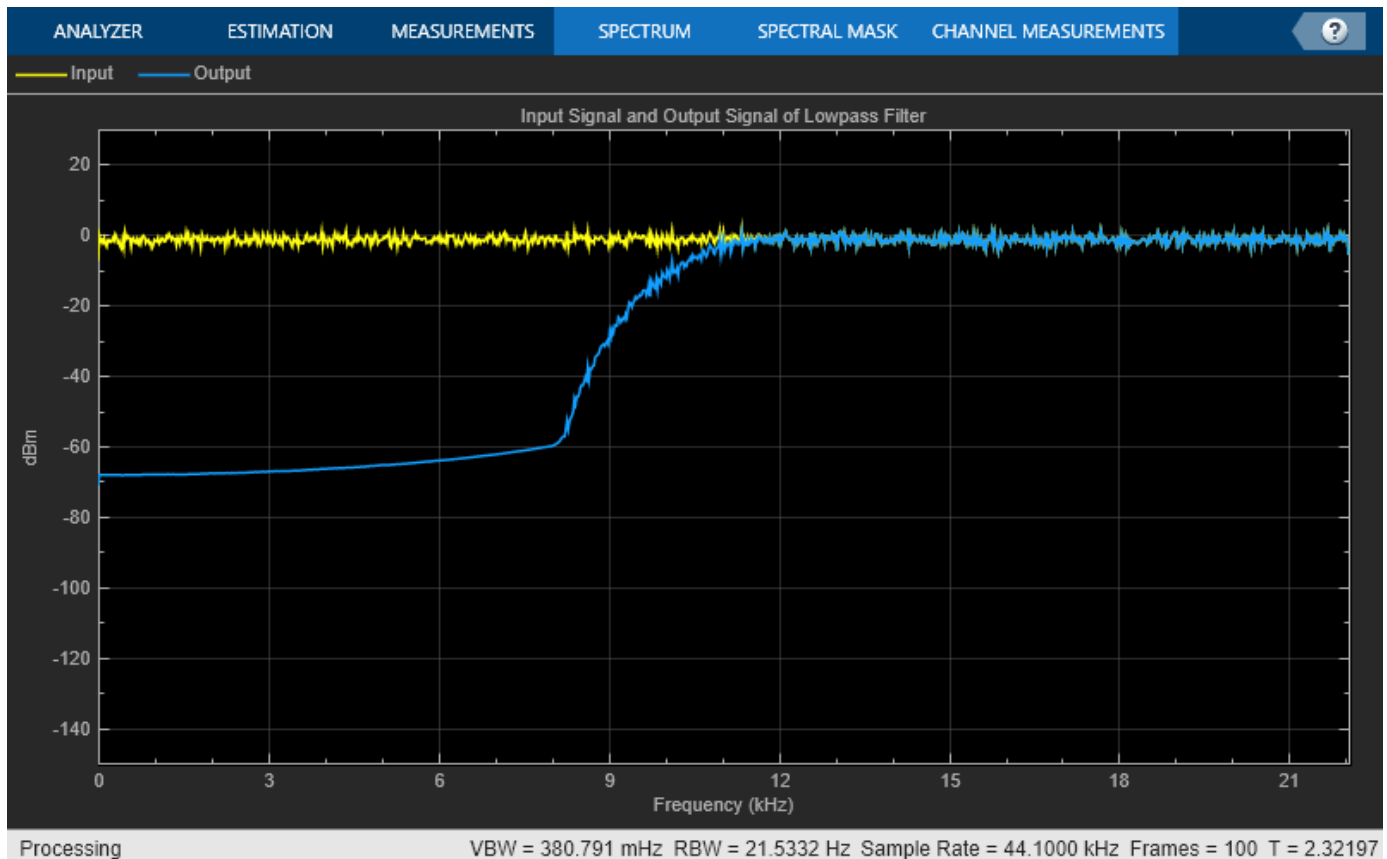
```
SA = spectrumAnalyzer(SampleRate=44.1e3,...
    PlotAsTwoSidedSpectrum=false,ShowLegend=true,...
    YLimits=[-150 30],...
    Title='Input Signal and Output Signal of Lowpass Filter');
SA.ChannelNames = {'Input','Output'};
```

Run the highpass filter algorithm to filter the white Gaussian noisy input signal. View the input and output signals using the spectrum analyzer.

```
for k = 1:100
    Input = randn(1024,1);

    Output = LPF(Input);

    SA([Input,Output]);
end
```



### Filter White Gaussian Noise with an IIR Highpass Filter

Set up the IIR highpass filter. The sampling rate of the white Gaussian noise is 44,100 Hz. The passband frequency of the filter is 12 kHz, the stopband frequency is 8 kHz, the passband ripple is 0.1 dB, and the stopband attenuation is 80 dB.

```
Fs = 44.1e3;
filtertype = 'IIR';
```

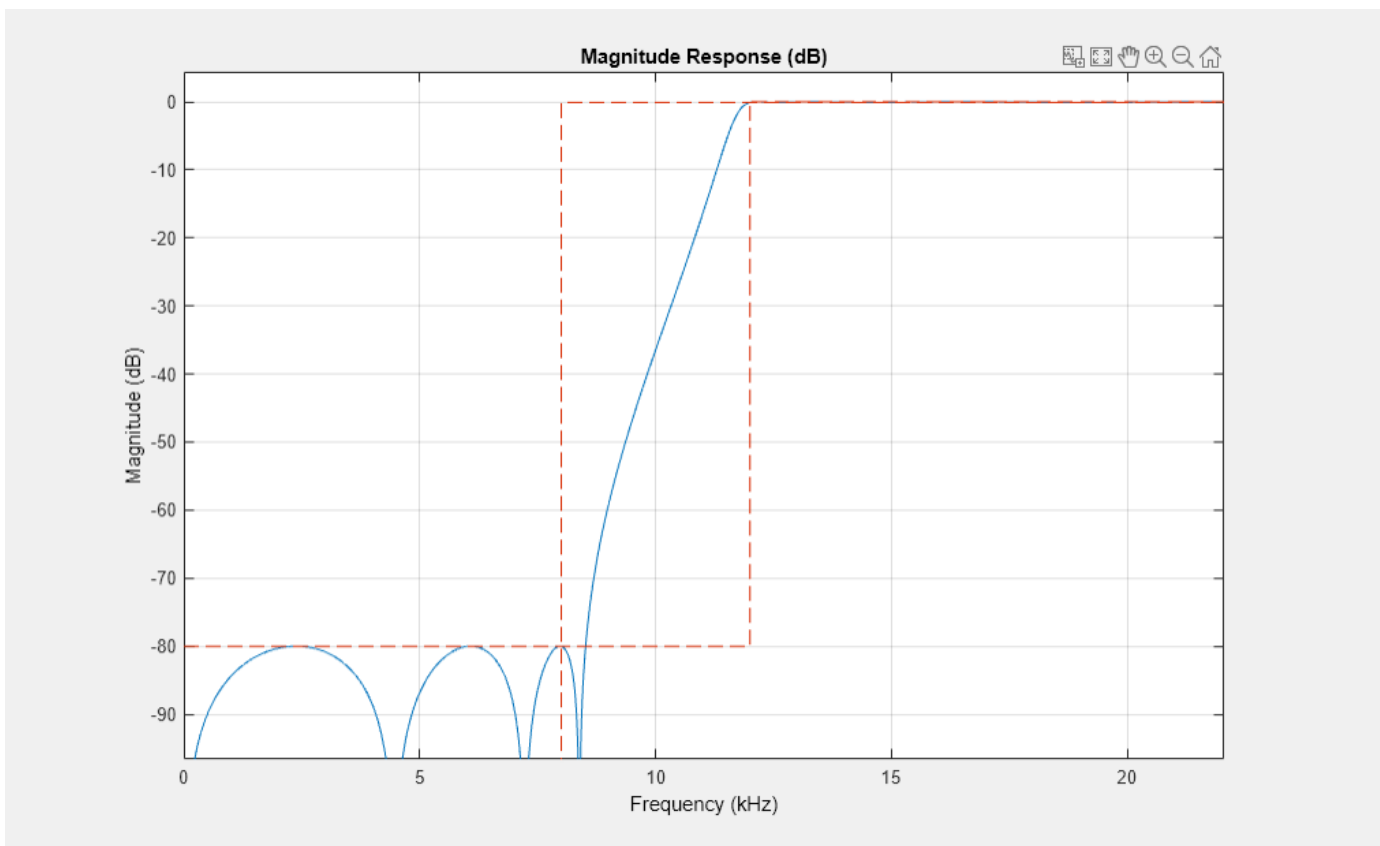
```

Fpass = 12e3;
Fstop = 8e3;
Rp = 0.1;
Astop = 80;
hpf = dsp.HighpassFilter(SampleRate=Fs,...
    FilterType=filtertype,...
    PassbandFrequency=Fpass,...
    StopbandFrequency=Fstop,...
    PassbandRipple=Rp,...
    StopbandAttenuation=Astop);

```

View the magnitude response of the highpass filter.

```
fvtool(hpf)
```



Create a spectrum analyzer object.

```

sa = spectrumAnalyzer(SampleRate=44.1e3,...
    PlotAsTwoSidedSpectrum=false,ShowLegend=true,...
    YLimits=[-150 30],...
    Title='Input Signal and Output Signal of IIR Highpass Filter');
sa.ChannelNames = {'Input','Output'};

```

Filter the white Gaussian noisy input signal. View the input and output signals using the spectrum analyzer.

```

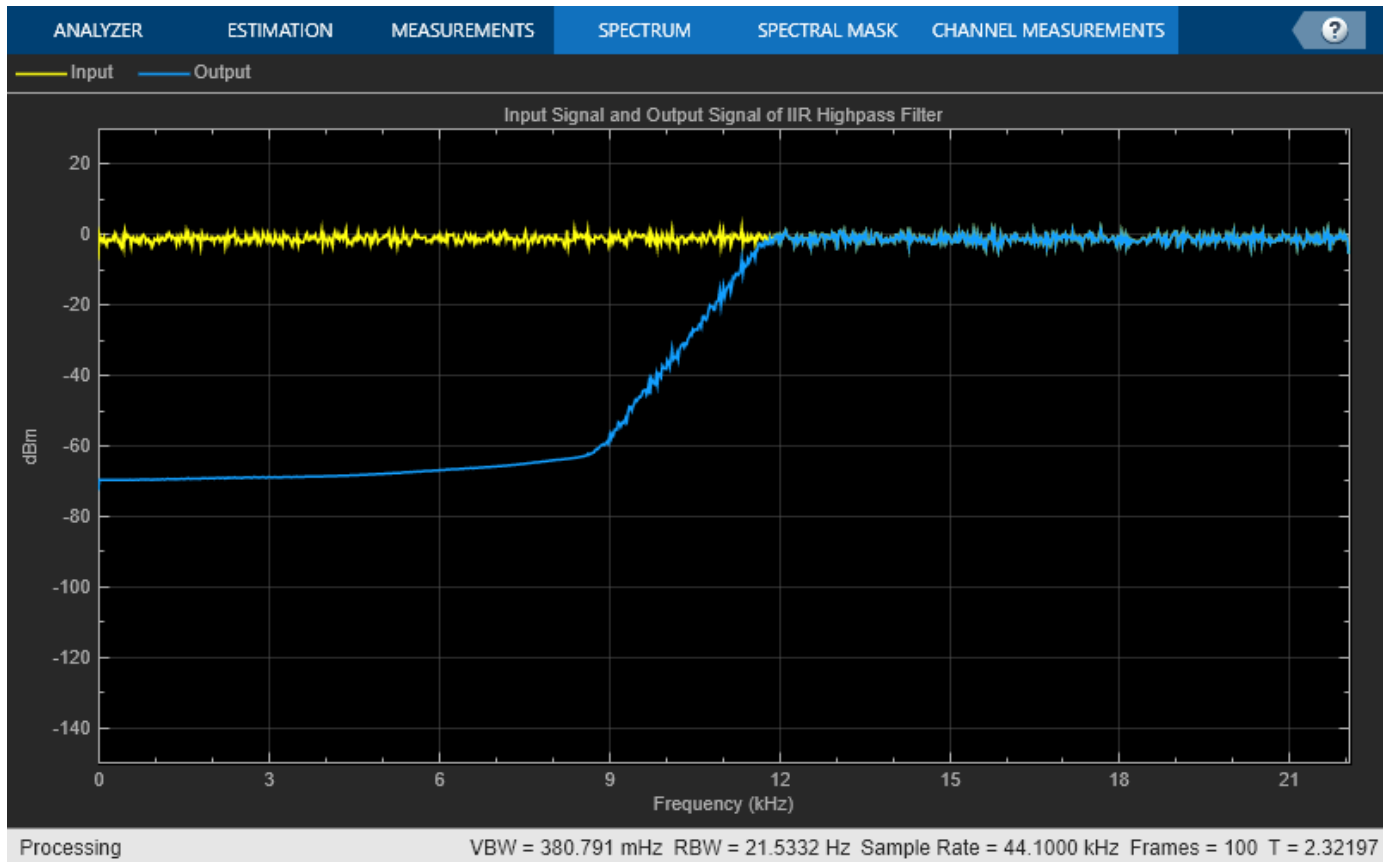
for k = 1:100
    Input = randn(1024,1);

```

```

Output = hpf(Input);
sa([Input,Output]);
end

```



### Measure Frequency Response Characteristics of Highpass Filter

Measure the frequency response characteristics of a highpass filter. Create a `dsp.HighpassFilter` System object with default properties. Measure the frequency response characteristics of the filter.

```
HPF = dsp.HighpassFilter
```

```
HPF =
dsp.HighpassFilter with properties:
```

```

    FilterType: 'FIR'
DesignForMinimumOrder: true
  StopbandFrequency: 8000
  PassbandFrequency: 12000
  StopbandAttenuation: 80
    PassbandRipple: 0.1000
  NormalizedFrequency: false
    SampleRate: 44100

```

```
Show all properties
```

```
HPFMeas = measure(HPF)
```

```
HPFMeas =  
Sample Rate      : 44.1 kHz  
Stopband Edge    : 8 kHz  
6-dB Point       : 10.418 kHz  
3-dB Point       : 10.8594 kHz  
Passband Edge    : 12 kHz  
Stopband Atten.  : 81.8558 dB  
Passband Ripple  : 0.08066 dB  
Transition Width : 4 kHz
```

## Algorithms

### FIR Highpass Filter

For the minimum order design, the algorithm uses generalized Remez FIR filter design algorithm. For the specified order design, the algorithm uses the constrained equiripple FIR filter design algorithm. The designed filter is then implemented as a linear phase Type-1 filter with a `Direct` form structure.

### IIR Highpass Filter

In the IIR configuration, the algorithm uses the elliptic design method to compute the SOS and scale values required to meet the filter design specifications. The algorithm uses the SOS and scale values to setup a `Direct` form I biquadratic IIR filter, which forms the basis of the IIR version of the highpass filter.

## Version History

### Introduced in R2015a

#### R2023a: Support for normalized frequencies

When you set the `NormalizedFrequency` property to `true`, you must specify the passband and stopband frequencies in normalized frequency units (0 to 1).

When you set the `NormalizedFrequency` property to `true` while creating the object, the passband and stopband frequency values are automatically set to normalized frequency units using the default sample rate of 44100 Hz.

```
hpFilter = dsp.HighpassFilter(NormalizedFrequency=true)
```

```
hpFilter =  
    dsp.HighpassFilter with properties:  
  
        FilterType: 'FIR'  
    DesignForMinimumOrder: true  
        StopbandFrequency: 0.3628  
        PassbandFrequency: 0.5442  
    StopbandAttenuation: 80  
        PassbandRipple: 0.1000  
    NormalizedFrequency: true
```

When you set the `NormalizedFrequency` property to `true` after you create the object, the passband and stopband frequencies must be manually set to the normalized frequency values before you run the object algorithm.

```
hpFilter = dsp.HighpassFilter

hpFilter =
  dsp.HighpassFilter with properties:
      FilterType: 'FIR'
  DesignForMinimumOrder: true
      StopbandFrequency: 8000
      PassbandFrequency: 12000
  StopbandAttenuation: 80
      PassbandRipple: 0.1000
  NormalizedFrequency: false
      SampleRate: 44100
```

To specify the normalized frequency values, set `NormalizedFrequency` to `true` and manually convert the frequency values in Hz to normalized values using the input sample rate in Hz. For example, if the input sample rate is 44100 Hz, the corresponding values in normalized units are computed using these equations.

```
hpFilter.NormalizedFrequency = true;
hpFilter.StopbandFrequency = 8000/(44100/2);
hpFilter.PassbandFrequency = 12000/(44100/2)

hpFilter =
  dsp.HighpassFilter with properties:
      FilterType: 'FIR'
  DesignForMinimumOrder: true
      StopbandFrequency: 0.3628
      PassbandFrequency: 0.5442
  StopbandAttenuation: 80
      PassbandRipple: 0.1000
  NormalizedFrequency: true
```

## References

- [1] Shpak, D.J., and A. Antoniou. "A generalized Remez method for the design of FIR digital filters." *IEEE Transactions on Circuits and Systems*. Vol. 37, Issue 2, Feb. 1990, pp. 161-174.
- [2] Selesnick, I.W., and C. S. Burrus. "Exchange algorithms that complement the Parks-McClellan algorithm for linear-phase FIR filter design." *IEEE Transactions on Circuits and Systems*. Vol. 44, Issue 2, Feb. 1997, pp. 137-143.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

This object supports code generation for ARM Cortex-M and ARM Cortex-A processors. To learn more about ARM Cortex code generation, see “Code Generation for ARM Cortex-M and ARM Cortex-A Processors”.

This object also supports SIMD code generation using Intel AVX2 technology under these conditions:

- `FilterType` is set to 'FIR'.
- Input signal has a data type of `single` or `double`.

The SIMD technology significantly improves the performance of the generated code.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

This object supports HDL code generation with the Filter Design HDL Coder product. For workflows and limitations, see “Generate HDL Code for Filter System Objects” (Filter Design HDL Coder).

### **See Also**

#### **Functions**

`freqz` | `fvtool` | `impz` | `info` | `coeffs` | `cost` | `grpdelay` | `generatehdl` | `measure` | `firceqrip` | `firgr`

#### **Objects**

`dsp.LowpassFilter`

#### **Blocks**

Highpass Filter

#### **Topics**

“Analysis Methods for Filter System Objects” on page 3-2

# dsp.IFFT

**Package:** dsp

Inverse discrete Fourier transform (IDFT)

## Description

The `dsp.IFFT` System object computes the inverse discrete Fourier transform (IDFT) of the input. The object uses one or more of the following fast Fourier transform (FFT) algorithms depending on the complexity of the input and whether the output is in linear or bit-reversed order:

- Double-signal algorithm
- Half-length algorithm
- Radix-2 decimation-in-time (DIT) algorithm
- Radix-2 decimation-in-frequency (DIF) algorithm
- An algorithm chosen from FFTW [1], [2]

To compute the IFFT of the input:

- 1 Create the `dsp.IFFT` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
ift = dsp.IFFT
ift = dsp.IFFT(Name,Value)
```

### Description

`ift = dsp.IFFT` returns an IFFT object, `ift`, that computes the IDFT of a column vector or  $N$ -D array. For column vectors or  $N$ -D arrays, the IFFT object computes the IDFT along the first dimension of the array. If the input is a row vector, the IFFT object computes a row of single-sample IDFTs and issues a warning.

`ift = dsp.IFFT(Name,Value)` returns an IFFT object, `ift`, with each property set to the specified value. Enclose each property name in single quotes. Unspecified properties have default values.

### Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

### **FFTImplementation — FFT implementation**

Auto (default) | Radix-2 | FFTW

Specify the implementation used for the FFT as Auto, Radix-2, or FFTW. When you set this property to Radix-2, the FFT length must be a power of two.

### **BitReversedInput — Enable bit-reversed order interpretation of input elements**

false (default) | true

Set this property to true if the order of Fourier transformed input elements to the IFFT object are in bit-reversed order. The default is false, which denotes linear ordering.

#### **Dependencies**

This property applies only when the FFTLengthSource property is Auto.

### **ConjugateSymmetricInput — Enable conjugate symmetric interpretation of input**

false (default) | true

Set this property to true if the input is conjugate symmetric to yield real-valued outputs. The discrete Fourier transform of a real valued sequence is conjugate symmetric, and setting this property to true optimizes the IDFT computation method. Setting this property to false for conjugate symmetric inputs may result in complex output values with nonzero imaginary parts. This occurs due to rounding errors. Setting this property to true for nonconjugate symmetric inputs results in invalid outputs.

#### **Dependencies**

This property applies only when the FFTLengthSource property is Auto.

### **Normalize — Enable dividing output by FFT length**

true (default) | false

Specify whether to divide the IFFT output by the FFT length. The default is true and each element of the output is divided by the FFT length.

### **FFTLengthSource — Source of FFT length**

Auto (default) | Property

Specify how to determine the FFT length as Auto or Property. When you set this property to Auto, the FFT length equals the number of rows of the input signal.

#### **Dependencies**

This property applies only when both the BitReversedInput and ConjugateSymmetricInput properties are false.

### **FFTLength — FFT length**

64 (default) | integer

Specify the FFT length as an integer greater than or equal to 2.



This property must be a power of two if any of these conditions apply:

- The input is a fixed-point data type.
- The FFTImplementation property is Radix-2.

#### Dependencies

This property applies when you set the BitReversedInput and ConjugateSymmetricInput properties to false, and the FFTLengthSource property to 'Property'.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

#### WrapInput — Boolean value of wrapping or truncating input

true (default) | false

Wrap input data when FFTLength is shorter than input length. If this property is set to true, modulo-length data wrapping occurs before the FFT operation, given FFTLength is shorter than the input length. If this property is set to false, truncation of the input data to the FFTLength occurs before the FFT operation.

#### Fixed-Point Properties

##### RoundingMethod — Rounding method for fixed-point operations

Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Specify the rounding method.

##### OverflowAction — Overflow action for fixed-point operations

Wrap (default) | Saturate

Specify the overflow action as Wrap or Saturate.

##### SineTableDataType — Sine table word and fraction lengths

Same word length as input (default) | Custom

Specify the sine table data type as Same word length as input or Custom.

##### CustomSineTableDataType — Sine table word and fraction lengths

numericType([],16) (default) | numericType

Specify the sine table fixed-point type as an unscaled numericType object with a Signedness of Auto.

#### Dependencies

This property applies when you set the SineTableDataType property to Custom.

##### ProductDataType — Product word and fraction lengths

Full precision (default) | Same as input | Custom

Specify the product data type as Full precision, Same as input, or Custom.

##### CustomProductDataType — Product word and fraction lengths

numericType([],32,30) (default) | numericType

Specify the product fixed-point type as a scaled numericType object with a Signedness of Auto.

**Dependencies**

This property applies when you set the `ProductDataType` property to `Custom`.

**AccumulatorDataType — Accumulator word and fraction lengths**

`Full precision (default) | Same as input | Same as product | Custom`

Specify the accumulator data type as `Full precision`, `Same as input`, `Same as product`, or `Custom`.

**CustomAccumulatorDataType — Accumulator word and fraction lengths**

`numerictype([],32,30) (default) | numerictype`

Specify the accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`.

**Dependencies**

This property applies when you set the `AccumulatorDataType` property to `Custom`.

**OutputDataType — Output word and fraction lengths**

`Full precision (default) | Same as input | Custom`

Specify the output data type as `Full precision`, `Same as input`, or `Custom`.

**CustomOutputDataType — Output word and fraction lengths**

`numerictype([],16,15) (default) | numerictype`

Specify the output fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`.

**Dependencies**

This property applies when you set the `OutputDataType` property to `Custom`.

**Usage****Syntax**

`y = ift(x)`

**Description**

`y = ift(x)` computes the inverse discrete Fourier transform (IDFT) , `y`, of the input `x` along the first dimension of `x`.

**Input Arguments****x — Data input**

`vector | matrix | N-D array`

Data input, specified as a vector, matrix, or *N*-D array.

When the `FFTLengthSource` property is `Auto`, the length of `x` along the first dimension must be a positive integer power of two. When the `FFTLengthSource` property is `'Property'`, the length of `x` along the first dimension can be any positive integer and the `FFTLength` property must be a positive integer power of two.

Variable-size input signals are only supported when the `FFTLengthSource` property is set to `'Auto'`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

## Output Arguments

### **y** — Inverse discrete Fourier transform of input signal

vector | matrix |  $N$ -D array

Inverse discrete Fourier transform of input signal, returned as a vector, matrix, or  $N$ -D array.

When `FFTLengthSource` property is set to `'Auto'`, the FFT length is same as the number of rows in the input signal. When `FFTLengthSource` property is set to `'Property'`, the FFT length is specified through the `FFTLength` property.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Common to All System Objects

|                      |  |
|----------------------|--|
| <code>step</code>    | Run System object algorithm  |
| <code>release</code> | Release resources and allow changes to System object property values and input characteristics |
| <code>reset</code>   | Reset internal states of System object   |

## Examples

### Construct a Sinusoidal Signal Using High Energy FFT Coefficients

Compute the FFT of a noisy sinusoidal input signal. The energy of the signal is stored as the magnitude square of the FFT coefficients. Determine the FFT coefficients which occupy 99.99% of the signal energy and reconstruct the time-domain signal by taking the IFFT of these coefficients. Compare the reconstructed signal with the original signal.

Consider a time-domain signal  $x[n]$ , which is defined over the finite time interval  $0 \leq n \leq N - 1$ . The energy of the signal  $x[n]$  is given by the following equation:

$$E_N = \sum_{n=0}^{N-1} |x[n]|^2$$

FFT coefficients,  $X[k]$ , are considered signal values in the frequency domain. The energy of the signal  $x[n]$  in the frequency-domain is therefore the sum of the squares of the magnitude of the FFT coefficients:

$$E_N = \frac{1}{N} \sum_{k=0}^{N-1} |X[k]|^2$$

According to Parseval's theorem, the total energy of the signal in time or frequency-domain is the same.

$$E_N = \sum_{n=0}^{N-1} |x[n]|^2 = \frac{1}{N} \sum_{k=0}^{N-1} |X[k]|^2$$

### Initialization

Initialize a `dsp.SineWave` System object to generate a sine wave sampled at 44.1 kHz and has a frequency of 1000 Hz. Construct a `dsp.FFT` and `dsp.IFFT` objects to compute the FFT and the IFFT of the input signal.

The `'FFTLengthSource'` property of each of these transform objects is set to `'Auto'`. The FFT length is hence considered as the input frame size. The input frame size in this example is 1020, which is not a power of 2, so select the `'FFTImplementation'` as `'FFTW'`.

```
L = 1020;
Sineobject = dsp.SineWave('SamplesPerFrame',L,...
    'PhaseOffset',10,...
    'SampleRate',44100,...
    'Frequency',1000);
ft = dsp.FFT('FFTImplementation','FFTW');
ift = dsp.IFFT('FFTImplementation','FFTW',...
    'ConjugateSymmetricInput',true);
rng(1);
```

### Streaming

Stream in the noisy input signal. Compute the FFT of each frame and determine the coefficients that constitute 99.99% energy of the signal. Take IFFT of these coefficients to reconstruct the time-domain signal.

```
numIter = 1000;
for Iter = 1:numIter
    Sinewave1 = Sineobject();
    Input = Sinewave1 + 0.01*randn(size(Sinewave1));
    FFTCoeff = ft(Input);
    FFTCoeffMagSq = abs(FFTCoeff).^2;

    EnergyFreqDomain = (1/L)*sum(FFTCoeffMagSq);
    [FFTCoeffSorted, ind] = sort((1/L)*FFTCoeffMagSq), ...
        1, 'descend');

    CumFFTCoeffs = cumsum(FFTCoeffSorted);
    EnergyPercent = (CumFFTCoeffs/EnergyFreqDomain)*100;
    Vec = find(EnergyPercent > 99.99);
    FFTCoeffsModified = zeros(L,1);
    FFTCoeffsModified(ind(1:Vec(1))) = FFTCoeff(ind(1:Vec(1)));
    ReonstrSignal = ift(FFTCoeffsModified);
end
```

99.99% of the signal energy can be represented by the number of FFT coefficients given by `Vec(1)`:

```
Vec(1)
```

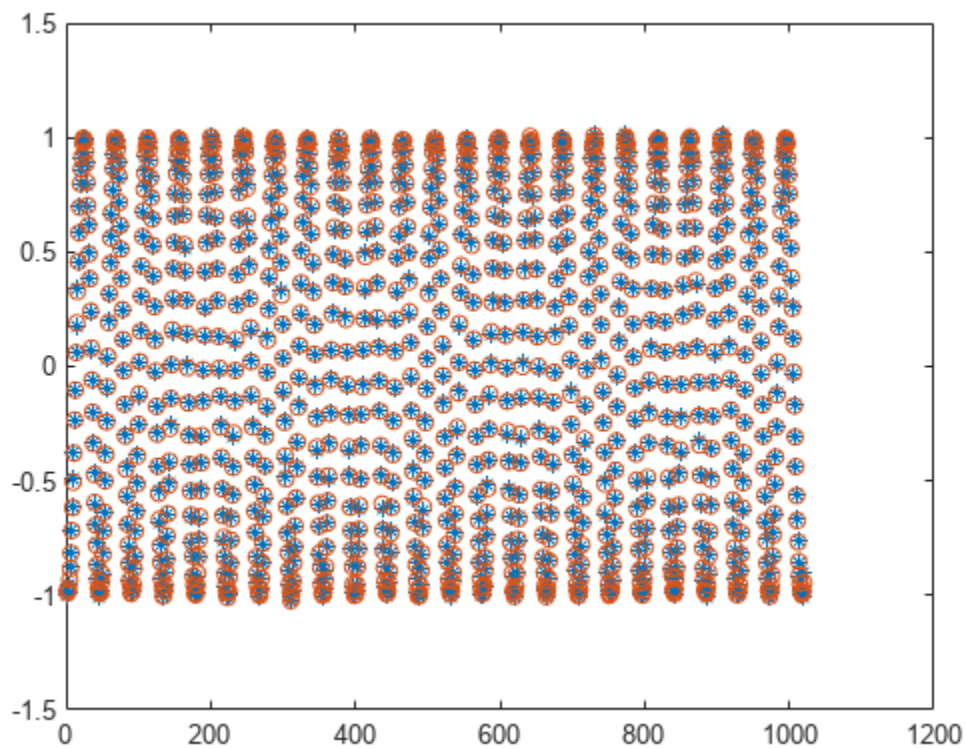
```
ans = 296
```

The signal is reconstructed efficiently using these coefficients. If you compare the last frame of the reconstructed signal with the original time-domain signal, you can see that the difference is very small and the plots match closely.

```
max(abs(Input-ReconstrSignal))
```

```
ans = 0.0431
```

```
plot(Input, '*');
hold on;
plot(ReconstrSignal, 'o');
hold off;
```



## Algorithms

This object implements the algorithm, inputs, and outputs described on the IFFT block reference page. The object properties correspond to the block parameters, except the **Output sampling mode** parameter is not supported by `dsp.IFFT`.

## Version History

Introduced in R2012a

### References

[1] FFTW (<https://www.fftw.org>)

[2] Frigo, M. and S. G. Johnson, "FFTW: An Adaptive Software Architecture for the FFT," *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Vol. 3, 1998, pp. 1381-1384.

### Extended Capabilities

#### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- See "System Objects in MATLAB Code Generation" (MATLAB Coder).
- When the following conditions apply, the executable generated from this System object relies on prebuilt dynamic library files (.dll files) included with MATLAB:
  - `FFTImplementation` is set to 'FFTW'.
  - `FFTImplementation` is set to 'Auto', `FFTLengthSource` is set to 'Property', and `FFTLength` is not a power of two.

Use the `packNGo` function to package the code generated from this System object and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see "How To Run a Generated Executable Outside MATLAB".

- When the FFT length is a power of two, you can generate standalone C and C++ code from this System object.

### See Also

#### Objects

`dsp.FFT`

# dsp.IIRFilter

**Package:** dsp

Infinite impulse response (IIR) filter

## Description

The `dsp.IIRFilter` System object filters each channel of the input using the specified filter. You can specify the filter to have a 'Direct form I', 'Direct form I transposed', 'Direct form II', or a 'Direct form II transposed' structure.

Use the “Numerator” on page 4-0 and “Denominator” on page 4-0 properties to specify the coefficients of the filter numerator and denominator coefficients. In addition to these coefficients, you can also specify nonzero initial filter states through the “InitialConditions” on page 4-0 property.

To filter a signal using an IIR filter:

- 1 Create the `dsp.IIRFilter` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
iir = dsp.IIRFilter
iir = dsp.IIRFilter(Name,Value)
```

### Description

`iir = dsp.IIRFilter` creates an infinite impulse response (IIR) filter System object that independently filters each channel of the input over time using a specified IIR filter implementation.

`iir = dsp.IIRFilter(Name,Value)` creates an IIR filter object with each specified property set to the specified value. Enclose each property name in single quotes.

Example: `iir = dsp.IIRFilter('Structure','Direct form I');`

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

**Structure — IIR filter structure**

'Direct form II transposed' (default) | 'Direct form I' | 'Direct form I transposed' | 'Direct form II'

IIR filter structure, specified as 'Direct form I', 'Direct form I transposed', 'Direct form II', or 'Direct form II transposed'.

**Numerator — Numerator coefficients**

[1 1] (default) | row vector

Numerator coefficients, specified as a row vector.

Example: [0.0296 0.1775 0.4438 0.5918 0.4438 0.1775 0.0296]

**Tunable:** Yes

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64  
Complex Number Support: Yes

**Denominator — Denominator coefficients**

[1 0.1] (default) | row vector

Denominator coefficients, specified as a row vector. The leading denominator coefficient must equal 1.

Example: [1.0000 -0.0000 0.7777 -0.0000 0.1142 -0.0000 0.0018]

**Tunable:** Yes

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64  
Complex Number Support: Yes

**InitialConditions — Initial conditions**

0 (default) | scalar | vector | matrix

Initial conditions of the filter states, specified as one of the following:

- scalar -- The object initializes all delay elements in the filter to the scalar value.
- vector -- The length of the vector equals the number of delay elements in the filter. Each vector element specifies a unique initial condition for the corresponding delay element. The object applies the same vector to each channel of the input signal.
- matrix -- The number of rows in the matrix must equal the number of delay elements in the filter. The number of columns in the matrix must equal the number of channels in the input. Each element specifies a unique initial condition for the corresponding delay element in the corresponding channel.

The number of filter states equals  $\max(N,M) - 1$ , where  $N$  is the number of poles, and  $M$  is the number of zeros.

**Tunable:** Yes

**Dependencies**

This property applies only when you set the "Structure" on page 4-0 property to 'Direct form II' or 'Direct form II transposed'.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64



Complex Number Support: Yes

### **NumeratorInitialConditions — Initial conditions on zeros side**

0 (default) | scalar | vector | matrix

Initial conditions of the filter states on the side of the filter structure with the zeros, specified as one of the following:

- scalar -- The object initializes all delay elements on the zeros side in the filter to the scalar value.
- vector -- The length of the vector equals the number of delay elements on the zeros side in the filter. Each vector element specifies a unique initial condition for the corresponding delay element on the zeros side. The object applies the same vector of initial conditions to each channel of the input signal.
- matrix -- The number of rows in the matrix must equal the number of delay elements on the zeros side in the filter. The number of columns in the matrix must equal the number of channels in the input signal. Each element specifies a unique initial condition for the corresponding delay element on the zeros side in the corresponding channel.

The number of filter states equals  $\max(N,M) - 1$ , where  $N$  is the number of poles, and  $M$  is the number of zeros, respectively.

**Tunable:** Yes

#### **Dependencies**

This property applies only when you set the “Structure” on page 4-0 property to 'Direct form I' or 'Direct form I transposed'.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64  
Complex Number Support: Yes

### **DenominatorInitialConditions — Initial conditions on poles side**

0 (default) | scalar | vector | matrix

Initial conditions of the filter states on the side of the filter structure with the poles, specified as one of the following:

- scalar -- The object initializes all delay elements on the poles side in the filter to the scalar value.
- vector -- The length of the vector equals the number of delay elements on the poles side in the filter. Each vector element specifies a unique initial condition for the corresponding delay element on the poles side. The object applies the same vector of initial conditions to each channel of the input signal.
- matrix -- The number of rows in the matrix must equal the number of delay elements on the poles side in the filter. The number of columns in the matrix must equal the number of channels in the input signal. Each element specifies a unique initial condition for the corresponding delay element on the poles side in the corresponding channel.

The number of filter states equals  $\max(N,M) - 1$ , where  $N$  is the number of poles, and  $M$  is the number of zeros, respectively.

**Tunable:** Yes

**Dependencies**

This property applies only when you set the “Structure” on page 4-0 property to 'Direct form I' or 'Direct form I transposed'.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`  
Complex Number Support: Yes

**Fixed-Point Properties****RoundingMethod — Rounding method**

'Floor' (default) | 'Ceiling' | 'Convergent' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'

Select the rounding mode for fixed-point operations.

**OverflowAction — Overflow action**

'Wrap' (default) | 'Saturate'

Overflow action for fixed-point operations, specified as one of the following:

- 'Wrap' -- The object wraps the result of its fixed-point operations.
- 'Saturate' -- The object saturates the result of its fixed-point operations.

For more details on overflow actions, see overflow mode for fixed-point operations.

**StateDataType — State data type**

'Same as input' (default) | 'Custom'

State data type, specified as one of the following:

- 'Same as input' -- The state data type is same as the input data type.
- 'Custom' -- The state output data type is an autosigned numeric type through the “CustomStateDataType” on page 4-0 property.

**CustomStateDataType — State word and fraction lengths**

`numericType([],16,15)` (default)

State word and fraction lengths, specified as an autosigned numeric type with a word length of 16 and a fraction length of 15.

**Dependencies**

This property applies only when you set “StateDataType” on page 4-0 to 'Custom'.

**NumeratorCoefficientsDataType — Data type of numerator coefficients**

'Same word length as input' (default) | 'Custom'

Data type of numerator coefficients, specified as one of the following:

- 'Same word length as input' -- The word length of the numerator coefficients is the same as the input word length. The fraction length is chosen to give the best possible precision.
- 'Custom' -- The data type of the numerator coefficients is the autosigned numeric type specified by the “CustomNumeratorCoefficientsDataType” on page 4-0 property.

### **CustomNumeratorCoefficientsDataType — Word and fraction lengths of the numerator coefficients**

`numericType([],16,15)` (default)

Word and fraction lengths of the numerator coefficients, specified as an autosigned numeric type with a word length of 16 and a fraction length of 15.

#### **Dependencies**

This property applies only when you set “NumeratorCoefficientsDataType” on page 4-0 to 'Custom'.

### **DenominatorCoefficientsDataType — Data type of the denominator coefficients**

`'Same word length as input'` (default) | `'Custom'`

Data type of the denominator coefficients, specified as one of the following:

- `'Same word length as input'` -- The word length of the denominator coefficients is the same as that of the input word length. The fraction length is chosen to give the best possible precision.
- `'Custom'` -- The data type of the denominator coefficients is the autosigned numeric type specified by the “CustomDenominatorCoefficientsDataType” on page 4-0 property.

### **CustomDenominatorCoefficientsDataType — Word and fraction lengths of denominator coefficients**

`numericType([],16,15)` (default)

Word and fraction lengths of denominator coefficients, specified as an autosigned numeric type with a word length of 16 and a fraction length of 15.

#### **Dependencies**

This property applies only when you set “DenominatorCoefficientsDataType” on page 4-0 to 'Custom'.

### **NumeratorProductDataType — Numerator product data type**

`'Full precision'` (default) | `'Same as input'` | `'Custom'`

Data type of the output of a product operation in the numerator polynomial of the IIR filter, specified as one of the following:

- `'Full precision'` -- The object computes the numerator product output data type using the full-precision rules. These rules provide the most accurate fixed-point numerics. No quantization occurs. Bits are added, as needed, to ensure that no roundoff or overflow occurs.
- `'Same as input'` -- The product output data type is the same as the input data type.
- `'Custom'` -- The product output data type is the custom numeric type specified by the “CustomNumeratorProductDataType” on page 4-0 property. The rounding method and the overflow action are specified by the “RoundingMethod” on page 4-0 and “OverflowAction” on page 4-0 properties.

### **CustomNumeratorProductDataType — Numerator product word and fraction lengths**

`numericType([],32,30)` (default)

Numerator product word and fraction lengths, specified as an autosigned numeric type with a word length of 32 and a fraction length of 30.

**Dependencies**

This property applies only when you set “NumeratorProductDataType” on page 4-0 to 'Custom'.

**DenominatorProductDataType — Denominator product data type**

'Full precision' (default) | 'Same as input' | 'Custom'

Data type of the output of a product operation in the denominator polynomial of the IIR filter, specified as one of the following:

- 'Full precision' -- The object computes the denominator product output data type using the full-precision rules. These rules provide the most accurate fixed-point numerics. No quantization occurs. Bits are added, as needed, to ensure that no roundoff or overflow occurs.
- 'Same as input' -- The product output data type is the same as the input data type.
- 'Custom' -- The product output data type is custom numeric type specified by the “CustomDenominatorProductDataType” on page 4-0 property. The rounding method and the overflow action are specified by the “RoundingMethod” on page 4-0 and “OverflowAction” on page 4-0 properties.

**CustomDenominatorProductDataType — Denominator product word and fraction lengths**

numericity([],32,30) (default)

Denominator product word and fraction lengths, specified as an autosigned numeric type with a word length of 32 and a fraction length of 30.

**Dependencies**

This property applies only when you set “DenominatorProductDataType” on page 4-0 to 'Custom'.

**NumeratorAccumulatorDataType — Numerator accumulator data type**

'Full precision' (default) | 'Same as input' | 'Same as product' | 'Custom'

Data type of the output of an accumulation operation in the numerator polynomial of the IIR filter, specified as one of the following:

- 'Full precision' -- The object computes the numerator accumulator data type using the full-precision rules. These rules provide the most accurate fixed-point numerics. No quantization occurs. Bits are added, as needed, to ensure that no roundoff or overflow occurs.
- 'Same as input' -- The accumulator data type is the same as the input data type.
- 'Same as product' -- The accumulator data type is the same as the product output data type.
- 'Custom' -- The accumulator data type is the custom numeric type specified by the “CustomNumeratorAccumulatorDataType” on page 4-0 property. The rounding method and the overflow action are specified by the “RoundingMethod” on page 4-0 and “OverflowAction” on page 4-0 properties.

**CustomNumeratorAccumulatorDataType — Numerator accumulator word and fraction lengths**

numericity([],32,30) (default)

Numerator accumulator word and fraction lengths, specified as an autosigned numeric type with a word length of 32 and a fraction length of 30.

**Dependencies**

This property applies only when you set “NumeratorAccumulatorDataType” on page 4-0 to 'Custom'.

**DenominatorAccumulatorDataType — Denominator accumulator data type**

'Full precision' (default) | 'Same as input' | 'Same as product' | 'Custom'

Data type of the output of an accumulation operation in the denominator polynomial of the IIR filter, specified as one of the following:

- 'Full precision' -- The object computes the denominator accumulator data type using the full-precision rules. These rules provide the most accurate fixed-point numerics. No quantization occurs. Bits are added, as needed, to ensure that no roundoff or overflow occurs.
- 'Same as input' -- The accumulator data type is the same as the input data type.
- 'Same as product' -- The accumulator data type is the same as the product output data type.
- 'Custom' -- The accumulator data type is the custom numeric type specified by the “CustomDenominatorAccumulatorDataType” on page 4-0 property. The rounding method and the overflow action are specified by the “RoundingMethod” on page 4-0 and “OverflowAction” on page 4-0 properties.

**CustomDenominatorAccumulatorDataType — Denominator accumulator word and fraction lengths**

numericType([],32,30) (default)

Denominator accumulator word and fraction lengths, specified as an autosigned numeric type with a word length of 32 and a fraction length of 30.

**Dependencies**

This property applies only when you set “DenominatorAccumulatorDataType” on page 4-0 to 'Custom'.

**OutputDataType — Output data type**

'Same as input' (default) | 'Full precision' | 'Custom'

Data type of the output of the dsp.IIRFilter object, specified as one of the following:

- 'Same as input' -- The output data type is the same as the input data type.
- 'Full precision' -- The object computes the output data type using the full-precision rules. These rules provide the most accurate fixed-point numerics. No quantization occurs. Bits are added, as needed, to ensure that no roundoff or overflow occurs.
- 'Custom' -- The output data type is the custom numeric type specified by the “CustomOutputDataType” on page 4-0 property. The rounding method and the overflow action are specified by the “RoundingMethod” on page 4-0 and “OverflowAction” on page 4-0 properties.

**CustomOutputDataType — Output word and fraction lengths**

numericType([],16,15) (default)

Output word and fraction lengths, specified as an autosigned numeric type with a word length of 16 and a fraction length of 15.

**Dependencies**

This property applies only when you set “OutputDataType” on page 4-0 to 'Custom'.

**MultiplicandDataType — Multiplicand data type**

'Same as input' (default) | 'Custom'

Multiplicand data type, specified as one of the following:

- 'Same as input' -- The multiplicand data type is the same as the input data type.
- 'Custom' -- The multiplicand data type is the autosigned numeric type specifies by the “CustomMultiplicandDataType” on page 4-0 property.

**CustomMultiplicandDataType — Multiplicand output word and fraction lengths**

numericType([],16,15) (default)

Multiplicand output word and fraction lengths, specified as an autosigned numeric type with a word length of 16 and a fraction length of 15.

**Dependencies**

This property applies only when you set “MultiplicandDataType” on page 4-0 to 'Custom'.

**Usage****Syntax**

```
iirOut = iir(input)
```

**Description**

`iirOut = iir(input)` filters the input signal using the specified filter to produce the filtered output. The System object filters each column of the input signal independently over time.

**Input Arguments****input — Data input**

vector | matrix

Data input that is filtered, specified as a vector or matrix.

Example: `randn(34,24)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `fi`

**Output Arguments****iirOut — Filtered output**

vector | matrix

Filtered output, returned as a vector or a matrix. The size, data type, and complexity of the output match that of the input.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `fi`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

### Specific to dsp.IIRFilter

|                     |  |
|---------------------|--|
| <code>freqz</code>  | Frequency response of discrete-time filter System object         |
| <code>fvtool</code> | Visualize frequency response of DSP filters                      |
| <code>impz</code>   | Impulse response of discrete-time filter System object           |
| <code>phasez</code> | Phase response of discrete-time filter System object (unwrapped) |
| <code>sos</code>    | Convert to second order sections                                 |

### Common to All System Objects

|                      |  |
|----------------------|--|
| <code>step</code>    | Run System object algorithm  |
| <code>release</code> | Release resources and allow changes to System object property values and input characteristics |
| <code>reset</code>   | Reset internal states of System object   |

For a list of filter analysis methods this object supports, type `dsp.IIRFilter.helpFilterAnalysis` in the MATLAB command prompt. For the corresponding function reference pages, see “Analysis Methods for Filter System Objects” on page 3-2.

## Examples

### Filter Noisy Signal Using IIR Filter

Filter a noisy sinusoidal signal using a lowpass butterworth IIR filter.

#### Input Signal

The input signal has three tones, 1 kHz, 5 kHz, and 12 kHz.

```
Sine1 = dsp.SineWave('Frequency',1e3,...
    'SampleRate',44.1e3);
Sine2 = dsp.SineWave('Frequency',5e3,...
    'SampleRate',44.1e3);
Sine3 = dsp.SineWave('Frequency',12e3,...
    'SampleRate',44.1e3);
```

#### Filter Design

Use the `butter` function to design a 10th order lowpass Butterworth filter.

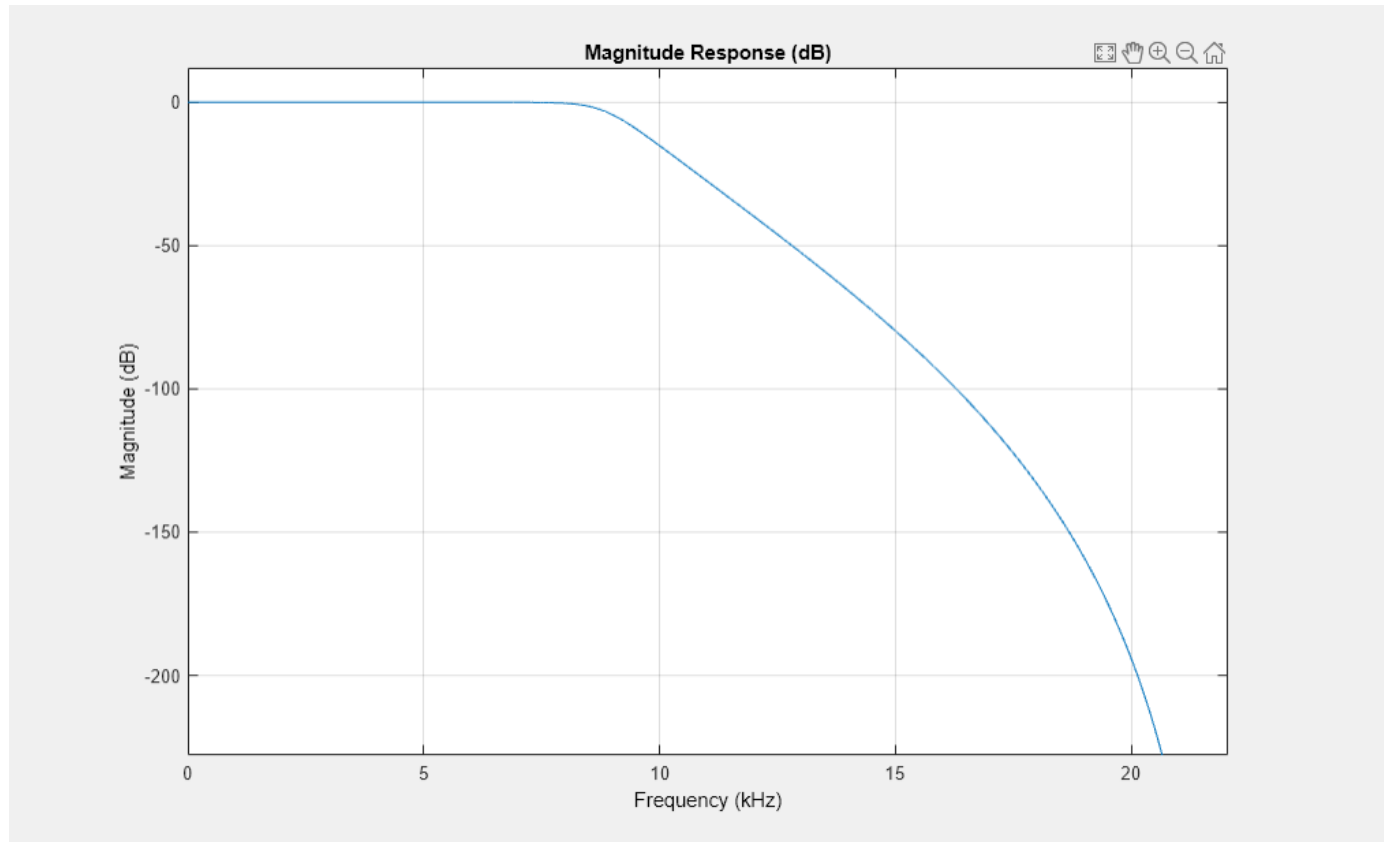
```
N = 10;
Fc = 0.4;
[b,a] = butter(N,Fc);
```

Create a `dsp.IIRFilter` object and assign the designed coefficients to the Numerator and the Denominator properties.

```
iir = dsp.IIRFilter('Numerator',b,...
    'Denominator',a);
```

View the magnitude response of the filter. The cutoff frequency is at  $0.4\pi$  rad/sample, which, with a sample rate of 44.1 kHz, translates to  $0.4 \times 44100/2$  or 8.82 kHz.

```
fvtool(iir,'Fs',Sine1.SampleRate)
```



View the power spectrum of the input and output signal using the Spectrum Analyzer.

```
sa = spectrumAnalyzer('SampleRate',Sine1.SampleRate,...
    'Method','welch',...
    'PlotAsTwoSidedSpectrum',false,...
    'OverlapPercent',80,...
    'SpectrumUnits','dBW',...
    'YLimits',[-220 -10]);
```

### Streaming

Add zero-mean white Gaussian noise with a standard deviation of 0.01 to the sum of sine waves. Filter the signal using the IIR filter.

The tones at 1 kHz and 5 kHz are unaffected since they fall in the passband. The frequency at 12 kHz is attenuated since it falls in the transition band of the filter.

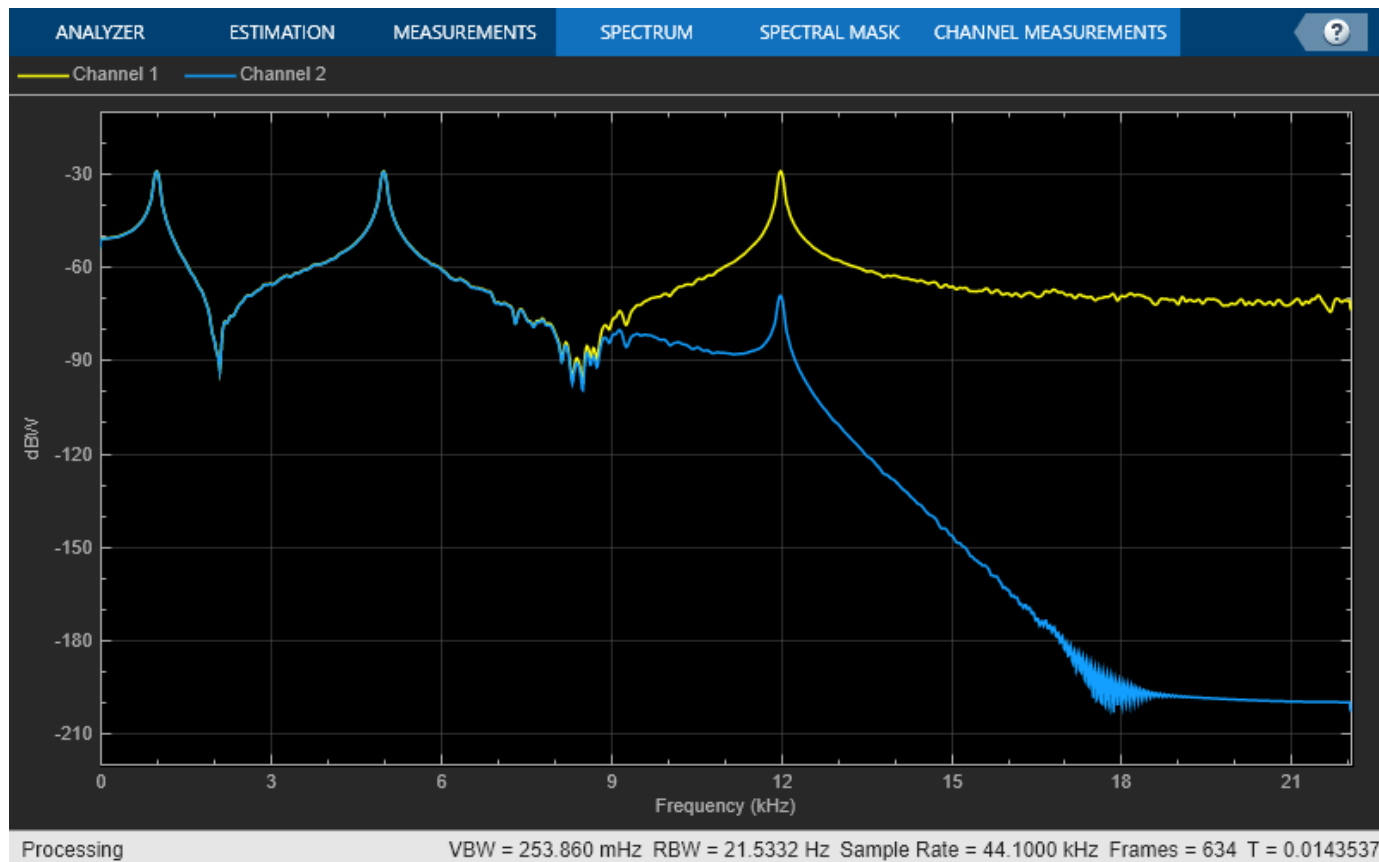
```
for i = 1:1000
    input = Sine1()+Sine2()+Sine3()+...
        0.01*randn(Sine1.SamplesPerFrame,1);
```



```

output = iir(input);
sa(input,output)
end

```



## Design an IIR Filter

Design a notching comb filter with 8 notches, and a notch bandwidth of 0.02 referenced to a -3 dB level.

Create a comb filter design specification object using the `fdesign.comb` function and specify these design parameters.

```
combSpecs = fdesign.comb('notch','N,BW',8,0.02);
```

Design the notching comb filter using the `design` function. The resulting filter is a `dsp.IIRFilter System` object™. For details on how to apply this filter on streaming data, refer to `dsp.IIRFilter`.

```
iirFilt = design(combSpecs,'Systemobject',true)
```

```
iirFilt =
dsp.IIRFilter with properties:
```

```

Structure: 'Direct form II'
Numerator: [0.8878 0 0 0 0 0 0 0 -0.8878]
Denominator: [1 0 0 0 0 0 0 0 -0.7757]

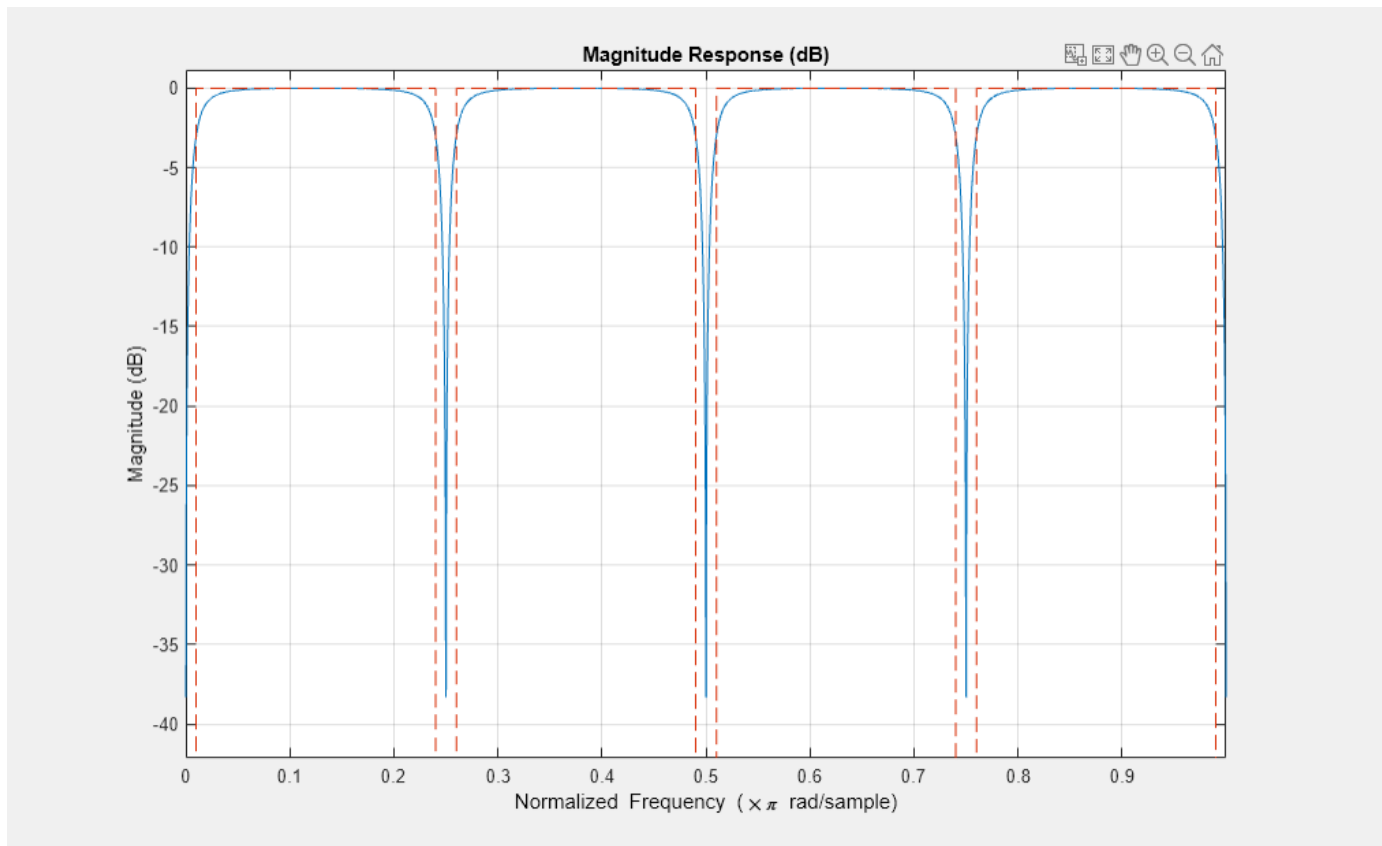
```

InitialConditions: 0

Show all properties

View magnitude response of the designed filter using `fvtool`.

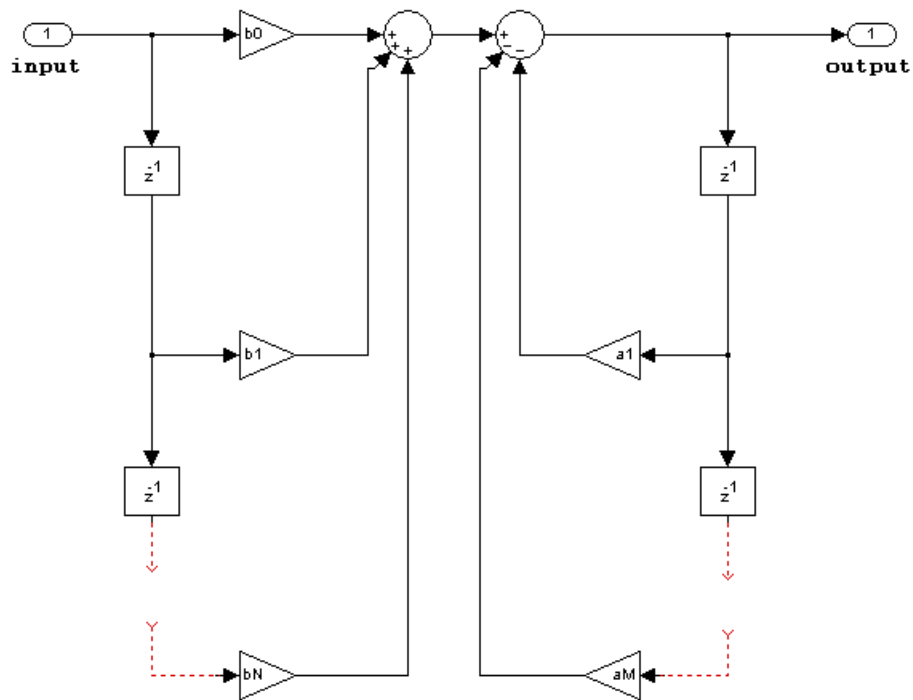
```
fvtool(iirFilt)
```



## More About

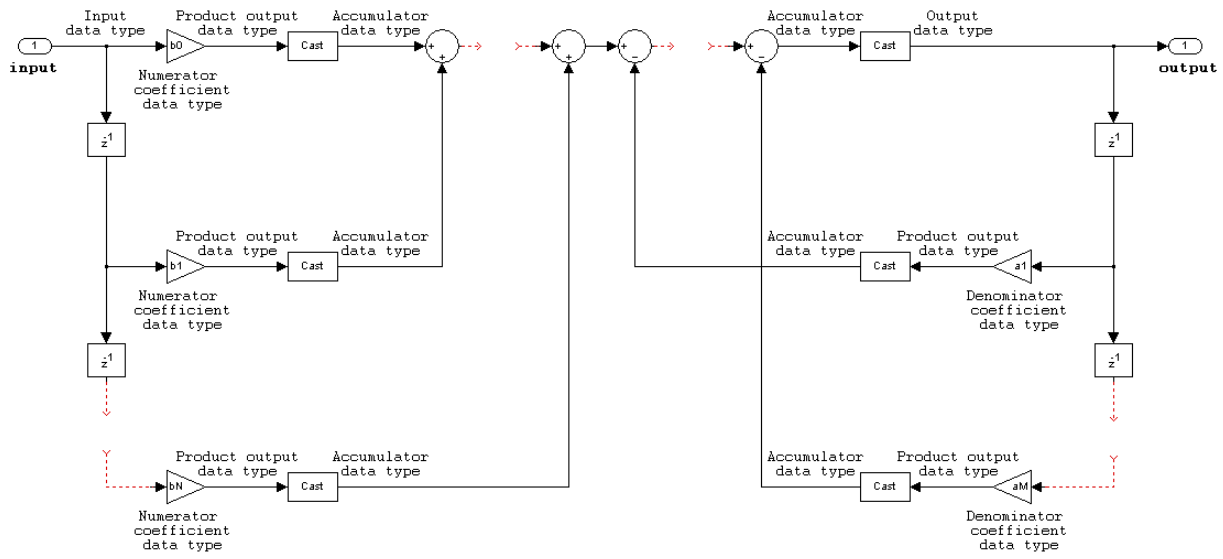
### Fixed Point

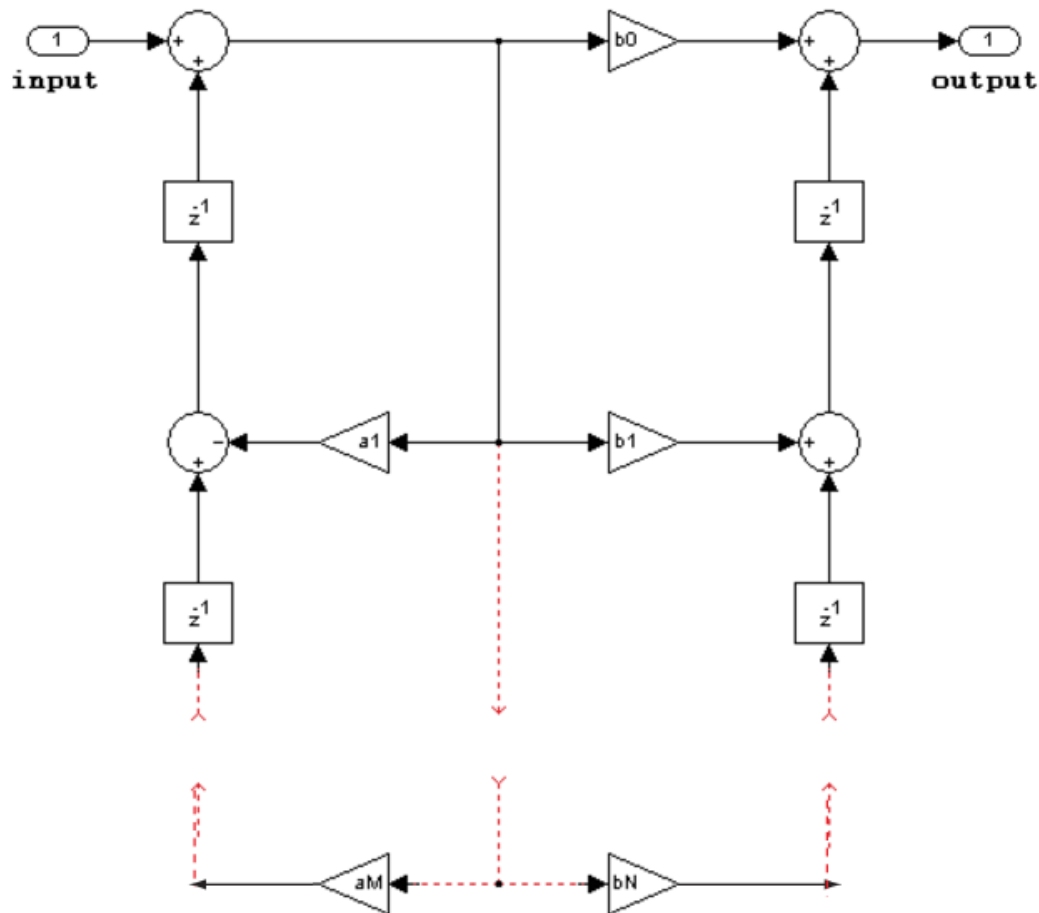
The `dsp.IIRFilter` System object supports the following filter structures. The diagrams in each section show the data types used in the filter structures for fixed-point signals. You can set the data types using the fixed-point properties of the object.

**Direct form I**

The following constraints apply when the Structure property is set to 'Direct form I':

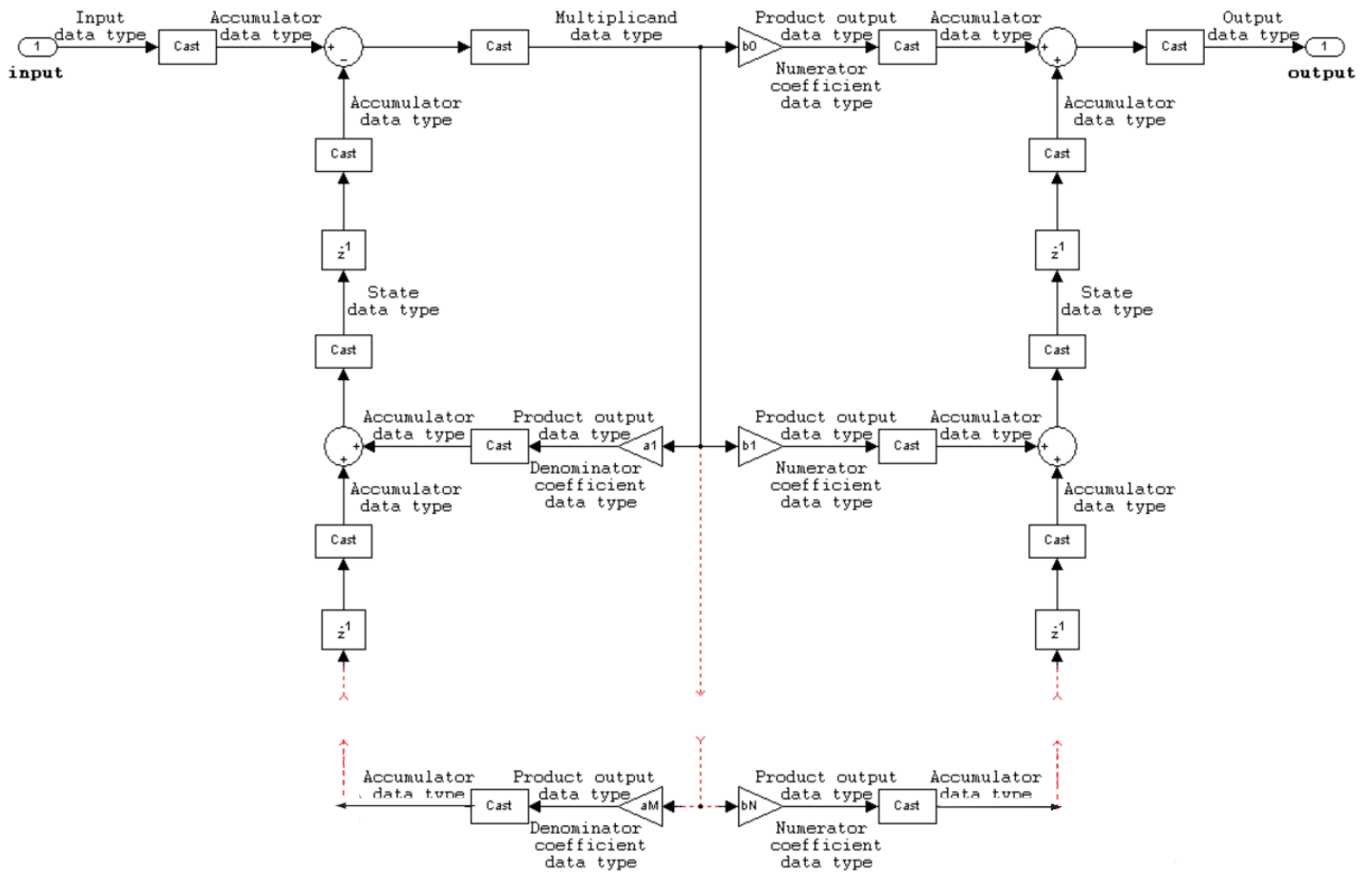
- Inputs can be real or complex.
- Numerator and denominator coefficients can be real or complex.
- Numerator and denominator coefficients must have the same complexity characteristics. When the numerator and denominator coefficients have different complexities from each other, the object processes the filter as if two sets of complex coefficients are provided. The real-valued coefficient set is treated as if it is a complex vector with zero-valued imaginary parts.
- The State data type cannot be specified for this structure. Doing so is not possible because the input and output states have the same data types as the input and output buffers.

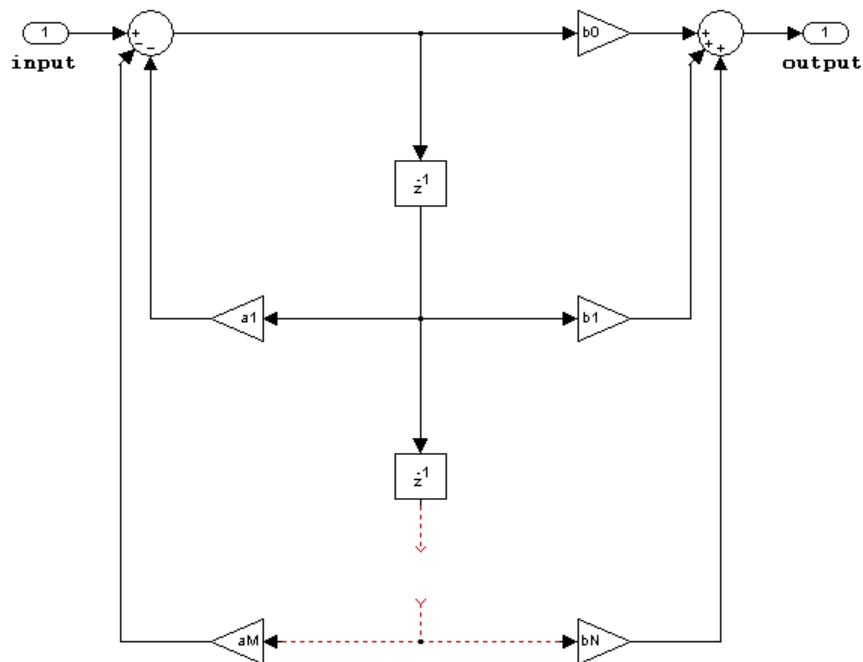


**Direct form I transposed**

The following constraints apply when the Structure property is set to 'Direct form I transposed':

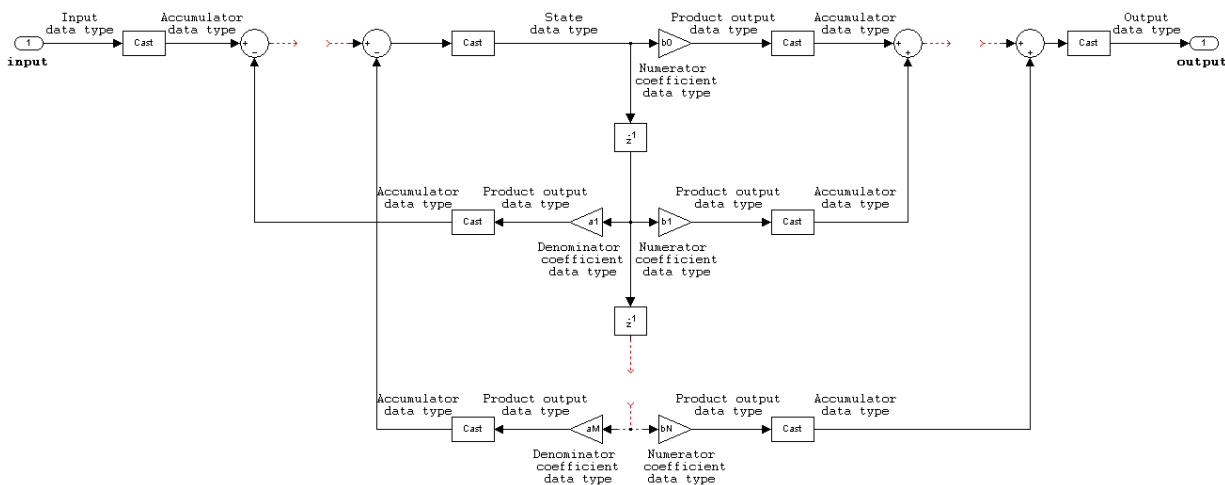
- Inputs can be real or complex.
- Numerator and denominator coefficients can be real or complex.
- Numerator and denominator coefficients must have the same complexity characteristics. When the numerator and denominator coefficients have different complexities from each other, the object processes the filter as if two sets of complex coefficients are provided. The real-valued coefficient set is treated as if it is a complex vector with zero-valued imaginary parts.
- States are complex when either the input or the coefficients are complex.

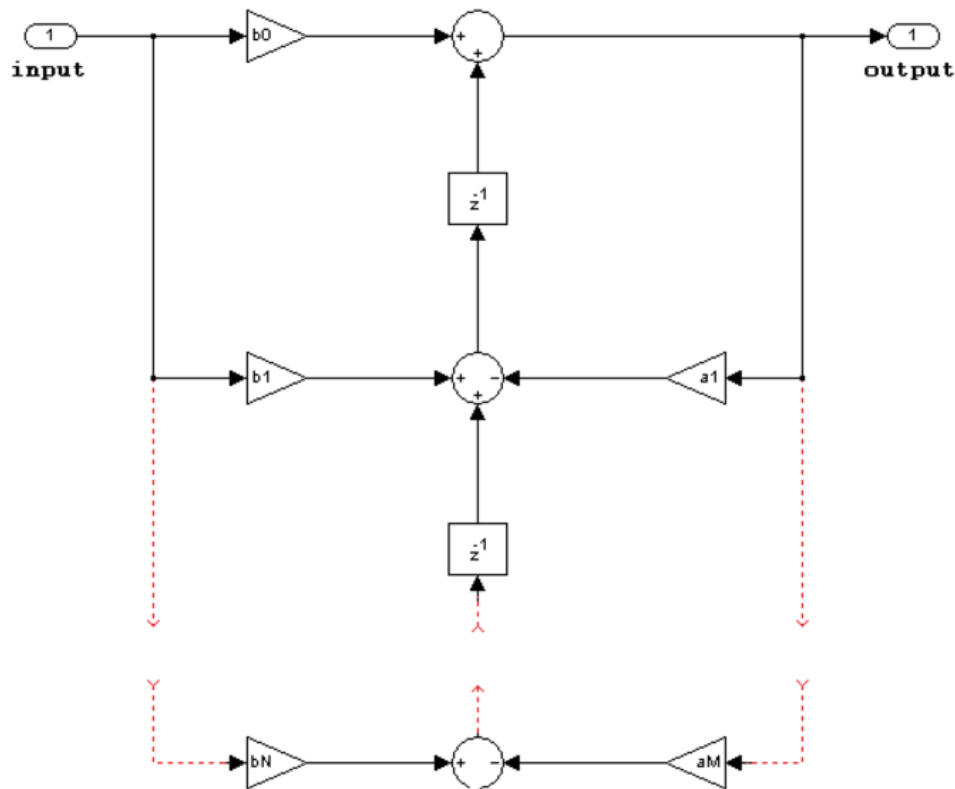


**Direct form II**

The following constraints apply when the Structure property is set to 'Direct form II':

- Inputs can be real or complex.
- Numerator and denominator coefficients can be real or complex.
- Numerator and denominator coefficients must have the same complexity characteristics. When the numerator and denominator coefficients have different complexities from each other, the object processes the filter as if two sets of complex coefficients are provided. The real-valued coefficient set is treated as if it is a complex vector with zero-valued imaginary parts.
- States are complex when either the inputs or the coefficients are complex.

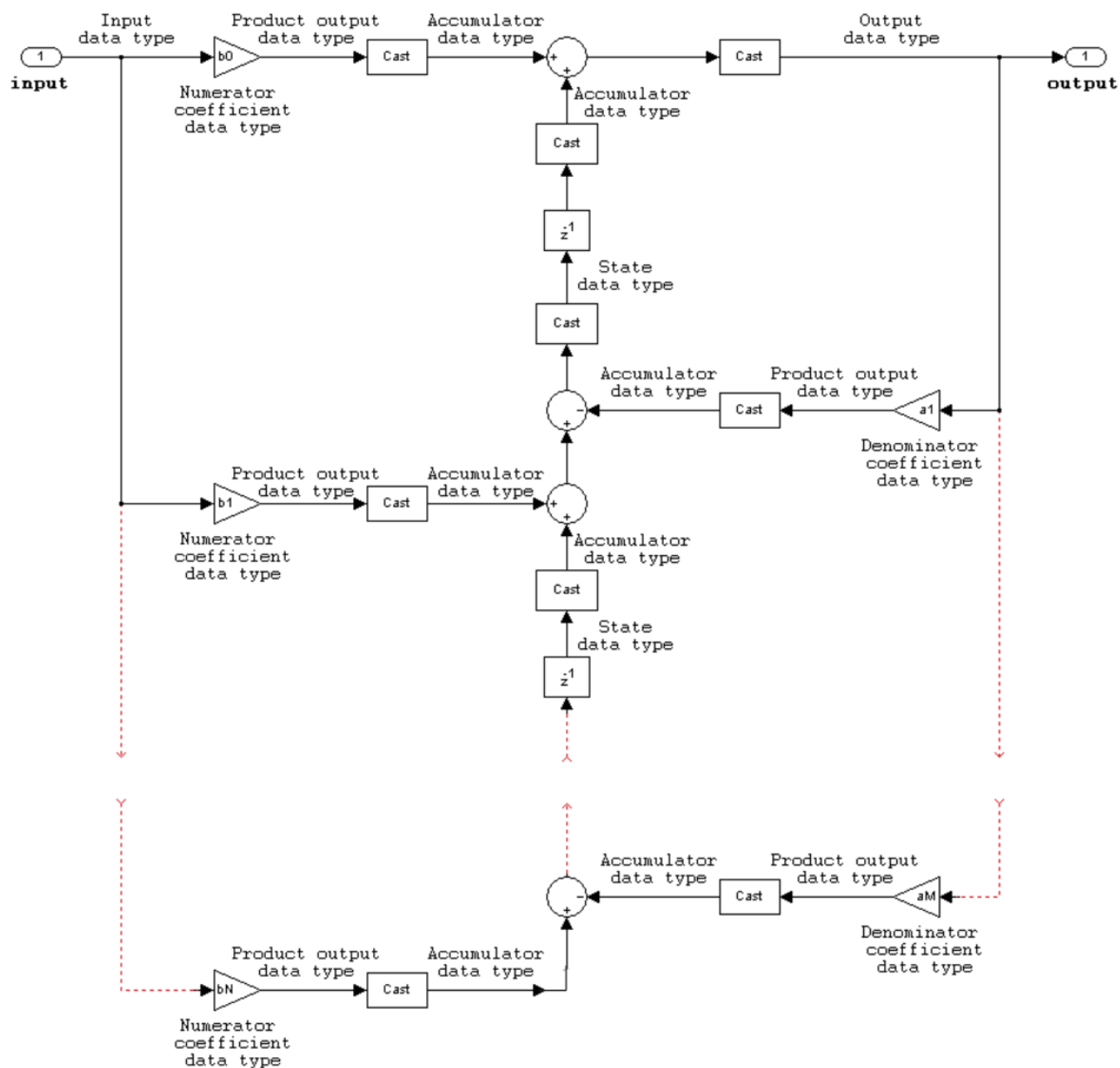


**Direct form II transposed**

The following constraints apply when the Structure property is set to 'Direct form II transposed':

- Inputs can be real or complex.
- Numerator and denominator coefficients can be real or complex.
- Numerator and denominator coefficients must have the same complexity characteristics. When the numerator and denominator coefficients have different complexities from each other, the object processes the filter as if two sets of complex coefficients are provided. The real-valued coefficient set is treated as if it is a complex vector with zero-valued imaginary parts.
- States are complex when either the inputs or the coefficients are complex.





## Version History

Introduced in R2012b

### Extended Capabilities

#### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Only the `Numerator` and `Denominator` properties are tunable for code generation.
- See “System Objects in MATLAB Code Generation” (MATLAB Coder).

### See Also

#### Functions

`freqz` | `fvtool` | `impz` | `phasez`

#### Objects

`dsp.SOSFilter` | `dsp.FIRFilter` | `dsp.AllpoleFilter`

# dsp.IIRHalfbandDecimator

**Package:** dsp

Decimate by factor of two using polyphase IIR

## Description

The `dsp.IIRHalfbandDecimator` System object performs efficient polyphase decimation of the input signal by a factor of two. To design the halfband filter, you can specify the object to use an elliptic design or a quasi-linear phase design. The object uses these design methods to compute the filter coefficients. To filter the inputs, the object uses a polyphase structure. The allpass filters in the polyphase structure are in a minimum multiplier form.

Elliptic design introduces nonlinear phase and creates the filter using fewer coefficients than quasi linear design. Quasi-linear phase design overcomes phase nonlinearity at the cost of additional coefficients.

Alternatively, instead of designing the halfband filter using a design method, you can specify the filter coefficients directly. When you choose this option, the allpass filters in the two branches of the polyphase implementation can be in a minimum multiplier form or in a wave digital form.

You can also use the `dsp.IIRHalfbandDecimator` object to implement the analysis portion of a two-band filter bank to filter a signal into lowpass and highpass subbands.

To filter and downsample your data:

- 1 Create the `dsp.IIRHalfbandDecimator` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
iirhalfbanddecim = dsp.IIRHalfbandDecimator
iirhalfbanddecim = dsp.IIRHalfbandDecimator(Name, Value)
```

### Description

`iirhalfbanddecim = dsp.IIRHalfbandDecimator` returns a halfband decimator, `iirhalfbanddecim`, with the default settings. Under the default settings, the System object filters and downsamples the input data with a halfband frequency of 22050 Hz, a transition width of 4100 Hz, and a stopband attenuation of 80 dB.

`iirhalfbanddecim = dsp.IIRHalfbandDecimator(Name, Value)` returns an IIR halfband decimator, with additional properties specified by one or more `Name, Value` pair arguments.

Example: `iirhalfbanddecim = dsp.IIRHalfbandDecimator('Specification','Filter order and stopband attenuation')` creates an IIR halfband decimator object with filter order set to 9 and stopband attenuation set to 80 dB.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see *System Design in MATLAB Using System Objects*.

### Specification — Filter design parameters

'Transition width and stopband attenuation' (default) | 'Filter order and stopband attenuation' | 'Filter order and transition width' | 'Coefficients'

Filter design parameters, specified as a character vector. When you set `Specification` to one of the filter design options, you can specify the filter design parameters using the corresponding `FilterOrder`, `StopbandAttenuation`, and `TransitionWidth` properties. Also, you can specify the design method using `DesignMethod`. When you set `Specification` to 'Coefficients', you can specify the coefficients directly.

### FilterOrder — Order of the IIR halfband filter

9 (default) | positive scalar integer

Order of the IIR halfband filter, specified as a positive scalar integer. If you set `DesignMethod` to 'Elliptic', then `FilterOrder` must be an odd integer greater than one. If you set `DesignMethod` to 'Quasi-linear phase', then `FilterOrder` must be a multiple of four.

#### Dependencies

This property applies when you set `Specification` to 'Filter order and stopband attenuation' or 'Filter order and transition width'.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### StopbandAttenuation — Minimum attenuation needed in stopband

80 (default) | positive real scalar

Minimum attenuation needed in the stopband of the IIR halfband filter, specified as a positive real scalar. Units are in dB.

#### Dependencies

This property applies only when you set `Specification` to 'Filter order and stopband attenuation' or 'Transition width and stopband attenuation'.

Data Types: `single` | `double`

### TransitionWidth — Transition width

4100 (default) | positive real scalar

Transition width of the IIR halfband filter, specified as a positive real scalar. Units are in Hz. The value of the transition width must be less than half the input sample rate.

**Dependencies**

This property applies only when you set `Specification` to 'Transition width and stopband attenuation' or 'Filter order and transition width'.

Data Types: `single` | `double`

**DesignMethod — Design method**

'Elliptic' (default) | 'Quasi-linear phase'

Design method for the IIR halfband filter, specified as 'Elliptic' or 'Quasi-linear phase'. When the property is set to 'Quasi-linear phase', the first branch of the polyphase structure is a pure delay, which results in an approximately linear phase response.

**Dependencies**

This property applies only when you set `Specification` to any accepted value except 'Coefficients'.

**SampleRate — Input sample rate**

44100 (default) | positive real scalar

Input sample rate, specified as a positive real scalar. Units are in Hz.

**Dependencies**

This property applies only when you set `Specification` to any accepted value except 'Coefficients'.

Data Types: `single` | `double`

**Structure — Internal allpass filter implementation structure**

'Minimum multiplier' (default) | 'Wave Digital Filter'

Internal allpass filter implementation structure, specified as 'Minimum multiplier' or 'Wave Digital Filter'.

This property is not tunable.

**Dependencies**

This property applies only when you set 'Specification' to 'Coefficients'. Each structure uses a different coefficients set, independently stored in the corresponding object property.

**AllpassCoefficients1 — Allpass polynomial filter coefficients of first branch**

[0.1284563; 0.7906755] (default) | [0.1284563 0.1534; 0.7906755 0.6745]

Allpass polynomial filter coefficients of the first branch, specified as an  $N$ -by-1 or  $N$ -by-2 matrix.  $N$  is the number of first-order or second-order allpass sections.

**Tunable:** Yes

**Dependencies**

This property applies only when you set `Specification` to 'Coefficients' and `Structure` to 'Minimum multiplier'.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**AllpassCoefficients2 — Allpass polynomial filter coefficients of second branch**`[0.4295667] (default) | [0.7906755 0.1534]`

Allpass polynomial filter coefficients of the second branch, specified as an  $N$ -by-1 or  $N$ -by-2 matrix.  $N$  is the number of first-order or second-order allpass sections.

**Tunable:** Yes

**Dependencies**

This property applies only when you set Specification to 'Coefficients' and Structure to 'Minimum multiplier'.

Data Types: `single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64`

**WDFCoefficients1 — Allpass filter coefficients of first branch in Wave Digital Filter form**`[0.1284563; 0.7906755] (default) | [0.1284563 0.1534; 0.7906755 0.6745]`

Allpass filter coefficients of the first branch in Wave Digital Filter form, specified as an  $N$ -by-1 or  $N$ -by-2 matrix.  $N$  is the number of first-order or second-order allpass sections. Each element must have an absolute value less than or equal to 1.

This property is not tunable.

**Dependencies**

This property applies only when you set Specification to 'Coefficients' and Structure to 'Wave Digital Filter'.

Data Types: `single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64`

**WDFCoefficients2 — Allpass filter coefficients of second branch in Wave Digital Filter form**`[0.4295667] (default) | [0.7906755 0.1534]`

Allpass filter coefficients of the second branch in Wave Digital Filter form, specified as the comma-separated pair consisting of 'WDFCoefficients2' and a  $N$ -by-1 or  $N$ -by-2 matrix.  $N$  is the number of first-order or second-order allpass sections. Each element must have an absolute value less than or equal to 1.

This property is not tunable.

**Dependencies**

This property applies only when you set 'Specification' to 'Coefficients' and 'Structure' to 'Wave Digital Filter'.

Data Types: `single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64`

**HasPureDelayBranch — Make first branch a pure delay**`false (default) | true`

Flag to make the first allpass branch a delay, specified as a logical scalar. When this property is true, the first branch is treated as a pure delay and the properties AllpassCoefficients1 and WDFCoefficients1 do not apply.

This property is not tunable.

**Dependencies**

This property applies only when you set `Specification` to `'Coefficients'`.

**Delay — Length of delay**

1 (default) | finite positive scalar

Length of the first branch delay, specified as a finite positive scalar. The value of this property specifies the number of samples by which you can delay the input to the first branch.

This property is not tunable.

**Dependencies**

This property applies only when you set `Specification` to `'Coefficients'` and `HasPureDelayBranch` to 1.

Data Types: `single` | `double`

**HasTrailingFirstOrderSection — Treat the last section of the second branch as first order**

false (default) | true

Option to treat the last section of the second branch as first order, specified as a logical scalar. When this property is 1 and the coefficients of the second branch are in an  $N$ -by-2 matrix, the object ignores the second element of the last row of the matrix. The last section of the second branch then becomes a first-order section. When this property is set to 0, the last section of the second branch is a second-order section. When the coefficients of the second branch are in an  $N$ -by-1 matrix, this property is ignored.

This property is not tunable.

**Dependencies**

This property applies only when you set `Specification` to `'Coefficients'`.

**Usage****Syntax**

```
y_low = iirhalfbanddecim(x)
[y_low,y_high] = iirhalfbanddecim(x)
```

**Description**

`y_low = iirhalfbanddecim(x)` filters the input signal, `x`, using the IIR halfband filter, `iirhalfbanddecim`, and downsamples the output by a factor of 2.

`[y_low,y_high] = iirhalfbanddecim(x)` computes the `y_low` and `y_high`, of the analysis filter bank, `iirhalfbanddecim` for input `x`. A  $K_i$ -by- $N$  input matrix is treated as  $N$  independent channels. The System object generates two power-complementary output signals by adding and subtracting the two polyphase branch outputs respectively. `y_low` and `y_high` are of the same size ( $K_o$ -by- $N$ ) and data type.  $K_o = K_i/2$ , where 2 is the decimation factor.

## Input Arguments

### **x** — Data input

column vector | matrix

Data input, specified as a column vector or a matrix. The number of rows in the input signal must be even since the decimation factor is always 2 for this object. If the input is a matrix, each column is treated as an independent channel.

Data Types: `single` | `double`

## Output Arguments

### **y<sub>low</sub>** — Lowpass subband of decimator output

column vector | matrix

Lowpass subband of decimator output, returned as a column vector or a matrix. The output, `ylow` is a lowpass halfband filtered and downsampled version of the input `x`. Due to the halfband nature of the filter, the downsampling factor is always 2.

Data Types: `single` | `double`

### **y<sub>high</sub>** — Highpass subband of decimator output

column vector | matrix

Highpass subband of decimator output, returned as a column vector or a matrix. The output, `yhigh` is a highpass halfband filtered and downsampled version of the input `x`. Due to the halfband nature of the filter, the downsampling factor is always 2.

Data Types: `single` | `double`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Specific to `dsp.IIRHalfbandDecimator`

|                        |  |
|------------------------|--|
| <code>freqz</code>     | Frequency response of discrete-time filter System object |
| <code>fvtool</code>    | Visualize frequency response of DSP filters              |
| <code>info</code>      | Information about filter System object                   |
| <code>cost</code>      | Estimate cost of implementing filter System object       |
| <code>polyphase</code> | Polyphase decomposition of multirate filter              |

## Common to All System Objects

|                      |  |
|----------------------|--|
| <code>step</code>    | Run System object algorithm  |
| <code>release</code> | Release resources and allow changes to System object property values and input characteristics |
| <code>reset</code>   | Reset internal states of System object   |

## Examples



## Frequency Response of Quasi-Linear Phase IIR Halfband Decimator

Create a minimum-order lowpass IIR halfband decimation filter for data sampled at 44.1 kHz. The filter has a transition width of 4.1 kHz, and a stopband attenuation of 80 dB.

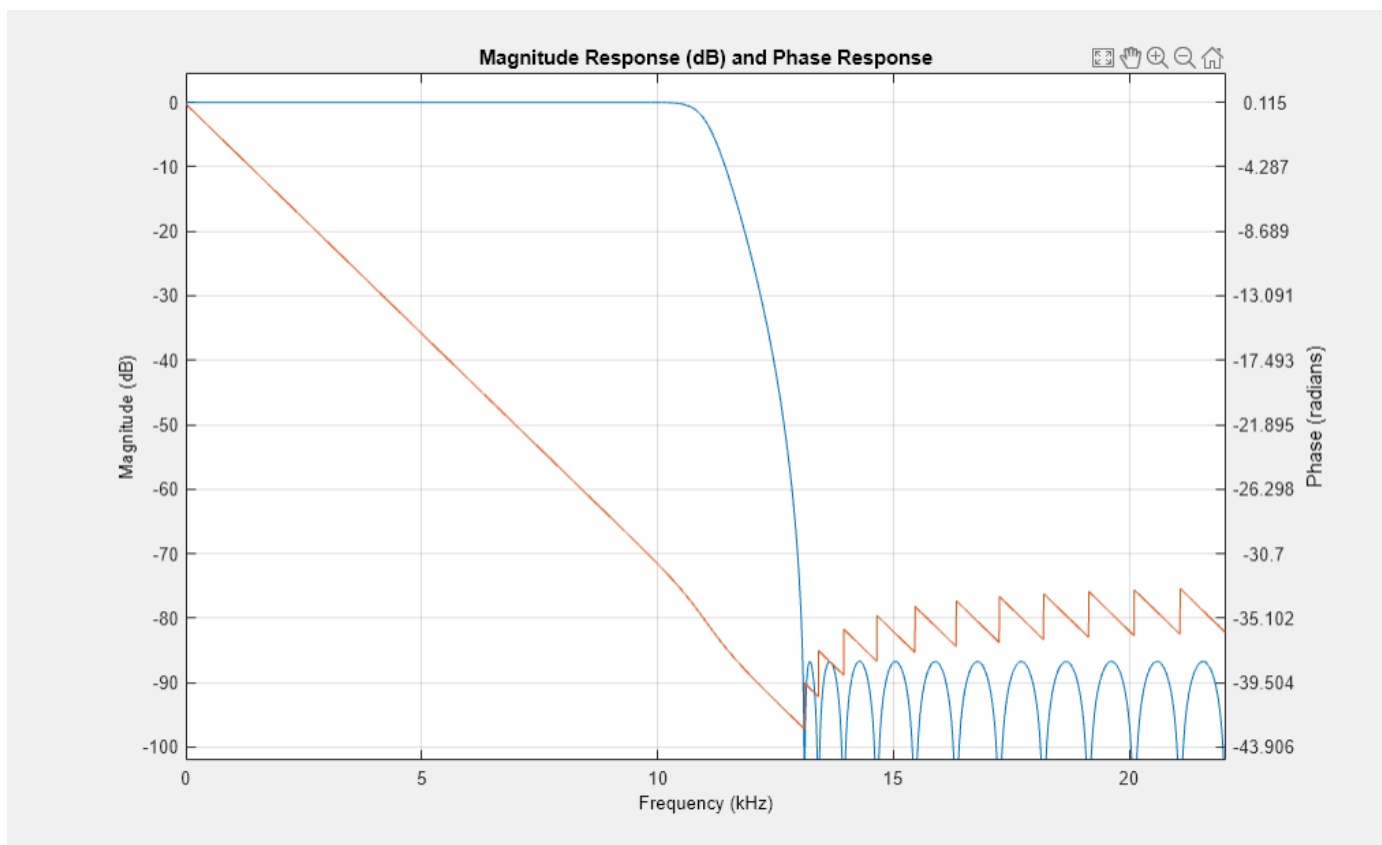
```
IIRHalfbandDecim = dsp.IIRHalfbandDecimator(...
    'DesignMethod', 'Quasi-linear phase');
```

Obtain the filter coefficients.

```
c = coeffs(IIRHalfbandDecim);
```

Plot the magnitude and phase response.

```
fvtool(IIRHalfbandDecim, 'Analysis', 'freq')
```



## Extract Low Frequency Subband from Speech

Use a halfband analysis filter bank and interpolation filter to extract the low frequency subband from a speech signal.

**Note:** The `audioDeviceWriter` System object™ is not supported in MATLAB Online.

Set up the audio file reader, the analysis filter bank, the audio device writer, and the interpolation filter. The sampling rate of the audio data is 22050 Hz. The halfband filter has an order of 21 and a transition width of 2 kHz.

```
afr = dsp.AudioFileReader('speech_dft.mp3','SamplesPerFrame',1024);
```

```
filterspec = 'Filter order and transition width';
Order = 21;
TW = 2000;
```

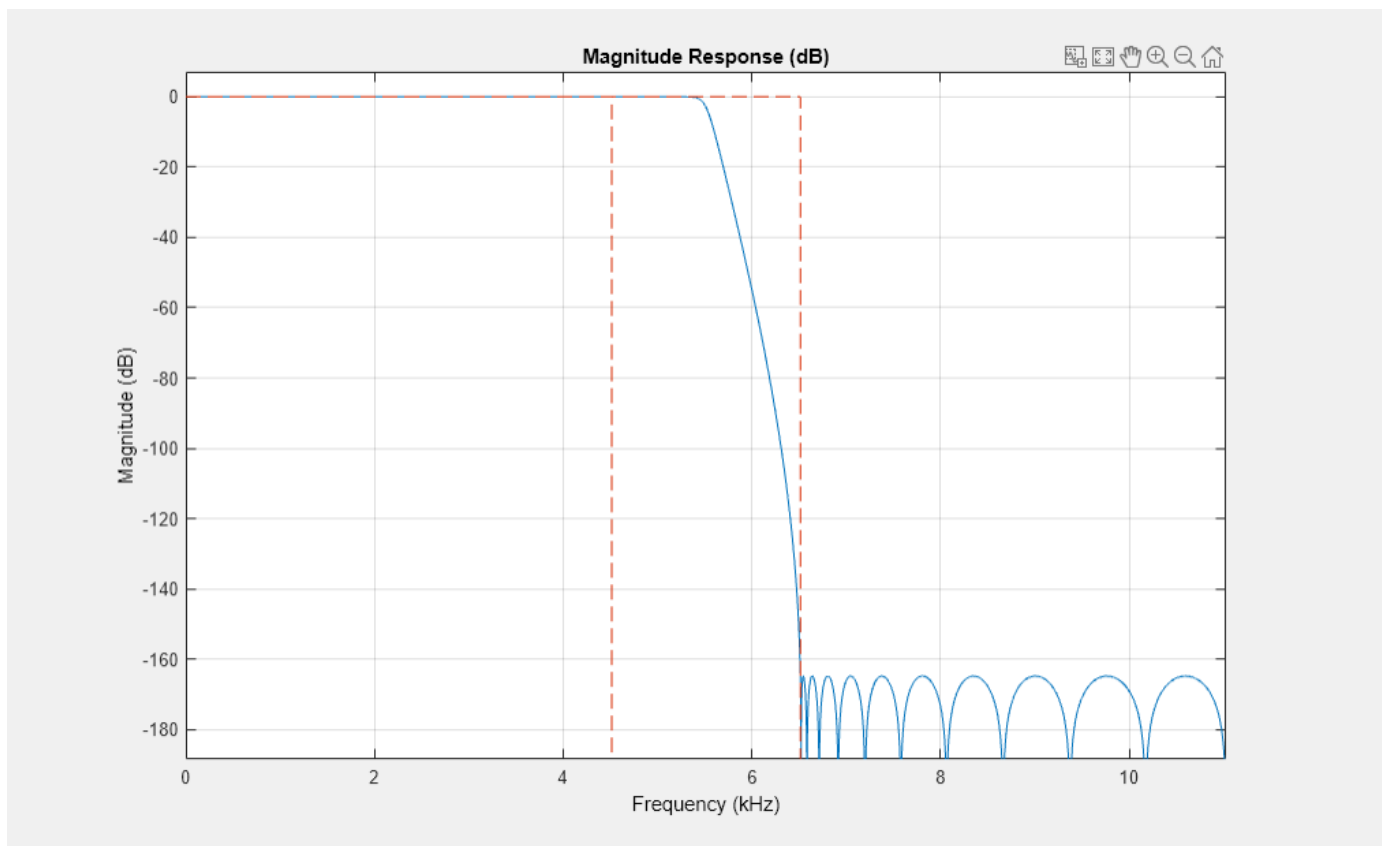
```
IIRHalfbandDecim = dsp.IIRHalfbandDecimator(...
    'Specification',filterspec,'FilterOrder',Order,...
    'TransitionWidth',TW,'SampleRate',afr.SampleRate);
```

```
IIRHalfbandInterp = dsp.IIRHalfbandInterpolator(...
    'Specification',filterspec,'FilterOrder',Order,...
    'TransitionWidth',TW,'SampleRate',afr.SampleRate/2);
```

```
ap = audioDeviceWriter('SampleRate',afr.SampleRate);
```

View the magnitude response of the halfband filter.

```
fvtool(IIRHalfbandDecim)
```



Read the speech signal from the audio file in frames of 1024 samples. Filter the speech signal into lowpass and highpass subbands with a halfband frequency of 5512.5 Hz. Reconstruct a lowpass approximation of the speech signal by interpolating the lowpass subband. Play the filtered output.

```

while ~isDone(afr)
    audioframe = afr();
    xlo = IIRHalfbandDecim(audioframe);
    ylow = IIRHalfbandInterp(xlo);
    ap(ylow);
end

```

Wait until the audio file ends, and then close the input file and release the audio output resource.

```

release(afr);
release(ap);

```

## Two-Channel Filter Bank

Use a halfband decimator and interpolator to implement a two-channel filter bank. This example uses an audio file input and shows that the power spectrum of the filter bank output does not differ significantly from the input.

**Note:** The `audioDeviceWriter` System object™ is not supported in MATLAB Online.

Set up the audio file reader and audio device writer. Construct the IIR halfband decimator and interpolator. Finally, set up the spectrum analyzer to display the power spectra of the filter-bank input and output.

```

AF = dsp.AudioFileReader('speech_dft.mp3','SamplesPerFrame',1024);
AP = audioDeviceWriter('SampleRate',AF.SampleRate);

```

```

filterspec = 'Filter order and transition width';
Order = 51;
TW = 2000;

```

```

IIRHalfbandDecim = dsp.IIRHalfbandDecimator(...
    'Specification',filterspec,'FilterOrder',Order,...
    'TransitionWidth',TW,'SampleRate',AF.SampleRate);

```

```

IIRHalfbandInterp = dsp.IIRHalfbandInterpolator(...
    'Specification',filterspec,'FilterOrder',Order,...
    'TransitionWidth',TW,'SampleRate',AF.SampleRate/2,...
    'FilterBankInputPort',true);

```

```

SpecAna = spectrumAnalyzer('SampleRate',AF.SampleRate,...
    'PlotAsTwoSidedSpectrum',false,...
    'ShowLegend',true,...
    'ChannelNames',{'Input signal','Filtered output signal'});

```

Read the audio 1024 samples at a time. Filter the input to obtain the lowpass and highpass subband signals decimated by a factor of two. This is the analysis filter bank. Use the halfband interpolator as the synthesis filter bank. Display the running power spectrum of the audio input and the output of the synthesis filter bank. Play the output.

```

while ~isDone(AF)
    audioInput = AF();
    [xlo,xhigh] = IIRHalfbandDecim(audioInput);
    audioOutput = IIRHalfbandInterp(xlo,xhigh);
    spectrumInput = [audioInput audioOutput];

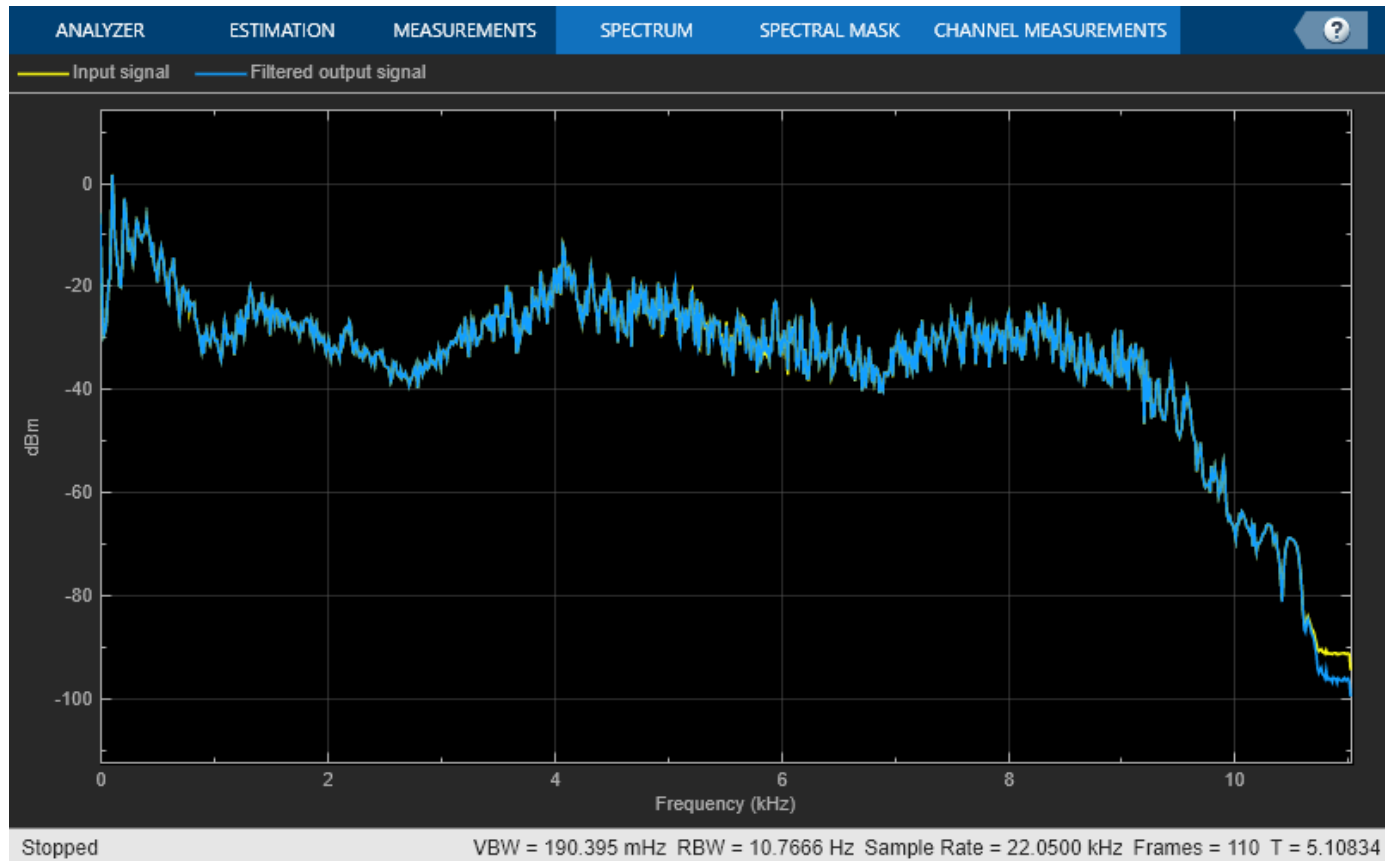
```

```

    SpecAna(spectrumInput);
    AP(audioOutput);
end

release(AF);
release(AP);
release(SpecAna);

```



### Filter Input into Lowpass and Highpass Subbands

Create a halfband decimator for data sampled at 44.1 kHz. Use a minimum-order design with a transition width of 2 kHz and a stopband attenuation of 60 dB.

```

IIRHalfbanddecim = dsp.IIRHalfbandDecimator(...
    'Specification','Transition width and stopband attenuation',...
    'TransitionWidth',2000,'StopbandAttenuation',60,'SampleRate',44.1e3);

```

Filter a two-channel input into lowpass and highpass subbands.

```

x = randn(1024,2);
[ylow,yhigh] = IIRHalfbanddecim(x);

```

## Algorithms

### Polyphase Implementation with Halfband Filters

When you filter your signal, the IIR halfband decimator uses an efficient polyphase implementation for halfband filters. You can use the polyphase implementation to move the downsample operation before filtering. This change enables you to filter at a lower sampling rate.

IIR halfband filters are generally modeled using two parallel allpass filter branches.

$$H(z) = 0.5 * [A_1(z^2) + z^{-1}A_2(z^2)]$$

### Elliptic Design

The allpass filters for elliptic IIR halfband filter are given as

$$A_1(z) = \prod_{k=1}^{K_1} \frac{a_k^{(1)} + z^{-1}}{1 + a_k^{(1)}z^{-1}}$$

$$A_2(z) = \prod_{k=1}^{K_2} \frac{a_k^{(2)} + z^{-1}}{1 + a_k^{(2)}z^{-1}}$$

### Quasi-Linear Phase Design

To achieve a near-linear phase response for IIR halfband filters, make one of the branches a pure delay. In this design, the cost of the filter increases.

The allpass filters for the quasi-linear phase IIR halfband filter are

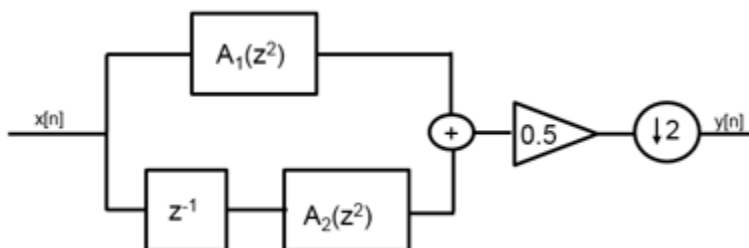
$$A_1(z) = z^{-k}$$

where  $k$  is the length of the delay.

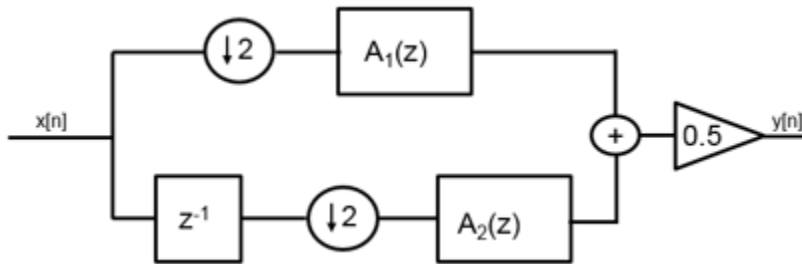
$$A_2(z) = \prod_{k=1}^{K_2^{(1)}} \frac{a_k + z^{-1}}{1 + a_k z^{-1}} \prod_{k=1}^{K_2^{(2)}} \frac{c_k + b_k z^{-1} + z^{-2}}{1 + b_k z^{-1} + c_k z^{-2}}$$

where  $N$  is the order of the IIR halfband filter.

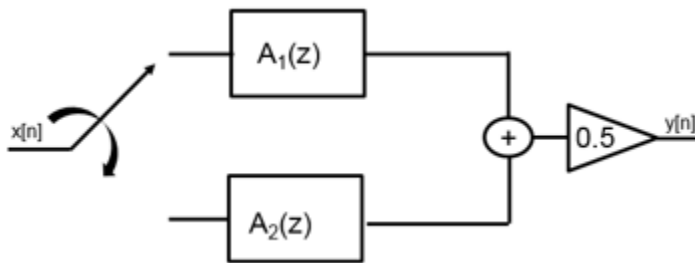
You can represent filtering the input signal and then downsampling it by 2 using this figure.



Using the multirate noble identity for downsampling, you can move the downsampling operation before the filtering operation. This change enables you to filter at a lower rate.



To implement the halfband decimator efficiently, this algorithm replaces the delay block and downsampling operator with a commutator switch.

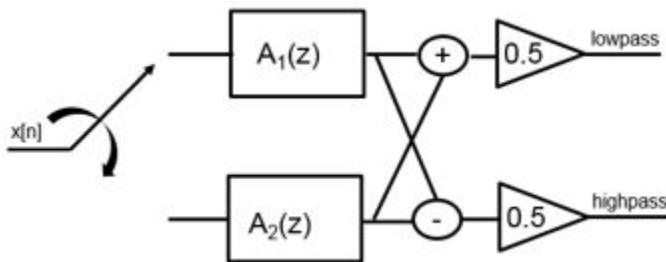


### Analysis Filter Bank

The transfer function of the complementary highpass filter branch of the analysis filter bank is given by

$$G(z) = 0.5 * [A_1(z^2) - z^{-1}A_2(z^2)]$$

You can represent the analysis filter bank as in this diagram.



The IIR halfband decimator generates two power-complementary output signals by adding and subtracting the two polyphase branch outputs respectively.

To summarize, the IIR halfband decimator:

- Decimates the input prior to filtering.
- Acts as an analysis filter bank.
- Has a nonlinear phase response and uses few coefficients with the elliptic design method.
- Has near-linear phase response at the cost of additional coefficients with the quasi-linear phase design method, where one of the branches is a pure delay

## Version History

Introduced in R2015b

## References

- [1] Lang, M. *Allpass Filter Design and Applications*. IEEE Transactions on Signal Processing. Vol. 46, No. 9, Sept 1998, pp. 2505-2514.
- [2] Harris, F.J. *Multirate Signal Processing for Communication Systems*. Prentice Hall. 2004, pp. 208-209.
- [3] Regalia, Phillip A., Sanjit K. Mitra, and P. P. Vaidyanathan. "The Digital All-Pass Filter: A Versatile Signal Processing Building Block." *Proceedings of the IEEE*. Vol. 76, Number 1, 1988, pp. 19-37.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

This object supports code generation for ARM Cortex-M and ARM Cortex-A processors.

## See Also

### Functions

freqz | fvtool | info | cost | polyphase

### Objects

dsp.IIRHalfbandInterpolator | dsp.FIRHalfbandDecimator |  
dsp.FIRHalfbandInterpolator

### Blocks

FIR Halfband Decimator | IIR Halfband Decimator

### Topics

"Analysis Methods for Filter System Objects" on page 3-2

# **dsp.IIRHalfbandInterpolator**

**Package:** dsp

Interpolate by a factor of two using polyphase IIR

## **Description**

The `dsp.IIRHalfbandInterpolator` System object performs efficient polyphase interpolation of the input signal by a factor of two. To design the halfband filter, you can specify the object to use an elliptic design or a quasi-linear phase design. The object uses these design methods to compute the filter coefficients. To filter the inputs, the object uses a polyphase structure. The allpass filters in the polyphase structure are in a minimum multiplier form.

Elliptic design introduces nonlinear phase and creates the filter using fewer coefficients than quasi linear design. Quasi-linear phase design overcomes phase nonlinearity at the cost of additional coefficients.

Alternatively, instead of designing the halfband filter using a design method, you can specify the filter coefficients directly. When you choose this option, the allpass filters in the two branches of the polyphase implementation can be in a minimum multiplier form or in a wave digital form.

You can also use `dsp.IIRHalfbandInterpolator` object to implement the synthesis portion of a two-band filter bank to synthesize a signal from lowpass and highpass subbands.

To upsample and interpolate your data:

- 1 Create the `dsp.IIRHalfbandInterpolator` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## **Creation**

### **Syntax**

```
iirhalfbandinterp = dsp.IIRHalfbandInterpolator  
iirhalfbandinterp = dsp.IIRHalfbandInterpolator(Name,Value)
```

### **Description**

`iirhalfbandinterp = dsp.IIRHalfbandInterpolator` returns an IIR halfband interpolation filter, `iirhalfbandinterp`, with the default settings. Under the default settings, the System object upsamples and interpolates the input data using a halfband frequency of 22050 Hz, a transition width of 4100 Hz, and a stopband attenuation of 80 dB.

`iirhalfbandinterp = dsp.IIRHalfbandInterpolator(Name,Value)` returns an IIR halfband interpolator, with additional properties specified by one or more `Name, Value` pair arguments.

Example: `iirhalfbandinterp = dsp.IIRHalfbandInterpolator('Specification','Filter order and stopband`



`attenuation'`) creates an IIR halfband interpolator object with filter order set to 9 and stopband attenuation set to 80 dB.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

### Specification — Filter design parameters

'Transition width and stopband attenuation' (default) | 'Filter order and stopband attenuation' | 'Filter order and transition width' | 'Coefficients'

Filter design parameters, specified as a character vector. When you set `Specification` to one of the filter design options, you can specify the filter design parameters using the corresponding `FilterOrder`, `StopbandAttenuation`, and `TransitionWidth` properties. Also, you can specify the design method using `DesignMethod`. When you set `Specification` to 'Coefficients', you can specify the coefficients directly.

### FilterOrder — Order of the IIR halfband filter

9 (default) | positive scalar integer

Order of the IIR halfband filter, specified as a positive scalar integer. If you set `DesignMethod` to 'Elliptic', then `FilterOrder` must be an odd integer greater than one. If you set `DesignMethod` to 'Quasi-linear phase', then `FilterOrder` must be a multiple of four.

#### Dependencies

This property applies when you set `Specification` to 'Filter order and stopband attenuation' or 'Filter order and transition width'.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### StopbandAttenuation — Minimum attenuation needed in stopband

80 (default) | positive real scalar

Minimum attenuation needed in the stopband of the IIR halfband filter, specified as a positive real scalar. Units are in dB.

#### Dependencies

This property applies only when you set `Specification` to 'Filter order and stopband attenuation' or 'Transition width and stopband attenuation'.

Data Types: `single` | `double`

### TransitionWidth — Transition width

4100 (default) | positive real scalar

Transition width of the IIR halfband filter, specified as a positive real scalar. Units are in Hz. The value of the transition width must be less than half the input sample rate.

**Dependencies**

This property applies only when you set `Specification` to 'Transition width and stopband attenuation' or 'Filter order and transition width'.

Data Types: `single` | `double`

**DesignMethod — Design method**

'Elliptic' (default) | 'Quasi-linear phase'

Design method for the IIR halfband filter, specified as 'Elliptic' or 'Quasi-linear phase'. When the property is set to 'Quasi-linear phase', the first branch of the polyphase structure is a pure delay, which results in an approximately linear phase response.

**Dependencies**

This property applies only when you set `Specification` to any accepted value except 'Coefficients'.

**SampleRate — Input sample rate**

44100 (default) | positive real scalar

Input sample rate, specified as a positive real scalar. Units are in Hz.

**Dependencies**

This property applies only when you set `Specification` to any accepted value except 'Coefficients'.

Data Types: `single` | `double`

**FilterBankInputPort — Option to use object as synthesis filter bank**

false (default) | true

Option to use object as synthesis filter bank, specified as a logical value. If this property is false, `dsp.IIRHalfbandInterpolator` acts as an interpolator. If this property is true, then `dsp.IIRHalfbandInterpolator` acts as a synthesis filter bank and the algorithm accepts two inputs: the lowpass and highpass subbands.

**Dependencies**

This property applies only when you set `Specification` to any accepted value except 'Coefficients'.

**Structure — Filter structure used in coefficient mode**

'Minimum multiplier' (default) | 'Wave Digital Filter'

Internal allpass filter implementation structure, specified as 'Minimum multiplier' or 'Wave Digital Filter'. Each structure uses a different coefficients set, independently stored in the corresponding object property.

This property is not tunable.

**Dependencies**

This property applies only when you set 'Specification' to 'Coefficients'.

**AllpassCoefficients1 — Allpass polynomial filter coefficients of first branch**

[0.1284563; 0.7906755] (default) | [0.1284563 0.1534; 0.7906755 0.6745]

Allpass polynomial filter coefficients of the first branch, specified as an  $N$ -by-1 or  $N$ -by-2 matrix.  $N$  is the number of first-order or second-order allpass sections.

**Tunable:** Yes**Dependencies**

This property applies only when you set Specification to 'Coefficients' and Structure to 'Minimum multiplier'.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**AllpassCoefficients2 — Allpass polynomial filter coefficients of second branch**

[0.4295667] (default) | [0.7906755 0.1534]

Allpass polynomial filter coefficients of the second branch, specified as an  $N$ -by-1 or  $N$ -by-2 matrix.  $N$  is the number of first-order or second-order allpass sections.

**Tunable:** Yes**Dependencies**

This property applies only when you set Specification to 'Coefficients' and Structure to 'Minimum multiplier'.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**WDFCoefficients1 — Allpass filter coefficients of first branch in Wave Digital Filter form**

[0.1284563; 0.7906755] (default) | [0.1284563 0.1534; 0.7906755 0.6745]

Allpass filter coefficients of the first branch in Wave Digital Filter form, specified as an  $N$ -by-1 or  $N$ -by-2 matrix.  $N$  is the number of first-order or second-order allpass sections. All elements must have an absolute value less than or equal to 1.

This property is not tunable.

**Dependencies**

This property applies only when you set Specification to 'Coefficients' and Structure to 'Wave Digital Filter'.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**WDFCoefficients2 — Allpass filter coefficients of second branch in Wave Digital Filter form**

[0.4295667] (default) | [0.7906755 0.1534]

Allpass filter coefficients of the second branch in Wave Digital Filter form, specified as an  $N$ -by-1 or  $N$ -by-2 matrix.  $N$  is the number of first-order or second-order allpass sections. All elements must have an absolute value less than or equal to 1.

This property is not tunable.

**Dependencies**

This property applies only when you set Specification to 'Coefficients' and Structure to 'Wave Digital Filter'.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **HasPureDelayBranch — Make the first branch a pure delay**

`false` (default) | `true`

Flag to make the first allpass branch a delay, specified as a logical scalar. When this property is true, the first branch is treated as a pure delay and the properties `AllpassCoefficients1` and `WDFCoefficients1` do not apply.

This property is not tunable.

#### **Dependencies**

This property applies only when you set `Specification` to `'Coefficients'`.

### **Delay — Length of the delay**

`1` (default) | finite positive scalar

Length of the first branch delay, specified as a finite positive scalar. The value of this property specifies the number of samples by which you can delay the input to the first branch.

This property is not tunable.

#### **Dependencies**

This property is applicable only when you set `'Specification'` to `'Coefficients'` and `HasPureDelayBranch` to 1.

Data Types: `single` | `double`

### **HasTrailingFirstOrderSection — Make the last section of the second branch as first order**

`false` (default) | `true`

Option to treat the last section of the second branch as first order, specified as a logical scalar. When this property is 1 and the coefficients of the second branch are in an  $N$ -by-2 matrix, the object ignores the second element of the last row of the matrix. The last section of the second branch then becomes a first-order section. When this property is set to 0, the last section of the second branch is a second-order section. When the coefficients of the second branch are in an  $N$ -by-1 matrix, this property is ignored.

This property is not tunable.

#### **Dependencies**

This property applies only when you set `Specification` to `'Coefficients'`.

## **Usage**

## **Syntax**

```
y = iirhalfbandinterp(x1)
y = iirhalfbandinterp(x1,x2)
```

## Description

`y = iirhalfbandinterp(x1)` upsamples by two and interpolates the input signal `x1` using the IIR halfband interpolator, `iirhalfbandinterp`.

`y = iirhalfbandinterp(x1,x2)` implements a halfband synthesis filter bank for the inputs `x1` and `x2`. `x1` is the lowpass output of a halfband analysis filter bank and `x2` is the highpass output of a halfband analysis filter bank. `dsp.IIRHalfbandInterpolator` implements a synthesis filter bank only when the `FilterBankInputPort` property is `true`.

## Input Arguments

### **x1** — Data input

column vector | matrix

Data input to the IIR halfband interpolator, specified as a column vector or a matrix. This signal is the lowpass output of a halfband analysis filter bank. If the input signal is a matrix, each column of the matrix is treated as an independent channel.

Data Types: `single` | `double`

### **x2** — Second data input

column vector | matrix

Second data input to the synthesis filter bank, specified as a column vector or a matrix. This signal is the highpass output of a halfband analysis filter bank. If the input signal is a matrix, each column of the matrix is treated as an independent channel.

The size, data type, and complexity of both the inputs must be the same.

Data Types: `single` | `double`

## Output Arguments

### **y** — Output of interpolator

column vector | matrix

Output of the interpolator, returned as a column vector or a matrix. The number of rows in the interpolator output is twice the number of rows in the input signal.

Data Types: `single` | `double`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Specific to `dsp.IIRHalfbandInterpolator`

|                     |  |
|---------------------|--|
| <code>freqz</code>  | Frequency response of discrete-time filter System object |
| <code>fvtool</code> | Visualize frequency response of DSP filters              |
| <code>info</code>   | Information about filter System object                   |
| <code>cost</code>   | Estimate cost of implementing filter System object       |

polyphase Polyphase decomposition of multirate filter

## Common to All System Objects

step Run System object algorithm  
 release Release resources and allow changes to System object property values and input characteristics  
 reset Reset internal states of System object

## Examples

### Frequency response of Quasi-linear Phase IIR Halfband Interpolator

Create a minimum order lowpass IIR half-band interpolation filter for data sampled at 44.1 kHz. The filter has a transition width of 4.1 kHz, and a stopband attenuation of 80 dB.

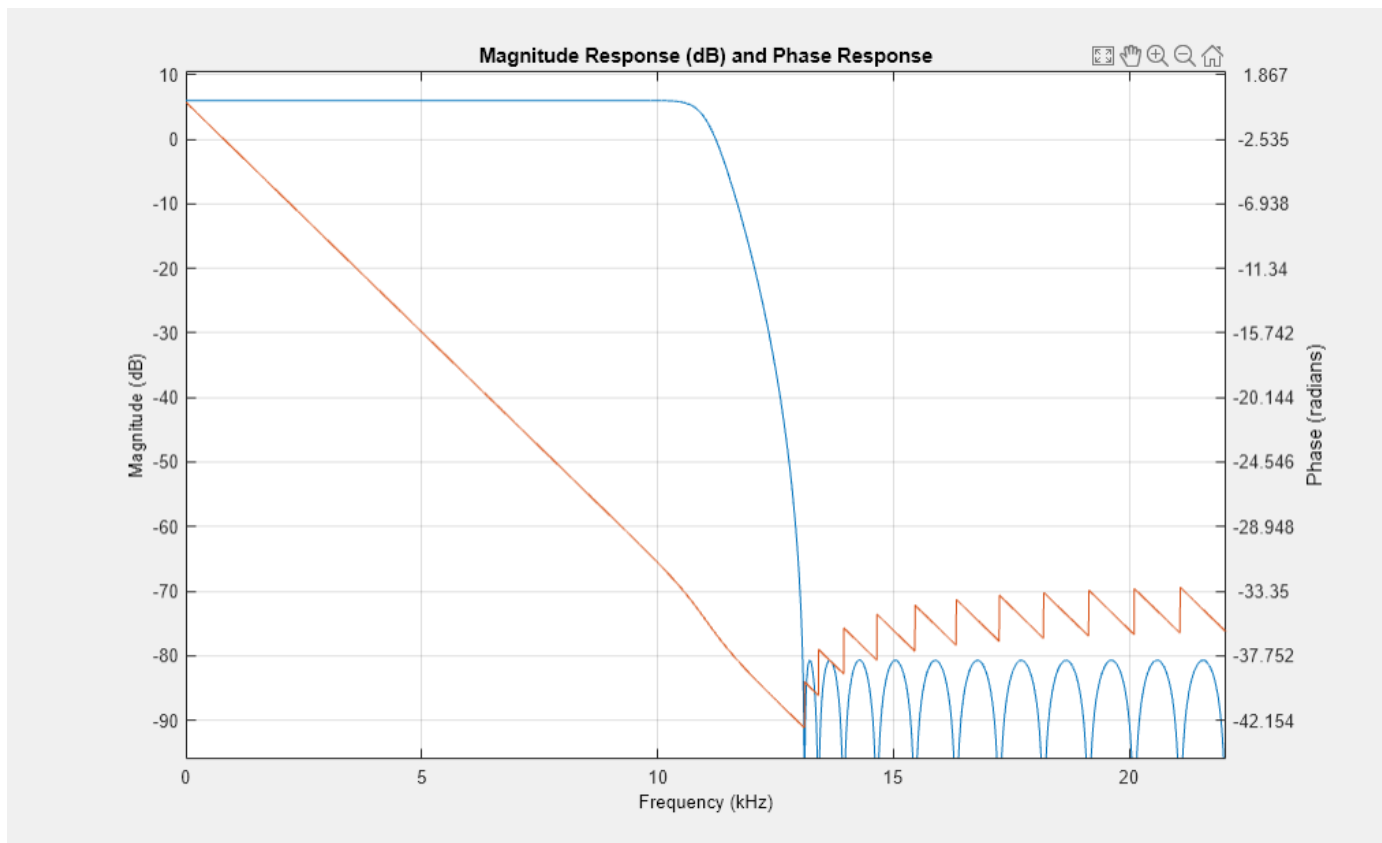
```
IIRHalfbandInterp = dsp.IIRHalfbandInterpolator(...
    'DesignMethod', 'Quasi-linear phase');
```

Obtain filter coefficients

```
c = coeffs(IIRHalfbandInterp);
```

Plot the Magnitude and Phase response

```
fvtool(IIRHalfbandInterp, 'Analysis', 'freq')
```



## Extract Low Frequency Subband from Speech

Use a halfband analysis filter bank and interpolation filter to extract the low frequency subband from a speech signal.

**Note:** The `audioDeviceWriter` System object™ is not supported in MATLAB Online.

Set up the audio file reader, the analysis filter bank, the audio device writer, and the interpolation filter. The sampling rate of the audio data is 22050 Hz. The halfband filter has an order of 21 and a transition width of 2 kHz.

```
afr = dsp.AudioFileReader('speech_dft.mp3','SamplesPerFrame',1024);
```

```
filterspec = 'Filter order and transition width';  
Order = 21;  
TW = 2000;
```

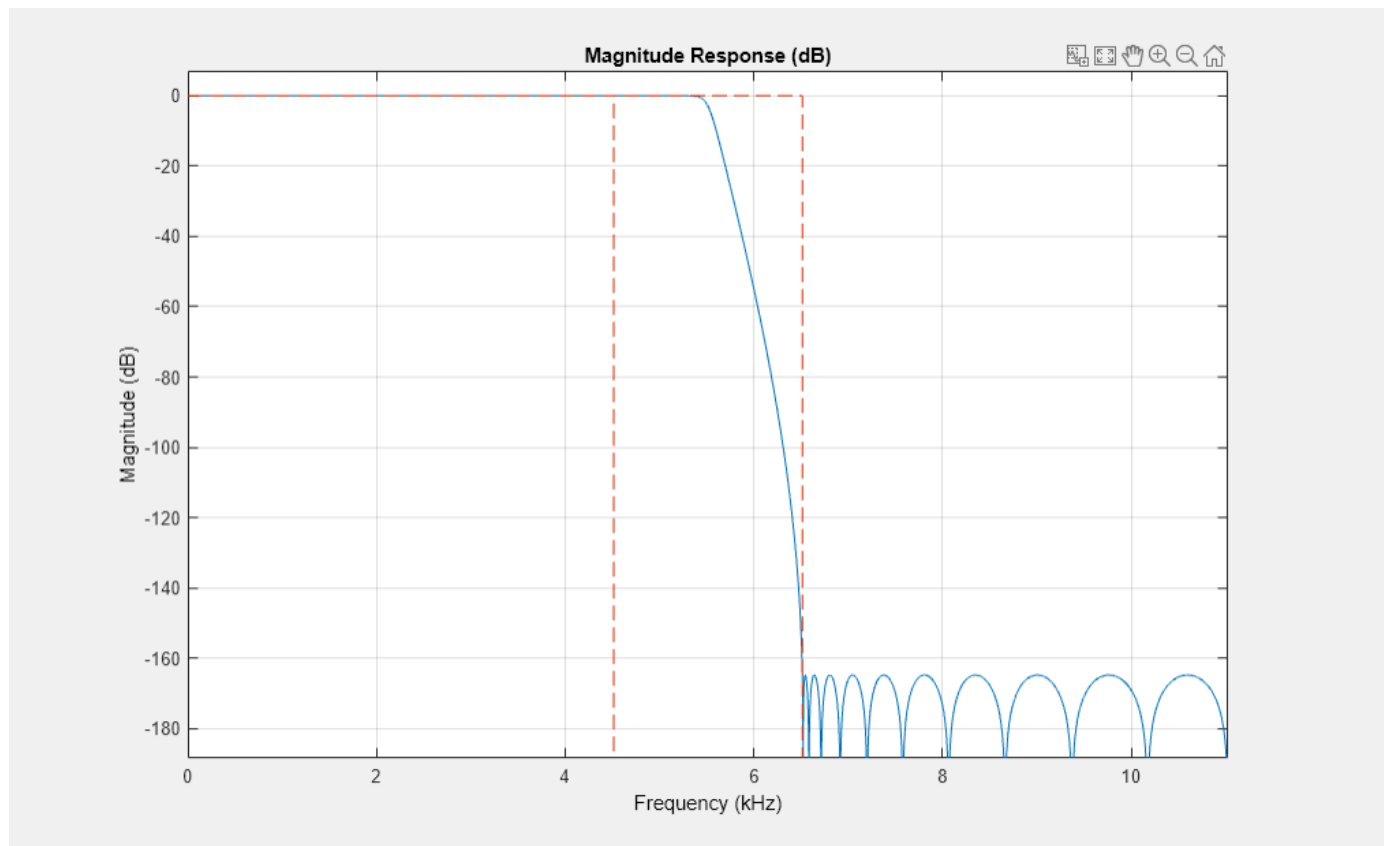
```
IIRHalfbandDecim = dsp.IIRHalfbandDecimator(...  
    'Specification',filterspec,'FilterOrder',Order,...  
    'TransitionWidth',TW,'SampleRate',afr.SampleRate);
```

```
IIRHalfbandInterp = dsp.IIRHalfbandInterpolator(...  
    'Specification',filterspec,'FilterOrder',Order,...  
    'TransitionWidth',TW,'SampleRate',afr.SampleRate/2);
```

```
ap = audioDeviceWriter('SampleRate',afr.SampleRate);
```

View the magnitude response of the halfband filter.

```
fvtool(IIRHalfbandDecim)
```



Read the speech signal from the audio file in frames of 1024 samples. Filter the speech signal into lowpass and highpass subbands with a halfband frequency of 5512.5 Hz. Reconstruct a lowpass approximation of the speech signal by interpolating the lowpass subband. Play the filtered output.

```
while ~isDone(afr)
    audioframe = afr();
    xlo = IIRHalfbandDecim(audioframe);
    ylow = IIRHalfbandInterp(xlo);
    ap(ylow);
end
```

Wait until the audio file ends, and then close the input file and release the audio output resource.

```
release(afr);
release(ap);
```

### Two-Channel Filter Bank

Use a halfband decimator and interpolator to implement a two-channel filter bank. This example uses an audio file input and shows that the power spectrum of the filter bank output does not differ significantly from the input.

**Note:** The `audioDeviceWriter` System object™ is not supported in MATLAB Online.



Set up the audio file reader and audio device writer. Construct the IIR halfband decimator and interpolator. Finally, set up the spectrum analyzer to display the power spectra of the filter-bank input and output.

```
AF = dsp.AudioFileReader('speech_dft.mp3','SamplesPerFrame',1024);
AP = audioDeviceWriter('SampleRate',AF.SampleRate);
```

```
filterspec = 'Filter order and transition width';
Order = 51;
TW = 2000;
```

```
IIRHalfbandDecim = dsp.IIRHalfbandDecimator(...
    'Specification',filterspec,'FilterOrder',Order,...
    'TransitionWidth',TW,'SampleRate',AF.SampleRate);
```

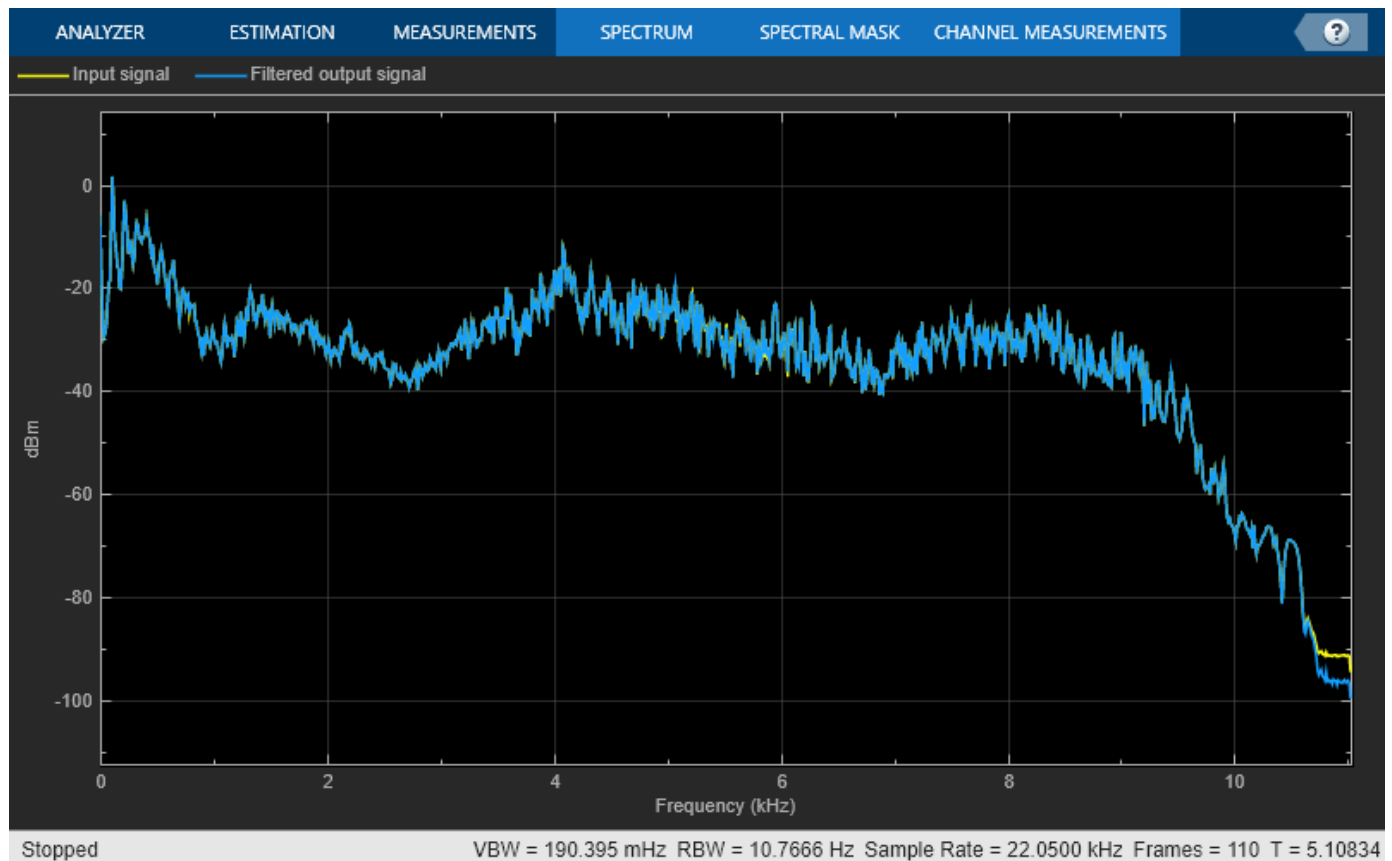
```
IIRHalfbandInterp = dsp.IIRHalfbandInterpolator(...
    'Specification',filterspec,'FilterOrder',Order,...
    'TransitionWidth',TW,'SampleRate',AF.SampleRate/2,...
    'FilterBankInputPort',true);
```

```
SpecAna = spectrumAnalyzer('SampleRate',AF.SampleRate,...
    'PlotAsTwoSidedSpectrum',false,...
    'ShowLegend',true,...
    'ChannelNames',{'Input signal','Filtered output signal'});
```

Read the audio 1024 samples at a time. Filter the input to obtain the lowpass and highpass subband signals decimated by a factor of two. This is the analysis filter bank. Use the halfband interpolator as the synthesis filter bank. Display the running power spectrum of the audio input and the output of the synthesis filter bank. Play the output.

```
while ~isDone(AF)
    audioInput = AF();
    [xlo,xhigh] = IIRHalfbandDecim(audioInput);
    audioOutput = IIRHalfbandInterp(xlo,xhigh);
    spectrumInput = [audioInput audioOutput];
    SpecAna(spectrumInput);
    AP(audioOutput);
end

release(AF);
release(AP);
release(SpecAna);
```



### Upsample and Interpolate Multichannel Input with IIR Halfband Interpolator

Create a half-band interpolation filter for data sampled at 44.1 kHz. The filter order is 51 with a transition width of 4.1 kHz. Use the filter to upsample and interpolate a multichannel input.

```

Fs = 44.1e3;
filterspec = 'Filter order and transition width';
Order = 51;
TW = 4.1e3;
iirhalfbandinterp = dsp.IIRHalfbandInterpolator(...
    'Specification',filterspec,...
    'FilterOrder',Order,...
    'TransitionWidth',TW,...
    'SampleRate',Fs);

x = randn(1024,4);
y = iirhalfbandinterp(x);

```

## Algorithms

### Polyphase Implementation with Halfband Filters

When you filter your signal, the IIR halfband interpolator uses an efficient polyphase implementation for halfband filters. You can use a polyphase implementation to move the upsampling operation after filtering. This change enables you to filter at a lower sampling rate.

IIR halfband filters are generally modeled using two parallel allpass filter branches.

$$H(z) = 0.5 * [A_1(z^2) + z^{-1}A_2(z^2)]$$

### Elliptic Design

The allpass filters for elliptic IIR halfband filter are given as

$$A_1(z) = \prod_{k=1}^{K_1} \frac{a_k^{(1)} + z^{-1}}{1 + a_k^{(1)}z^{-1}}$$

$$A_2(z) = \prod_{k=1}^{K_2} \frac{a_k^{(2)} + z^{-1}}{1 + a_k^{(2)}z^{-1}}$$

### Quasi-Linear Phase Design

A near-linear phase response for IIR halfband filters is achieved by making one of the branches a pure delay. In this design, the cost of the filter increases.

The allpass filters for quasi-linear phase IIR halfband filter are

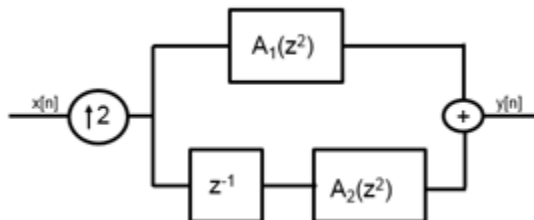
$$A_1(z) = z^{-k}$$

where,  $k$  is the length of the delay.

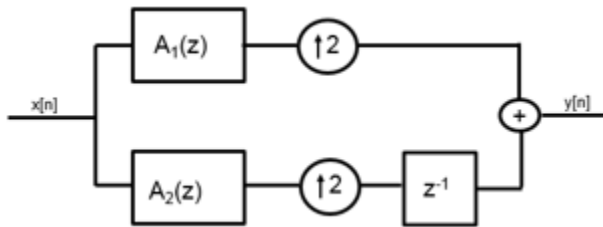
$$A_2(z) = \prod_{K=1}^{K_2^{(1)}} \frac{a_k + z^{-1}}{1 + a_k z^{-1}} \prod_{K=1}^{K_2^{(2)}} \frac{c_k + b_k z^{-1} + z^{-2}}{1 + b_k z^{-1} + c_k z^{-2}}$$

where  $N$  is the order of the IIR halfband filter.

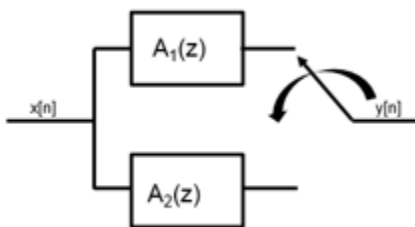
You can represent the upsampling-by-2 operation followed by the filtering operation using this figure.



Using the multirate noble identity for upsampling, you can move the upsampling operation after filtering. This enables you to filter at a lower rate.



To efficiently implement the halfband interpolator, this algorithm replaces the upsampling operator, delay block, and adder with a commutator switch. The commutator switch operates at twice the input sample rate. This is shown in the following figure.



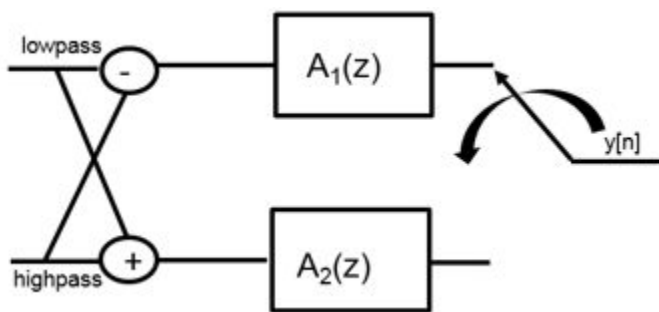
The commutator switch takes input samples from the two branches alternately, one sample at a time. This doubles the sampling rate of the input signal.

### Synthesis Filter Bank

Transfer function of the complementary high-pass filter branch of the synthesis filter bank is given by

$$G(z) = 0.5 * [A_1(z^2) - z^{-1}A_2(z^2)]$$

You can represent the synthesis filter bank as in this diagram.



The IIR halfband interpolator implements the synthesis portion of a two-band filter bank to synthesize a signal from lowpass and highpass subbands.

To summarize, the IIR halfband interpolator:

- Filters the input before upsampling
- Acts as a synthesis filter bank
- Has a nonlinear phase response and uses few coefficients with the elliptic design method

- Has near-linear phase response at the cost of additional coefficients with the quasi-linear phase design method, where one of the branches is a pure delay

## Version History

Introduced in R2015b

## References

- [1] Lang, M. *Allpass Filter Design and Applications*. IEEE Transactions on Signal Processing. Vol. 46, No. 9, Sept 1998, pp. 2505-2514.
- [2] Harris, F.J. *Multirate Signal Processing for Communication Systems*. Prentice Hall. 2004, pp. 208-209.
- [3] Regalia, Phillip A., Sanjit K. Mitra, and P. P. Vaidyanathan. "The Digital All-Pass Filter: A Versatile Signal Processing Building Block." *Proceedings of the IEEE*. Vol. 76, Number 1, 1988, pp. 19-37.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

This object supports code generation for ARM Cortex-M and ARM Cortex-A processors.

## See Also

### Functions

freqz | fvtool | info | cost | polyphase

### Objects

dsp.IIRHalfbandDecimator | dsp.FIRHalfbandDecimator |  
dsp.FIRHalfbandInterpolator

### Blocks

IIR Halfband Interpolator

### Topics

"Analysis Methods for Filter System Objects" on page 3-2

## dsp.Interpolator

**Package:** dsp

(Removed) Linear or polyphase FIR interpolation

---

**Note** `dsp.Interpolator` has been removed. Use `dsp.FIRInterpolator`, `interp1` instead. For more information, see “Compatibility Considerations”.

---

### Description

The `dsp.Interpolator` System object interpolates values between real-valued input samples using linear or polyphase FIR interpolation. Specify which values to interpolate by providing a vector of interpolation points. An interpolation point of 1 refers to the first sample in the input. To interpolate the value halfway between the second and third sample in the input, specify an interpolation point of 2.5. Interpolation points that are not within the valid range are replaced with the closest value in the valid range.

To interpolate a real-valued input signal:

- 1 Create the `dsp.Interpolator` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

### Creation

#### Syntax

```
interp = dsp.Interpolator  
interp = dsp.Interpolator(Name, Value)
```

#### Description

`interp = dsp.Interpolator` creates an interpolation System object, `interp`, to interpolate values between real-valued input samples using linear interpolation.

`interp = dsp.Interpolator(Name, Value)` creates an interpolation System object, `interp`, with each specified property set to the specified value. Enclose each property name in single quotes.

Example: `interp = dsp.Interpolator('InterpolationPointsSource', 'Input port')`

### Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

### InterpolationPointsSource — Source of interpolation points

'Property' (default) | 'Input port'

Method to specify the interpolation points, specified as one of the following:

- 'Property' — Specify the interpolation points through the “InterpolationPoints” on page 4-0 property.
- 'Input port' — Pass the interpolation points as an input to the System object algorithm.

### InterpolationPoints — Interpolation points

[1.1;4.8;2.67;1.6;3.2] (default) | vector | matrix |  $N$ -D array

Interpolation points, specified as a vector, matrix, or an  $N$ -D array. The valid range of the values in the interpolation vector is from 1 to the number of samples in each channel of the input. If you specify interpolation points outside the valid range, the object *clips* the point to the nearest point in the valid range. For example, if the input is [2;3.1;-2.1], the valid range of interpolation points is from 1 to 3. If you specify a [-1;1.5;2;2.5;3;3.5] vector of interpolation points, the interpolator object clips -1 to 1 and 3.5 to 3. This clipping results in the interpolation points [1 1.5 2 2.5 3 3].

For details on the dimension of the interpolation points array and how that influences the dimension of the output, see the tables in the “*ipts*” on page 4-0 input of the System object.

**Tunable:** Yes

### Dependencies

This property applies only when you set the “InterpolationPointsSource” on page 4-0 property to 'Property'.

### Method — Interpolation method

'Linear' (default) | 'FIR'

Interpolation method, specified as one of the following:

- 'Linear' -- The object interpolates data values by assuming that the data varies linearly between samples taken at adjacent sample times.
- 'FIR' -- The object uses polyphase interpolation to replace filtering (convolution) at the upsampled rate with a series of convolutions at the lower rate. If the input has insufficient low-rate samples to perform FIR interpolation, the interpolator object performs linear interpolation. For more details, see the “FilterHalfLength” on page 4-0 property.

### FilterHalfLength — Half-length of interpolation filter

3 (default) | integer scalar greater than 0

For a filter half-length of  $P$ , the polyphase FIR subfilters have length  $2P$ . FIR interpolation always requires  $2P$  low-rate samples for every interpolation point.

- If the interpolation point does not correspond to a low-rate sample, FIR interpolation requires  $P$  low-rate samples below and  $P$  low-rate samples above the interpolation point.
- If the interpolation point corresponds to a low-rate sample, the  $2P$ -sample requirement includes the low-rate sample.

- If the input has less than  $2P$  neighboring low-rate samples, the interpolator object uses linear interpolation.

For example, for an input `[1 4 1 4 1 4 1 4]`, upsampling by a factor of 4 results in equally spaced interpolation points, `InterP = [1:0.25:8]`. The points `InterP(9:12)` are `[3.0 3.25 3.5 3.75]`. If you set `FilterHalfLength` to 2, interpolating at these points uses the 4 low-rate samples from the input with indices `(2,3,4,5)`. If you set `FilterHalfLength` to 4, the interpolator object uses linear interpolation, because the input does not have enough low-rate samples to perform FIR interpolation.

The longer the `FilterHalfLength` property, the better the quality of the interpolation. However, increasing the filter half-length increases computation time and requires more low-rate samples below and above the interpolation point. In general, setting the `FilterHalfLength` property between 4 and 6 provides a reasonably accurate interpolation.

### Dependencies

This property applies only when you set the “Method” on page 4-0 property to 'FIR'.

### InterpolationPointsPerSample — Upsampling factor

3 (default) | integer scalar greater than 0

Upsampling factor, specified as an integer scalar greater than 0. An upsampling factor of  $L$  inserts  $L - 1$  zeros between low-rate samples. Interpolation results from filtering the upsampled sequence with a lowpass anti-imaging filter. The interpolator object uses a polyphase FIR implementation with `InterpolationPointsPerSample` subfilters of length  $2P$ , where  $P$  is the value you specify in the “FilterHalfLength” on page 4-0 property. For  $nL$  low-rate samples in the upsampled input, where  $n=1,2,\dots$ , the interpolator object uses exactly one of the `InterpolationPointsPerSample` subfilters to interpolate at the points  $nL+i/L$ , where  $i = 0, 1, 2, \dots, L - 1$ .

If you specify interpolation points that do not correspond to a polyphase subfilter, the object rounds the point down to the nearest interpolation point associated with a polyphase subfilter. Suppose you set the `InterpolationPointsPerSample` property to 4 and interpolate at the points `[3 3.2 3.4 3.6 3.8]`. The interpolator object uses the first polyphase subfilter for the points `[3.0 3.2]`, the second subfilter for the point 3.4, the third subfilter for the point 3.6, and the fourth subfilter for the point 3.8.

### Dependencies

This property applies only when you set the “Method” on page 4-0 property to 'FIR'.

### Bandwidth — Normalized input bandwidth

0.5 (default) | real scalar greater than 0 and less than or equal to 1

Bandwidth to which the interpolated output samples must be constrained, specified as a real scalar greater than 0 and less than or equal to 1. A value of 1 equals the Nyquist frequency, or half the sampling frequency,  $F_s$ . Use this property to take advantage of the bandlimited frequency content of the input. For example, if the input signal does not have frequency content above  $F_s/4$ , you can specify a value of 0.5 for the `Bandwidth` property.

### Dependencies

This property applies only when you set the “Method” on page 4-0 property to 'FIR'.



## Usage

### Syntax

```
interpOut = interp(input)
interpOut = interp(input, ipts)
```

### Description

`interpOut = interp(input)` outputs the interpolated sequence, `interpOut`, of the input vector or matrix `input`, as specified in the “InterpolationPoints” on page 4-0 property. Each column of `input` is treated as an independent channel of the input.

`interpOut = interp(input, ipts)` outputs the interpolated sequence as specified by `ipts`.

To specify the interpolation points, set the “InterpolationPointsSource” on page 4-0 property to 'Input port'.

```
t = 0:.0001:.0511;
x = sin(2*pi*20*t);
x1 = x(1:50:end);
ipts = 1:0.1:length(x1);
interp = dsp.Interpolator('InterpolationPointsSource', 'Input port');
interpOut = interp(x1', ipts');
```

### Input Arguments

#### **input** — Data input

vector | matrix | *N*-D array

Input that is interpolated by the object, specified as a vector, matrix, or *N*-D array.

Example: `t = 0:0.0001:0.0511; input = sin(2*pi*20*t);`

Data Types: `single` | `double`

#### **ipts** — Interpolation points

vector | matrix | *N*-D array

Interpolation array  $I_{pts}$ , specified as a vector, matrix, or *N*-D array. The interpolation array represents the points in time at which to interpolate values of the input signal. An entry of 1 in  $I_{pts}$  refers to the first sample of the input, an entry of 2.5 refers to the sample halfway between the second and third input sample, and so on. In most cases, when  $I_{pts}$  is a vector, it can be of any length.

Valid values in the interpolation array  $I_{pts}$  range from 1 to the number of samples in each channel of the input. For instance, given a length-5 input vector  $D$ , all entries of  $I_{pts}$  must range from 1 to 5.  $I_{pts}$  cannot contain entries such as 7 or -9, because  $D$  does not have a seventh or ninth entry.

The algorithm replaces any out-of-range values in  $I_{pts}$  with the closest value in the valid range (from 1 to the number of input samples). Then it performs the interpolation using the clipped version of  $I_{pts}$ .

Consider the following input data and interpolation points vector:

- `D = [11 22 33 44]'`

- $I_{Pts} = [10 \ 2.6 \ -3]'$

Because D has four samples, valid interpolation points range from 1 to 4. The algorithm clips interpolation point 10 down to 4 and the point -3 up to 1. The result is the clipped interpolation vector  $I_{PtsClipped} = [4 \ 2.6 \ 1]'$ .

Depending on the dimension of the input and the dimension of  $I_{Pts}$ , the algorithm applies  $I_{Pts}$  to the input in one of the following ways:

- If  $I_{Pts}$  is an array, the object applies  $I_{Pts}$  across the first dimension of an  $N$ -D array, resulting in an  $N$ -D array output.
- If  $I_{Pts}$  is a vector, the object applies  $I_{Pts}$  to each input vector (as if the input vector were a single channel), resulting in a vector output with the same orientation as the input (row or column).

The following tables summarize how the object applies the interpolation array  $I_{Pts}$  to all the possible types of inputs. The table also shows the resulting output dimensions.

This table describes the behavior when InterpolationPointsSource is set to 'Property'.

| Input Dimensions                   | Valid Dimensions of Interpolation Array $I_{Pts}$             | How Object Applies $I_{Pts}$ to Input  | Output Dimensions   |
|------------------------------------|---|--|---|
| $M$ -by- $N$ -by- $K$ matrix       | $P$ -by-1 column  | Applies $I_{Pts}$ to the first dimension of the input                                    | $P$ -by- $N$ -by- $K$ array                                       |
|                                    | $P$ -by- $N$ -by- $K$ matrix                                  | Applies each column element of $I_{Pts}$ to the corresponding column of the input matrix | $P$ -by- $N$ -by- $K$ array                                       |
| $M$ -by- $N$ matrix                | 1-by- $N$ row   | Applies each column element of $I_{Pts}$ to the corresponding column of the input matrix | 1-by- $N$ row   |
|                                    | $P$ -by-1 column  | Applies $I_{Pts}$ to each input column   | $P$ -by- $N$ matrix   |
|                                    | $P$ -by- $N$ matrix   | Applies the columns of $I_{Pts}$ to the corresponding columns of the input matrix        |   |
| $M$ -by-1 column                   | 1-by- $P$ row<br>(the algorithm treats $I_{Pts}$ as a column) | Applies $I_{Pts}$ to the input column  | $P$ -by-1 column  |
|                                    | $P$ -by-1 column  |  |   |
| 1-by- $N$ row<br>(not recommended) | 1-by- $N$ row   | Not applicable. Object copies input vector.  | 1-by- $N$ row, a copy of the input vector                         |
|                                    | $P$ -by-1 column  |  | $P$ -by- $N$ matrix, where each row is a copy of the input vector |
|                                    | $P$ -by- $N$ matrix   |  |   |

This table describes the behavior when InterpolationPointsSource is set to 'Input port'.

| Input Dimensions                   | Valid Dimensions of Interpolation Array $I_{pts}$ | How Object Applies $I_{pts}$ to Input  | Output Dimensions   |
|------------------------------------|---|--|---|
| $M$ -by- $N$ -by- $K$ matrix       | Column vector of length $P$                       | Applies $I_{pts}$ to the first dimension of the input                                    | $P$ -by- $N$ -by- $K$ array                                       |
|                                    | $P$ -by- $N$ -by- $K$ matrix                      | Applies each column element of $I_{pts}$ to the corresponding column of the input matrix | $P$ -by- $N$ -by- $K$ array                                       |
| $M$ -by- $N$ matrix                | 1-by- $N$ row                                     | Applies each column element of $I_{pts}$ to the corresponding column of the input matrix | 1-by- $N$ row   |
|                                    | $P$ -by-1 column                                  | Applies $I_{pts}$ to each input column   | $P$ -by- $N$ matrix   |
|                                    | $P$ -by- $N$ matrix                               | Applies the columns of $I_{pts}$ to the corresponding columns of the input matrix        |   |
| $M$ -by-1 column                   | 1-by- $P$ row                                     | Applies $I_{pts}$ to the input column  | $P$ -by-1 column  |
|                                    | $P$ -by-1 column                                  |  |   |
| 1-by- $N$ row<br>(not recommended) | 1-by- $N$ row                                     | Not applicable. Object copies input vector.  | 1-by- $N$ row, a copy of the input vector                         |
|                                    | $P$ -by-1 column                                  |  | $P$ -by- $N$ matrix, where each row is a copy of the input vector |
|                                    | $P$ -by- $N$ matrix                               |  |   |

Example: `ipts = [1:10];`

Data Types: `single` | `double`

## Output Arguments

### **interpOut** — Interpolated sequence

vector | matrix |  $N$ -D array

Interpolated sequence, returned as a vector, matrix, or  $N$ -D array. The dimension of the output depends on the dimensions of the input and the interpolation points array. For more details on the dimensions, see the tables in “`ipts`” on page 4-0 .

Data Types: `single` | `double`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Common to All System Objects

|         |  |
|---------|--|
| step    | Run System object algorithm  |
| release | Release resources and allow changes to System object property values and input characteristics |
| reset   | Reset internal states of System object   |

## Examples

### Interpolate a Sum of Sinusoids

Interpolate a sum of sinusoids with FIR interpolation, and with 'Input port' as the source of interpolation points.

```

Fs = 1000;
t = 0:(1/Fs):0.1-(1/Fs);
x = cos(2*pi*50*t)+0.5*sin(2*pi*100*t);
x1 = x(1:4:end);
I = 1:(1/4):length(x1);
interp = dsp.Interpolator('Method','FIR',...
'FilterHalfLength',3,'InterpolationPointsSource','Input Port');
y = interp(x1,I);
stem(I,y,'r');
hold on;
axis([0 25 -2 2]);
stem(x1,'b','linewidth',2);
legend('Interpolated Signal','Original',...
'Location','Northeast');

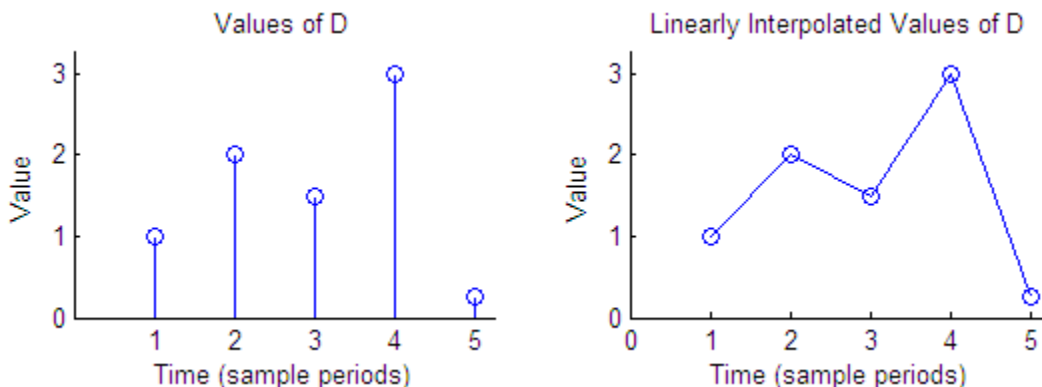
```

## Algorithms

### Linear Interpolation Mode

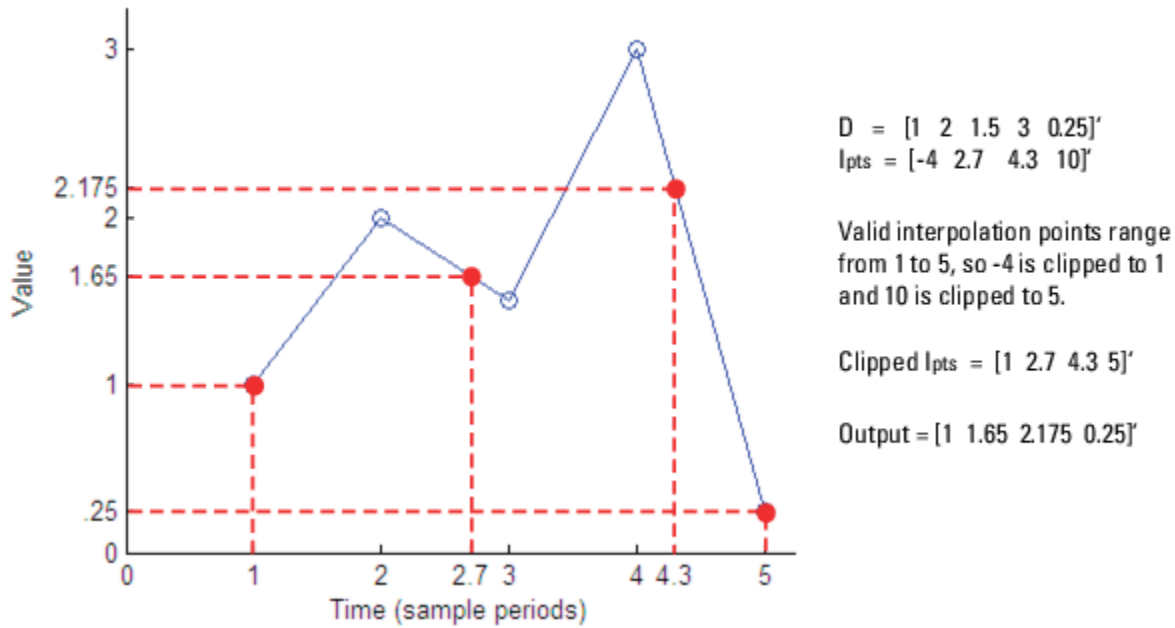
In the linear interpolation mode, the algorithm interpolates data values by assuming that the data varies linearly between samples taken at adjacent sample times.

Suppose the input signal is  $D = [1 \ 2 \ 1.5 \ 3 \ 0.25]'$ . The left plot shows the samples in D and the right plot shows the linearly interpolated values between the samples in D.



When the interpolation points are out of range, the algorithm clips the invalid interpolation points. Consider an input signal,  $D = [1 \ 2 \ 1.5 \ 3 \ 0.25]'$ , and an interpolation vector,  $I_{pts} = [-4 \ 2.7 \ 4.3 \ 10]'$ . The interpolated output is given by  $[1 \ 1.65 \ 2.175 \ 0.25]'$ .

### Interpolated Values of D at Clipped Interpolation Points



### FIR Interpolation Mode

In the FIR interpolation mode, the algorithm interpolates data values using an FIR interpolation filter. The FIR filter is implemented using a polyphase structure. A polyphase implementation *splits* the lowpass FIR filter impulse response into a number of different subfilters.

Let  $L$  represent the number of interpolation points per sample, or the upsampling factor. Let  $P$  represent the half length of the polyphase subfilters. Indexing from zero, if  $h(n)$  is the impulse response of the FIR filter, the  $k$ th subfilter is:

$$h_k(n) = h(k + nL) \quad k = 0, 1, \dots, L - 1 \quad n = 0, 1, \dots, 2P - 1$$

The table describes the decomposition of an 18-coefficient FIR filter into 3 polyphase subfilters of length 6, the defaults for the FIR interpolator object.

| Coefficients                     | Polyphase Subfilter |
|----------------------------------|---------------------|
| $h(0), h(3), h(6), \dots, h(15)$ | $h_0(n)$            |
| $h(1), h(4), h(7), \dots, h(16)$ | $h_1(n)$            |
| $h(2), h(5), h(8), \dots, h(17)$ | $h_2(n)$            |

An upsampling factor of  $L$  inserts  $L - 1$  zeros between low-rate samples. Interpolation results from filtering the upsampled sequence with a lowpass anti-imaging filter.

## Version History

### Introduced in R2012a

#### R2021b: `dsp.Interpolator` System object will be removed

*Warns starting in R2021b*

`dsp.Interpolator` System object will be removed in a future release. Use `dsp.FIRInterpolator`, `interp1` instead.

#### R2023a: `dsp.Interpolator` System object has been removed

*Errors starting in R2023a*

The `dsp.Interpolator` System object has been removed. Use `dsp.FIRInterpolator`, `interp1` instead.

### Update Code

This table shows how to update existing code to use the replacement features.

| Discouraged Usage   | Replacement using <code>dsp.FIRInterpolator</code>   |
|---|--|
| <p><b>Linear interpolation</b></p> <p>Create the input signal.</p> <pre>n = 0:11; x = sin(2*pi*n/length(n));</pre> <p>Create the <code>dsp.Interpolator</code> object and pass the input signal to the object.</p> <pre>L = 10; t = (n(1):1/L:n(end)+1-1/L)+1/L; interpolator = dsp.Interpolator(Interpolator); y = interpolator(x');</pre> | <p><b>Linear interpolation</b></p> <p>Create a triangle interpolation filter. This filter yields a linear interpolation template. Apply this filter to the input signal.</p> <pre>hLin = triang(2*L-1); firInterpolator = dsp.FIRInterpolator(L,hLin); yf = firInterpolator(x');</pre> <p>Plot and compare the interpolated outputs.</p> <pre>subplot(2,1,1); plot(t,x); hold on; scatter(t,yf,'+'); hold off; legend('dsp.Interpolator','dsp.FIRInterpolator');</pre> |

| Discouraged Usage   | Replacement using dsp.FIRInterpolator  |
|---|--|
| <p><b>FIR interpolation</b></p> <p>Create the input signal.</p> <pre>n = 1:60; x = sin(10*pi*n/length(n))';</pre> <p>Create the dsp.Interpolator object and pass the input signal to the object.</p> <pre>L = 5; P = 6; t = 1:1/L:length(x); intr = dsp.Interpolator(Method='FIR',...     FilterHalfLength=P,...     InterpolationPoints=t',...     InterpolationPointsPerSample=L); y_int = intr(x);</pre>   | <p><b>FIR interpolation</b></p> <p>Determine the FIR coefficients of the interpolator to use with dsp.FIRInterpolator.</p> <pre>FIRcoeffs = intfilt(L,P,0.5); hFIR = [zeros(2*L-mod(length(FIRcoeffs),2*L),1); FIRcoeffs(:)];</pre> <p>Create a dsp.FIRInterpolator object and pass the input signal to the object.</p> <pre>firintr = dsp.FIRInterpolator(L,hFIR); y_firint = firintr(x);</pre> <p>Plot and compare the input signal and the interpolated outputs.</p> <pre>i0 = floor(length(hFIR)/2); % Latency tfir = (1:1/L:length(x)+1-1/L) - i0/L;</pre> <pre>figure; hold on;</pre> <pre>scatter(n,x,50,'filled','Dk') scatter(t,y_int,50,'red'); scatter(tfir,y_firint,50,'bx');</pre> <pre>legend('Low rate input','dsp.Interpolator',... 'dsp.FIRInterpolator','Location','Northeast');</pre> |
| <p><b>Discouraged Usage</b></p> <p>Create the input signal.</p> <pre>n = (1:60); x = sin(10*pi*n/length(n))';</pre> <p>For uniformly spaced interpolation points, use this t vector.</p> <pre>t = 1:1/10:60;</pre> <p>For nonuniformly spaced interpolation points, use this t vector instead.</p> <pre>t = sort(n(1)+(n(end)-n(1))*rand(1,500));</pre> <p>Create the dsp.Interpolator object and pass the input signal to the object.</p> <pre>intr = dsp.Interpolator(Method='Linear',InterpolationPoints=t'); y_int = intr(x);</pre> | <p><b>Replacement using interp1</b></p> <p>Pass the input signal to the interp1 function.</p> <pre>y_interp1 = interp1(n,x,t);</pre> <p>Plot and compare the input signal and the interpolated outputs.</p> <pre>figure; hold on;</pre> <pre>scatter(1+(0:length(x)-1),x,50,'filled','Dk') scatter(t,y_int,50,'red'); scatter(t,y_interp1,50,'blue','x');</pre> <pre>legend('Low rate input','dsp.Interpolator',... 'interp1','Location','Northeast');</pre>   |

### Extended Capabilities

#### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

### See Also

#### Objects

`dsp.FIRInterpolator` | `dsp.CICCompensationInterpolator` | `dsp.CICInterpolator` | `dsp.FIRHalfbandInterpolator` | `dsp.IIRHalfbandInterpolator` | `dsp.VariableFractionalDelay`

#### Blocks

CIC Compensation Interpolator | CIC Interpolation | FIR Halfband Interpolator | IIR Halfband Interpolator | Interpolation



# dsp.KalmanFilter

**Package:** dsp

(Removed) Estimate system measurements and states using Kalman filter

---

**Note** dsp.KalmanFilter has been removed. Use the Kalman filter functionality in Sensor Fusion and Tracking Toolbox™ instead.

---

## Description

The dsp.KalmanFilter System object is an estimator used to recursively obtain a solution for linear optimal filtering. This estimation is made without precise knowledge of the underlying dynamic system. The Kalman filter implements the following linear discrete-time process with state,  $x$ , at the  $k^{\text{th}}$  time-step:  $x(k) = Ax(k-1) + Bu(k-1) + w(k-1)$  (state equation). This measurement,  $z$ , is given as:  $z(k) = Hx(k) + v(k)$  (measurement equation).

The Kalman filter algorithm computes the following two steps recursively:

- Prediction: Process parameters  $x$  (state) and  $P$  (state error covariance) are estimated using the previous state.
- Correction: The state and error covariance are corrected using the current measurement.

To filter each channel of the input:

- 1 Create the dsp.KalmanFilter object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
kalman = dsp.KalmanFilter
kalman = dsp.KalmanFilter(STMatrix, MMatrix, PNCovariance, MNCovariance,
CIMatrix)
kalman = dsp.KalmanFilter(Name,Value)
```

### Description

kalman = dsp.KalmanFilter returns the Kalman filter System object, kalman, with default values for the parameters.

kalman = dsp.KalmanFilter(STMatrix, MMatrix, PNCovariance, MNCovariance, CIMatrix) returns a Kalman filter System object, kalman. The StateTransitionMatrix property is set to STMatrix, the MeasurementMatrix property is set to MMatrix, the ProcessNoiseCovariance property is set to PNCovariance, the

MeasurementNoiseCovariance property is set to MNCovariance, and the ControlInputMatrix property is set to CIMatrix.

`kalman = dsp.KalmanFilter(Name,Value)` returns an Kalman filter System object, `kalman`, with each property set to the specified value. Enclose each property name in single quotes. Unspecified properties have default values.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

### StateTransitionMatrix — Model of state transition

1 (default) | scalar | square matrix

Specify  $A$  in the state equation that relates the state at the previous time step to the state at current time step.  $A$  is a square matrix with each dimension equal to the number of states.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### ControlInputMatrix — Model of relation between control input and states

1 (default) | column vector

Specify  $B$  in the state equation that relates the control input to the state.  $B$  is a column vector with a number of rows equal to the number of states.

### Dependencies

This property is activated only when the `ControlInputPort` property value is `true`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### MeasurementMatrix — Model of relation between states and measurement output

1 (default) | row vector

Specify  $H$  in the measurement equation that relates the states to the measurements.  $H$  is a row vector with a number of columns equal to the number of measurements.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### ProcessNoiseCovariance — Covariance of process noise

0.1 (default) | scalar | square matrix

Specify  $Q$  as a square matrix with each dimension equal to the number of states.  $Q$  is the covariance of the white Gaussian process noise,  $w$ , in the state equation.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### MeasurementNoiseCovariance — Covariance of measurement noise

0.1 (default) | scalar | square matrix

Specify  $R$  as a square matrix with each dimension equal to the number of states.  $R$  is the covariance of the white Gaussian process noise,  $v$ , in the measurement equation.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

#### **InitialStateEstimate — Initial value for states**

0 (default) | scalar | column vector

Specify an initial estimate of the states of the model as a column vector with length equal to the number of states.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

#### **InitialErrorCovarianceEstimate — Initial value for state error covariance**

0.1 (default) | scalar | square matrix

Specify an initial estimate for covariance of the state error, as a square matrix with each dimension equal to the number of states.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

#### **DisableCorrection — Disable port for filters**

false (default) | true

Specify as a scalar logical value, disabling System object filters from performing the correction step after the prediction step in the Kalman filter algorithm.

#### **ControlInputPort — Presence of a control input**

true (default) | false

Specify if the control input is present, using a scalar logical value. The default value is true.

## Usage

### Syntax

```
[zEst, xEst, MSE_Est, zPred, xPred, MSE_Pred] = kalman(z,u)
```

### Description

`[zEst, xEst, MSE_Est, zPred, xPred, MSE_Pred] = kalman(z,u)` carries out the iterative Kalman filter algorithm over measurements  $z$  and control inputs  $u$ . The columns in  $z$  and  $u$  are treated as inputs to separate parallel filters, whose correction (or update) step can be disabled by the `DisableCorrection` property. The values returned are estimated measurements `zEst`, estimated states `xEst`, MSE of estimated states `MSE_Est`, predicted measurements `zPred`, predicted states `xPred`, and MSE of predicted states `MSE_Pred`.

### Input Arguments

#### **z — Measurement input**

vector | matrix

Measurement input, specified as a vector or a matrix.

The ratio of the number of rows of the measurement input to the number of rows of the `MeasurementMatrix` property must be equal to the ratio of the number of rows of the control input to the number of columns of the `ControlInputMatrix` property.

The measurement signal can be a variable-size input. Once the object is locked, you can change the size of each input channel, but the number of channels cannot change.

Data Types: `single` | `double`

### **u — Control input**

vector | matrix

Control input, specified as a vector or a matrix.

The ratio of the number of rows of the control input to the number of columns of the `ControlInputMatrix` property must be equal to the ratio of the number of rows of the measurement input to the number of rows of the `MeasurementMatrix` property.

The control signal can be a variable-size input. Once the object is locked, you can change the size of each input channel, but the number of channels cannot change.

Data Types: `single` | `double`

### **Output Arguments**

#### **zEst — Estimated measurements**

vector | matrix

Estimated measurements, returned as a vector or matrix.

Data Types: `single` | `double`

#### **xEst — Estimated state**

vector | matrix

Estimated state, returned as a vector or matrix.

Data Types: `single` | `double`

#### **MSE\_Est — MSE of estimated states**

scalar | column vector

Mean-squared error of estimated states, returned as a scalar or column vector. If the input is a row vector, the MSE of the estimated states is a scalar.

Data Types: `single` | `double`

#### **zPred — Predicted measurements**

vector | matrix

Predicted measurements, returned as a vector or a matrix.

Data Types: `single` | `double`

#### **xPred — Predicted states**

vector | matrix

Predicted states, returned as a vector or a matrix.

Data Types: `single` | `double`

### MSE\_Pred — MSE of predicted states

scalar | column vector

Mean-squared error of predicted states, returned as a scalar or a column vector. If the input is a row vector, the MSE of the estimated states is a scalar.

Data Types: `single` | `double`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Common to All System Objects

`step`     Run System object algorithm  
`release`   Release resources and allow changes to System object property values and input characteristics  
`reset`     Reset internal states of System object

## Examples

### Estimate Changing Scalar

Create the System objects for the changing scalar input, the Kalman filter, and the scope (for plotting).

```
numSamples = 4000;
R = 0.02;
src = dsp.SignalSource;
src.Signal = [ones(numSamples/4,1); -3*ones(numSamples/4,1); ...
    4*ones(numSamples/4,1); -0.5*ones(numSamples/4,1)];
tScope = timescope('NumInputPorts',3,...
    'TimeSpanSource','Property','TimeSpan',numSamples, ...
    'TimeUnits','Seconds','YLimits',[-5 5], ...
    'ShowLegend',true); % Create the Time Scope
kalman = dsp.KalmanFilter('ProcessNoiseCovariance', 0.0001,...
    'MeasurementNoiseCovariance',R,...
    'InitialStateEstimate',5,...
    'InitialErrorCovarianceEstimate',1,...
    'ControlInputPort',false); %Create Kalman filter
```

Add noise to the scalar, and pass the result to the Kalman filter. Stream the data, and plot the filtered signal.

```
while(~isDone(src))
    trueVal = src();
    noisyVal = trueVal + sqrt(R)*randn;
    estVal = kalman(noisyVal);
```

```
tScope(noisyVal,trueVal,estVal);  
end
```

### Algorithms

This object implements the algorithm, inputs, and outputs described on the Kalman Filter block reference page. The object properties correspond to the block parameters.

### Version History

#### Introduced in R2013b

#### **R2021b: dsp.KalmanFilter System object will be removed**

*Warns starting in R2021b*

`dsp.KalmanFilter` System object will be removed in a future release. Use the Kalman filter functionality in the Sensor Fusion and Tracking Toolbox instead.

#### **R2023a: dsp.KalmanFilter System object has been removed**

*Errors starting in R2023a*

The `dsp.KalmanFilter` System object has been removed. Use the Kalman filter functionality in Sensor Fusion and Tracking Toolbox instead.

### References

- [1] Greg Welch and Gary Bishop, *An Introduction to the Kalman Filter*, Technical Report TR95 041. University of North Carolina at Chapel Hill: Chapel Hill, NC., 1995.

### Extended Capabilities

#### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

### See Also

#### **Blocks**

Kalman Filter

# dsp.LDLFactor

**Package:** dsp

(Removed) Factor square Hermitian positive definite matrices into components

---

**Note** dsp.LDLFactor has been removed. Use `ldl` instead. For more information, see “Compatibility Considerations”.

---

## Description

The `LDLFactor` object factors square Hermitian positive definite matrices into lower, upper, and diagonal components. The object uses only the lower triangle of  $S$ .

To factor these matrices into lower, upper, and diagonal components:

- 1 Define and set up your LDL factor object. See “Construction” on page 4-713.
- 2 Call `step` to factor the matrices according to the properties of `dsp.LDLFactor`. The behavior of `step` is specific to each object in the toolbox.

---

**Note** Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj, x)` and `y = obj(x)` perform equivalent operations.

---

## Construction

`ldl = dsp.LDLFactor` returns an LDL factor System object, `ldl`, that computes unit lower triangular  $L$  and diagonal  $D$  such that  $S = LDL$  for square, symmetric/Hermitian, positive definite input matrix  $S$ .

`ldl = dsp.LDLFactor('PropertyName', PropertyValue, ...)` returns an LDL factor System object, `ldl`, with each specified property set to the specified value.

## Properties

### Fixed-Point Properties

#### AccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as `Full precision`, `Same as input`, `Same as product` or `Custom`. The default is `Full precision`

#### CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `AccumulatorDataType` property to `Custom`. The default is `numericType([], 32, 30)`.

### **CustomIntermediateProductDataType**

Intermediate product word and fraction lengths

Specify the intermediate product fixed-point type as a signed, scaled `numericType` object. This property applies when you set the `IntermediateProductDataType` property to `Custom`. The default is `numericType(true, 16, 15)`.

### **CustomOutputDataType**

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `OutputDataType` property to `Custom`. The default is `numericType([], 16, 15)`.

### **CustomProductDataType**

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `ProductDataType` property to `Custom`. The default is `numericType([], 32, 30)`.

### **IntermediateProductDataType**

Intermediate product word and fraction lengths

Specify the intermediate product fixed-point data type as `Same as input` or `Custom`. The default is `Same as input`.

### **OutputDataType**

Output word and fraction lengths

Specify the output fixed-point data type as `Same as input` or `Custom`. The default is `Same as input`.

### **OverflowAction**

Overflow action for fixed-point operations

Specify the overflow action as one of `Wrap` or `Saturate`. The default is `Wrap`.

### **ProductDataType**



Product word and fraction lengths

Specify the product fixed-point data type as `Full precision`, `Same as input` or `Custom`. The default is `Full precision`.

### RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as: `Ceiling`, `Convergent`, `Floor`, `Nearest`, `Round`, `Simplest` or `Zero`. The default is `Floor`.

## Methods

`step`      Decompose matrix into components

| Common to All System Objects |  |
|------------------------------|--|
| <code>release</code>         | Allow System object property value changes |

## Examples

### Decompose a Matrix

**Note** If you are using R2016a or an earlier release, replace each call to the object with the equivalent step syntax. For example, `obj(x)` becomes `step(obj,x)`.

Decompose a square Hermitian positive definite matrix using LDL factor.

```
A = gallery('randcorr',5);
ldl = dsp.LDLFactor;
y = ldl(A)
```

```
y = 5x5
 1.0000  -0.3884   0.0296  -0.6152  -0.3903
-0.3884   0.8491  -0.6386  -0.4901   0.3137
 0.0296  -0.6386   0.6529   0.3611   0.5770
-0.6152  -0.4901   0.3611   0.3324   0.5660
-0.3903   0.3137   0.5770   0.5660   0.4403
```

## Algorithms

This object implements the algorithm, inputs, and outputs described on the LDL Factorization block reference page. The object properties correspond to the block parameters, except: No object property that corresponds to the **Non-positive definite input** block parameter. The object does not issue any alerts for nonpositive definite inputs. The output is not a valid factorization. A partial factorization is in the upper left corner of the output.

## Version History

Introduced in R2012a

### R2021b: dsp.LDLFactor System object will be removed

Warns starting in R2021b

`dsp.LDLFactor` System object will be removed in a future release. Use the `ldl` function instead.

### R2023a: dsp.LDLFactor System object has been removed

Errors starting in R2023a

The `dsp.LDLFactor` System object has been removed. Use the `ldl` function instead.

### Update Code

This table shows how to update existing code to use the `ldl` function.

| Discouraged Usage   | Recommended Replacement   |
|---|---|
| <p>The output <code>y</code> of the object is a composite matrix with <math>L</math> as its lower triangular portion, <math>D</math> as the diagonal, and <math>L'</math> as its upper triangular portion.</p> <pre>A = gallery('randcorr',5); ldlObj = dsp.LDLFactor; y = ldlObj(A)</pre> <p><math>y = 5 \times 5</math></p> <pre>1.0000 -0.3884 0.0296 -0.6152 -0.3903 -0.3884 0.8491 -0.6386 -0.4901 0.3107 0.0296 -0.6386 0.6529 0.3611 0.5770 -0.6152 -0.4901 0.3611 0.3324 0.5660 -0.3903 0.3137 0.5770 0.5660 0.4403</pre> | <p>The function outputs the lower triangular matrix <math>L</math> and the diagonal <math>D</math> separately.</p> <pre>[L,D,P] = ldl(A)</pre> <p><math>L = 5 \times 5</math></p> <pre>1.0000 0 0 0 0 -0.3884 1.0000 0 0 0 0.0296 -0.6386 1.0000 0 0 -0.3903 0.3137 -0.4901 0.3611 1.0000 0.5660 0.5770 0.3324 0.3611 0.3137</pre> <p><math>D = 5 \times 5</math></p> <pre>0.4403 0 0 0 0 0 0.8491 0 0 0 0 0 0.6529 0 0 0 0 0 0.3324 0 0 0 0 0 0.4403</pre> <p><math>P = 5 \times 5</math></p> <pre>1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1</pre> |

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

## See Also

ldl

## step

**System object:** dsp.LDLFactor

**Package:** dsp

Decompose matrix into components

### Syntax

```
Y = step(ldl,S)
```

### Description

---

**Note** Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

---

`Y = step(ldl,S)` decomposes the matrix `S` into lower, upper, and diagonal components. The output `Y` is a composite matrix with the `L` as its lower triangular part and `D` as the diagonal and `L'` as its upper triangular part. If `S` is not positive definite the output `Y` is not a valid factorization.

---

**Note** `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

# dsp.LevinsonSolver

**Package:** dsp

(Removed) Solve linear system of equations using Levinson-Durbin recursion

---

**Note** `dsp.LevinsonSolver` has been removed. Use `levinson` instead. For more information, see “Compatibility Considerations”.

---

## Description

The `LevinsonSolver` object solves linear systems of equations using Levinson-Durbin recursion.

To solve linear systems of equations using Levinson-Durbin recursion:

- 1 Define and set up your System object. See “Construction” on page 4-719.
- 2 Call `step` to solve the system of equations according to the properties of `dsp.LevinsonSolver`. The behavior of `step` is specific to each object in the toolbox.

---

**Note** Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

---

## Construction

`levinson = dsp.LevinsonSolver` returns a System object, `levinson`, that solves a Hermitian Toeplitz system of equations using the Levinson-Durbin recursion.

`levinson = dsp.LevinsonSolver('PropertyName',PropertyValue,...)` returns a Levinson-Durbin object, `levinson`, with each specified property set to the specified value.

## Properties

### AOutputPort

Enable polynomial coefficients output

Set this property to `true` to output the polynomial coefficients *A*. Both `AOutputPort` and `KOutputPort` properties cannot be `false` at the same time. For scalar inputs, set the `AOutputPort` property to `true`. The default is `false`.

### KOutputPort

Enable reflection coefficients output

Set this property to `true` to output the reflection coefficients  $K$ . You cannot set both the `AOutputPort` and `KOutputPort` properties to `false` at the same time. For scalar inputs, you must set the `KOutputPort` property to `false`. The default is `true`.

### **PredictionErrorOutputPort**

Enable prediction error output

Set this property to `true` to output the prediction error. The default is `false`.

### **ZerothLagZeroAction**

Action when value of lag zero is zero

Specify the output for an input with the first coefficient as zero. Select `Ignore` or `Use zeros`. The default is `Use zeros`.

### **Fixed-Point Properties**

#### **RoundingMethod**

Rounding method for fixed-point operations

Specify the rounding method as `Ceiling`, `Convergent`, `Floor`, `Nearest`, `Round`, `Simplest`, or `Zero`. The default is `Floor`.

#### **OverflowAction**

Overflow action for fixed-point operations

Specify the overflow action as `Wrap`, `Saturate`. The default is `Wrap`.

#### **ACoefficientDataType**

A coefficient word and fraction lengths

This constant property has a value of `Custom`.

#### **CustomACoefficientDataType**

A coefficient word and fraction lengths

Specify the A coefficient fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. The default is `numericType([], 16, 15)`.

#### **KCoefficientDataType**

K coefficient word and fraction lengths

This constant property has a value of Custom.

### **CustomKCoefficientDataType**

*K* coefficient word and fraction lengths

Specify the *K* coefficient fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. The default is `numericType([], 16, 15)`.

### **PredictionErrorDataType**

Prediction error power word and fraction lengths

Specify the prediction error power fixed-point data type as `Same as input` or `Custom`. The default is `Same as input`.

### **CustomPredictionErrorDataType**

Prediction error power word and fraction lengths

Specify the prediction error power fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies only when the `PredictionErrorDataType` property is `Custom`. The default is `numericType([], 16, 15)`.

### **ProductDataType**

Product word and fraction lengths

Specify the product fixed-point data type as `Same as input` or `Custom`. The default is `Custom`

### **CustomProductDataType**

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies only when the `ProductDataType` property is `Custom`. The default is `numericType([], 32, 30)`.

### **AccumulatorDataType**

Accumulator word and fraction lengths

Specify the Accumulator fixed-point data type as `Same as input`, `Same as product`, or `Custom`. The default is `Custom`.

### **CustomAccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies only when the `AccumulatorDataType` property is `Custom`. The default is `numericType([], 32, 30)`.

## Methods

`step` Reflection coefficients corresponding to columns of input

### Common to All System Objects

|                      |  |
|----------------------|--|
| <code>release</code> | Allow System object property value changes |
|----------------------|--|

## Examples

### Compute Polynomial Coefficients

---

**Note** If you are using R2016a or an earlier release, replace each call to the object with the equivalent `step` syntax. For example, `obj(x)` becomes `step(obj, x)`.

---

Use the Levinson solver to compute polynomial coefficients from autocorrelation coefficients.

```
lev = dsp.LevinsonSolver;
lev.AOutputPort = true;
lev.KOutputPort = false;
x = (1:100)';
a = xcorr(x,10);
% Consider the autocorrelation computed over nonnegative lags [0 10].
c = lev(a(11:end)); % Compute polynomial coefficients
```

## Algorithms

This object implements the algorithm, inputs, and outputs described on the Levinson-Durbin block reference page. The object properties correspond to the block parameters, except: **Output(s)** block parameter corresponds to the `AOutputPort` and the `KOutputPort` object properties.

## Version History

### Introduced in R2012a

#### R2021b: `dsp.LevinsonSolver` System object will be removed

*Warns starting in R2021b*

`dsp.LevinsonSolver` System object will be removed in a future release. Use the `levinson` function instead.

#### R2023a: `dsp.LevinsonSolver` System object has been removed

*Errors starting in R2023a*

The `dsp.LevinsonSolver` System object has been removed. Use the `levinson` function instead.



## Update Code

This table shows how to update existing code to use the `levinson` function.

| Discouraged Usage  | Recommended Replacement   |
|--|---|
| <pre>lev = dsp.LevinsonSolver; lev.AOutputPort = true; lev.KOutputPort = false; x = (1:100)';</pre> <p>The System object computes autocorrelation over the lags [-10 10].</p> <pre>a = xcorr(x,10);</pre> <p>Display the autocorrelation computed over nonnegative lags [0 10].</p> <pre>c = lev(a(11:end));</pre> <pre>c = 10x1  1.0000 -0.9920  0.0001  0.0001  0.0001  0.0001  0.0001  0.0001  0.0001  0.0001  0.0001  0.0070</pre> | <p>The function returns the polynomial coefficients in the form of a row vector.</p> <pre>cfn = levinson(a(11:end))</pre> <pre>cfn = 1x10  1.0000 -0.9920 0.0001 0.0001 0.0001 0.0001 0.0001...  0.0001 0.0001 0.0001 0.0001 0.0070</pre> |

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

### See Also

`levinson`

## step

**System object:** dsp.LevinsonSolver

**Package:** dsp

Reflection coefficients corresponding to columns of input

### Syntax

```
K = step(levinson,X)
A = step(levinson,X)
[A, K] = step(levinson,X)
[... , P] = step(levinson,X)
```

### Description

---

**Note** Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

---

`K = step(levinson,X)` returns reflection coefficients `K` corresponding to the columns of input `X`. `X` is typically a column or matrix of autocorrelation coefficients with lag  $\theta$  as the first element.

`A = step(levinson,X)` returns polynomial coefficients `A` when the `AOutputPort` property is `true` and the `KOutputPort` property is `false`.

`[A, K] = step(levinson,X)` returns polynomial coefficients `A` and reflection coefficients `K` when both the `AOutputPort` and `KOutputPort` properties are `true`.

`[... , P] = step(levinson,X)` also returns the error power `P` when the `PredictionErrorOutputPort` property is `true`.

---

**Note** `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

# dsp.LMSFilter

**Package:** dsp

Compute output, error, and weights of LMS adaptive filter

## Description

The `dsp.LMSFilter` System object implements an adaptive finite impulse response (FIR) filter that converges an input signal to the desired signal using one of the following algorithms:

- LMS
- Normalized LMS
- Sign-Data LMS
- Sign-Error LMS
- Sign-Sign LMS

For more details on each of these methods, see “Algorithms” on page 4-768.

The filter adapts its weights until the error between the primary input signal and the desired signal is minimal. The mean square of this error (MSE) is computed using the `mseim` function. The predicted version of the MSE is determined using a Wiener filter in the `msepred` function. The `maxstep` function computes the maximum adaptation step size, which controls the speed of convergence.

For an overview of the adaptive filter methodology, and the most common applications the adaptive filters are used in, see “Overview of Adaptive Filters and Applications”.

To filter a signal using an adaptive FIR filter:

- 1 Create the `dsp.LMSFilter` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

Under specific conditions, this System object also supports SIMD code generation. For details, see “Code Generation” on page 4-769.

## Creation

### Syntax

```
lms = dsp.LMSFilter
lms = dsp.LMSFilter(L)
lms = dsp.LMSFilter(Name,Value)
```

## Description

`lms = dsp.LMSFilter` returns an LMS filter object, `lms`, that computes the filtered output, filter error, and the filter weights for a given input and a desired signal using the least mean squares (LMS) algorithm.

`lms = dsp.LMSFilter(L)` returns an LMS filter object with the “Length” on page 4-0 property set to `L`.

`lms = dsp.LMSFilter(Name, Value)` returns an LMS filter object with each specified property set to the specified value. Enclose each property name in single quotes. You can use this syntax with the previous input argument.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

### Method — Method to calculate filter weights

'LMS' (default) | 'Normalized LMS' | 'Sign-Data LMS' | 'Sign-Error LMS' | 'Sign-Sign LMS'

Method to calculate filter weights, specified as one of the following:

- 'LMS' -- Solves the Wiener-Hopf equation and finds the filter coefficients for an adaptive filter.
- 'Normalized LMS' -- Normalized variation of the LMS algorithm.
- 'Sign-Data LMS' -- Correction to the filter weights at each iteration depends on the sign of the input `x`.
- 'Sign-Error LMS' -- Correction applied to the current filter weights for each successive iteration depends on the sign of the error, `err`.
- 'Sign-Sign LMS' -- Correction applied to the current filter weights for each successive iteration depends on both the sign of `x` and the sign of `err`.

For more details on the algorithms, see “Algorithms” on page 4-768.

### Length — Length of FIR filter weights vector

32 (default) | positive integer

Length of the FIR filter weights vector, specified as a positive integer.

Example: 64

Example: 16

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### StepSizeSource — Method to specify adaptation step size

'Property' (default) | 'Input port'

Method to specify the adaptation step size, specified as one of the following:

- 'Property' -- The property "StepSize" on page 4-0 specifies the size of each adaptation step.
- 'Input port' -- Specify the adaptation step size as one of the inputs to the object.

### StepSize — Adaptation step size

0.1 (default) | non-negative scalar

Adaptation step size factor, specified as a non-negative scalar. For convergence of the normalized LMS method, the step size must be greater than 0 and less than 2.

A small step size ensures a small steady state error between the output "y" on page 4-0 and the desired signal "d" on page 4-0. If the step size is small, the convergence speed of the filter decreases. To improve the convergence speed, increase the step size. Note that if the step size is large, the filter can become unstable. To compute the maximum step size the filter can accept without becoming unstable, use the `maxstep` function.

**Tunable:** Yes

### Dependencies

This property applies when you set "StepSizeSource" on page 4-0 to 'Property'.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### LeakageFactor — Leakage factor used in leaky LMS method

1 (default) | [0 1]

Leakage factor used when implementing the leaky LMS method, specified as a scalar in the range [0 1]. When the value equals 1, there is no leakage in the adapting method. When the value is less than 1, the filter implements a leaky LMS method.

Example: 0.5

**Tunable:** Yes

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### InitialConditions — Initial conditions of filter weights

0 (default) | scalar | vector

Initial conditions of filter weights, specified as a scalar or a vector of length equal to the value of the "Length" on page 4-0 property. When the input is real, the value of this property must be real.

Example: 0

Example: [1 3 1 2 7 8 9 0 2 2 8 2]

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

Complex Number Support: Yes

### AdaptInputPort — Flag to adapt filter weights

false (default) | true

Flag to adapt filter weights, specified as one of the following:

- false -- The object continuously updates the filter weights.

- `true` -- An adaptation control input is provided to the object when you call its algorithm. If the value of this input is non-zero, the object continuously updates the filter weights. If the value of this input is zero, the filter weights remain at their current value.

### WeightsResetInputPort — Flag to reset filter weights

`false` (default) | `true`

Flag to reset filter weights, specified as one of the following:

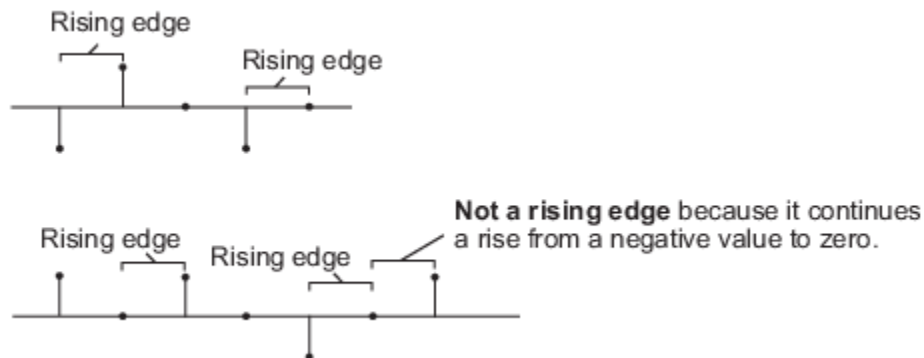
- `false` -- The object does not reset weights.
- `true` -- A reset control input is provided to the object when you call its algorithm. This setting enables the “WeightsResetCondition” on page 4-0 property. The object resets the filter weights based on the values of the `WeightsResetCondition` property and the reset input provided to the object algorithm.

### WeightsResetCondition — Event to reset filter weights

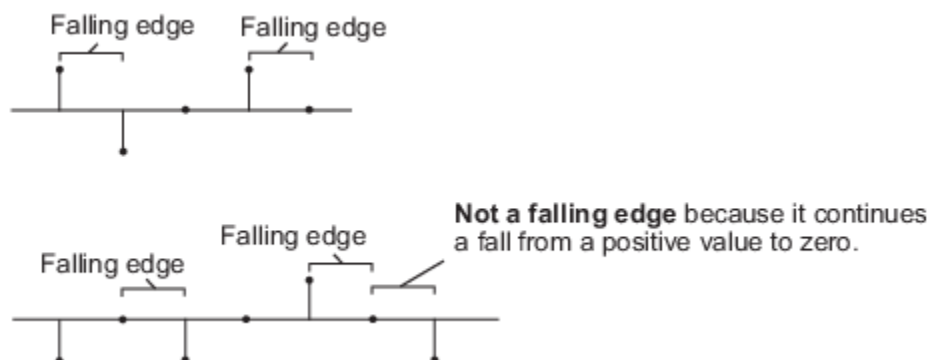
`'Non-zero'` (default) | `'Rising edge'` | `'Falling edge'` | `'Either edge'`

Event that triggers the reset of the filter weights, specified as one of the following. The object resets the filter weights whenever a reset event is detected in its reset input.

- `'Non-zero'` -- Triggers a reset operation at each sample, when the reset input is not zero.
- `'Rising edge'` -- Triggers a reset operation when the reset input does one of the following:
  - Rises from a negative value to either a positive value or zero.
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero.



- `'Falling edge'` -- Triggers a reset operation when the reset input does one of the following:
  - Falls from a positive value to a negative value or zero.
  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero.



- 'Either edge' -- Triggers a reset operation when the reset input is a rising edge or a falling edge.

The object resets the filter weights based on the value of this property and the reset input `r` provided to the object algorithm.

#### Dependencies

This property applies when you set the “WeightsResetInputPort” on page 4-0 property to `true`.

#### WeightsOutput — Method to output adapted filter weights

'Last' (default) | 'None' | 'All'

Method to output adapted filter weights, specified as one of the following:

- 'Last' (default) — The object returns a column vector of weights corresponding to the last sample of the data frame. The length of the weights vector is the value given by the “Length” on page 4-0 property.
- 'All' — The object returns a *FrameLength*-by-*Length* matrix of weights. The matrix corresponds to the full sample-by-sample history of weights for all *FrameLength* samples of the input values. Each row in the matrix corresponds to a set of LMS filter weights calculated for the corresponding input sample.
- 'None' — This setting disables the weights output.

#### Fixed-Point Properties

##### RoundingMethod — Rounding method for fixed-point operations

'Floor' (default) | 'Ceiling' | 'Convergent' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'

Specify the rounding mode for fixed-point operations. For more details, see rounding mode.

##### OverflowAction — Overflow action for fixed-point operations

'Wrap' (default) | 'Saturate'

Overflow action for fixed-point operations, specified as one of the following:

- 'Wrap' -- The object wraps the result of its fixed-point operations.
- 'Saturate' -- The object saturates the result of its fixed-point operations.

For more details on overflow actions, see overflow mode for fixed-point operations.

##### StepSizeDataType — Step size word length and fraction length settings

'Same word length as first input' (default) | 'Custom'

Step size word length and fraction length settings, specified as one of the following:

- 'Same word length as first input' -- The object specifies the word length of step size to be the same as that of the first input. The fraction length is computed to get the best possible precision.
- 'Custom' -- The step size data type is specified as a custom numeric type through the "CustomStepSizeDataType" on page 4-0 property.

For more information on the step size data type this object uses, see the "Fixed Point" on page 4-765 section.

#### **CustomStepSizeDataType — Word and fraction lengths of step size**

`numerictype([],16,15)` (default)

Word and fraction lengths of the step size, specified as an autosigned numeric type with a word length of 16 and a fraction length of 15.

Example: `numerictype([],32)`

#### **Dependencies**

This property applies under the following conditions:

- "StepSizeSource" on page 4-0 property set to 'Property' and "StepSizeDataType" on page 4-0 set to 'Custom'.
- StepSizeSource property set to 'Input port'.

#### **LeakageFactorDataType — Leakage factor word length and fraction length settings**

'Same word length as first input' (default) | 'Custom'

Leakage factor word length and fraction length settings, specified as one of the following:

- 'Same word length as first input' -- The object specifies the word length of leakage factor to be the same as that of the first input. The fraction length is computed to get the best possible precision.
- 'Custom' -- The leakage factor data type is specified as a custom numeric type through the "CustomLeakageFactorDataType" on page 4-0 property.

For more information on the leakage factor data type this object uses, see the "Fixed Point" on page 4-765 section.

#### **CustomLeakageFactorDataType — Word and fraction lengths of the leakage factor**

`numerictype([],16,15)` (default)

Word and fraction lengths of the leakage factor, specified as an autosigned numeric type with a word length of 16 and a fraction length of 15.

Example: `numerictype([],32)`

#### **Dependencies**

This property applies when you set the "LeakageFactorDataType" on page 4-0 property to 'Custom'.

#### **WeightsDataType — Weights word length and fraction length settings**

'Same as first input' (default) | 'Custom'



Weights word length and fraction length settings, specified as one of the following:

- 'Same as first input' -- The object specifies the data type of the filter weights to be the same as that of the first input.
- 'Custom' -- The data type of filter weights is specified as a custom numeric type through the "CustomWeightsDataType" on page 4-0 property.

For more information on the filter weights data type this object uses, see the "Fixed Point" on page 4-765 section.

### **CustomWeightsDataType — Word and fraction lengths of filter weights**

`numericType([],16,15)` (default)

Word and fraction lengths of the filter weights, specified as an autosigned numeric type with a word length of 16 and a fraction length of 15.

Example: `numericType([],32,20)`

#### **Dependencies**

This property applies when you set the "WeightsDataType" on page 4-0 property to 'Custom'.

### **EnergyProductDataType — Energy product word length and fraction length settings**

'Same as first input' (default) | 'Custom'

Energy product word length and fraction length settings, specified as one of the following:

- 'Same as first input' -- The object specifies the data type of the energy product to be the same as that of the first input.
- 'Custom' -- The data type of the energy product is specified as a custom numeric type through the "CustomEnergyProductDataType" on page 4-0 property.

For more information on the energy product data type this object uses, see the "Fixed Point" on page 4-765 section.

#### **Dependencies**

This property applies when you set the "Method" on page 4-0 property to 'Normalized LMS'.

### **CustomEnergyProductDataType — Word and fraction lengths of energy product**

`numericType([],32,20)` (default)

Word and fraction lengths of the energy product, specified as an autosigned numeric type with a word length of 32 and a fraction length of 20.

#### **Dependencies**

This property applies when you set the "Method" on page 4-0 property to 'Normalized LMS' and "EnergyProductDataType" on page 4-0 property to 'Custom'.

### **EnergyAccumulatorDataType — Energy accumulator word length and fraction length settings**

'Same as first input' (default) | 'Custom'

Energy accumulator word length and fraction length settings, specified as one of the following:

- 'Same as first input' -- The object specifies the data type of the energy accumulator to be the same as that of the first input.
- 'Custom' -- The data type of the energy accumulator is specified as a custom numeric type through the “CustomEnergyAccumulatorDataType” on page 4-0 property.

For more information on the energy accumulator data type this object uses, see the “Fixed Point” on page 4-765 section.

### Dependencies

This property applies when you set the “Method” on page 4-0 property to 'Normalized LMS'.

### CustomEnergyAccumulatorDataType — Word and fraction lengths of energy accumulator

numericity([],32,20) (default)

Word and fraction lengths of the energy accumulator, specified as an autosigned numeric type with a word length of 32 and a fraction length of 20.

### Dependencies

This property applies when you set the “Method” on page 4-0 property to 'Normalized LMS' and “EnergyAccumulatorDataType” on page 4-0 property to 'Custom'.

### ConvolutionProductDataType — Convolution product word length and fraction length settings

'Same as first input' (default) | 'Custom'

Convolution product word length and fraction length settings, specified as one of the following:

- 'Same as first input' -- The object specifies the data type of the convolution product to be the same as that of the first input.
- 'Custom' -- The data type of the convolution product is specified as a custom numeric type through the “CustomConvolutionProductDataType” on page 4-0 property.

For more information on the convolution product data type this object uses, see the “Fixed Point” on page 4-765 section.

### CustomConvolutionProductDataType — Word and fraction lengths of convolution product

numericity([],32,20) (default)

Word and fraction lengths of the convolution product, specified as an autosigned numeric type with a word length of 32 and a fraction length of 20.

### Dependencies

This property applies when you set the “ConvolutionProductDataType” on page 4-0 property to 'Custom'.

### ConvolutionAccumulatorDataType — Convolution accumulator word length and fraction length settings

'Same as first input' (default) | 'Custom'

Convolution accumulator word length and fraction length settings, specified as one of the following:

- 'Same as first input' -- The object specifies the data type of the convolution accumulator to be the same as that of the first input.

- 'Custom' -- The data type of the convolution accumulator is specified as a custom numeric type through the “CustomConvolutionAccumulatorDataType” on page 4-0 property.

For more information on the convolution accumulator data type this object uses, see the “Fixed Point” on page 4-765 section.

#### **CustomConvolutionAccumulatorDataType — Word and fraction lengths of convolution accumulator**

`numericType([],32,20)` (default)

Word and fraction lengths of the convolution accumulator, specified as an autosigned numeric type with a word length of 32 and a fraction length of 20.

#### **Dependencies**

This property applies when you set the “ConvolutionAccumulatorDataType” on page 4-0 property to 'Custom'.

#### **StepSizeErrorProductDataType — Step size error product word length and fraction length settings**

`'Same as first input'` (default) | `'Custom'`

Step size error product word length and fraction length settings, specified as one of the following:

- 'Same as first input' -- The object specifies the data type of the step size error product to be the same as that of the first input.
- 'Custom' -- The data type of the step size error product is specified as a custom numeric type through the “CustomStepSizeErrorProductDataType” on page 4-0 property.

For more information on the step size error product data type this object uses, see the “Fixed Point” on page 4-765 section.

#### **CustomStepSizeErrorProductDataType — Word and fraction lengths of step size error product**

`numericType([],32,20)` (default)

Word and fraction lengths of the step size error product, specified as an autosigned numeric type with a word length of 32 and a fraction length of 20.

#### **Dependencies**

This property applies when you set the “StepSizeErrorProductDataType” on page 4-0 property to 'Custom'.

#### **WeightsUpdateProductDataType — Filter weights update product word length and fraction length settings**

`'Same as first input'` (default) | `'Custom'`

Word and fraction length settings of the filter weights update product, specified as one of the following:

- 'Same as first input' -- The object specifies the data type of the filter weights update product to be the same as that of the first input.
- 'Custom' -- The data type of the filter weights update product is specified as a custom numeric type through the “CustomWeightsUpdateProductDataType” on page 4-0 property.

For more information on the filter weights update product data type this object uses, see the “Fixed Point” on page 4-765 section.

### **CustomWeightsUpdateProductDataType — Word and fraction lengths of filter weights update product**

`numericType([],32,20)` (default)

Word and fraction lengths of the filter weights update product, specified as an autosigned numeric type with a word length of 32 and a fraction length of 20.

#### **Dependencies**

This property applies when you set the “WeightsUpdateProductDataType” on page 4-0 property to 'Custom'.

### **QuotientDataType — Quotient word length and fraction length settings**

'Same as first input' (default) | 'Custom'

Quotient word length and fraction length settings, specified as one of the following:

- 'Same as first input' -- The object specifies the quotient data type to be the same as that of the first input.
- 'Custom' -- The quotient data type is specified as a custom numeric type through the “CustomQuotientDataType” on page 4-0 property.

For more information on the quotient data type this object uses, see the “Fixed Point” on page 4-765 section.

#### **Dependencies**

This property applies when you set the “Method” on page 4-0 property to 'Normalized LMS'.

### **CustomQuotientDataType — Word and fraction lengths of quotient**

`numericType([],32,20)` (default)

Word and fraction lengths of the filter weights update product, specified as an autosigned numeric type with a word length of 32 and a fraction length of 20.

#### **Dependencies**

This property applies when you set the “Method” on page 4-0 property to 'Normalized LMS' and “QuotientDataType” on page 4-0 property to 'Custom'.

## **Usage**

### **Syntax**

```
[y,err,wts] = lms(x,d)
[y,err] = lms(x,d)
[___] = lms(x,d,mu)
[___] = lms(x,d,a)
[___] = lms(x,d,r)
[y,err,wts] = lms(x,d,mu,a,r)
```

## Description

`[y,err,wts] = lms(x,d)` filters the input signal, `x`, using `d` as the desired signal, and returns the filtered output in `y`, the filter error in `err`, and the estimated filter weights in `wts`. The LMS filter object estimates the filter weights needed to minimize the error between the output signal and the desired signal.

`[y,err] = lms(x,d)` filters the input signal, `x`, using `d` as the desired signal, and returns the filtered output in `y` and the filter error in `err` when the “WeightsOutput” on page 4-0 property is set to 'None'.

`[ ___ ] = lms(x,d,mu)` filters the input signal, `x`, using `d` as the desired signal and `mu` as the step size, when the “StepSizeSource” on page 4-0 property is set to 'Input port'. These inputs can be used with either of the previous sets of outputs.

`[ ___ ] = lms(x,d,a)` filters the input signal, `x`, using `d` as the desired signal and `a` as the adaptation control when the “AdaptInputPort” on page 4-0 property is set to `true`. When `a` is nonzero, the System object continuously updates the filter weights. When `a` is zero, the filter weights remain constant.

`[ ___ ] = lms(x,d,r)` filters the input signal, `x`, using `d` as the desired signal and `r` as a reset signal when the “WeightsResetInputPort” on page 4-0 property is set to `true`. The “WeightsResetCondition” on page 4-0 property can be used to set the reset trigger condition. If a reset event occurs, the System object resets the filter weights to their initial values.

`[y,err,wts] = lms(x,d,mu,a,r)` filters the input signal, `x`, using `d` as the desired signal, `mu` as the step size, `a` as the adaptation control, and `r` as the reset signal, and returns the filtered output in `y`, the filter error in `err`, and the adapted filter weights in `wts`.

## Input Arguments

### **x** — Data input

scalar | column vector

The signal to be filtered by the LMS filter. The input, `x`, and the desired signal, `d` must have the same size, data type, and complexity. If the input is fixed-point, the data type must be signed and must have the same word length as the desired signal.

The input, `x` can be a variable-size signal. You can change the number of elements in the column vector even when the object is locked. The System object locks when you call the object to run its algorithm.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `fi`

### **d** — Desired signal

scalar | column vector

The LMS filter adapts its filter weights, `wts`, to minimize the error, `err`, and converge the input signal `x` to the desired signal `d` as closely as possible.

The input, `x`, and the desired signal, `d`, must have the same size, data type, and complexity. If the desired signal is fixed-point, the data type must be signed and must have the same word length as the data input.

The input, `d` can be a variable-size signal. You can change the number of elements in the column vector even when the object is locked. The System object locks when you call the object to run its algorithm.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `fi`

**mu — Step size**

nonnegative scalar

Adaptation step size factor, specified as a scalar, nonnegative numeric value. For convergence of the normalized LMS method, the step size should be greater than 0 and less than 2. The data type of the step size input must match the data type of `x` and `d`. If the data type is fixed-point, the data type must be signed.

A small step size ensures a small steady state error between the output `y` and the desired signal `d`. If the step size is small, the convergence speed of the filter decreases. To improve the convergence speed, increase the step size. Note that if the step size is large, the filter can become unstable. To compute the maximum step size the filter can accept without becoming unstable, use the `maxstep` function.

**Dependencies**

This input is required when the “StepSizeSource” on page 4-0 property is set to `'Input port'`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `fi`

**a — Adaptation control**

scalar

Adaptation control input that controls how the filter weights are updated. If the value of this input is non-zero, the object continuously updates the filter weights. If the value of this input is zero, the filter weights remain at their current value.

**Dependencies**

This input is required when the “AdaptInputPort” on page 4-0 property is set to `true`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `logical`

**r — Reset signal**

scalar

Reset signal that resets the filter weights based on the values of the “WeightsResetCondition” on page 4-0 property.

**Dependencies**

This input is required when the “WeightsResetInputPort” on page 4-0 property is set to `true`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `logical`

**Output Arguments****y — Filtered output**

scalar | column vector

Filtered output, returned as a scalar or a column vector. The object adapts its filter weights to converge the input signal `x` to match the desired signal `d`. The filter outputs the converged signal.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `fi`

### **err — Difference between output and desired signal**

scalar | column vector

Difference between the output signal `y` and the desired signal `d`, returned as a scalar or a column vector. The data type of `err` matches the data type of `y`. The objective of the adaptive filter is to minimize this error. The object adapts its weights to converge towards optimal filter weights that produce an output signal that matches closely with the desired signal. For more details on how `err` is computed, see “Algorithms” on page 4-768.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `fi`

### **wts — Adaptive filter weights**

scalar | column vector

Adaptive filter weights, returned as a scalar or a column vector of length specified by the value in “Length” on page 4-0 .

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `fi`

## **Object Functions**

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

### **Specific to dsp.LMSFilter**

`maxstep` Maximum step size for LMS adaptive filter convergence  
`msepred` Predicted mean squared error for LMS adaptive filter  
`msestim` Estimated mean squared error for adaptive filters

### **Common to All System Objects**

`step` Run System object algorithm  
`release` Release resources and allow changes to System object property values and input characteristics  
`reset` Reset internal states of System object

## **Examples**

### **Predict Mean Squared Error for LMS Filter**

The mean squared error (MSE) measures the average of the squares of the errors between the desired signal and the primary signal input to the adaptive filter. Reducing this error converges the primary input to the desired signal. Determine the predicted value of MSE and the simulated value of MSE at each time instant using the `msepred` and `msestim` functions. Compare these MSE values with each other and with respect to the minimum MSE and steady-state MSE values. In addition, compute the sum of the squares of the coefficient errors given by the trace of the coefficient covariance matrix.

### Initialization

Create a `dsp.FIRFilter` System object™ that represents the unknown system. Pass the signal, `x`, to the FIR filter. The output of the unknown system is the desired signal, `d`, which is the sum of the output of the unknown system (FIR filter) and an additive noise signal, `n`.

```
num = fir1(31,0.5);
fir = dsp.FIRFilter('Numerator',num);
iir = dsp.IIRFilter('Numerator',sqrt(0.75),...
    'Denominator',[1 -0.5]);
x = iir(sign(randn(2000,25)));
n = 0.1*randn(size(x));
d = fir(x) + n;
```

### LMS Filter

Create a `dsp.LMSFilter` System object to create a filter that adapts to output the desired signal. Set the length of the adaptive filter to 32 taps, step size to 0.008, and the decimation factor for analysis and simulation to 5. The variable `simmse` represents the simulated MSE between the output of the unknown system, `d`, and the output of the adaptive filter. The variable `mse` gives the corresponding predicted value.

```
l = 32;
mu = 0.008;
m = 5;

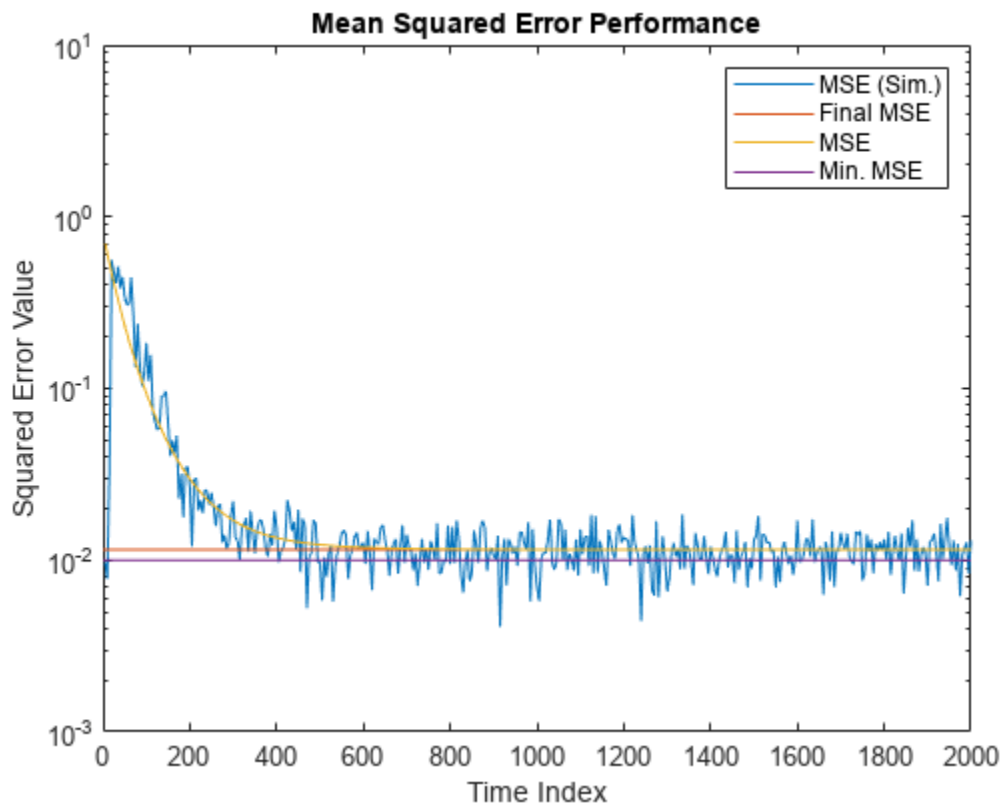
lms = dsp.LMSFilter('Length',l,'StepSize',mu);
[mmse,emse,meanW,mse,traceK] = msepred(lms,x,d,m);
[simmse,meanWsim,Wsim,traceKsim] = msesim(lms,x,d,m);
```

### Plot the MSE Results

Compare the values of simulated MSE, predicted MSE, minimum MSE, and the final MSE. The final MSE value is given by the sum of minimum MSE and excess MSE.

```
nn = m:m:size(x,1);
semilogy(nn,simmse,[0 size(x,1)],[(emse+mmse)...
    (emse+mmse)],nn,mse,[0 size(x,1)],[mmse mmse])
title('Mean Squared Error Performance')
axis([0 size(x,1) 0.001 10])
legend('MSE (Sim.)','Final MSE','MSE','Min. MSE')
xlabel('Time Index')
ylabel('Squared Error Value')
```





The predicted MSE follows the same trajectory as the simulated MSE. Both these trajectories converge with the steady-state (final) MSE.

### Plot the Coefficient Trajectories

meanWsim is the mean value of the simulated coefficients given by msesim. meanW is the mean value of the predicted coefficients given by msepred.

Compare the simulated and predicted mean values of LMS filter coefficients 12,13,14, and 15.

```
plot(nn,meanWsim(:,12),'b',nn,meanW(:,12),'r',nn,...
meanWsim(:,13:15),'b',nn,meanW(:,13:15),'r')
PlotTitle ={'Average Coefficient Trajectories for';...
            'W(12), W(13), W(14), and W(15)'}

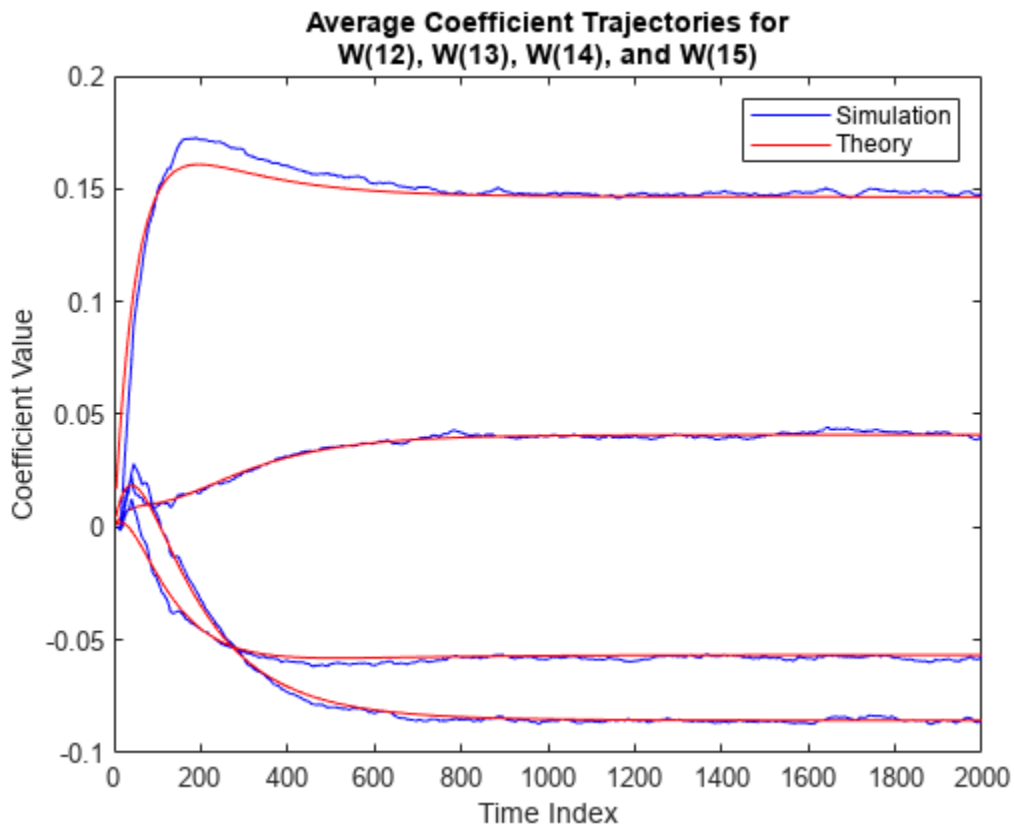
```

```
PlotTitle = 2x1 cell
            {'Average Coefficient Trajectories for'}
            {'W(12), W(13), W(14), and W(15)'}

```

```
title(PlotTitle)
legend('Simulation','Theory')
xlabel('Time Index')
ylabel('Coefficient Value')

```

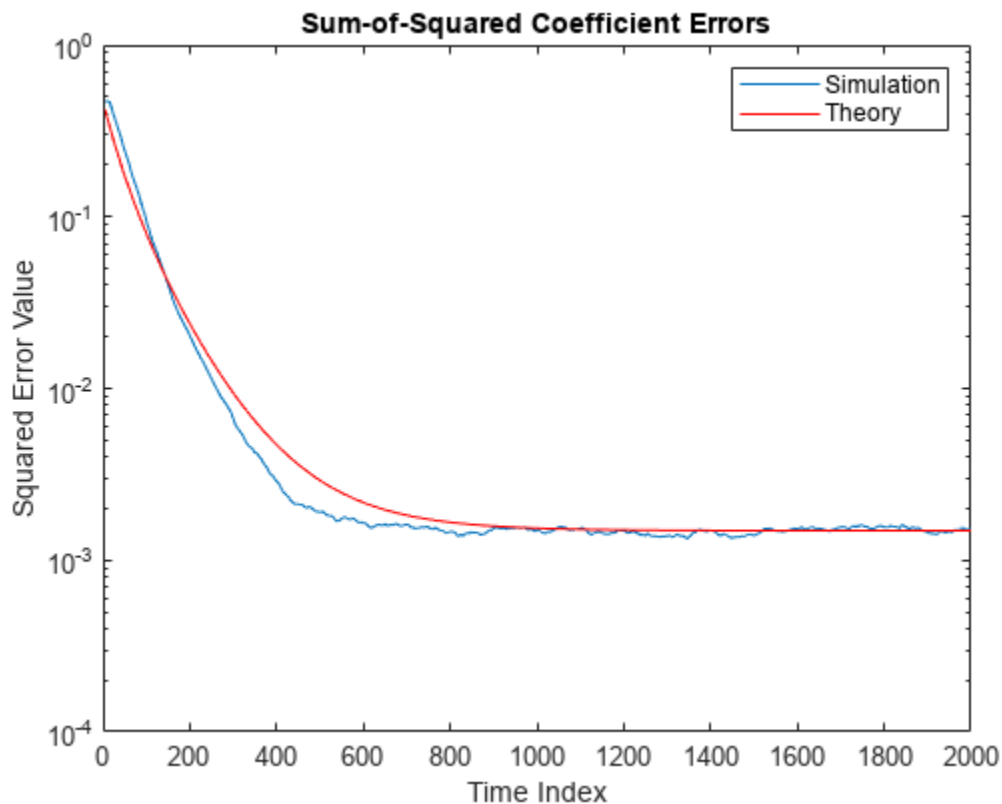


In steady state, both the trajectories converge.

### Sum of Squared Coefficient Errors

Compare the sum of the squared coefficient errors given by `msepred` and `msesim`. These values are given by the trace of the coefficient covariance matrix.

```
semilogy(nn,traceKsim,nn,traceK,'r')
title('Sum-of-Squared Coefficient Errors')
axis([0 size(x,1) 0.0001 1])
legend('Simulation','Theory')
xlabel('Time Index')
ylabel('Squared Error Value')
```



### Compute Maximum Step of LMS Adaptive Filter

The `maxstep` function computes the maximum step size of the adaptive filter. This step size keeps the filter stable at the maximum possible speed of convergence. Create the primary input signal, `x`, by passing a signed random signal to an IIR filter. Signal `x` contains 50 frames of 2000 samples each frame. Create an LMS filter with 32 taps and a step size of 0.1.

```
x = zeros(2000,50);
IIRFilter = dsp.IIRFilter('Numerator',sqrt(0.75),...
    'Denominator',[1 -0.5]);
for k = 1:size(x,2)
    x(:,k) = IIRFilter(sign(randn(size(x,1),1)));
end
mu = 0.1;
LMSFilter = dsp.LMSFilter('Length',32,...
    'StepSize',mu);
```

Compute the maximum adaptation step size and the maximum step size in mean-squared sense using the `maxstep` function.

```
[mumax,mumaxmse] = maxstep(LMSFilter,x)

mumax = 0.0625
mumaxmse = 0.0536
```

## System Identification of FIR Filter Using LMS Algorithm

System identification is the process of identifying the coefficients of an unknown system using an adaptive filter. The general overview of the process is shown in “System Identification -- Using an Adaptive Filter to Identify an Unknown System”. The main components involved are:

- The adaptive filter algorithm. In this example, set the `Method` property of `dsp.LMSFilter` to 'LMS' to choose the LMS adaptive filter algorithm.
- An unknown system or process to adapt to. In this example, the filter designed by `firband` is the unknown system.
- Appropriate input data to exercise the adaptation process. For the generic LMS model, these are the desired signal  $d(k)$  and the input signal  $x(k)$ .

The objective of the adaptive filter is to minimize the error signal between the output of the adaptive filter  $y(k)$  and the output of the unknown system (or the system to be identified)  $d(k)$ . Once the error signal is minimized, the adapted filter resembles the unknown system. The coefficients of both the filters match closely.

### Unknown System

Create a `dsp.FIRFilter` object that represents the system to be identified. Use the `firband` function to design the filter coefficients. The designed filter is a lowpass filter constrained to 0.2 ripple in the stopband.

```
filt = dsp.FIRFilter;  
filt.Numerator = firband(12,[0 0.4 0.5 1],[1 1 0 0],[1 0.2],...  
{'w' 'c'});
```

Pass the signal  $x$  to the FIR filter. The desired signal  $d$  is the sum of the output of the unknown system (FIR filter) and an additive noise signal  $n$ .

```
x = 0.1*randn(250,1);  
n = 0.01*randn(250,1);  
d = filt(x) + n;
```

### Adaptive Filter

With the unknown filter designed and the desired signal in place, create and apply the adaptive LMS filter object to identify the unknown filter.

Preparing the adaptive filter object requires starting values for estimates of the filter coefficients and the LMS step size ( $\mu$ ). You can start with some set of nonzero values as estimates for the filter coefficients. This example uses zeros for the 13 initial filter weights. Set the `InitialConditions` property of `dsp.LMSFilter` to the desired initial values of the filter weights. For the step size, 0.8 is a good compromise between being large enough to converge well within 250 iterations (250 input sample points) and small enough to create an accurate estimate of the unknown filter.

Create a `dsp.LMSFilter` object to represent an adaptive filter that uses the LMS adaptive algorithm. Set the length of the adaptive filter to 13 taps and the step size to 0.8.

```
mu = 0.8;  
lms = dsp.LMSFilter(13,'StepSize',mu)
```

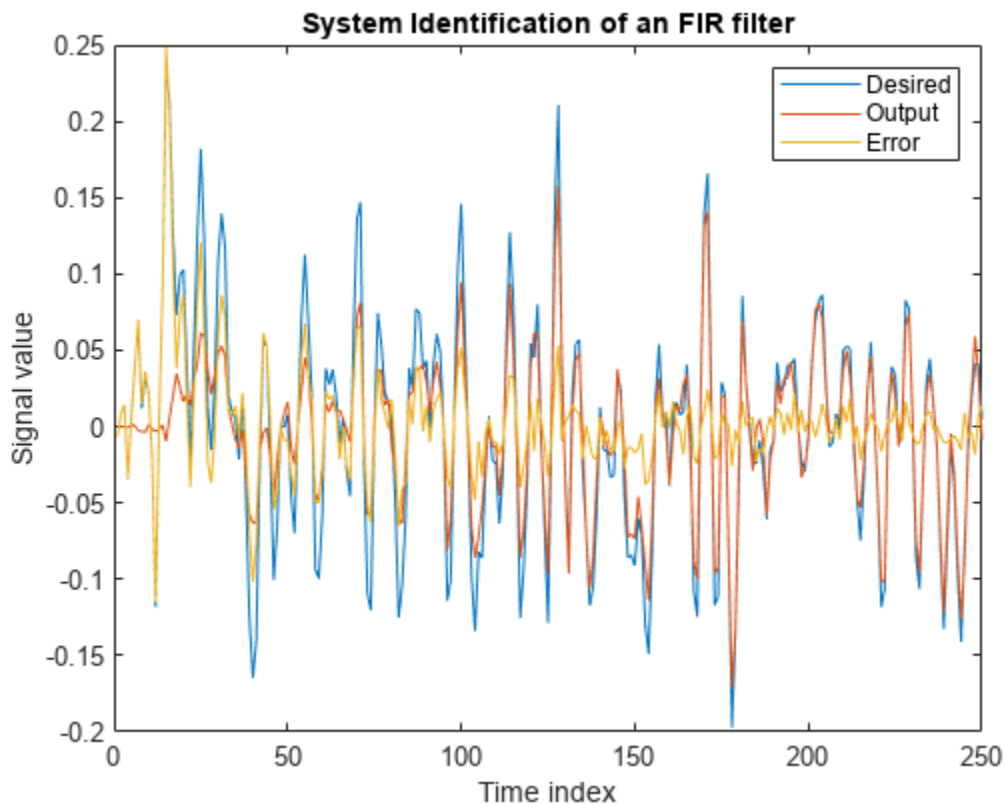
```
lms =  
  dsp.LMSFilter with properties:  
  
          Method: 'LMS'  
         Length: 13  
  StepSizeSource: 'Property'  
         StepSize: 0.8000  
    LeakageFactor: 1  
  InitialConditions: 0  
    AdaptInputPort: false  
  WeightsResetInputPort: false  
        WeightsOutput: 'Last'
```

Show all properties

Pass the primary input signal  $x$  and the desired signal  $d$  to the LMS filter. Run the adaptive filter to determine the unknown system. The output  $y$  of the adaptive filter is the signal converged to the desired signal  $d$  thereby minimizing the error  $e$  between the two signals.

Plot the results. The output signal does not match the desired signal as expected, making the error between the two nontrivial.

```
[y,e,w] = lms(x,d);  
plot(1:250, [d,y,e])  
title('System Identification of an FIR filter')  
legend('Desired','Output','Error')  
xlabel('Time index')  
ylabel('Signal value')
```

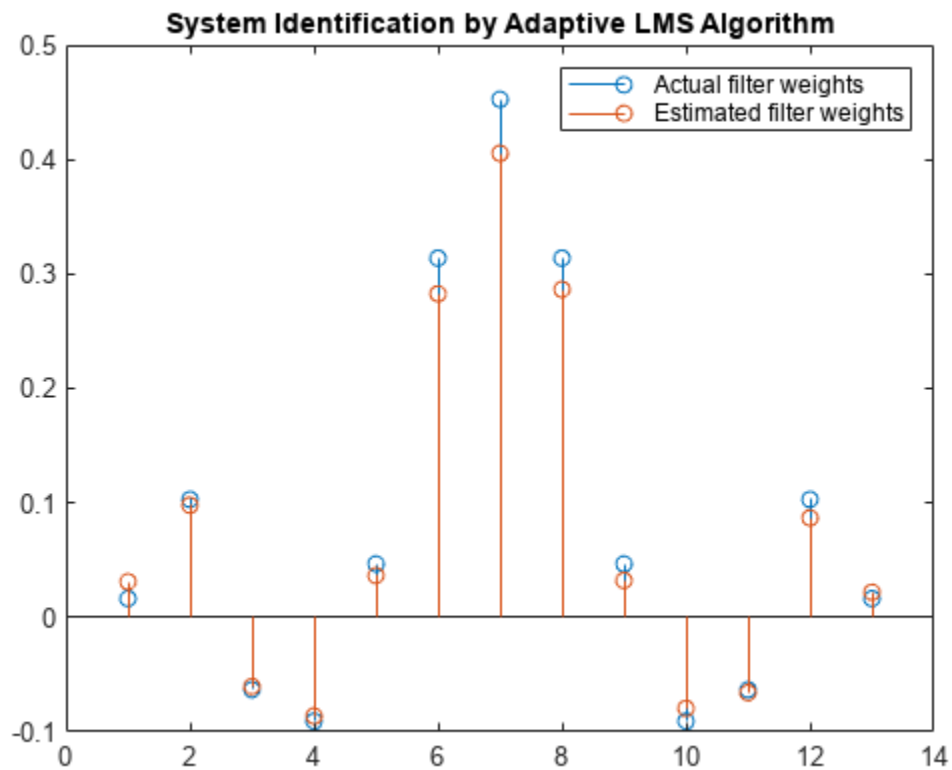


### Compare the Weights

The weights vector  $w$  represents the coefficients of the LMS filter that is adapted to resemble the unknown system (FIR filter). To confirm the convergence, compare the numerator of the FIR filter and the estimated weights of the adaptive filter.

The estimated filter weights do not closely match the actual filter weights, confirming the results seen in the previous signal plot.

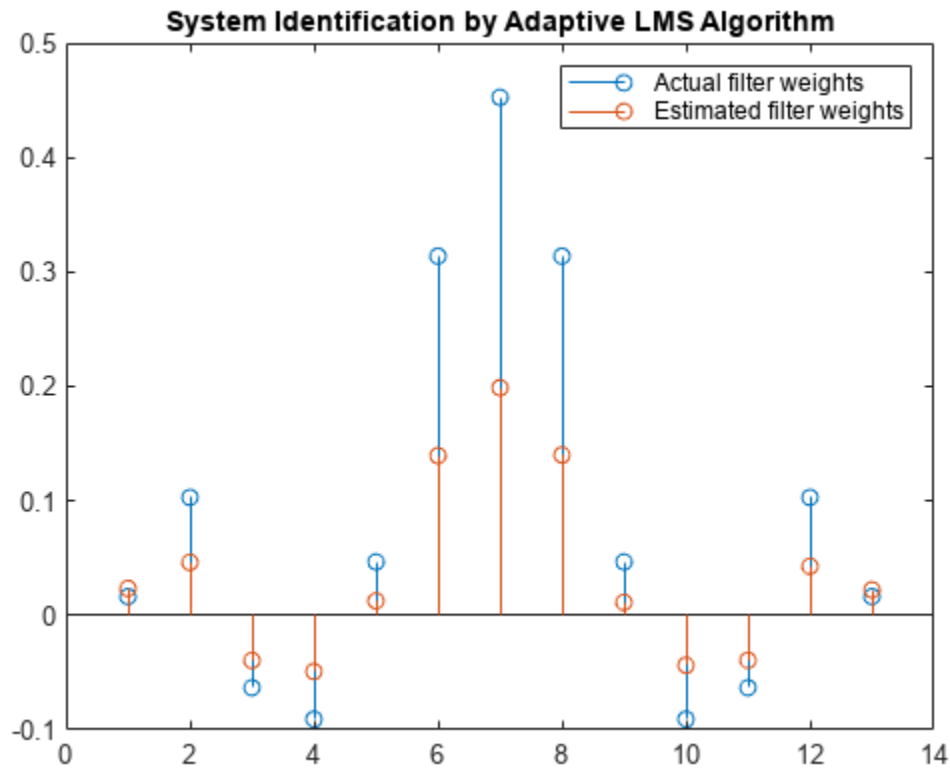
```
stem([(filt.Numerator). ' w'])
title('System Identification by Adaptive LMS Algorithm')
legend('Actual filter weights', 'Estimated filter weights', ...
       'Location', 'NorthEast')
```



### Changing the Step Size

As an experiment, change the step size to 0.2. Repeating the example with  $\mu = 0.2$  results in the following stem plot. The filters do not converge, and the estimated weights are not good approximations of the actual weights.

```
mu = 0.2;
lms = dsp.LMSFilter(13, 'StepSize', mu);
[~,~,w] = lms(x,d);
stem([(filt.Numerator).' w])
title('System Identification by Adaptive LMS Algorithm')
legend('Actual filter weights', 'Estimated filter weights', ...
       'Location', 'NorthEast')
```

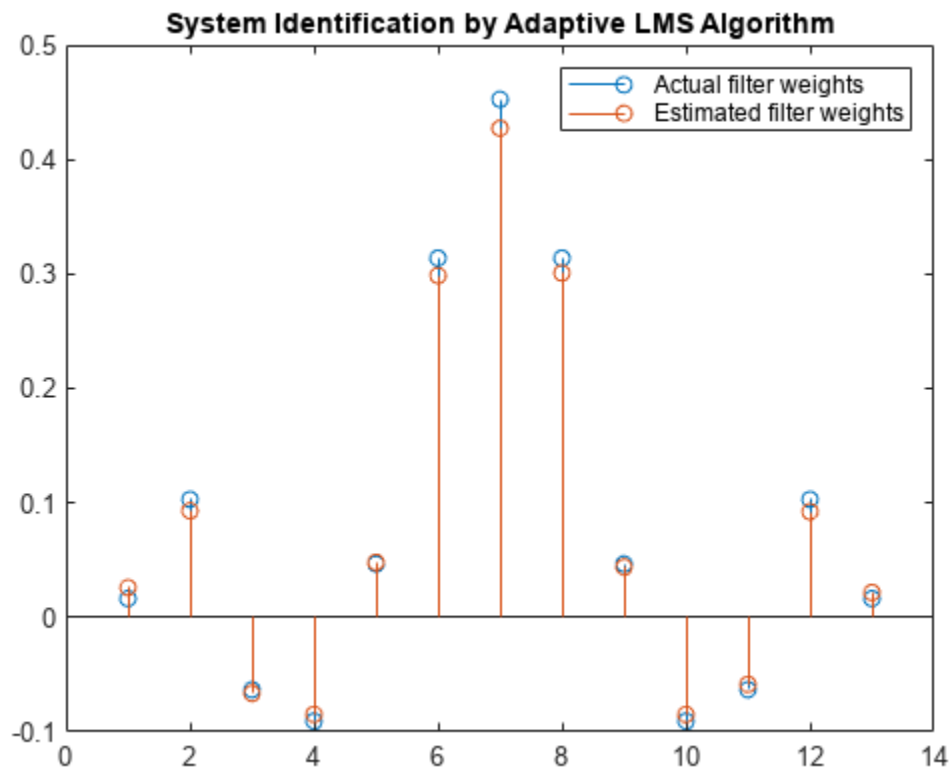


### Increase the Number of Data Samples

Increase the frame size of the desired signal. Even though this increases the computation involved, the LMS algorithm now has more data that can be used for adaptation. With 1000 samples of signal data and a step size of 0.2, the coefficients are aligned closer than before, indicating an improved convergence.

```
release(filt);
x = 0.1*randn(1000,1);
n = 0.01*randn(1000,1);
d = filt(x) + n;
[y,e,w] = lms(x,d);
stem([filt.Numerator.' w])
title('System Identification by Adaptive LMS Algorithm')
legend('Actual filter weights','Estimated filter weights',...
       'Location','NorthEast')
```





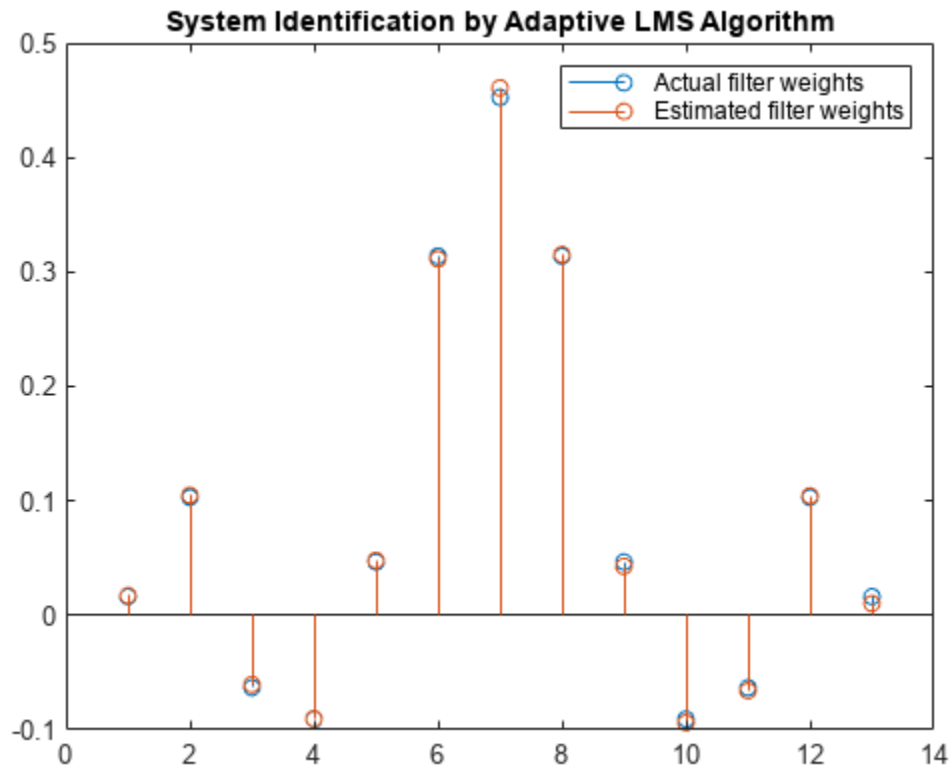
Increase the number of data samples further by inputting the data through iterations. Run the algorithm on 4000 samples of data, passed to the LMS algorithm in batches of 1000 samples over 4 iterations.

Compare the filter weights. The weights of the LMS filter match the weights of the FIR filter very closely, indicating a good convergence.

```

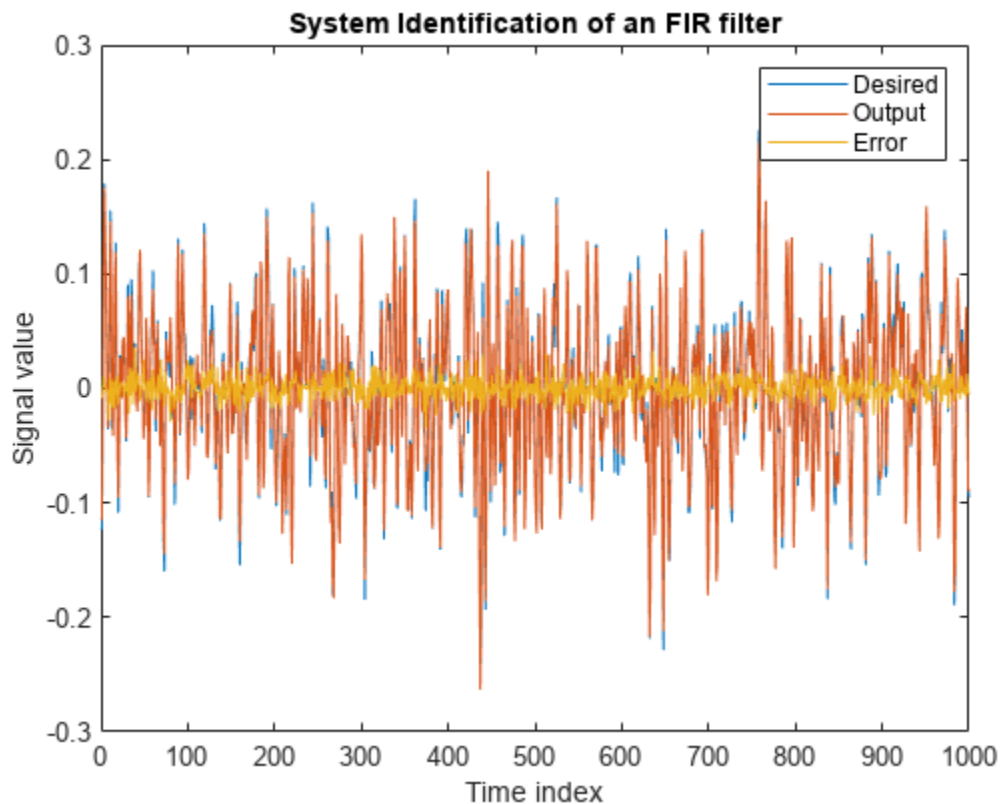
release(filt);
n = 0.01*randn(1000,1);
for index = 1:4
    x = 0.1*randn(1000,1);
    d = filt(x) + n;
    [y,e,w] = lms(x,d);
end
stem([(filt.Numerator).' w])
title('System Identification by Adaptive LMS Algorithm')
legend('Actual filter weights','Estimated filter weights',...
       'Location','NorthEast')

```



The output signal matches the desired signal very closely, making the error between the two close to zero.

```
plot(1:1000, [d,y,e])  
title('System Identification of an FIR filter')  
legend('Desired', 'Output', 'Error')  
xlabel('Time index')  
ylabel('Signal value')
```



### System Identification of FIR Filter Using Normalized LMS Algorithm

To improve the convergence performance of the LMS algorithm, the normalized variant (NLMS) uses an adaptive step size based on the signal power. As the input signal power changes, the algorithm calculates the input power and adjusts the step size to maintain an appropriate value. The step size changes with time, and as a result, the normalized algorithm converges faster with fewer samples in many cases. For input signals that change slowly over time, the normalized LMS algorithm can be a more efficient LMS approach.

For an example using the LMS approach, see “System Identification of FIR Filter Using LMS Algorithm”.

### Unknown System

Create a `dsp.FIRFilter` object that represents the system to be identified. Use the `firband` function to design the filter coefficients. The designed filter is a lowpass filter constrained to 0.2 ripple in the stopband.

```
filt = dsp.FIRFilter;
filt.Numerator = firband(12,[0 0.4 0.5 1],[1 1 0 0],[1 0.2],...
{'w' 'c'});
```

Pass the signal `x` to the FIR filter. The desired signal `d` is the sum of the output of the unknown system (FIR filter) and an additive noise signal `n`.

```
x = 0.1*randn(1000,1);  
n = 0.001*randn(1000,1);  
d = filt(x) + n;
```

### Adaptive Filter

To use the normalized LMS algorithm variation, set the `Method` property on the `dsp.LMSFilter` to 'Normalized LMS'. Set the length of the adaptive filter to 13 taps and the step size to 0.2.

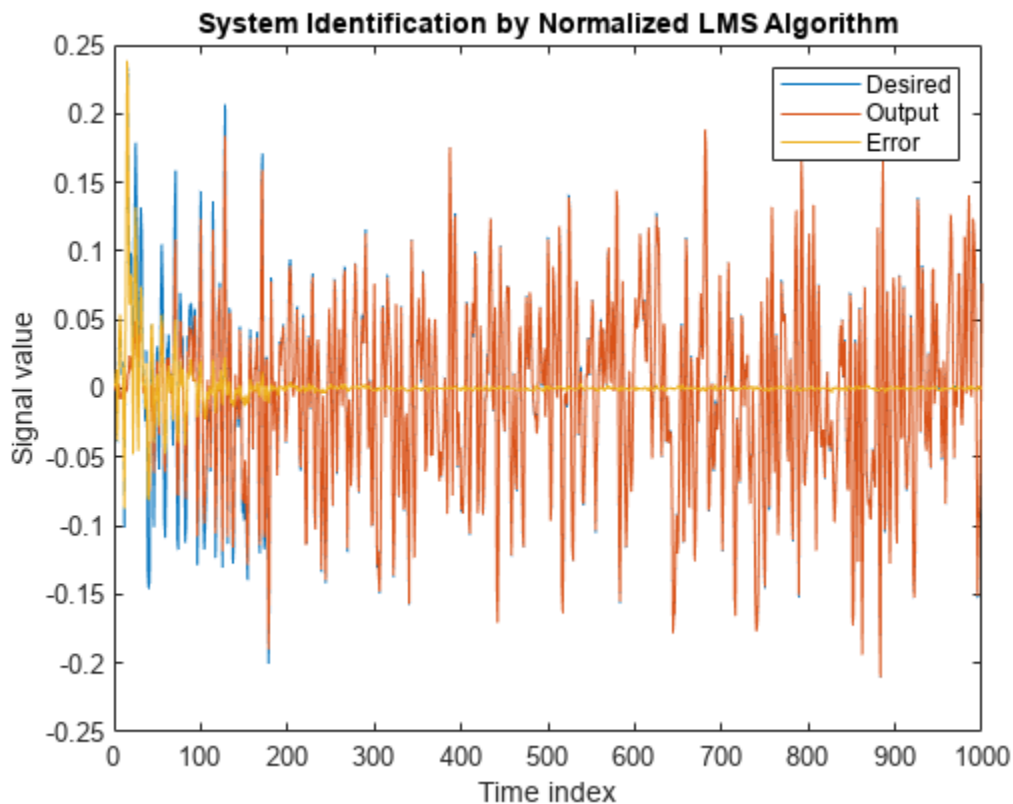
```
mu = 0.2;  
lms = dsp.LMSFilter(13, 'StepSize', mu, 'Method', ...  
    'Normalized LMS');
```

Pass the primary input signal `x` and the desired signal `d` to the LMS filter.

```
[y,e,w] = lms(x,d);
```

The output `y` of the adaptive filter is the signal converged to the desired signal `d` thereby minimizing the error `e` between the two signals.

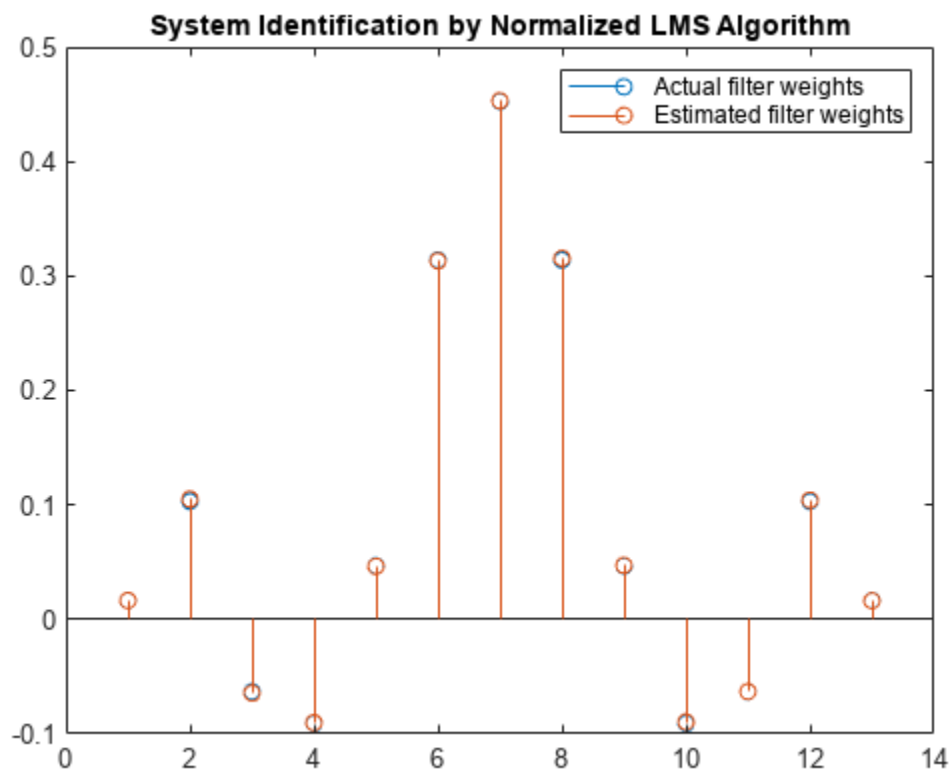
```
plot(1:1000, [d,y,e])  
title('System Identification by Normalized LMS Algorithm')  
legend('Desired', 'Output', 'Error')  
xlabel('Time index')  
ylabel('Signal value')
```



### Compare the Adapted Filter to the Unknown System

The weights vector  $w$  represents the coefficients of the LMS filter that is adapted to resemble the unknown system (FIR filter). To confirm the convergence, compare the numerator of the FIR filter and the estimated weights of the adaptive filter.

```
stem([filt.Numerator.' w])
title('System Identification by Normalized LMS Algorithm')
legend('Actual filter weights', 'Estimated filter weights', ...
       'Location', 'NorthEast')
```



### Compare Convergence Performance Between LMS Algorithm and Normalized LMS Algorithm

An adaptive filter adapts its filter coefficients to match the coefficients of an unknown system. The objective is to minimize the error signal between the output of the unknown system and the output of the adaptive filter. When these two outputs converge and match closely for the same input, the coefficients are said to match closely. The adaptive filter at this state resembles the unknown system. This example compares the rate at which this convergence happens for the normalized LMS (NLMS) algorithm and the LMS algorithm with no normalization.

#### Unknown System

Create a `dsp.FIRFilter` that represents the unknown system. Pass the signal  $x$  as an input to the unknown system. The desired signal  $d$  is the sum of the output of the unknown system (FIR filter) and an additive noise signal  $n$ .

```
filt = dsp.FIRFilter;  
filt.Numerator = fircband(12,[0 0.4 0.5 1],[1 1 0 0],[1 0.2],...  
{'w' 'c'});  
x = 0.1*randn(1000,1);  
n = 0.001*randn(1000,1);  
d = filt(x) + n;
```

### Adaptive Filter

Create two `dsp.LMSFilter` objects, with one set to the LMS algorithm, and the other set to the normalized LMS algorithm. Choose an adaptation step size of 0.2 and set the length of the adaptive filter to 13 taps.

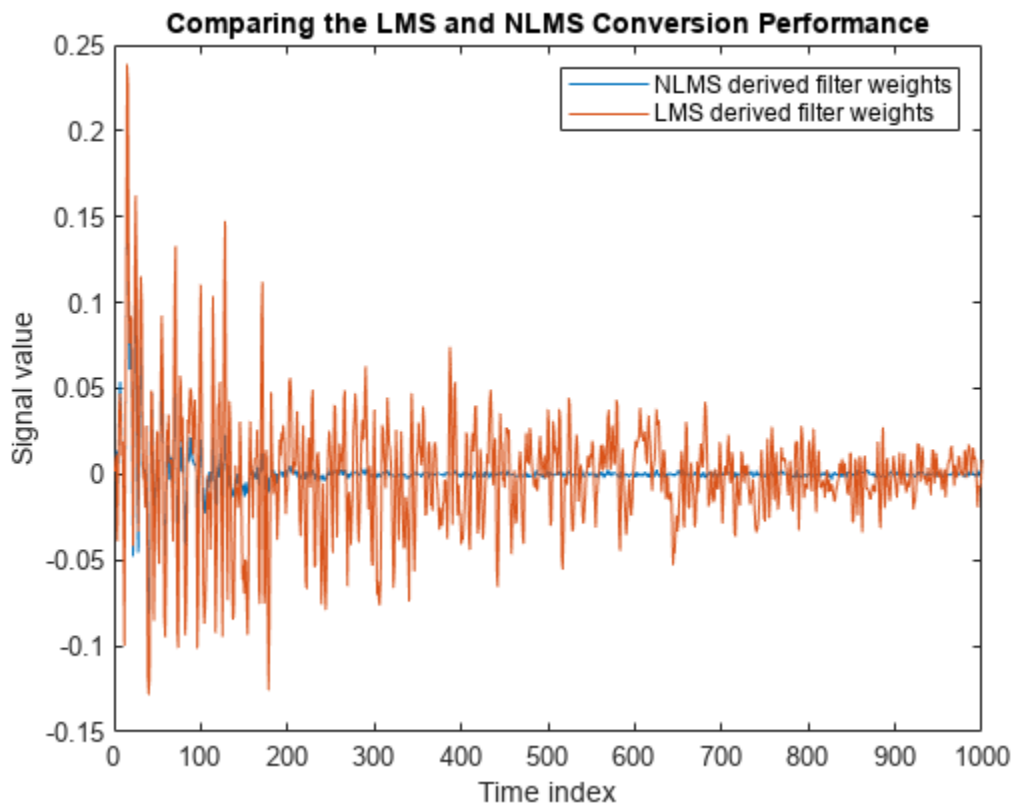
```
mu = 0.2;  
lms_nonnormalized = dsp.LMSFilter(13,'StepSize',mu,...  
    'Method','LMS');  
lms_normalized = dsp.LMSFilter(13,'StepSize',mu,...  
    'Method','Normalized LMS');
```

Pass the primary input signal `x` and the desired signal `d` to both the variations of the LMS algorithm. The variables `e1` and `e2` represent the error between the desired signal and the output of the normalized and nonnormalized filters, respectively.

```
[~,e1,~] = lms_normalized(x,d);  
[~,e2,~] = lms_nonnormalized(x,d);
```

Plot the error signals for both variations. The error signal for the NLMS variant converges to zero much faster than the error signal for the LMS variant. The normalized version adapts in far fewer iterations to a result almost as good as the nonnormalized version.

```
plot([e1,e2]);  
title('Comparing the LMS and NLMS Conversion Performance');  
legend('NLMS derived filter weights', ...  
    'LMS derived filter weights','Location','NorthEast');  
xlabel('Time index')  
ylabel('Signal value')
```



### Cancel Noise Using LMS Filter

Cancel additive noise,  $n$ , added to an unknown system using an LMS adaptive filter. The LMS filter adapts its coefficients until its transfer function matches the transfer function of the unknown system as closely as possible. The difference between the output of the adaptive filter and the output of the unknown system represents the error signal,  $e$ . Minimizing this error signal is the objective of the adaptive filter.

The unknown system and the LMS filter process the same input signal,  $x$ , and produce outputs  $d$  and  $y$ , respectively. If the coefficients of the adaptive filter match the coefficients of the unknown system, the error,  $e$ , in effect represents the additive noise.

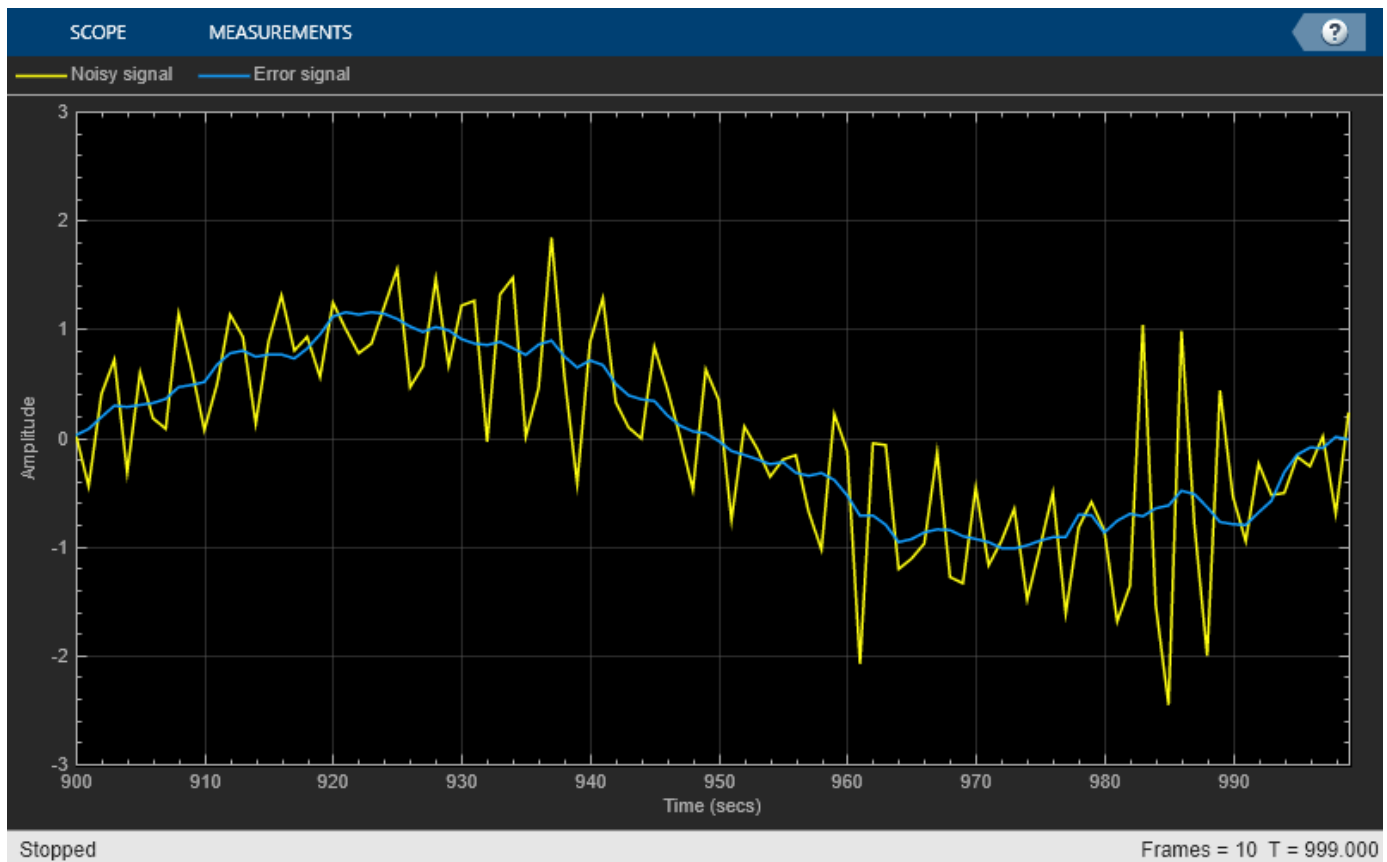
Create a `dsp.FIRFilter` System object to represent the unknown system. Create a `dsp.LMSFilter` object and set the length to 11 taps and the step size to 0.05. Create a sine wave to represent the noise added to the unknown system. View the signals in a time scope.

```
FrameSize = 100;
NIter = 10;
lmsfilt2 = dsp.LMSFilter('Length',11,'Method','Normalized LMS', ...
    'StepSize',0.05);
firfilt2 = dsp.FIRFilter('Numerator', fir1(10,[.5, .75]));
sinewave = dsp.SineWave('Frequency',0.01, ...
    'SampleRate',1,'SamplesPerFrame',FrameSize);
scope = timescope('TimeUnits','Seconds',...
    'YLimits',[-3 3],'BufferLength',2*FrameSize*NIter, ...
```

```
'ShowLegend',true,'ChannelNames', ...
{'Noisy signal', 'Error signal'});
```

Create a random input signal,  $x$  and pass the signal to the FIR filter. Add a sine wave to the output of the FIR filter to generate the noisy signal,  $d$ . The signal,  $d$  is the output of the unknown system. Pass the noisy signal and the primary input signal to the LMS filter. View the noisy signal and the error signal in the time scope.

```
for k = 1:NIter
    x = randn(FrameSize,1);
    d = firfilt2(x) + sinewave();
    [y,e,w] = lmsfilt2(x,d);
    scope([d,e])
end
release(scope)
```



The error signal,  $e$ , is the sinusoidal noise added to the unknown system. Minimizing the error signal minimizes the noise added to the system.

### Noise Cancellation Using Sign-Data LMS Algorithm

When the amount of computation required to derive an adaptive filter drives your development process, the sign-data variant of the LMS (SDLMS) algorithm might be a very good choice, as demonstrated in this example.



In the standard and normalized variations of the LMS adaptive filter, coefficients for the adapting filter arise from the mean square error between the desired signal and the output signal from the unknown system. The sign-data algorithm changes the mean square error calculation by using the sign of the input data to change the filter coefficients.

When the error is positive, the new coefficients are the previous coefficients plus the error multiplied by the step size  $\mu$ . If the error is negative, the new coefficients are again the previous coefficients minus the error multiplied by  $\mu$  — note the sign change.

When the input is zero, the new coefficients are the same as the previous set.

In vector form, the sign-data LMS algorithm is:

$$w(k+1) = w(k) + \mu e(k) \text{sgn}(x(k)),$$

where

$$\text{sgn}(x(k)) = \begin{cases} 1, & x(k) > 0 \\ 0, & x(k) = 0 \\ -1, & x(k) < 0 \end{cases}$$

with vector  $w$  containing the weights applied to the filter coefficients and vector  $x$  containing the input data. The vector  $e$  is the error between the desired signal and the filtered signal. The objective of the SDLMS algorithm is to minimize this error. Step size is represented by  $\mu$ .

With a smaller  $\mu$ , the correction to the filter weights gets smaller for each sample, and the SDLMS error falls more slowly. A larger  $\mu$  changes the weights more for each step, so the error falls more rapidly, but the resulting error does not approach the ideal solution as closely. To ensure a good convergence rate and stability, select  $\mu$  within the following practical bounds.

$$0 < \mu < \frac{1}{N\{\text{InputSignalPower}\}},$$

where  $N$  is the number of samples in the signal. Also, define  $\mu$  as a power of two for efficient computing.

**Note:** How you set the initial conditions of the sign-data algorithm profoundly influences the effectiveness of the adaptation process. Because the algorithm essentially quantizes the input signal, the algorithm can become unstable easily.

A series of large input values, coupled with the quantization process might result in the error growing beyond all bounds. Restrain the tendency of the sign-data algorithm to get out of control by choosing a small step size ( $\mu \ll 1$ ) and setting the initial conditions for the algorithm to nonzero positive and negative values.

In this noise cancellation example, set the Method property of `dsp.LMSFilter` to 'Sign-Data LMS'. This example requires two input data sets:

- Data containing a signal corrupted by noise. In the block diagram under “Noise or Interference Cancellation -- Using an Adaptive Filter to Remove Noise from an Unknown System”, this is the desired signal  $d(k)$ . The noise cancellation process removes the noise from the signal.
- Data containing random noise. In the block diagram under “Noise or Interference Cancellation -- Using an Adaptive Filter to Remove Noise from an Unknown System”, this is  $x(k)$ . The signal  $x(k)$

is correlated with the noise that corrupts the signal data. Without the correlation between the noise data, the adapting algorithm cannot remove the noise from the signal.

For the signal, use a sine wave. Note that `signal` is a column vector of 1000 elements.

```
signal = sin(2*pi*0.055*(0:1000-1)');
```

Now, add correlated white noise to `signal`. To ensure that the noise is correlated, pass the noise through a lowpass FIR filter and then add the filtered noise to the signal.

```
noise = randn(1000,1);  
filt = dsp.FIRFilter;  
filt.Numerator = fir1(11,0.4);  
fnoise = filt(noise);  
d = signal + fnoise;
```

`fnoise` is the correlated noise and `d` is now the desired input to the sign-data algorithm.

To prepare the `dsp.LMSFilter` object for processing, set the initial conditions of the filter weights and `mu` (`StepSize`). As noted earlier in this section, the values you set for `coeffs` and `mu` determine whether the adaptive filter can remove the noise from the signal path.

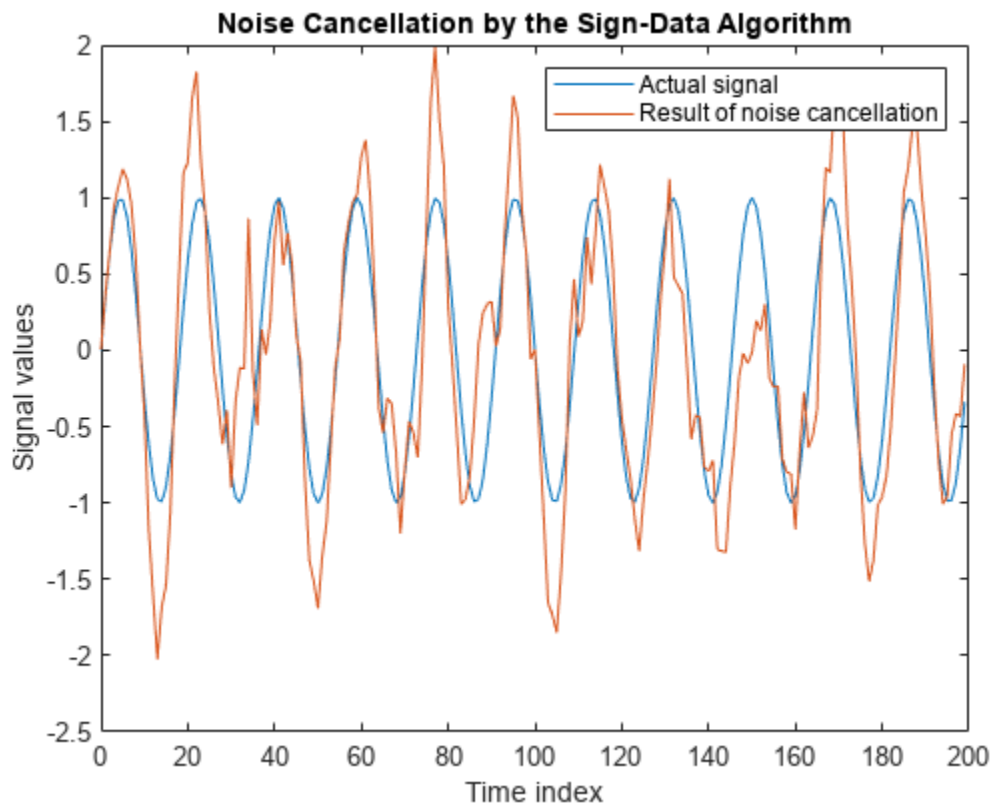
In “System Identification of FIR Filter Using LMS Algorithm” on page 4-742, you constructed a default filter that sets the filter coefficients to zeros. In most cases that approach does not work for the sign-data algorithm. The closer you set your initial filter coefficients to the expected values, the more likely it is that the algorithm remains well behaved and converges to a filter solution that removes the noise effectively.

For this example, start with the coefficients used in the noise filter (`filt.Numerator`), and modify them slightly so the algorithm has to adapt.

```
coeffs = (filt.Numerator).'-0.01; % Set the filter initial conditions.  
mu = 0.05; % Set the step size for algorithm updating.
```

With the required input arguments for `dsp.LMSFilter` prepared, construct the LMS filter object, run the adaptation, and view the results.

```
lms = dsp.LMSFilter(12,'Method','Sign-Data LMS',...  
    'StepSize',mu,'InitialConditions',coeffs);  
[~,e] = lms(noise,d);  
L = 200;  
plot(0:L-1,signal(1:L),0:L-1,e(1:L));  
title('Noise Cancellation by the Sign-Data Algorithm');  
legend('Actual signal','Result of noise cancellation',...  
    'Location','NorthEast');  
xlabel('Time index')  
ylabel('Signal values')
```



When `dsp.LMSFilter` runs, it uses far fewer multiplication operations than either of the standard LMS algorithms. Also, performing the sign-data adaptation requires only multiplication by bit shifting when the step size is a power of two.

Although the performance of the sign-data algorithm as shown in this plot is quite good, the sign-data algorithm is much less stable than the standard LMS variations. In this noise cancellation example, the processed signal is a very good match to the input signal, but the algorithm could very easily grow without bound rather than achieve good performance.

Changing the weight initial conditions (`InitialConditions`) and  $\mu$  (`StepSize`), or even the lowpass filter you used to create the correlated noise, can cause noise cancellation to fail.

### Noise Cancellation Using Sign-Error LMS Algorithm

In the standard and normalized variations of the LMS adaptive filter, the coefficients for the adapting filter arise from calculating the mean square error between the desired signal and the output signal from the unknown system, and applying the result to the current filter coefficients. The sign-error LMS (SELMS) algorithm replaces the mean square error calculation by using the sign of the error to modify the filter coefficients.

When the error is positive, the new coefficients are the previous coefficients plus the error multiplied by the step size  $\mu$ . If the error is negative, the new coefficients are the previous coefficients minus the error multiplied by  $\mu$  — note the sign change. When the input is zero, the new coefficients are the same as the previous set.

In vector form, the sign-error LMS algorithm is:

$$w(k+1) = w(k) + \mu \text{sgn}(e(k))x(k),$$

where

$$\text{sgn}(e(k)) = \begin{cases} 1, & e(k) > 0 \\ 0, & e(k) = 0 \\ -1, & e(k) < 0 \end{cases}$$

with vector  $w$  containing the weights applied to the filter coefficients and vector  $x$  containing the input data. The vector  $e$  is the error between the desired signal and the filtered signal. The objective of the SELMS algorithm is to minimize this error.

With a smaller  $\mu$ , the correction to the filter weights gets smaller for each sample and the SELMS error falls more slowly. A larger  $\mu$  changes the weights more for each step so the error falls more rapidly, but the resulting error does not approach the ideal solution as closely. To ensure a good convergence rate and stability, select  $\mu$  within the following practical bounds.

$$0 < \mu < \frac{1}{N\{\text{InputSignalPower}\}}$$

where  $N$  is the number of samples in the signal. Also, define  $\mu$  as a power of two for efficient computation.

**Note:** How you set the initial conditions of the sign-error algorithm profoundly influences the effectiveness of the adaptation process. Because the algorithm essentially quantizes the error signal, the algorithm can become unstable easily.

A series of large error values, coupled with the quantization process might result in the error growing beyond all bounds. Restrain the tendency of the sign-error algorithm to become unstable by choosing a small step size ( $\mu \ll 1$ ) and setting the initial conditions for the algorithm to nonzero positive and negative values.

In this noise cancellation example, set the Method property of `dsp.LMSFilter` to 'Sign-Error LMS'. This example requires two input data sets:

- Data containing a signal corrupted by noise. In the block diagram under “Noise or Interference Cancellation -- Using an Adaptive Filter to Remove Noise from an Unknown System”, this is the desired signal  $d(k)$ . The noise cancellation process removes the noise from the signal.
- Data containing random noise. In the block diagram under “Noise or Interference Cancellation -- Using an Adaptive Filter to Remove Noise from an Unknown System”, this is  $x(k)$ . The signal  $x(k)$  is correlated with the noise that corrupts the signal data. Without the correlation between the noise data, the adapting algorithm cannot remove the noise from the signal.

For the signal, use a sine wave. Note that `signal` is a column vector of 1000 elements.

```
signal = sin(2*pi*0.055*(0:1000-1)');
```

Now, add correlated white noise to `signal`. To ensure that the noise is correlated, pass the noise through a lowpass FIR filter and then add the filtered noise to the signal.

```
noise = randn(1000,1);
filt = dsp.FIRFilter;
```

```
filt.Numerator = fir1(11,0.4);
fnoise = filt(noise);
d = signal + fnoise;
```

fnoise is the correlated noise and d is now the desired input to the sign-error algorithm.

To prepare the dsp.LMSFilter object for processing, set the initial conditions of the filter weights (InitialConditions) and mu (StepSize). As noted earlier in this section, the values you set for coeffs and mu determine whether the adaptive filter can remove the noise from the signal path.

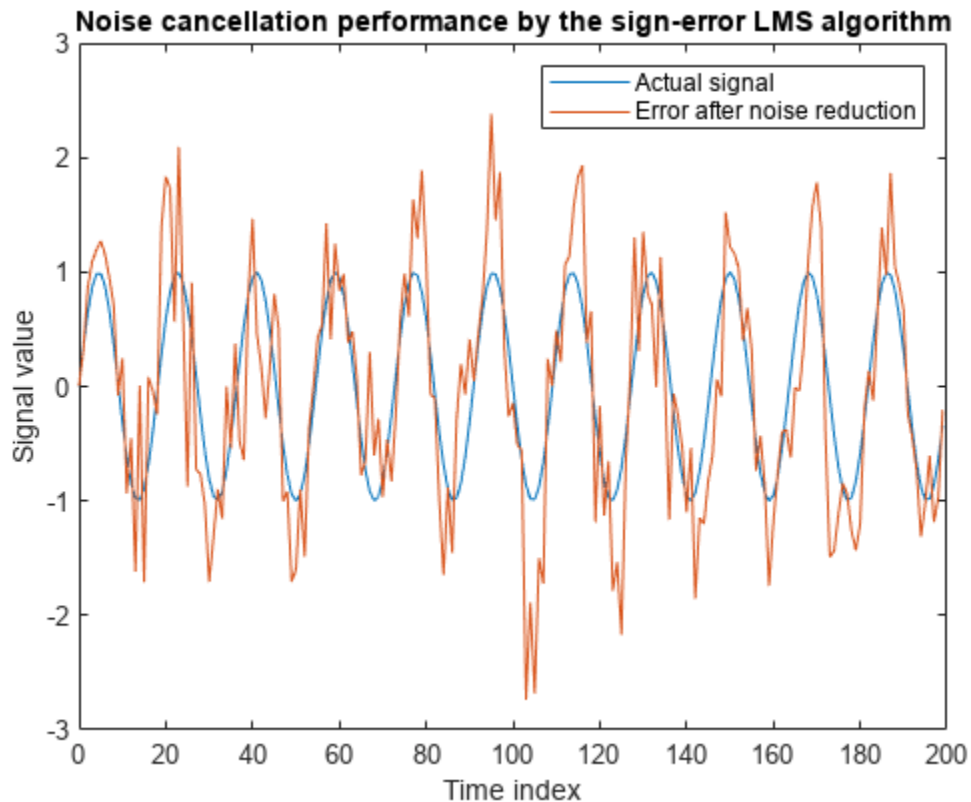
In “System Identification of FIR Filter Using LMS Algorithm” on page 4-742, you constructed a default filter that sets the filter coefficients to zeros. In most cases that approach does not work for the sign-error algorithm. The closer you set your initial filter coefficients to the expected values, the more likely it is that the algorithm remains well behaved and converges to a filter solution that removes the noise effectively.

For this example, start with the coefficients used in the noise filter (filt.Numerator) and modify them slightly so the algorithm has to adapt.

```
coeffs = (filt.Numerator).'-0.01; % Set the filter initial conditions.
mu = 0.05; % Set the step size for algorithm updating.
```

With the required input arguments for dsp.LMSFilter prepared, run the adaptation and view the results.

```
lms = dsp.LMSFilter(12,'Method','Sign-Error LMS',...
    'StepSize',mu,'InitialConditions',coeffs);
[~,e] = lms(noise,d);
L = 200;
plot(0:199,signal(1:200),0:199,e(1:200));
title('Noise cancellation performance by the sign-error LMS algorithm');
legend('Actual signal','Error after noise reduction',...
    'Location','NorthEast')
xlabel('Time index')
ylabel('Signal value')
```



When the sign-error LMS algorithm runs, it uses far fewer multiplication operations than either of the standard LMS algorithms. Also, performing the sign-error adaptation requires only bit shifting multiples when the step size is a power of two.

Although the performance of the sign-error algorithm as shown in this plot is quite good, the sign-error algorithm is much less stable than the standard LMS variations. In this noise cancellation example, the adapted signal is a very good match to the input signal, but the algorithm could very easily become unstable rather than achieve good performance.

Changing the weight initial conditions (`InitialConditions`) and  $\mu$  (`StepSize`), or even the lowpass filter you used to create the correlated noise, can cause noise cancellation to fail and the algorithm to become useless.

### Noise Cancellation Using Sign-Sign LMS Algorithm

The sign-sign LMS algorithm (SSLMS) replaces the mean square error calculation by using the sign of the input data to change the filter coefficients. When the error is positive, the new coefficients are the previous coefficients plus the error multiplied by the step size  $\mu$ . If the error is negative, the new coefficients are the previous coefficients minus the error multiplied by  $\mu$  — note the sign change. When the input is zero, the new coefficients are the same as the previous set.

In essence, the algorithm quantizes both the error and the input by applying the sign operator to them.

In vector form, the sign-sign LMS algorithm is:

$$w(k+1) = w(k) + \mu \text{sgn}(e(k)) \text{sgn}(x(k)),$$

where

$$\text{sgn}(z(k)) = \begin{cases} 1, & z(k) > 0 \\ 0, & z(k) = 0 \\ -1, & z(k) < 0 \end{cases}$$

$$z(k) = e(k) \text{sgn}(x(k))$$

Vector  $w$  contains the weights applied to the filter coefficients and vector  $x$  contains the input data. The vector  $e$  is the error between the desired signal and the filtered signal. The objective of the SSLMS algorithm is to minimize this error.

With a smaller  $\mu$ , the correction to the filter weights gets smaller for each sample and the SSLMS error falls more slowly. A larger  $\mu$  changes the weights more for each step, so the error falls more rapidly, but the resulting error does not approach the ideal solution as closely. To ensure a good convergence rate and stability, select  $\mu$  within the following practical bounds.

$$0 < \mu < \frac{1}{N\{\text{InputSignalPower}\}}$$

where  $N$  is the number of samples in the signal. Also, define  $\mu$  as a power of two for efficient computation

**Note:**

How you set the initial conditions of the sign-sign algorithm profoundly influences the effectiveness of the adaptation process. Because the algorithm essentially quantizes the input signal and the error signal, the algorithm can become unstable easily.

A series of large error values, coupled with the quantization process might result in the error growing beyond all bounds. Restrain the tendency of the sign-sign algorithm to become unstable by choosing a small step size ( $\mu \ll 1$ ) and setting the initial conditions for the algorithm to nonzero positive and negative values.

In this noise cancellation example, set the `Method` property of `dsp.LMSFilter` to 'Sign-Sign LMS'. This example requires two input data sets:

- Data containing a signal corrupted by noise. In the block diagram under "Noise or Interference Cancellation -- Using an Adaptive Filter to Remove Noise from an Unknown System", this is the desired signal  $d(k)$ . The noise cancellation process removes the noise from the signal.
- Data containing random noise. In the block diagram under "Noise or Interference Cancellation -- Using an Adaptive Filter to Remove Noise from an Unknown System", this is  $x(k)$ . The signal  $x(k)$  is correlated with the noise that corrupts the signal data. Without the correlation between the noise data, the adapting algorithm cannot remove the noise from the signal.

For the signal, use a sine wave. Note that `signal` is a column vector of 1000 elements.

```
signal = sin(2*pi*0.055*(0:1000-1)');
```

Now, add correlated white noise to `signal`. To ensure that the noise is correlated, pass the noise through a lowpass FIR filter, then add the filtered noise to the signal.

```
noise = randn(1000,1);
filt = dsp.FIRFilter;
filt.Numerator = fir1(11,0.4);
fnoise = filt(noise);
d = signal + fnoise;
```

`fnoise` is the correlated noise and `d` is now the desired input to the sign-sign algorithm.

To prepare the `dsp.LMSFilter` object for processing, set the initial conditions of the filter weights (`InitialConditions`) and `mu` (`StepSize`). As noted earlier in this section, the values you set for `coeffs` and `mu` determine whether the adaptive filter can remove the noise from the signal path. In “System Identification of FIR Filter Using LMS Algorithm” on page 4-742, you constructed a default filter that sets the filter coefficients to zeros. Usually that approach does not work for the sign-sign algorithm.

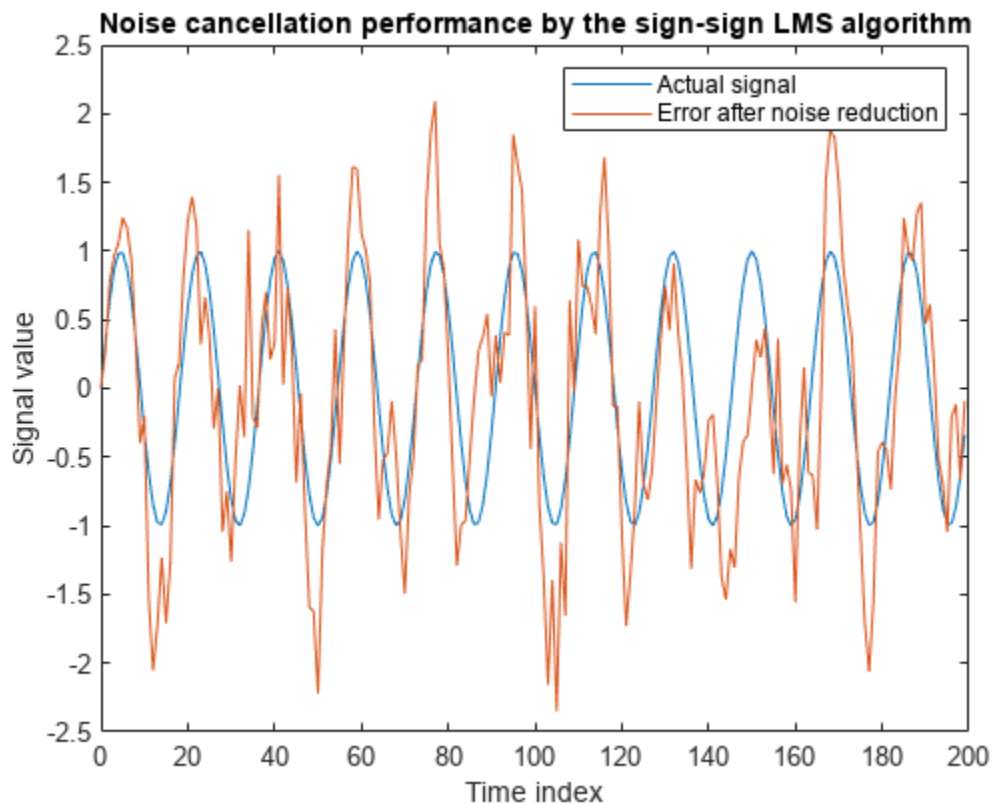
The closer you set your initial filter coefficients to the expected values, the more likely it is that the algorithm remains well behaved and converges to a filter solution that removes the noise effectively. For this example, you start with the coefficients used in the noise filter (`filt.Numerator`), and modify them slightly so the algorithm has to adapt.

```
coeffs = (filt.Numerator).' -0.01; % Set the filter initial conditions.
mu = 0.05;
```

With the required input arguments for `dsp.LMSFilter` prepared, run the adaptation and view the results.

```
lms = dsp.LMSFilter(12,'Method','Sign-Sign LMS',...
    'StepSize',mu,'InitialConditions',coeffs);
[~,e] = lms(noise,d);
L = 200;
plot(0:199,signal(1:200),0:199,e(1:200));
title('Noise cancellation performance by the sign-sign LMS algorithm');
legend('Actual signal','Error after noise reduction',...
    'Location','NorthEast')
xlabel('Time index')
ylabel('Signal value')
```





When `dsp.LMSFilter` runs, it uses far fewer multiplication operations than either of the standard LMS algorithms. Also, performing the sign-sign adaptation requires only bit shifting multiples when the step size is a power of two.

Although the performance of the sign-sign algorithm as shown in this plot is quite good, the sign-sign algorithm is much less stable than the standard LMS variations. In this noise cancellation example, the adapted signal is a very good match to the input signal, but the algorithm could very easily become unstable rather than achieve good performance.

Changing the weight initial conditions (`InitialConditions`) and  $\mu$  (`StepSize`), or even the lowpass filter you used to create the correlated noise, can cause noise cancellation to fail and the algorithm to become useless.

### Access Full History of LMS Filter Weights

**Note:** This example runs only in R2017a or later. If you are using a release earlier than R2017a, the object does not output a full sample-by-sample history of filter weights.

Initialize the `dsp.LMSFilter` System object and set the `WeightsOutput` property to `'All'`. This setting enables the LMS filter to output a matrix of weights with dimensions `[FrameLength Length]`, corresponding to the full sample-by-sample history of weights for all `FrameLength` samples of input values.

```
FrameSize = 15000;
lmsfilt3 = dsp.LMSFilter('Length',63,'Method','LMS', ...
    'StepSize',0.001,'LeakageFactor',0.99999, ...
    'WeightsOutput','All'); % full Weights history

w_actual = fir1(64,[0.5 0.75]);
firfilt3 = dsp.FIRFilter('Numerator',w_actual);
sinewave = dsp.SineWave('Frequency',0.01, ...
    'SampleRate',1,'SamplesPerFrame',FrameSize);

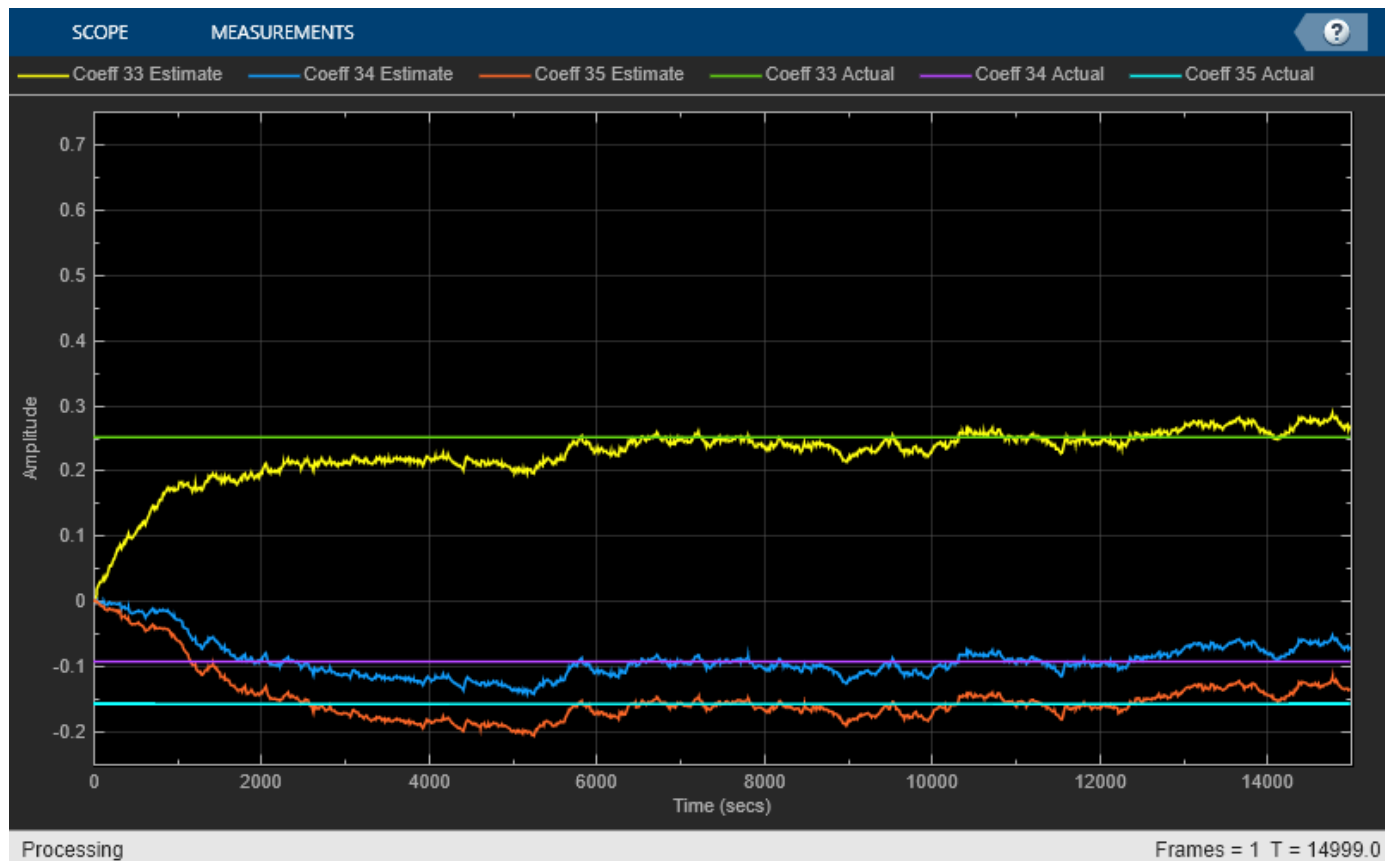
scope = timescope('TimeUnits','Seconds', ...
    'YLimits',[-0.25 0.75],'BufferLength',2*FrameSize, ...
    'ShowLegend',true,'ChannelNames', ...
    {'Coeff 33 Estimate','Coeff 34 Estimate','Coeff 35 Estimate', ...
    'Coeff 33 Actual','Coeff 34 Actual','Coeff 35 Actual'});
```

Run one frame and output the full adaptive weights history, w.

```
x = randn(FrameSize,1); % Input signal
d = firfilt3(x) + sinewave(); % Noise + Signal
[~,~,w] = lmsfilt3(x,d);
```

Each row in w is a set of weights estimated for the respective input sample. Each column in w gives the complete history of a specific weight. Plot the actual weight and the entire history of the 33rd, 34th, and 35th weight. In the plot, you can see that the estimated weight output eventually converges with the actual weight as the adaptive filter receives input samples and continues to adapt.

```
idxBeg = 33;
idxEnd = 35;
scope([w(:,idxBeg:idxEnd), repmat(w_actual(idxBeg:idxEnd),FrameSize,1)])
```



## More About

### Fixed Point

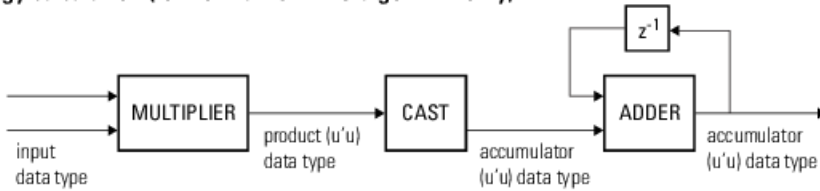
The following diagrams show the data types used within the `dsp.LMSFilter` object for fixed-point signals. The table summarizes the definitions of the variables used in the diagrams:

| Variable                            | Definition  |
|-------------------------------------|---|
| $\mathbf{u}$                        | Input vector  |
| $\mathbf{W}$                        | Vector of filter weights                                    |
| $\mu$                               | Step size   |
| $e$                                 | Error   |
| $Q$                                 | Quotient, $Q = \frac{\mu \cdot e}{\mathbf{u}'\mathbf{u}}$   |
| Product $\mathbf{u}'\mathbf{u}$     | Product data type in Energy calculation diagram             |
| Accumulator $\mathbf{u}'\mathbf{u}$ | Accumulator data type in Energy calculation diagram         |
| Product $\mathbf{W}'\mathbf{u}$     | Product data type in Convolution diagram                    |
| Accumulator $\mathbf{W}'\mathbf{u}$ | Accumulator data type in Convolution diagram                |
| Product $\mu \cdot e$               | Product data type in Product of step size and error diagram |

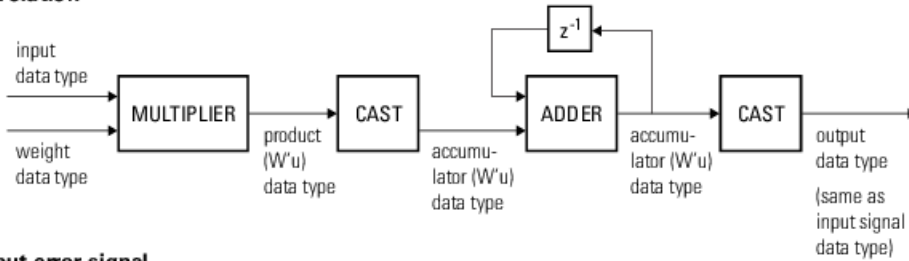
| Variable            | Definition   |
|---------------------|--|
| Product $Q \cdot u$ | Product and accumulator data type in Weight update diagram. <sup>1</sup> |

<sup>1</sup>The accumulator data type for this quantity is automatically set to be the same as the product data type. The minimum, maximum, and overflow information for this accumulator is logged as part of the product information. Autoscaling treats this product and accumulator as one data type.

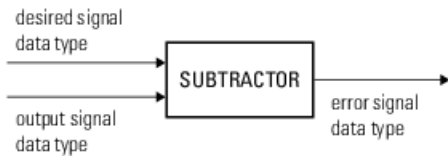
**Energy calculation (for normalized LMS algorithm only)**

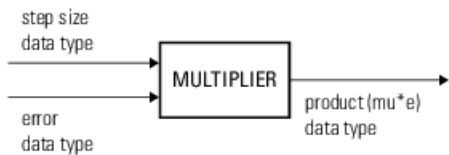
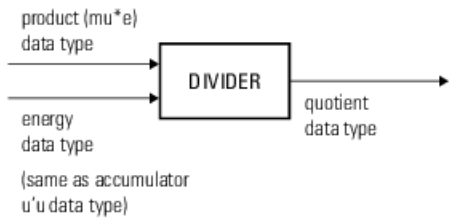
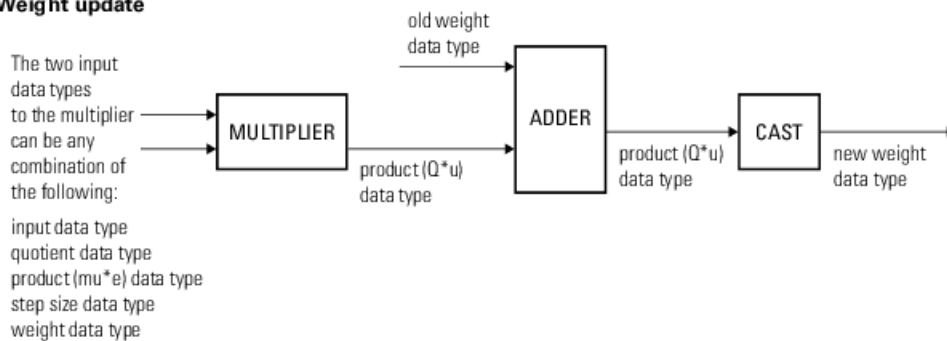


**Convolution**



**Output error signal**



**Product of step size and error (for LMS and Sign-Data LMS algorithms only)****Quotient (for normalized LMS only)****Weight update**

You can set the data type of the properties, weights, products, quotient, and accumulators in the System object properties. Fixed-point inputs, outputs, and System object properties must have the following characteristics:

- The input signal and the desired signal must have the same word length, but their fraction lengths can differ.
- The step size and leakage factor must have the same word length, but their fraction lengths can differ.
- The output signal and the error signal have the same word length and the same fraction length as the desired signal.
- The quotient and the product output of the  $\mathbf{u}'\mathbf{u}$ ,  $\mathbf{W}'\mathbf{u}$ ,  $\mu \cdot e$ , and  $Q \cdot \mathbf{u}$  operations must have the same word length, but their fraction lengths can differ.
- The accumulator data type of the  $\mathbf{u}'\mathbf{u}$  and  $\mathbf{W}'\mathbf{u}$  operations must have the same word length, but their fraction lengths can differ.

The output of the multiplier is in the product output data type if at least one of the inputs to the multiplier is real. If both of the inputs to the multiplier are complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication performed, see “Multiplication Data Types”.

## Algorithms

The LMS filter algorithm is defined by the following equations.

$$\begin{aligned}y(n) &= \mathbf{w}^T(n-1)\mathbf{u}(n) \\e(n) &= d(n) - y(n) \\ \mathbf{w}(n) &= \alpha\mathbf{w}(n-1) + f(\mathbf{u}(n), e(n), \mu)\end{aligned}$$

The various LMS adaptive filter algorithms available in this System object are defined as:

- LMS -- Solves the Wiener-Hopf equation and finds the filter coefficients for an adaptive filter.

$$f(\mathbf{u}(n), e(n), \mu) = \mu e(n)\mathbf{u}^*(n)$$

- Normalized LMS -- Normalized variation of the LMS algorithm.

$$f(\mathbf{u}(n), e(n), \mu) = \mu e(n) \frac{\mathbf{u}^*(n)}{\varepsilon + \mathbf{u}^H(n)\mathbf{u}(n)}$$

In Normalized LMS, to overcome potential numerical instability in the update of the weights, a small positive constant,  $\varepsilon$ , has been added in the denominator. For double-precision floating-point input,  $\varepsilon$  is the output of the `eps` function. For single-precision floating-point input,  $\varepsilon$  is the output of `eps('single')`. For fixed-point input,  $\varepsilon$  is 0.

- Sign-Data LMS -- Correction to the filter weights at each iteration depends on the sign of the input  $\mathbf{u}(n)$ .

$$f(\mathbf{u}(n), e(n), \mu) = \mu e(n)\text{sign}(\mathbf{u}(n))$$

where  $\mathbf{u}(n)$  is real.

- Sign-Error LMS -- Correction applied to the current filter weights for each successive iteration depends on the sign of the error,  $e(n)$ .

$$f(\mathbf{u}(n), e(n), \mu) = \mu \text{sign}(e(n))\mathbf{u}^*(n)$$

- Sign-Sign LMS -- Correction applied to the current filter weights for each successive iteration depends on both the sign of  $\mathbf{u}(n)$  and the sign of  $e(n)$ .

$$f(\mathbf{u}(n), e(n), \mu) = \mu \text{sign}(e(n))\text{sign}(\mathbf{u}(n))$$

where  $\mathbf{u}(n)$  is real.

The variables are as follows:

| Variable          | Description   |
|-------------------|---|
| $n$               | The current time index  |
| $\mathbf{u}(n)$   | The vector of buffered input samples at step $n$                          |
| $\mathbf{u}^*(n)$ | The complex conjugate of the vector of buffered input samples at step $n$ |
| $\mathbf{w}(n)$   | The vector of filter weight estimates at step $n$                         |
| $y(n)$            | The filtered output at step $n$   |
| $e(n)$            | The estimation error at step $n$  |
| $d(n)$            | The desired response at step $n$  |

| Variable      | Description  |
|---------------|--|
| $\mu$         | The adaptation step size   |
| $\alpha$      | The leakage factor ( $0 < \alpha \leq 1$ )   |
| $\varepsilon$ | A constant that corrects any potential numerical instability that occurs during the update of weights. |

## Version History

Introduced in R2012a

## References

[1] Hayes, M.H. *Statistical Digital Signal Processing and Modeling*. New York: John Wiley & Sons, 1996.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

The dsp.LMSFilter System object supports SIMD code generation using Intel AVX2 technology under these conditions:

- Method is set to 'LMS' or 'Normalized LMS'.
- WeightsOutput is set to 'None' or 'Last'.
- Input signal is real-valued.
- Input signal has a data type of single or double.

The SIMD technology significantly improves the performance of the generated code.

## See Also

### Functions

msepred | msesim | maxstep

### Objects

dsp.BlockLMSFilter | dsp.FIRFilter | dsp.FilteredXLMSFilter |  
 dsp.FrequencyDomainAdaptiveFilter | dsp.AdaptiveLatticeFilter |  
 dsp.AffineProjectionFilter | dsp.FastTransversalFilter | dsp.RLSFilter

### Blocks

LMS Filter | LMS Update

### Topics

“Overview of Adaptive Filters and Applications”

“Signal Enhancement Using LMS and NLMS Algorithms”  
“Variable-Size Signal Support DSP System Objects”



# **dsp.LogicAnalyzer**

**Package:** dsp

Visualize, measure, and analyze transitions and states over time

## **Description**

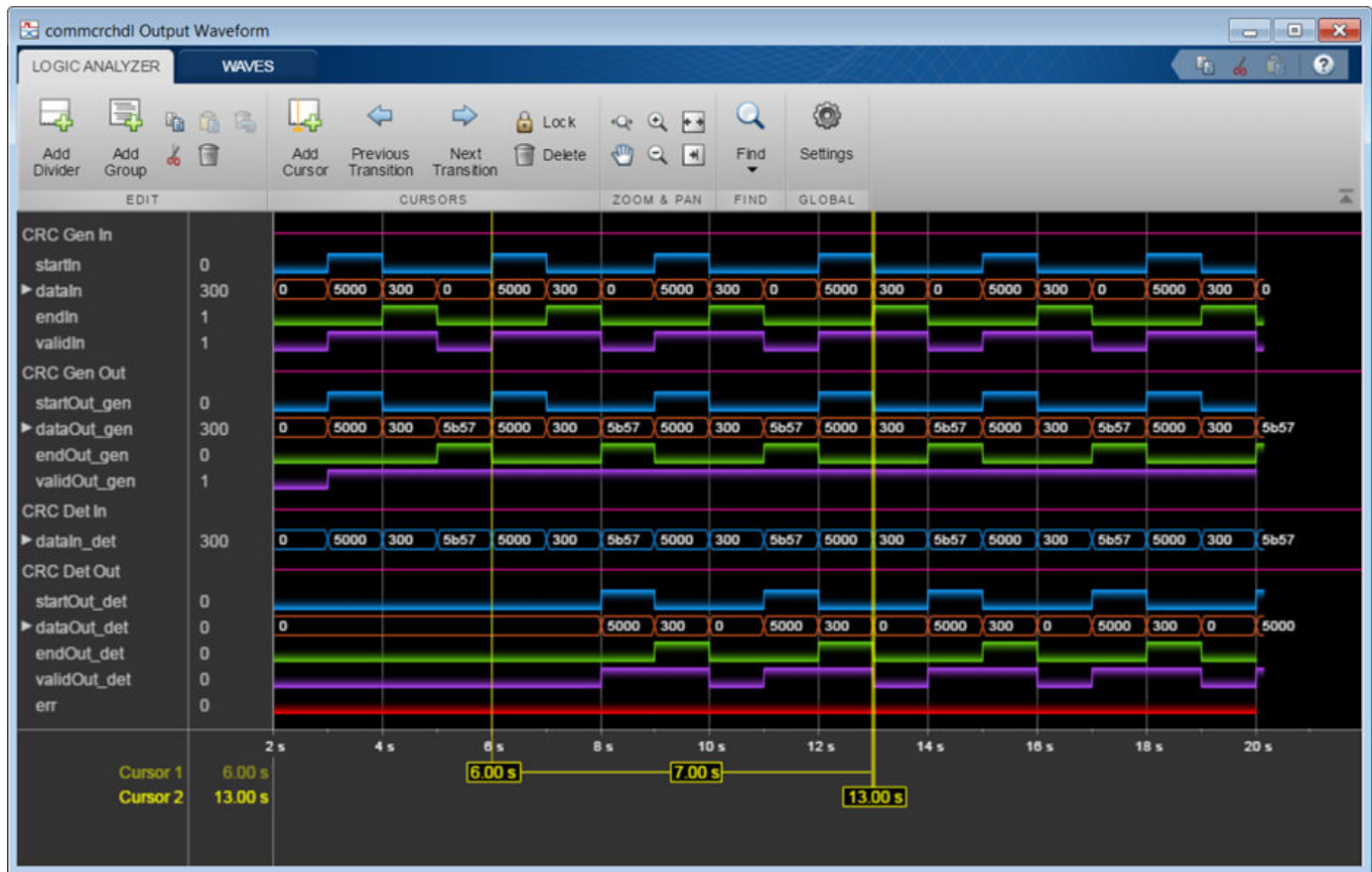
The Logic Analyzer System object displays the transitions in time-domain signals. Using `dsp.LogicAnalyzer`, you can:

- Debug and analyze models
- Trace and correlate 96 signals simultaneously
- Detect and analyze timing violations
- Trace system execution
- Detect signal changes using triggers

To display the transitions of signals in the Logic Analyzer:

- 1** Create the `dsp.LogicAnalyzer` object and set its properties.
- 2** Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)



For more information about how to configure and customize the Logic Analyzer, see **Logic Analyzer**.

## Creation

### Syntax

```
scope = dsp.LogicAnalyzer
scope = dsp.LogicAnalyzer(Name,Value)
```

### Description

`scope = dsp.LogicAnalyzer` creates a Logic Analyzer System object, `scope`.

`scope = dsp.LogicAnalyzer(Name,Value)` sets properties using one or more name-value pairs. Enclose each property name in single quotes. For example, `scope = dsp.LogicAnalyzer('BackgroundColor','White','NumInputPorts',4)`.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

### **BackgroundColor** — Background color for display

'Black' (default) | 'White'

Background color of the display, specified as 'Black' or 'White'.

**Tunable:** Yes

Data Types: char | string

### **DisplayChannelColor** — Color for channels in display

[0.0588 1 1] (default) | RGB triplet

Color for channels in the display, specified as an RGB triplet.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0,1]; for example, [0.4 0.6 0.7].

**Tunable:** Yes

Data Types: double | single | uint8 | uint16 | uint32 | uint64 | int8 | int16 | int32 | int64

### **DisplayChannelFontSize** — Font size for channels in display

10 (default) | nonnegative scalar integer

Font size for channels in the display, in points, specified as a nonnegative integer.

**Tunable:** Yes

Data Types: double | single | uint8 | uint16 | uint32 | uint64 | int8 | int16 | int32 | int64

### **DisplayChannelFormat** — Format for channels in display

'Automatic' (default) | 'Analog' | 'Digital'

Format for channels in the display, specified as one of the following:

- 'Automatic' — Displays floating-point signals in `Analog` format and integer and fixed-point signals in `Digital` format. Boolean signals are displayed as zero or one.
- 'Analog' — Shows values as an analog plot.
- 'Digital' — Shows values as digital transitions.

**Tunable:** Yes

Data Types: char | string

### **DisplayChannelHeight** — Channel height in display

12 (default) | positive real scalar

Channel height in the display, in pixels, specified as a positive real scalar in the range [8, 200].

**Tunable:** Yes

Data Types: double | single | uint8 | uint16 | uint32 | uint64 | int8 | int16 | int32 | int64

**DisplayChannelRadix — Base used to display values**

'Hexadecimal' (default) | 'Binary' | 'Octal' | 'Signed decimal' | 'Unsigned decimal'

This property applies only to fixed-point (fi) values.

**Tunable:** Yes

Data Types: char | string

**DisplayChannelSpacing — Spacing between channels in display (pixels)**

4 (default) | positive integer

Spacing between channels in the display, in pixels, specified as a positive scalar integer.

**Tunable:** Yes

Data Types: double | single | uint8 | uint16 | uint32 | uint64 | int8 | int16 | int32 | int64

**Name — Caption to display on scope window**

'Logic Analyzer' (default) | character vector | string scalar

Caption to display on the scope window, specified as a character vector or string.

**Tunable:** Yes

Data Types: char | string

**NumInputPorts — Number of input ports**

1 (default) | integer between [1, 96]

Number of input ports, specified as a positive integer. Each signal coming through a separate input becomes a separate channel in the scope. You must invoke the scope with the same number of inputs as the value of this property.

**Position — Scope window position**

[left bottom width height] vector

Position of the scope window on your screen, in pixels, specified as a [left bottom width height] vector. The default position depends on your screen resolution. By default, the scope window appears in the center of your screen, with a width of 800 pixels and height of 600 pixels.

**Tunable:** Yes

Data Types: double | single | uint8 | uint16 | uint32 | uint64 | int8 | int16 | int32 | int64

**SampleTime — Input sample time**

1 (default) | scalar

Sample time of inputs in seconds, specified as a finite numeric scalar. The same sample time is used for all inputs.

Data Types: double | single | uint8 | uint16 | uint32 | uint64 | int8 | int16 | int32 | int64

**TimeDisplayOffset — Time display offset**

0 (default) | nonnegative scalar

Time display offset in seconds, specified as a nonnegative scalar.

**Tunable:** Yes

Data Types: double | single | uint8 | uint16 | uint32 | uint64 | int8 | int16 | int32 | int64

### **TimeSpan — Time span**

10 | positive scalar

Time span in seconds, specified as a positive scalar. The x-axis limits are calculated as follows:

- Minimum x-axis limit = `min(TimeDisplayOffset)`
- Maximum x-axis limit = `max(TimeDisplayOffset) + TimeSpan`

`TimeDisplayOffset` and `TimeSpan` are the values of their respective properties.

**Tunable:** Yes

Data Types: double | single | uint8 | uint16 | uint32 | uint64 | int8 | int16 | int32 | int64

## **Usage**

### **Syntax**

```
scope(signal)
scope(signal1,signal2,...signalN)
```

### **Description**

`scope(signal)` displays the signal `signal` in the Logic Analyzer scope.

`scope(signal1,signal2,...signalN)` displays multiple signals in the Logic Analyzer when you set the `NumInputPorts` property to `N`. Each signal can have different data types and dimensions.

### **Input Arguments**

#### **signal — Input signal or signals to visualize**

scalar | vector | matrix

Specify one or more input signals to visualize in the `dsp.LogicAnalyzer`. Signals can have different data types and dimensions.

Integers are supported up to 64 bits and fixed-point signals are supported up to 128 bits.

Example: `scope(signal1,signal2)`

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | logical | struct | table | cell

## **Object Functions**

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

### Specific to dsp.LogicAnalyzer

|                       |   |
|-----------------------|---|
| addCursor             | Add cursor to Logic Analyzer                        |
| addDivider            | Add divider to Logic Analyzer                       |
| addWave               | Add wave to Logic Analyzer                          |
| deleteCursor          | Delete Logic Analyzer cursor                        |
| deleteDisplayChannel  | Delete Logic Analyzer channel                       |
| getCursorInfo         | Return settings for Logic Analyzer cursor           |
| getCursorTags         | Return all Logic Analyzer cursor tags               |
| getDisplayChannelInfo | Return settings for Logic Analyzer display channel  |
| getDisplayChannelTags | Return all Logic Analyzer display channel tags      |
| modifyCursor          | Modify properties of Logic Analyzer cursor          |
| modifyDisplayChannel  | Modify properties of Logic Analyzer display channel |
| moveDisplayChannel    | Move position of Logic Analyzer display channel     |

### Specific to Scopes

|           |                               |
|-----------|-------------------------------|
| show      | Display scope window          |
| hide      | Hide scope window             |
| isVisible | Determine visibility of scope |

### Common to All System Objects

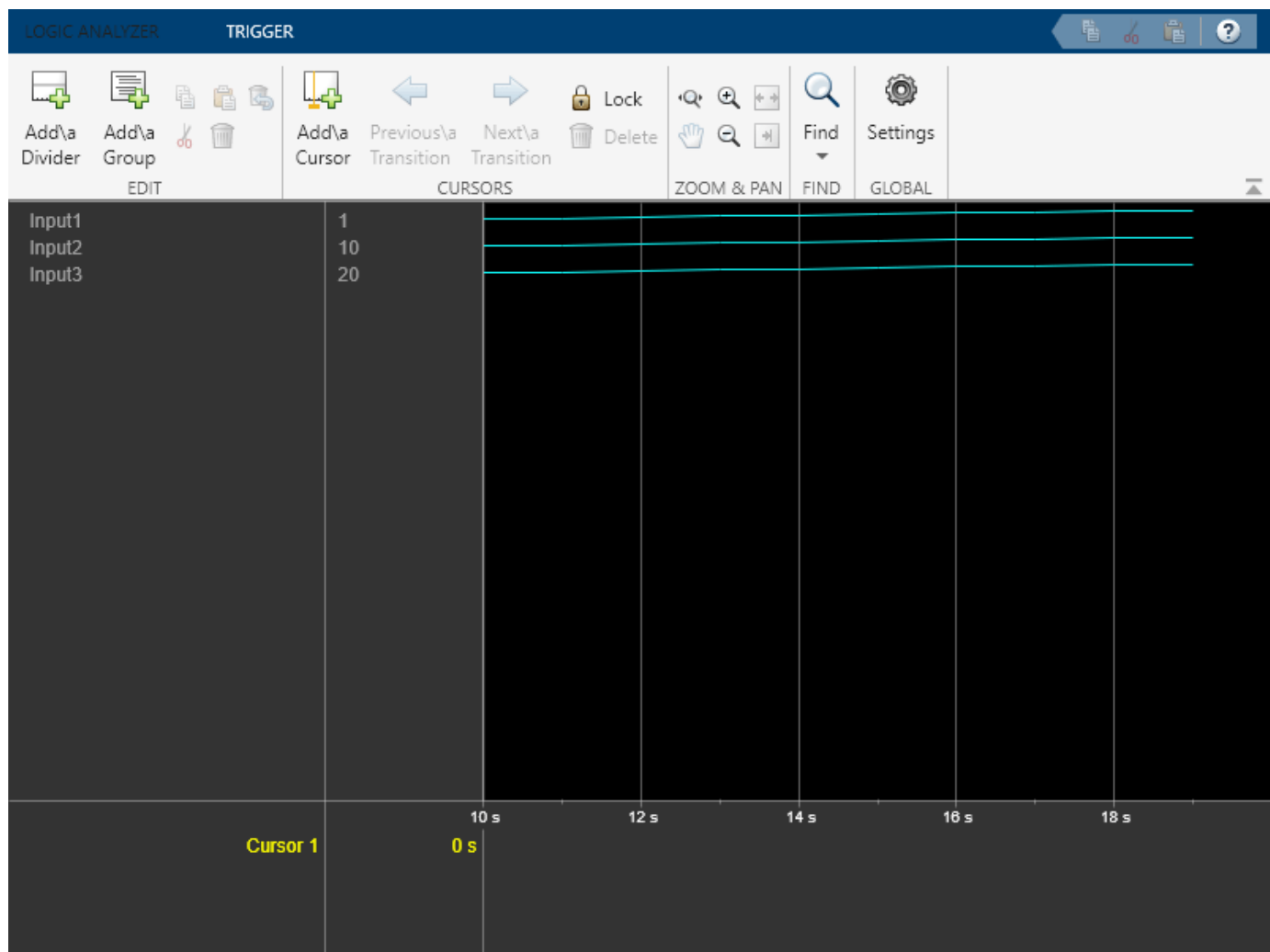
|         |  |
|---------|--|
| step    | Run System object algorithm  |
| release | Release resources and allow changes to System object property values and input characteristics |
| reset   | Reset internal states of System object   |

## Examples

### Display Simple Ramp Signals

Create a dsp.LogicAnalyzer object. Call the scope in a loop to display the signals.

```
scope = dsp.LogicAnalyzer('NumInputPorts',3);  
for ii = 1:20  
    scope(ii,10*ii,20*ii);  
end
```



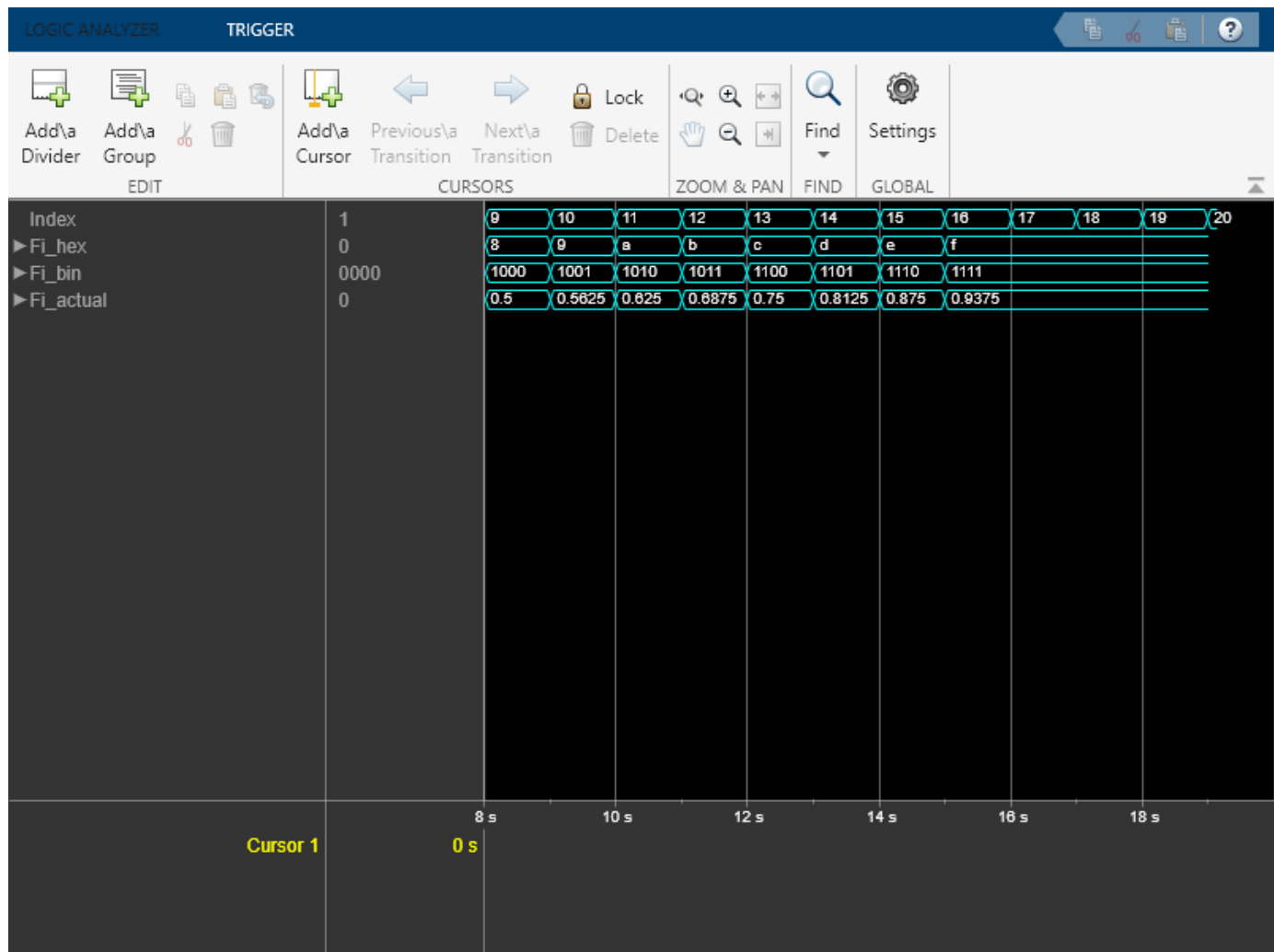
### Display Fixed-Point Signals

Create a `dsp.LogicAnalyzer` object with four channels. Call `modifyDisplayChannel` to set the radix of each of the channels. Run the scope in a loop to display the waves.

```
scope = dsp.LogicAnalyzer('NumInputPorts',4,'DisplayChannelFormat','Digital');
scope.TimeSpan = 12;
```

```
modifyDisplayChannel(scope,1,'Name','Index','Radix','Unsigned decimal');
modifyDisplayChannel(scope,2,'Name','Fi_hex','Radix','Hexadecimal');
modifyDisplayChannel(scope,3,'Name','Fi_bin','Radix','Binary');
modifyDisplayChannel(scope,4,'Name','Fi_actual','Radix','Signed decimal');
```

```
for ii = 1:20
    fival = fi((ii-1)/16,0,4,4);
    scope(ii,fival,fival,fival);
end
```



### Display Vector, Complex, and Enumerated Signals

Define a `WeekDaysInt` class to hold an enumerated list of weekday values. Create and save the following class definition file.

```
classdef WeekDaysInt < int32
    enumeration
        Monday(1), Tuesday(2), Wednesday(3), Thursday(4), Friday(5)
    end
end
```

Create a `dsp.LogicAnalyzer` object and configure the vector, complex, and enumerated data signals.

```
scope = dsp.LogicAnalyzer('NumInputPorts',6);
waves = getDisplayChannelTags(scope);

modifyDisplayChannel(scope,waves{1},'InputChannel',1,'Name','Vector Digital');
```



```

modifyDisplayChannel(scope,waves{2}, 'InputChannel',2, 'Name', 'Vector Analog', ...
    'Format', 'Analog', 'Height',80);
modifyDisplayChannel(scope,waves{3}, 'InputChannel',3, 'Name', 'Complex Digital');
modifyDisplayChannel(scope,waves{4}, 'InputChannel',4, 'Name', 'Complex Analog', ...
    'Format', 'Analog', 'Height',80, 'Color', 'Green');
modifyDisplayChannel(scope,waves{5}, 'InputChannel',5, 'Name', 'Enum Digital');
modifyDisplayChannel(scope,waves{6}, 'InputChannel',6, 'Name', 'Enum Analog', ...
    'Format', 'Analog', 'Height',80);

```

Call the scope object in a loop to display the signals.

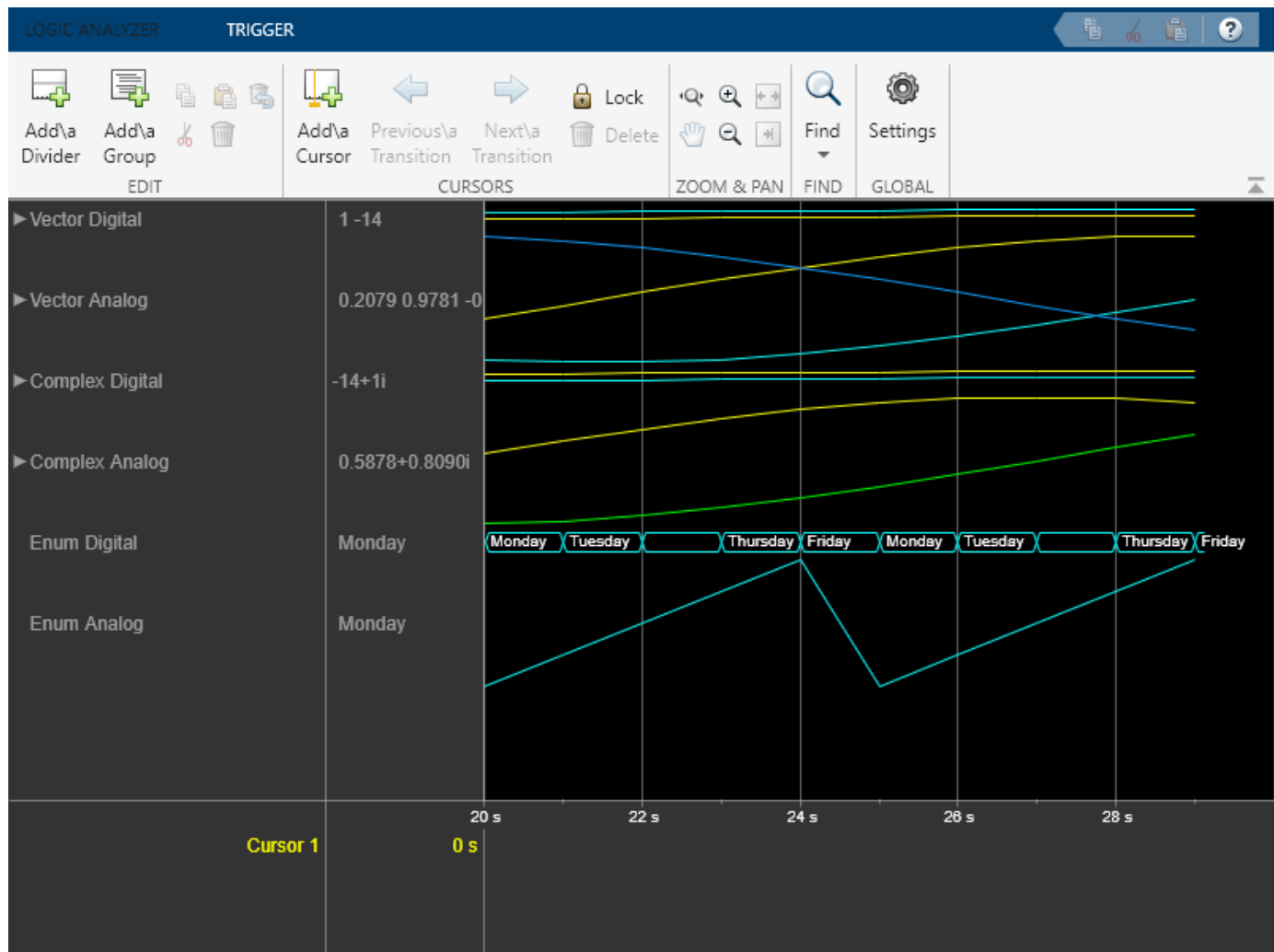
```

stop = 30;
for count = 1:stop
    sinValVec      = sin(count/stop*2*pi);
    cosValVec      = cos(count/stop*2*pi);
    cosValVecOffset = cos((count+10)/stop*2*pi);
    sinValReal     = sin((count+2)/stop*2*pi);
    cosValImag     = cos((count+2)/stop*2*pi);

    % Create a weekday enumerated value by wrapping the index
    day = WeekDaysInt(1+mod(count-1,5));

    scope(...
        [count (count-(stop/2))],...           % digital vector
        [sinValVec cosValVec cosValVecOffset],... % analog vector
        complex((count-(stop/2)),count),...   % digital complex
        complex(sinValReal, cosValImag),...   % analog complex
        day,...                               % digital enum
        day...                                 % analog enum
    )
end

```



## Tips

To close the logic analyzer window and clear its associated data, use the MATLAB `clear` function.

## Version History

Introduced in R2013a

## See Also

### Objects

`dsp.DynamicFilterVisualizer`

### Objects

`spectrumAnalyzer` | `timescope` | `dsp.ArrayPlot`

### Blocks

**Logic Analyzer**

**Topics**

“Inspect and Measure Transitions Using the Logic Analyzer”

## dsp.LowerTriangularSolver

**Package:** dsp

(Removed) Solve lower-triangular matrix equation

---

**Note** `dsp.LowerTriangularSolver` has been removed. Use the `mldivide` function or the `\` operator instead. For more information, see “Compatibility Considerations”.

---

### Description

The `LowerTriangularSolver` object solves  $LX = B$  for  $X$  when  $L$  is a square, lower-triangular matrix with the same number of rows as  $B$ .

To solve  $LX = B$  for  $X$ :

- 1 Create the `dsp.LowerTriangularSolver` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

### Creation

#### Syntax

```
lowtriang = dsp.LowerTriangularSolver  
lowtriang = dsp.LowerTriangularSolver(Name,Value)
```

#### Description

`lowtriang = dsp.LowerTriangularSolver` returns a linear system solver, `lowtriang`, used to solve the linear system  $LX = B$ , where  $L$  is a lower (or unit-lower) triangular matrix.

`lowtriang = dsp.LowerTriangularSolver(Name,Value)` returns a linear system solver, `lowtriang`, with each specified property set to the specified value.

### Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

#### **OverwriteDiagonal — Replace diagonal elements of input with ones**

false (default) | true

When you set this property to `true`, the linear system solver replaces the elements on the diagonal of the input,  $L$ , with ones. This property is useful when matrix  $L$  is the result of another operation, such as an LDL decomposition, that uses the diagonal elements to represent the  $D$  matrix.

### **RealDiagonalElements — Indicate that diagonal of complex input is real**

`false` (default) | `true`

When you set this property to `true`, the linear system solver optimizes computation speed if the input  $L$  is complex, but its diagonal elements are real. Set this property to either `true` or `false`.

#### **Dependencies**

This property applies only when you set the `OverwriteDiagonal` property to `false`.

### **Fixed-Point Properties**

#### **RoundingMethod — Rounding method for fixed-point operations**

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

Specify the rounding method as `Ceiling`, `Convergent`, `Floor`, `Nearest`, `Round`, `Simplest`, or `Zero`.

#### **OverflowAction — Overflow action for fixed-point operations**

`Wrap` (default) | `Saturate`

Specify the overflow action as `Wrap` or `Saturate`.

#### **ProductDataType — Data type of product**

`Full precision` (default) | `Same as input` | `Custom`

Specify the product data type as `Full precision`, `Same as input`, or `Custom`.

#### **CustomProductDataType — Product word and fraction lengths**

`numerictype([],32,30)` (default) | `numerictype`

Specify the product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`.

#### **Dependencies**

This property applies only when you set the `ProductDataType` property to `Custom`.

#### **AccumulatorDataType — Data type of accumulator**

`Full precision` (default) | `Same as first input` | `Same as product` | `Custom`

Specify the accumulator data type as `Full precision`, `Same as first input`, `Same as product`, or `Custom`.

#### **CustomAccumulatorDataType — Accumulator word and fraction lengths**

`numerictype([],32,30)` (default) | `numerictype`

Specify the accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`.

#### **Dependencies**

This property applies only when you set the `AccumulatorDataType` property to `Custom`.

**OutputDataType — Data type of output**

Same as first input (default) | Custom

Specify the output data type as Same as first input or Custom.

**CustomOutputDataType — Output word and fraction lengths**

numericType([],16,15) (default) | numericType

Specify the output fixed-point type as a scaled numericType object with a Signedness of Auto.

**Dependencies**

This property applies only when you set the OutputDataType property to Custom.

**Usage****Syntax**`X = lowtriang(L,B)`**Description**

`X = lowtriang(L,B)` computes the solution,  $X$ , of the matrix equation  $LX = B$ , where  $L$  is a square, lower-triangular matrix with the same number of rows as the matrix  $B$ .

**Input Arguments****L — Lower-triangular matrix**

matrix

Lower-triangular square matrix of size  $M$ -by- $M$ .

If the matrix is of fixed-point data type, it must be signed fixed point.

Data Types: single | double | int8 | int16 | int32 | int64 | fi

**B — Input B**

vector | matrix

Input  $B$  in the equation  $LX = B$ , where  $B$  is an  $M$ -by- $N$  matrix.

If the matrix is of fixed-point data type, it must be signed fixed point.

Data Types: single | double | int8 | int16 | int32 | int64 | fi

**Output Arguments****X — Solution of the equation**

vector | matrix

Solution of the  $LX = B$  equation, returned as an  $M$ -by- $N$  output matrix. The object uses only the elements in the lower triangle of input  $L$  and ignores the upper elements. When you set `OverwriteDiagonal` to `true`, the object replaces the elements on the diagonal of the input,  $L$ , with ones.

If the matrix is of fixed-point data type, it must be signed fixed point.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `fi`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Common to All System Objects

|                      |  |
|----------------------|--|
| <code>step</code>    | Run System object algorithm  |
| <code>release</code> | Release resources and allow changes to System object property values and input characteristics |
| <code>reset</code>   | Reset internal states of System object   |

## Examples

### Solve a Lower Triangular Matrix

```
rng(1);
lowtriang = dsp.LowerTriangularSolver;
u = tril(rand(4,4));
b = rand(4,1);
```

Check that result is the solution to the linear equations.

```
x1 = u\b
```

```
x1 = 4×1
```

```
    1.0007
   -1.7557
    1.1148
   -0.3901
```

```
x = lowtriang(u,b)
```

```
x = 4×1
```

```
    1.0007
   -1.7557
    1.1148
   -0.3901
```

## Algorithms

This object implements the algorithm, inputs, and outputs described on the Forward Substitution block reference page. The object properties correspond to the block parameters.

## Version History

**Introduced in R2012a**

**R2021b: dsp.LowerTriangularSolver System object will be removed***Warns starting in R2021b*

`dsp.LowerTriangularSolver` System object will be removed in a future release. Use the `mldivide` function or the `\` operator instead.

**R2023a: dsp.LowerTriangularSolver System object has been removed***Errors starting in R2023a*

The `dsp.LowerTriangularSolver` System object has been removed. Use the `mldivide` function or the `\` operator instead.

**Update Code**

This table shows how to update existing code to use the `mldivide` function or the `\` operator.

| Discouraged Usage   | Recommended Replacement  |
|---|--|
| <pre>rng(1); lowtriang = dsp.LowerTriangularSolver; u = tril(rand(4,4)); b = rand(4,1); x = lowtriang(u,b)  x = 4x1     1.0007    -1.7557     1.1148    -0.3901</pre> | <pre>x1 = u\b  x1 = 4x1     1.0007    -1.7557     1.1148    -0.3901</pre> <p>Alternatively, you can also use the <code>mldivide</code> function.</p> <pre>x1 = mldivide(u,b)  x1 = 4x1     1.0007    -1.7557     1.1148    -0.3901</pre> |

**Extended Capabilities****C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

**See Also****Objects**

`mldivide`



# dsp.LowpassFilter

**Package:** dsp

FIR or IIR lowpass filter

## Description

The `dsp.LowpassFilter` object independently filters each channel of the input over time using the given design specifications. You can set the `FilterType` property to 'FIR' or 'IIR' to implement the object as an FIR or an IIR lowpass filter.

When the `FilterType` property is set to 'FIR', using this object is an alternative to using the `firceqrip` and `firgr` functions with `dsp.FIRFilter`. The `dsp.LowpassFilter` object condenses the two-step process into one. You can use `measure` to verify that the design meets the prescribed specifications.

To filter each channel of your input:

- 1 Create the `dsp.LowpassFilter` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
LPF = dsp.LowpassFilter  
LPF = dsp.LowpassFilter(Name=Value)
```

### Description

`LPF = dsp.LowpassFilter` returns a minimum order FIR lowpass filter with the default filter settings. Calling the object with the default property settings filters the input data with a passband frequency of 8 kHz, a stopband frequency of 12 kHz, a passband ripple of 0.1 dB, and a stopband attenuation of 80 dB.

`LPF = dsp.LowpassFilter(Name=Value)` returns a lowpass filter with additional properties specified by one or more `Name-Value` pair arguments. `Name` is the property name and `Value` is the corresponding value. For example, `PassbandFrequency=8000` sets the passband frequency specification of the filter to 8000 Hz.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see *System Design in MATLAB Using System Objects*.

**FilterType — Type of filter response**

'FIR' (default) | 'IIR'

Type of filter response, specified as one of these options:

- 'FIR' — The object designs an FIR lowpass filter.
- 'IIR' — The object designs an IIR lowpass (biquad) filter.

**DesignForMinimumOrder — Flag to design minimum order filter**

true (default) | false

Flag to design minimum order filter, specified as:

- true -- The object designs the minimum order filter that meets the filter design specifications.
- false -- The object designs the filter with the order that you specify in the `FilterOrder` property.

**FilterOrder — Order of the FIR or IIR filter**

50 (default) | positive integer

Order of the FIR or IIR filter, specified as a positive integer.

**Dependencies**

To enable this property, set `DesignForMinimumOrder` to false.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**PassbandFrequency — Filter passband edge frequency**

8000 (default) | real positive scalar

Filter passband edge frequency, specified as a real positive scalar in Hz or in normalized frequency units.

If you set the `NormalizedFrequency` property to:

- false -- The value of the passband edge frequency is in Hz and must be less than the stopband frequency and half the `SampleRate` property value.
- true -- The value of the passband edge frequency is in normalized frequency units. The value must be a positive scalar less than 1.0.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**StopbandFrequency — Filter stopband edge frequency**

12000 (default) | real positive scalar

Filter stopband edge frequency, specified as a real positive scalar in Hz or in normalized frequency units.

If you set the `NormalizedFrequency` property to:

- false -- The value of the stopband edge frequency is in Hz. The value must be greater than the passband edge frequency and less than half the `SampleRate` property value.

- `true` -- The value of the stopband edge frequency is in normalized frequency units. The value must be a positive scalar greater than the passband edge frequency and less than `1.0`.

### Dependencies

To enable this property, set the `DesignForMinimumOrder` property to `true`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### PassbandRipple — Maximum ripple of filter response in the passband

`0.1` (default) | real positive scalar

Maximum ripple of filter response in the passband, specified as a real positive scalar in dB.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### StopbandAttenuation — Minimum attenuation in the stopband

`80` (default) | real positive scalar

Minimum attenuation in the stopband, specified as a real positive scalar in dB.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### NormalizedFrequency — Flag to set frequencies in normalized units

`false` (default) | `true`

Flag to set frequencies in normalized units, specified as one of these values:

- `true` -- The passband edge and stopband edge frequencies must be in the normalized frequency units and less than `1.0`.
- `false` -- The passband edge and stopband edge frequencies are in Hz. You can specify the input sample rate through the `SampleRate` property.

Data Types: `logical`

### SampleRate — Input sample rate

`44100` (default) | real positive scalar

Input sample rate in Hz, specified as a real positive scalar.

### Dependency

To enable this property, set `NormalizedFrequency` to `false`.

Data Types: `single` | `double`

### Fixed-Point Properties

#### RoundingMethod — Rounding method for output fixed-point operations

'Floor' (default) | 'Ceiling' | 'Convergent' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'

Rounding method for output fixed-point operations, specified as a character vector. For more information on the rounding modes, see "Precision and Range".

#### CoefficientsDataType — Word and fraction lengths of coefficients

`numericType(1,16)` (default) | `numericType` object

Word and fraction lengths of coefficients, specified as a `numericType` object. The default, `numericType(1,16)` corresponds to a signed numeric type object with 16-bit coefficients and a fraction length determined based on the coefficient values, to give the best possible precision.

This property is not tunable.

Word length of the output is same as the word length of the input. Fraction length of the output is computed such that the entire dynamic range of the output can be represented without overflow. For details on how the fraction length of the output is computed, see “Fixed-Point Precision Rules for Avoiding Overflow in FIR Filters”.

## Usage

### Syntax

```
y = LPF(x)
```

### Description

`y = LPF(x)` lowpass filters the input signal, `x`. `y` is a lowpass-filtered version of `x`.

### Input Arguments

#### **x** — Noisy data input

vector | matrix

Noisy data input, specified as a vector or a matrix. If the input signal is a matrix, each column of the matrix is treated as an independent channel. The number of rows in the input signal denote the channel length. This object accepts variable-size inputs. After the object is locked, you can change the size of each input channel, but you cannot change the number of channels.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

### Output Arguments

#### **y** — Filtered output

vector | matrix

Filtered output, returned as a vector or a matrix. The output has the same size, data type, and complexity characteristics as the input.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Specific to dsp.LowpassFilter

|             |   |
|-------------|---|
| freqz       | Frequency response of discrete-time filter System object                      |
| fvtool      | Visualize frequency response of DSP filters                                   |
| impz        | Impulse response of discrete-time filter System object                        |
| info        | Information about filter System object  |
| coeffs      | Returns the filter System object coefficients in a structure                  |
| cost        | Estimate cost of implementing filter System object                            |
| grpdelay    | Group delay response of discrete-time filter System object                    |
| generatehdl | Generate HDL code for quantized DSP filter (requires Filter Design HDL Coder) |
| measure     | Measure frequency response characteristics of filter System object            |

## Common to All System Objects

|         |  |
|---------|--|
| step    | Run System object algorithm  |
| release | Release resources and allow changes to System object property values and input characteristics |
| reset   | Reset internal states of System object   |

## Examples

### Impulse and Frequency Response of FIR and IIR Lowpass Filters

Create a minimum-order FIR lowpass filter for data sampled at 44.1 kHz. Specify a passband frequency of 8 kHz, a stopband frequency of 12 kHz, a passband ripple of 0.1 dB, and a stopband attenuation of 80 dB.

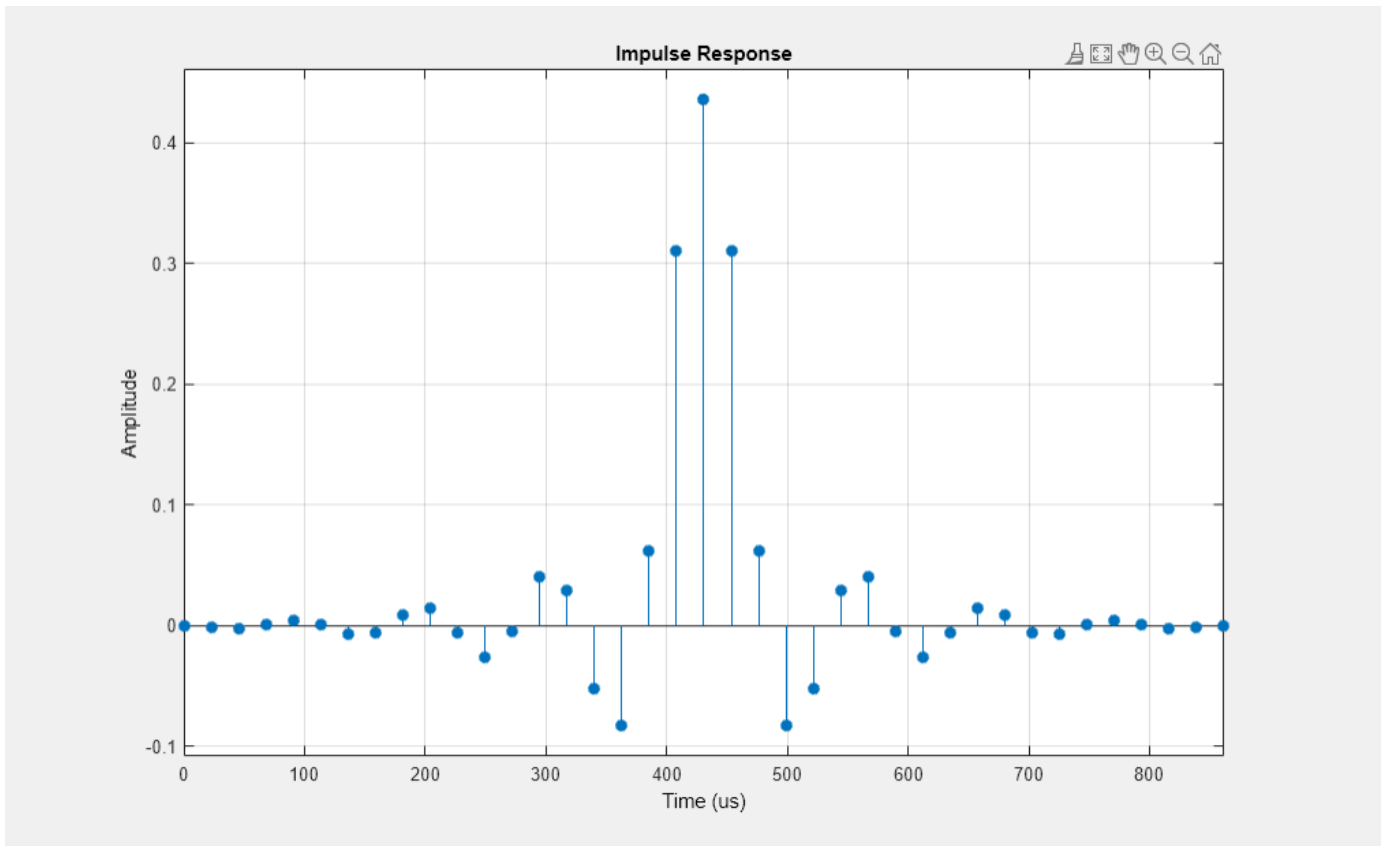
```
Fs = 44.1e3;
filtertype = 'FIR';
Fpass = 8e3;
Fstop = 12e3;
Rp = 0.1;
Astop = 80;
FIRLPF = dsp.LowpassFilter(SampleRate=Fs,...
                           FilterType=filtertype,...
                           PassbandFrequency=Fpass,...
                           StopbandFrequency=Fstop,...
                           PassbandRipple=Rp,...
                           StopbandAttenuation=Astop);
```

Design a minimum-order IIR lowpass filter with the same properties as the FIR lowpass filter. Change the `FilterType` property of the cloned filter to `IIR`.

```
IIRLPF = clone(FIRLPF);
IIRLPF.FilterType = 'IIR';
```

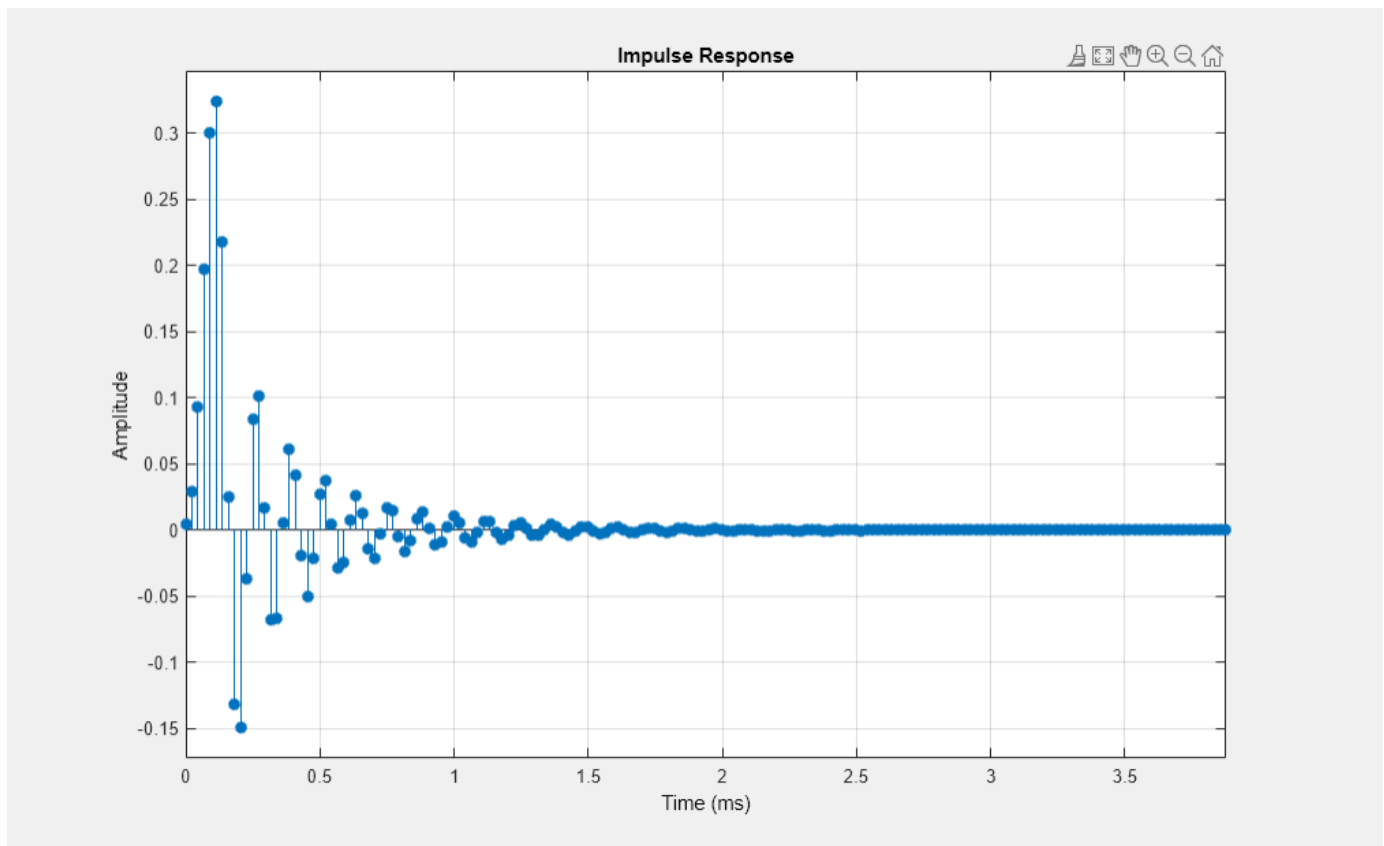
Plot the impulse response of the FIR lowpass filter. The zeroth-order coefficient is delayed by 19 samples, which is equal to the group delay of the filter. The FIR lowpass filter is a causal FIR filter.

```
fvtool(FIRLPF,Analysis='impulse')
```



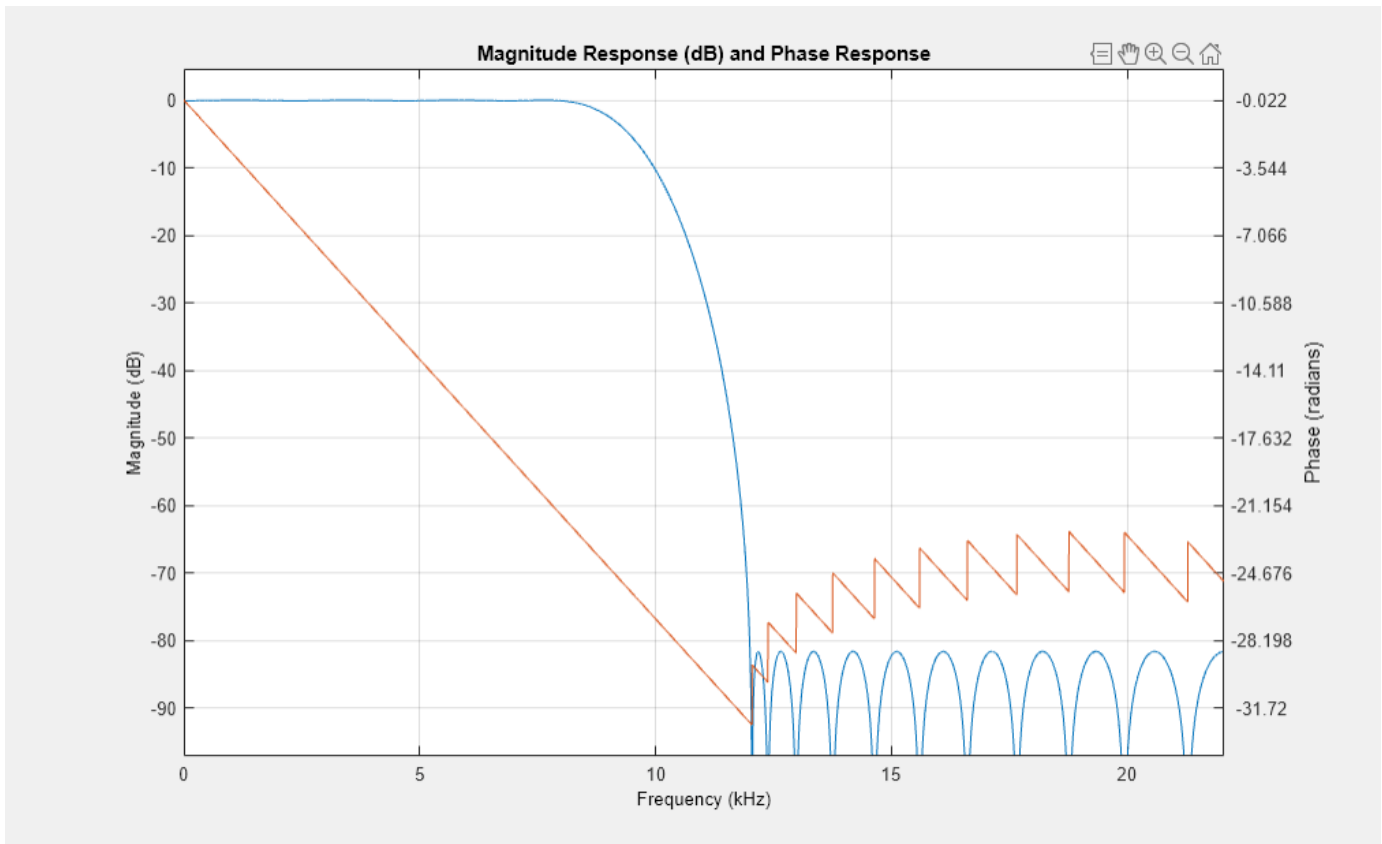
Plot the impulse response of the IIR lowpass filter.

```
fvtool(IIRLPF,Analysis='impulse')
```



Plot the magnitude and phase response of the FIR lowpass filter.

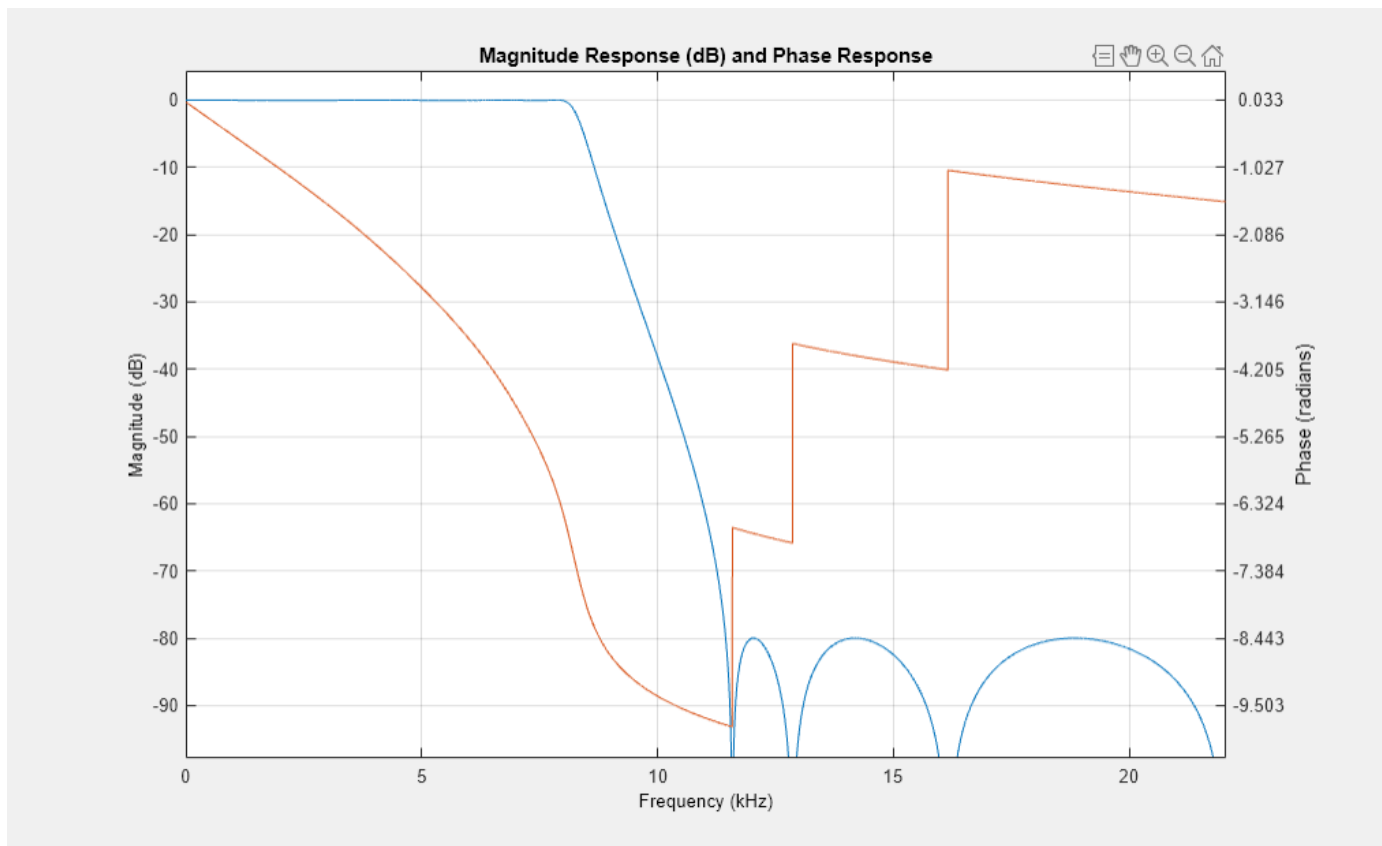
```
fvtool(FIRLPF,Analysis='freq')
```



Plot the magnitude and phase response of the IIR lowpass filter.

```
fvtool(IIRLPF,Analysis='freq')
```





Calculate the cost of implementing the FIR lowpass filter.

```
cost(FIRLPF)
```

```
ans = struct with fields:
    NumCoefficients: 39
    NumStates: 38
    MultiplicationsPerInputSample: 39
    AdditionsPerInputSample: 38
```

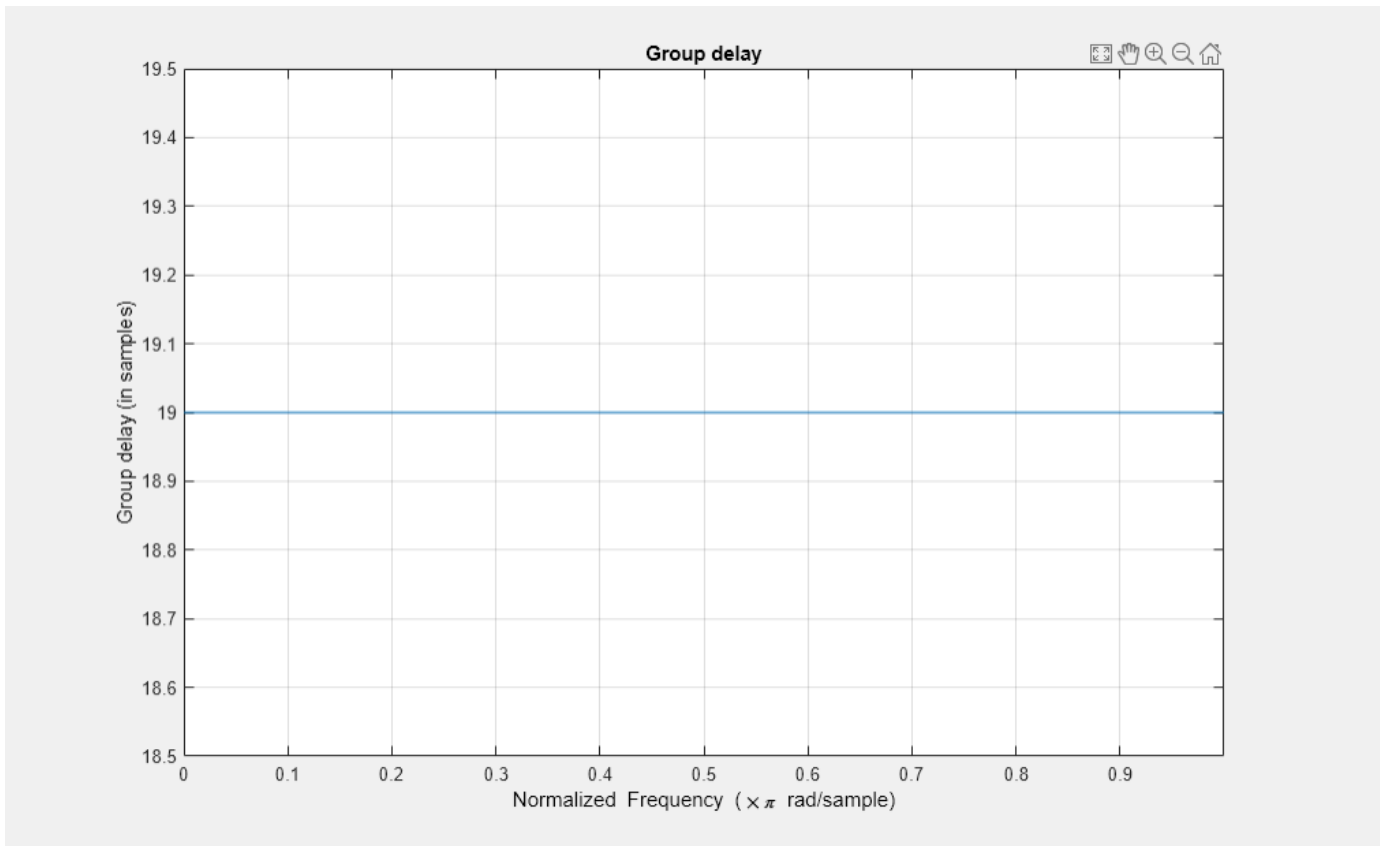
Calculate the cost of implementing the IIR lowpass filter. The IIR filter is more efficient to implement than the FIR filter.

```
cost(IIRLPF)
```

```
ans = struct with fields:
    NumCoefficients: 18
    NumStates: 14
    MultiplicationsPerInputSample: 18
    AdditionsPerInputSample: 14
```

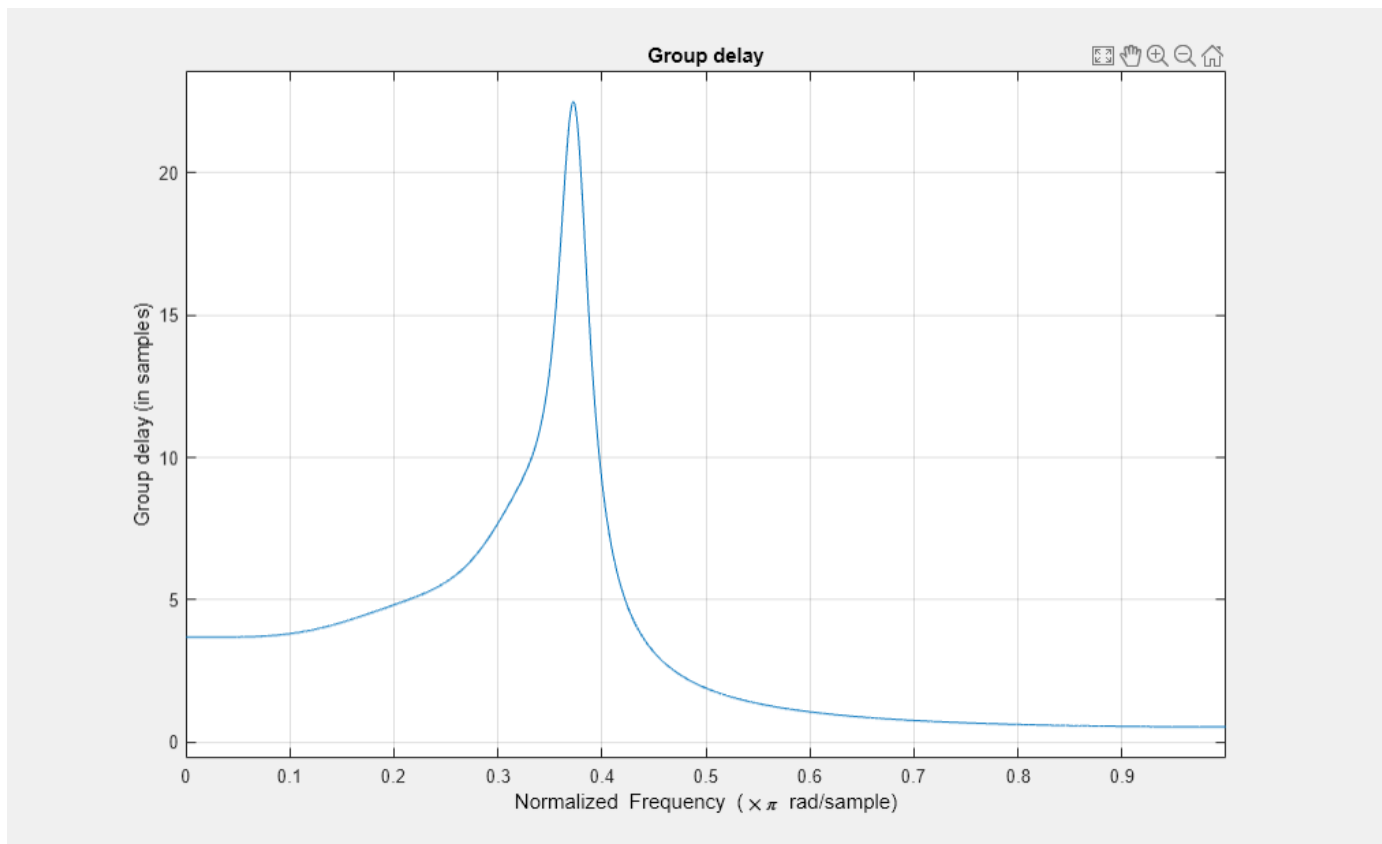
Calculate the group delay of the FIR lowpass filter.

```
grpdelay(FIRLPF)
```



Calculate the group delay of the IIR lowpass filter. The FIR filter has a constant group delay (linear phase), while its IIR counterpart does not.

```
grpdelay(IIRLPF)
```



### Filter White Gaussian Noise Signal With FIR Lowpass Filter

Create a lowpass filter using the `dsp.LowpassFilter` System object™. Setting the `NormalizedFrequency` property to `true` designs the filter with frequency specifications in normalized frequency units.

```
LPF = dsp.LowpassFilter(NormalizedFrequency=true)
```

```
LPF =  
dsp.LowpassFilter with properties:
```

```
    FilterType: 'FIR'  
    DesignForMinimumOrder: true  
    PassbandFrequency: 0.3628  
    StopbandFrequency: 0.5442  
    PassbandRipple: 0.1000  
    StopbandAttenuation: 80  
    NormalizedFrequency: true
```

```
Show all properties
```

Create a `spectrumAnalyzer` object to visualize the input and output signal spectra. With a sample rate of  $44.1 \times 10^3$  Hz, the passband frequency and the stopband frequency of the filter translate to 8000 Hz and 12000 Hz, respectively.

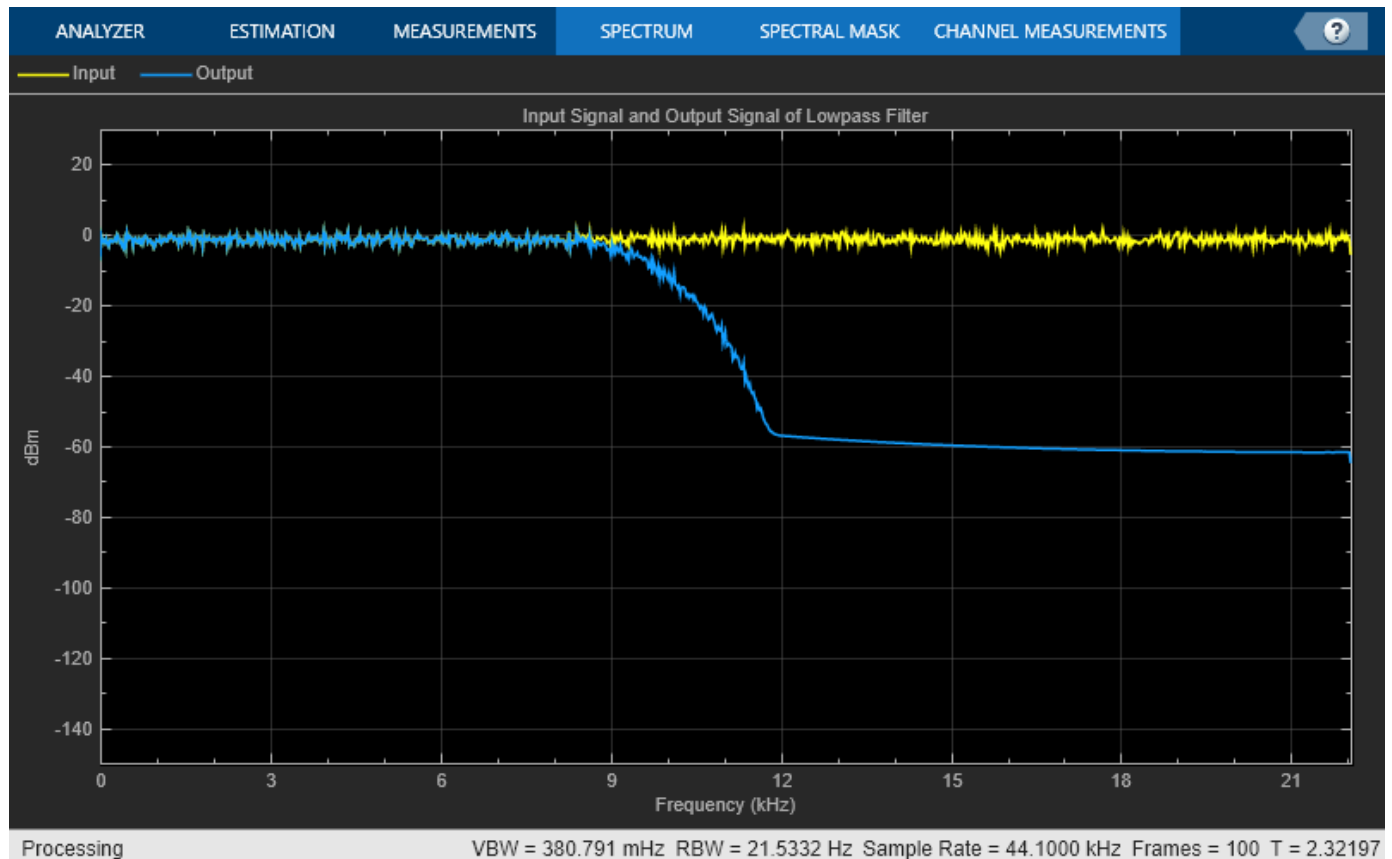
```
SA = spectrumAnalyzer(SampleRate=44.1e3,...
    PlotAsTwoSidedSpectrum=false,ShowLegend=true,...
    YLimits=[-150 30],...
    Title='Input Signal and Output Signal of Lowpass Filter');
SA.ChannelNames = {'Input','Output'};
```

Run the lowpass filter algorithm to filter the white Gaussian noisy input signal. View the input and output signals using the spectrum analyzer.

```
for k = 1:100
    Input = randn(1024,1);

    Output = LPF(Input);

    SA([Input,Output]);
end
```



### Filter White Gaussian Noise with IIR Lowpass Filter

Set up the IIR lowpass filter. The sampling rate of the white Gaussian noise is 44,100 Hz. The passband frequency of the filter is 8 kHz, the stopband frequency is 12 kHz, the passband ripple is 0.1 dB, and the stopband attenuation is 80 dB.

```
Fs = 44.1e3;
filtertype = 'IIR';
```

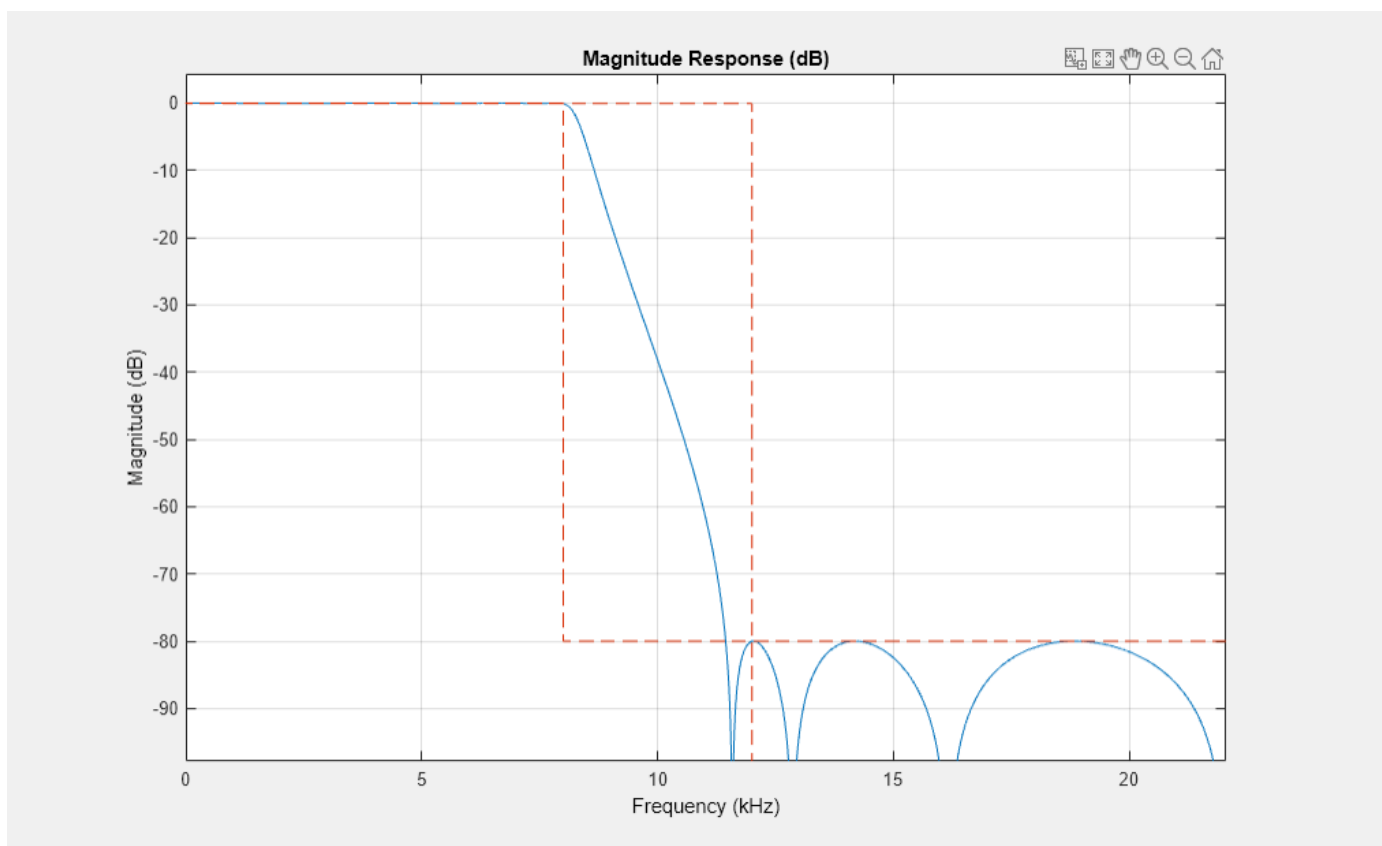
```

Fpass = 8e3;
Fstop = 12e3;
Rp = 0.1;
Astop = 80;
LPF = dsp.LowpassFilter(SampleRate=Fs,...
    FilterType=filtertype,...
    PassbandFrequency=Fpass,...
    StopbandFrequency=Fstop,...
    PassbandRipple=Rp,...
    StopbandAttenuation=Astop);

```

View the magnitude response of the lowpass filter.

```
fvtool(LPf)
```



Create a spectrum analyzer object.

```

SA = spectrumAnalyzer(SampleRate=Fs,...
    PlotAsTwoSidedSpectrum=false,...
    ShowLegend=true,YLimits=[-150 30],...
    Title='Input Signal and Output Signal of IIR Lowpass Filter');
SA.ChannelNames = {'Input','Output'};

```

Filter the white Gaussian noisy input signal. View the input and output signals using the spectrum analyzer.

```

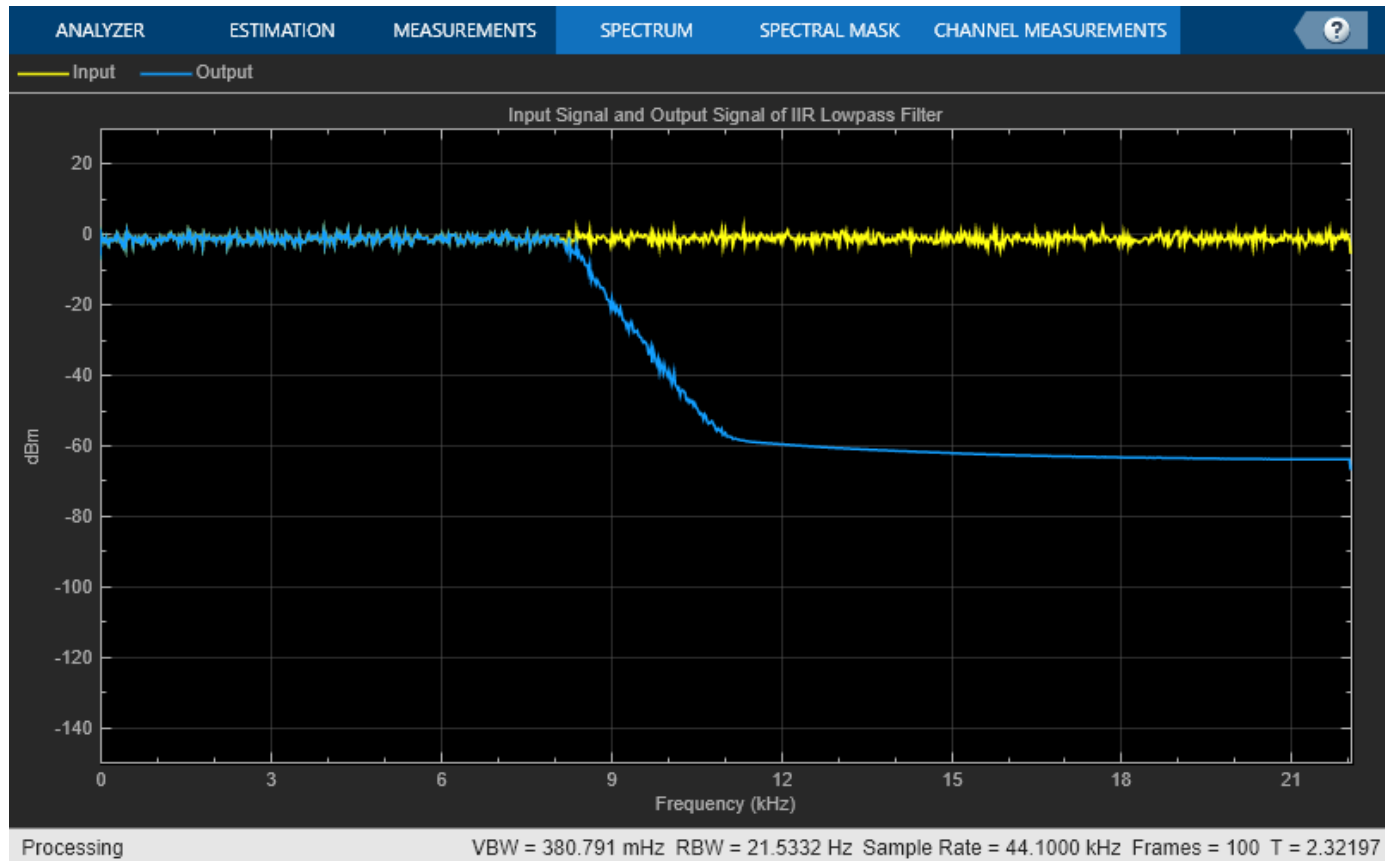
for k = 1:100
    Input = randn(1024,1);

```

```

Output = LPF(Input);
SA([Input,Output]);
end

```



### Measure Frequency Response Characteristics of Lowpass Filter

Measure the frequency response characteristics of a lowpass filter. Create a `dsp.LowpassFilter` System object with default properties. Measure the frequency response characteristics of the filter.

```
LPF = dsp.LowpassFilter
```

```
LPF =
    dsp.LowpassFilter with properties:
```

```

    FilterType: 'FIR'
    DesignForMinimumOrder: true
    PassbandFrequency: 8000
    StopbandFrequency: 12000
    PassbandRipple: 0.1000
    StopbandAttenuation: 80
    NormalizedFrequency: false
    SampleRate: 44100

```

Show all properties

```
LPFMeas = measure(LPF)
```

```
LPFMeas =
Sample Rate      : 44.1 kHz
Passband Edge    : 8 kHz
3-dB Point       : 9.1311 kHz
6-dB Point       : 9.5723 kHz
Stopband Edge    : 12 kHz
Passband Ripple  : 0.08289 dB
Stopband Atten.  : 81.6141 dB
Transition Width : 4 kHz
```

## Algorithms

### FIR Lowpass Filter

For the minimum order design, the algorithm uses generalized Remez FIR filter design algorithm. For the specified order design, the algorithm uses the constrained equiripple FIR filter design algorithm. The designed filter is then implemented as a linear phase Type-1 filter with a `Direct` form structure.

### IIR Lowpass Filter

In the IIR configuration, the algorithm uses the elliptic design method to compute the SOS and scale values required to meet the filter design specifications. The algorithm uses the SOS and scale values to setup a `Direct form I` biquadratic IIR filter, which forms the basis of the IIR version of the lowpass filter.

## Version History

### Introduced in R2015a

#### R2023a: Support for normalized frequencies

When you set the `NormalizedFrequency` property to `true`, you must specify the passband and stopband frequencies in normalized frequency units (0 to 1).

When you set the `NormalizedFrequency` property to `true` while creating the object, the passband and stopband frequency values are automatically set to normalized frequency units using the default sample rate of 44100 Hz.

```
lpFilter = dsp.LowpassFilter(NormalizedFrequency=true)
```

```
lpFilter =
dsp.LowpassFilter with properties:
    FilterType: 'FIR'
    DesignForMinimumOrder: true
    PassbandFrequency: 0.3628
```

```
StopbandFrequency: 0.5442
PassbandRipple: 0.1000
StopbandAttenuation: 80
NormalizedFrequency: true
```

When you set the `NormalizedFrequency` property to `true` after you create the object, the passband and stopband frequencies must be manually set to the normalized frequency values before you run the object algorithm.

```
lpFilter = dsp.LowpassFilter
```

```
lpFilter =
dsp.LowpassFilter with properties:
```

```
FilterType: 'FIR'
DesignForMinimumOrder: true
PassbandFrequency: 8000
StopbandFrequency: 12000
PassbandRipple: 0.1000
StopbandAttenuation: 80
NormalizedFrequency: false
SampleRate: 44100
```

To specify the normalized frequency values, set `NormalizedFrequency` to `true` and manually convert the frequency values in Hz to normalized values using the input sample rate in Hz. For example, if the input sample rate is 44100 Hz, the corresponding values in normalized units are computed using these equations.

```
lpFilter.NormalizedFrequency = true;
lpFilter.PassbandFrequency = 8000/(44100/2);
lpFilter.StopbandFrequency = 12000/(44100/2)
```

```
lpFilter =
dsp.LowpassFilter with properties:
```

```
FilterType: 'FIR'
DesignForMinimumOrder: true
PassbandFrequency: 0.3628
StopbandFrequency: 0.5442
PassbandRipple: 0.1000
StopbandAttenuation: 80
NormalizedFrequency: true
```

## References

- [1] Shpak, D.J., and A. Antoniou. "A generalized Remez method for the design of FIR digital filters." *IEEE Transactions on Circuits and Systems*. Vol. 37, Issue 2, Feb. 1990, pp. 161-174.
- [2] Selesnick, I.W., and C. S. Burrus. "Exchange algorithms that complement the Parks-McClellan algorithm for linear-phase FIR filter design." *IEEE Transactions on Circuits and Systems*. Vol. 44, Issue 2, Feb. 1997, pp. 137-143.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.



Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

This object supports code generation for ARM Cortex-M and ARM Cortex-A processors. To learn more about ARM Cortex code generation, see “Code Generation for ARM Cortex-M and ARM Cortex-A Processors”.

This object also supports SIMD code generation using Intel AVX2 technology under these conditions:

- `FilterType` is set to `'FIR'`.
- Input signal has a data type of `single` or `double`.

The SIMD technology significantly improves the performance of the generated code.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

This object supports HDL code generation with the Filter Design HDL Coder product. For workflows and limitations, see “Generate HDL Code for Filter System Objects” (Filter Design HDL Coder).

## **See Also**

### **Functions**

`freqz` | `fvtool` | `impz` | `info` | `coeffs` | `cost` | `grpdelay` | `generatehdl` | `measure` | `firceqrip` | `firgr`

### **Objects**

`dsp.HighpassFilter`

### **Blocks**

Lowpass Filter

### **Topics**

“Lowpass Filter Design in MATLAB”

“Analysis Methods for Filter System Objects” on page 3-2

## dsp.LPCToLSF

**Package:** dsp

(Removed) Convert linear prediction coefficients to line spectral frequencies

---

**Note** dsp.LPCToLSF has been removed. Use `poly2lsf` from Signal Processing Toolbox™ instead. For more information, see “Compatibility Considerations”.

---

### Description

The LPCToLSF object converts linear prediction coefficients to line spectral frequencies.

To convert LPC to LSF:

- 1 Define and set up your LPC to LSF converter. See “Construction” on page 4-804.
- 2 Call `step` to convert LPC according to the properties of `dsp.LPCToLSF`. The behavior of `step` is specific to each object in the toolbox.

---

**Note** Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj, x)` and `y = obj(x)` perform equivalent operations.

---

### Construction

`lpc2lsf = dsp.LPCToLSF` returns a System object, `lpc2lsf`, that converts linear prediction coefficients (LPCs) to line spectral frequencies (LSFs).

`lpc2lsf = dsp.LPCToLSF('PropertyName', PropertyValue, ...)` returns an LPC to LSF System object, `lpc2lsf`, with each specified property set to the specified value.

### Properties

#### NumCoarseGridPoints

Number of coarse subintervals used for finding roots (LSP values)

Specify the number of coarse subintervals, *n*, used for finding line spectral pairs (LSP) values as a positive scalar integer. LSPs, which are the roots of two particular polynomials related to the input LPC polynomial, always lie in the range (-1, 1). The System object finds these roots using the Chebyshev polynomial root finding method. To compute LSF outputs, the object computes the arc cosine of the LSPs, outputting values ranging from 0 to pi radians. The object divides the interval (-1, 1) into *n* subintervals and looks for roots in each subinterval. If you set *n* too small in relation to the LPC polynomial order, the object can fail to find some of the roots. The default is 64. This property is tunable.

**NumBisects**

Value of bisection refinement used for finding roots

Specify the root bisection refinement value,  $k$ , used in the Chebyshev polynomial root finding method, where each line spectral pair (LSP) output is within

$$\frac{1}{(n \cdot 2^k)}$$

of the actual LSP value. Here  $n$  is the value of the `NumCoarseGridPoints` property, and the object searches a maximum of  $k \cdot (n - 1)$  points for finding the roots. You must set the `NumBisects` property value  $k$ , to a positive scalar integer. The default is 4. This property is tunable.

**ExceptionOutputPort**

Produces output with validity status of LSF output

Set this property to `true` to return a second output that indicates whether the computed LSF values are valid. The output is a vector with a length equal to the number of channels. A logical value of 1 indicates valid output. A logical value of 0 indicates invalid output. The LSF outputs are invalid when the object fails to find all the LSF values or when the input LPCs are unstable. The default is `false`.

**OverwriteInvalidOutput**

Enable overwriting invalid output with previous output

Specify the action that the System object should take for invalid LSF outputs. When you set this property to `true`, the object overwrites the invalid output with the previous output. When you set this property to `false`, the object does not take any action on invalid outputs and ignores the outputs.

**FirstOutputValuesSource**

Source of values for first output when output is invalid

Specify the source of values for the first output when the output is invalid as `Auto` or `Property`. This property applies when you set the `OverwriteInvalidOutput` property to `true`. The default is `Auto`. When you set this property to `Auto`, the object uses a default value for the first output. The default value corresponds to the LSF representation of an allpass filter.

**FirstOutputValues**

Value of the first output

Specify a numeric vector of LSF values for overwriting an invalid first output. The length of this vector must be one less than the length of the input LPC vector. For multichannel inputs, you can set this property to a matrix with the same number of channels as the input, or one vector that is applied to every channel. The default is an empty vector. This property applies when you set the `OverwriteInvalidOutput` property to `true` and the `FirstOutputValuesSource` property to `Property`.

## NonUnityFirstCoefficientAction

Action to take when first LPC coefficient is not 1

Specify the action the object takes when the first coefficient of each channel of the LPC input is not 1 as `Replace with 1` or `Normalize`. The default is `Replace with 1`.

## Methods

`reset`    Reset values for overwriting invalid outputs to their initial values  
`step`     Convert LPC coefficients to line spectral frequencies

### Common to All System Objects

|                      |  |
|----------------------|--|
| <code>release</code> | Allow System object property value changes |
|----------------------|--|

## Examples

### Convert LPC Coefficients To LSF Coefficients

**Note** This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, `myObject(x)` becomes `step(myObject,x)`.

Convert LPC to LSF coefficients

```
a = [1.0000 0.6149 0.9899 0.0000 0.0031 -0.0082]';
lpc2lsf = dsp.LPCToLSF;
y = lpc2lsf(a);
display(y);
```

$y = 5 \times 1$

```
0.7842
1.5605
1.8776
1.8984
2.3593
```

## Algorithms

This object implements the algorithm, inputs, and outputs described on the LPC to LSF/LSP Conversion block reference page. The object properties correspond to the block parameters, except: There is no object property that corresponds to the **Output** block parameter. The object only supports LSF outputs in the range  $(0, \infty)$

## Version History

Introduced in R2012a

**R2022b: dsp.LPCToLSF System object has been removed***Errors starting in R2022b*

`dsp.LPCToLSF` System object has been removed. Use `poly2lsf` instead.

**Update Code**

This table shows typical usage of the System object and explains how to update existing code to use the equivalent function.

| Discouraged Usage   | Recommended Replacement  |
|---|--|
| <pre>a = [1.0000 0.6149 0.9899 0.0000 0.0031]; lpc2lsf = dsp.LPCToLSF lsfObj = lpc2lsf(a)</pre> <p>If you are using a release prior to R2016b, replace <code>lpc2lsf(a)</code> with <code>step(lpc2lsf,a)</code>.</p> | <pre>a = [1.0000 0.6149 0.9899 0.0000 0.0031 -0.0082]; lsfFn = poly2lsf(a)</pre> |

**Extended Capabilities****C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

**See Also****Functions**

`poly2lsf`

## **reset**

Reset values for overwriting invalid outputs to their initial values

### **Syntax**

```
reset(lpc2lsf)
```

### **Description**

`reset(lpc2lsf)` resets the values for overwriting the invalid outputs to their initial values.

## step

**System object:** dsp.LPCToLSF

**Package:** dsp

Convert LPC coefficients to line spectral frequencies

### Syntax

```
LSF = step(lpc2lsf,A)
[... , STATUS] = step(lpc2lsf,A)
```

### Description

---

**Note** Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

---

`LSF = step(lpc2lsf,A)` converts the LPC coefficients, `A`, to line spectral frequencies, `LSF`, in the range  $(0 \pi)$ . The System object operates along the columns of the input `A`.

`[... , STATUS] = step(lpc2lsf,A)` also returns the status flag, `STATUS`, indicating if the current output is valid when the `ExceptionOutputPort` property is `true`.

---

**Note** `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

## dsp.LUFactor

**Package:** dsp

(Removed) Factor square matrix into lower and upper triangular matrices

---

**Note** dsp.LUFactor has been removed. Use lu instead. For more information, see “Compatibility Considerations”.

---

### Description

The LUFactor object factors a square matrix into lower and upper triangular matrices.

To factor a square matrix into lower and upper triangular matrices:

- 1 Define and set up your System object. See “Construction” on page 4-810.
- 2 Call step to factor the square matrix according to the properties of dsp.LUFactor. The behavior of step is specific to each object in the toolbox.

---

**Note** Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example,  $y = \text{step}(\text{obj}, x)$  and  $y = \text{obj}(x)$  perform equivalent operations.

---

### Construction

$\text{lu} = \text{dsp.LUFactor}$  returns an LUFactor System object, lu, which factors a row permutation of a square input matrix  $A$  as  $A_p = L \cdot U$ , where  $L$  is the unit-lower triangular matrix, and  $U$  is the upper triangular matrix. The row-pivoted matrix  $A_p$  contains the rows of  $A$  permuted as indicated by the permutation index vector  $P$ . The equivalent MATLAB code is  $A_p = A(P, :)$ .

$\text{lu} = \text{dsp.LUFactor}('PropertyName', PropertyValue, \dots)$  returns an LUFactor object, lu, with each specified property set to the specified value.

### Properties

#### ExceptionOutputPort

Set to true to output singularity of input

Set this property to true to output the singularity of the input as logical data type values of true or false. An output of true indicates that the current input is singular, and an output of false indicates the current input is nonsingular.

#### Fixed-Point Properties

#### RoundingMethod



Rounding method for fixed-point operations

Specify the rounding method as `|Ceiling|Convergent|Floor|Nearest |Round | Simplest | Zero|`. The default is `Floor`.

### **OverflowAction**

Overflow action for fixed-point operations

Specify the overflow action as `Wrap` or `Saturate`. The default is `Wrap`.

### **ProductDataType**

Product word and fraction lengths

Specify the product fixed-point data type as `Full precision`, `Same as input` or `Custom`. The default is `Full precision`.

### **CustomProductDataType**

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `ProductDataType` property to `Custom`. The default is `numericType([], 32, 30)`.

### **AccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as `Full precision`, `Same as input`, `Same as product` or `Custom`. The default is `Full precision`.

### **CustomAccumulatorDataType**

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`. This property applies when you set the `AccumulatorDataType` property to `Custom`. The default is `numericType([], 32, 30)`.

### **OutputDataType**

Output word and fraction lengths

Specify the output fixed-point data type as `Same as input` or `Custom`. The default is `Same as input`.

### **CustomOutputDataType**

Output word and fraction lengths

Specify the output fixed-point type as a scaled numeric type object with a Signedness of Auto. This property applies when you set the "OutputDataType" on page 4-0 property to Custom. The default is `numericType([], 16, 15)`.

## Methods

`step` Decompose matrix into lower and upper triangular matrices

### Common to All System Objects

|                      |  |
|----------------------|--|
| <code>release</code> | Allow System object property value changes |
|----------------------|--|

## Examples

### Decompose a Square Matrix

**Note** If you are using R2016a or an earlier release, replace each call to the object with the equivalent `step` syntax. For example, `obj(x)` becomes `step(obj, x)`.

Decompose a square matrix into the lower and upper components.

```
lu = dsp.LUFactor;
x = rand(4)

x = 4x4

    0.8147    0.6324    0.9575    0.9572
    0.9058    0.0975    0.9649    0.4854
    0.1270    0.2785    0.1576    0.8003
    0.9134    0.5469    0.9706    0.1419

[LU, P] = lu(x);
L = tril(LU, -1)+diag(ones(size(LU,1),1));
U = triu(LU);
y = L*U

y = 4x4

    0.9134    0.5469    0.9706    0.1419
    0.9058    0.0975    0.9649    0.4854
    0.8147    0.6324    0.9575    0.9572
    0.1270    0.2785    0.1576    0.8003
```

Check back whether `y` equals the permuted `x`

```
xp = x(P, :)

xp = 4x4
```

|        |        |        |        |
|--------|--------|--------|--------|
| 0.9134 | 0.5469 | 0.9706 | 0.1419 |
| 0.9058 | 0.0975 | 0.9649 | 0.4854 |
| 0.8147 | 0.6324 | 0.9575 | 0.9572 |
| 0.1270 | 0.2785 | 0.1576 | 0.8003 |

## Algorithms

This object implements the algorithm, inputs, and outputs described on the LU Factorization block reference page. The object properties correspond to the block parameters.

## Version History

### Introduced in R2012a

#### **R2021b: dsp.LUFactor System object will be removed**

*Warns starting in R2021b*

`dsp.LUFactor` System object will be removed in a future release. Use the `lu` function instead.

#### **R2023a: dsp.LUFactor System object has been removed**

*Errors starting in R2023a*

The `dsp.LUFactor` System object has been removed. Use the `lu` function instead.

### Update Code

This table shows how to update existing code to use the `lu` function.

| Discouraged Usage   | Recommended Replacement  |
|---|--|
| <pre>rng(1) lu = dsp.LUFactor; x = rand(4)  x = 4x4      0.4170    0.1468    0.3968    0.2045     0.7203    0.0923    0.5388    0.8781     0.0001    0.1863    0.4192    0.0274     0.3023    0.3456    0.6852    0.6705  [LU,P] = lu(x)  LU = 4x4     0.7203    0.0923    0.5388    0.8781     0.4197    0.3068    0.4591    0.3019     0.0002    0.6070    0.1404   -0.1560     0.5789    0.3041   -0.3900   -0.4566  P = 4x1      2      4      3      1</pre> | <pre>[LUfn,Pfn] = lu(x)  LUfn = 4x4     0.7203    0.0923    0.5388    0.8781     0.4197    0.3068    0.4591    0.3019     0.0002    0.6070    0.1404   -0.1560     0.5789    0.3041   -0.3900   -0.4566  Pfn = 4x1      2      4      3      1</pre> |

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

### See Also

lu

## step

**System object:** dsp.LUFactor

**Package:** dsp

Decompose matrix into lower and upper triangular matrices

### Syntax

```
[LU,P] = step(lu,A)
[LU,P,S] = step(lu,A)
```

### Description

---

**Note** Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

---

`[LU,P] = step(lu,A)` decomposes the matrix `A` into lower and upper triangular matrices. The output `LU` is a composite matrix with lower triangle elements from `L` and upper triangle elements from `U`. The permutation vector `P` is the second output.

`[LU,P,S] = step(lu,A)` returns an additional output `S` indicating if the input is singular when the `ExceptionOutputPort` property is set to `true`.

---

**Note** `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

# dsp.MatFileReader

**Package:** dsp

Read MAT file

## Description

The `dsp.MatFileReader` System object reads V7.3 MAT files.

To read V7.3 MAT files:

- 1 Create the `dsp.MatFileReader` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
mfr = dsp.MatFileReader
mfr = dsp.MatFileReader(fname,vname,framesize)
mfr = dsp.MatFileReader( ___,Name,Value)
```

### Description

`mfr = dsp.MatFileReader` returns a System object, `mfr`, to read a stream of scalar data from a V7.3 MAT file.

`mfr = dsp.MatFileReader(fname,vname,framesize)` reads frames of MAT file data, using the specified file name, variable name, and frame size.

`mfr = dsp.MatFileReader( ___,Name,Value)` reads MAT file data with each specified property set to the specified value.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

### Filename — Name of MAT file

'Untitled.mat' (default) | character vector | string scalar

Name of the MAT file from which to read, specified as a character vector or a string scalar. Specify the full path for the file only if the file is not on the MATLAB path.

**VariableName — Name of variable to read**

'x' (default) | character vector | string scalar

Name of the variable to read from the MAT file, specified as a character vector or a string scalar.

**SamplesPerFrame — Number of samples per output frame**

1 (default) | scalar

Number of samples per output frame to read from the MAT file on each call to the object algorithm, specified as a positive, integer-valued scalar.

## Usage

### Syntax

```
data = mfr()
```

### Description

`data = mfr()` reads data from a specified variable stored in a MAT-file. The variable is assumed to be  $N$ -dimensional and a MATLAB built-in data type. The data is read into MATLAB by reading along the first dimension.

### Output Arguments

**data — Data read from MAT file**

scalar | vector | matrix

Data read from the MAT file, returned as a scalar, vector, or a matrix. The data can be an  $N$ -dimensional array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

### Specific to dsp.MatFileReader

`isDone` End-of-data status

### Common to All System Objects

`step` Run System object algorithm  
`release` Release resources and allow changes to System object property values and input characteristics  
`reset` Reset internal states of System object

## Examples

### Read MAT File

Read a MAT file using the `MatFileReader` object.

```
filename = [tempname '.mat']; % Create variable name
originalData = rand(40,2);
save(filename,'originalData','-v7.3'); % Write to MAT file

mfr = dsp.MatFileReader(filename,'VariableName',...
    'originalData','SamplesPerFrame', 4);
while ~isDone(mfr) % Stream data into MATLAB
    finalData = mfr();
end
```

## Version History

Introduced in R2012b

### See Also

#### Objects

`dsp.MatFileWriter`

#### Topics

“Remove High-Frequency Noise from Gyroscope Data”

“Outlier Removal Techniques with ECG Signals”

“Measure Statistics of Streaming Signals”



# dsp.MatFileWriter

**Package:** dsp

Write MAT file

## Description

The `dsp.MatFileWriter` System object writes data to a V7.3 MAT file.

To write data to a V7.3 MAT file:

- 1 Create the `dsp.MatFileWriter` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
mfw = dsp.MatFileWriter  
mfw = dsp.MatFileWriter(fname,vname)  
mfw = dsp.MatFileWriter(Name,Value)
```

### Description

`mfw = dsp.MatFileWriter` returns a MAT file writer System object, `mfw`, that writes data to a V7.3 MAT file.

`mfw = dsp.MatFileWriter(fname,vname)` returns a MAT file writer System object with the `Filename` property set to `fname` and the `VariableName` property set to `vname`.

`mfw = dsp.MatFileWriter(Name,Value)` returns a MAT file writer System object with each specified property set to the specified value. Unspecified properties have default values.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

### Filename — Name of MAT file to write

'Untitled.mat' (default) | character vector | string scalar

Specify the name of a MAT file as a character vector or a string scalar. Specify the full path for the file only if the file is not on the MATLAB path.

**VariableName — Name of variable to write**

'x' (default) | character vector | string scalar

Name of the variable to which to write, returned as a character vector or a string scalar. This variable is stored in the MAT file. You cannot overwrite a variable that is already in an existing MAT file.

## Usage

### Syntax

```
mfw(data)
```

### Description

`mfw(data)` writes one frame of data to the variable stored in the MAT file. The variable is assumed to be  $N$ -dimensional and a MATLAB built-in data type. The data is written to the file by concatenating along the first dimension.

### Input Arguments

**data — Data to be written to MAT file**

scalar | vector | matrix

Data to be written to the MAT file, specified as a scalar, vector, or a matrix. The data can be an  $N$ -dimensional array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Common to All System Objects

|                      |  |
|----------------------|--|
| <code>step</code>    | Run System object algorithm  |
| <code>release</code> | Release resources and allow changes to System object property values and input characteristics |
| <code>reset</code>   | Reset internal states of System object   |

## Examples

### Write Data Into a MAT File

First, create a variable name.

```
filename = [tempname '.mat'];
```

Next, write that variable to a MAT-file.

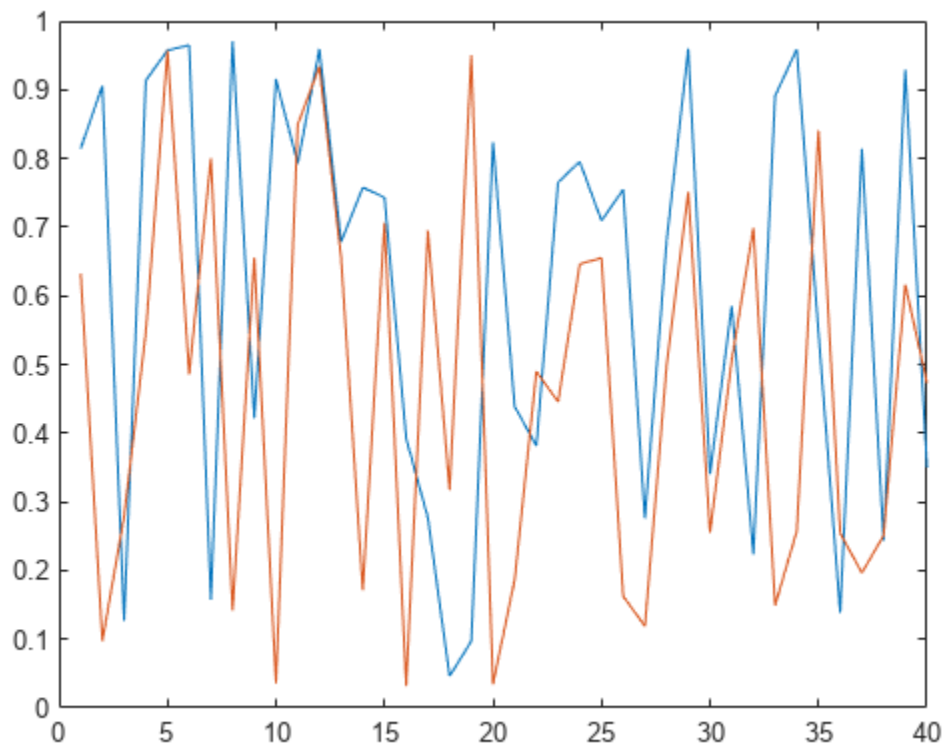
```
mfw = dsp.MatFileWriter(filename, 'VariableName', 'originalData');  
for i = 1:10  
    originalData = rand(4,2);  
    mfw(originalData);  
end  
release(mfw); % This will close the MAT file
```

Finally, load the variable back into MATLAB.

```
data = load(filename, 'originalData');
```

Plot the data.

```
plot(data.originalData);
```



## Version History

Introduced in R2012b

## See Also

### Objects

dsp.MatFileReader

# dsp.MatrixViewer

**Package:** dsp

Visualize matrix data

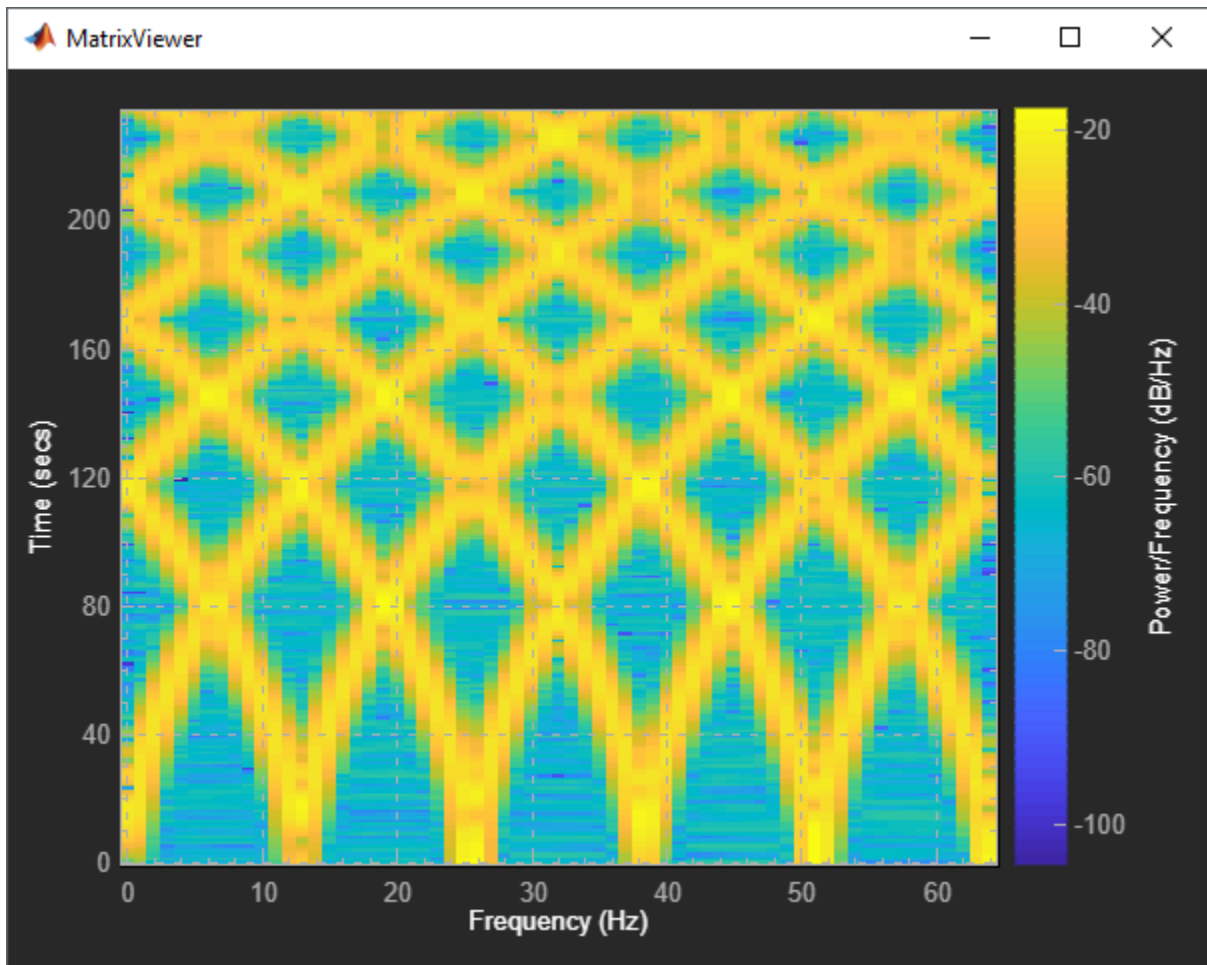
## Description

The `dsp.MatrixViewer` visualizes matrix data by mapping the matrix elements to a specified range of colors.

To visualize matrix data in the Matrix Viewer:

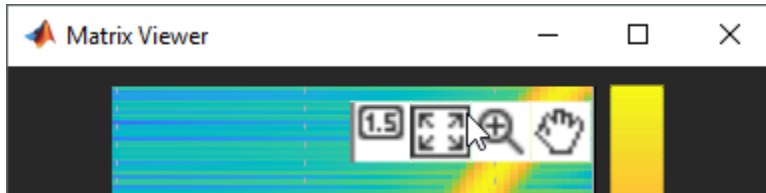
- 1 Create the `dsp.MatrixViewer` object and set its properties.
- 2 Call the object with arguments, as if it were a function.




To learn more about how System objects work, see [What Are System Objects?](#)



## Zoom and Pan

To scale the plot axes, you can use the scroll button on your mouse to zoom in/out of the plot and **CTRL+Click** and drag to pan around the plot. Additionally, you can use the buttons that appear when you hover over the upper right corner of the plot window.

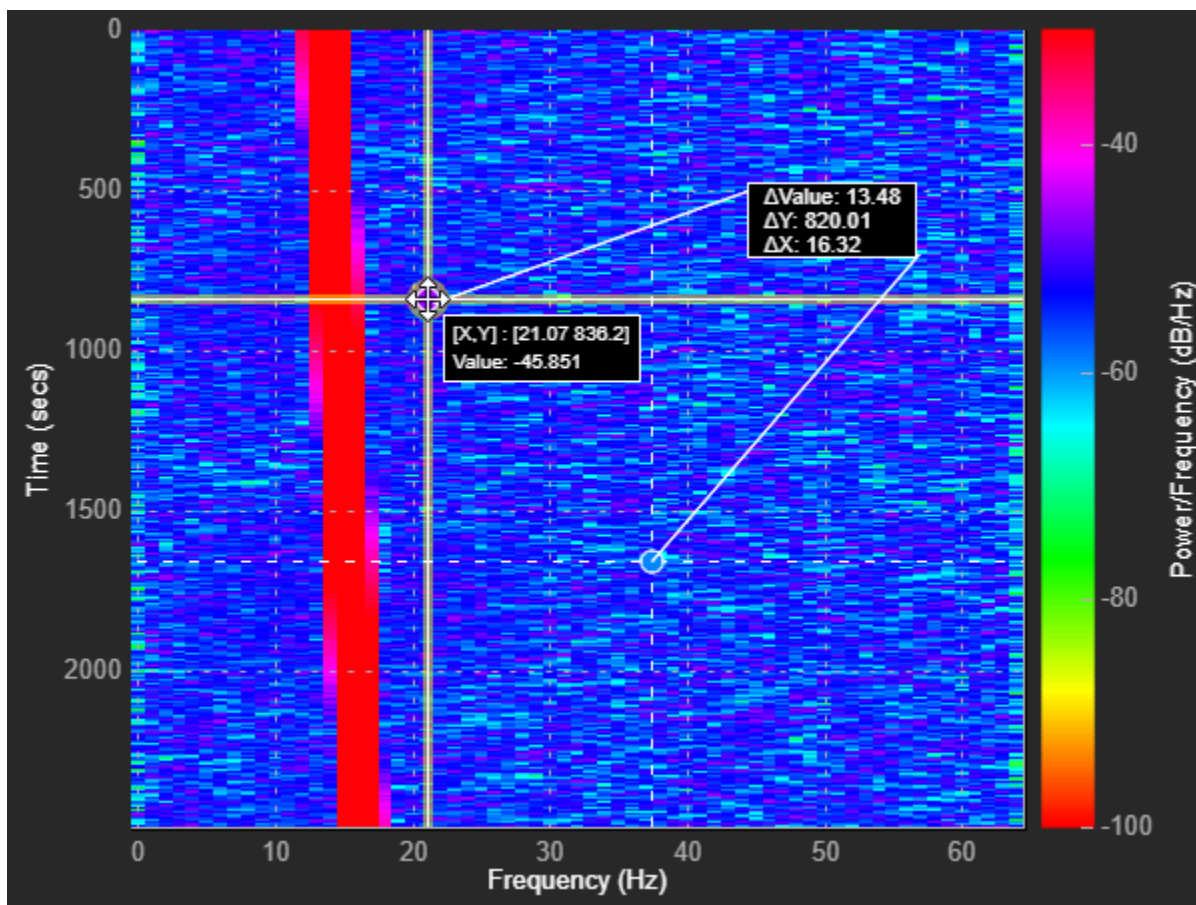


-  — Autoscale the axes to fit the data.
-  — Zoom in to the plot.
-  — Pan around the axes.

## Cursor Measurements

Activate cursor measurements by hovering over the matrix viewer and selecting the cursor button





Two horizontal and two vertical cursors appear on the plot. A dialog box shows the difference between the two intersection points. Hovering over an intersection point shows the value at that intersection point. Move the cursors by clicking and dragging the cursor lines or the intersection points to your desired location. Additionally, you can **Alt+Click** and drag the cursor dialog box to move the cursors while keeping the distance between the cursor lines constant.

## Creation

### Syntax

```
scope = dsp.MatrixViewer
scope = dsp.MatrixViewer(Name, Value)
```

### Description

`scope = dsp.MatrixViewer` creates a Matrix Viewer System object, `scope`.

`scope = dsp.MatrixViewer(Name, Value)` sets properties using one or more name-value pairs. Enclose each property name in single quotes. For example, `scope = dsp.MatrixViewer("AxisOrigin", "Lower left corner")`

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see *System Design in MATLAB Using System Objects*.

### Matrix Information

#### XDataMode — x-axis numbering mode

"Offset and resolution" (default) | "Custom"

Specify the x-axis numbering mode.

- "Offset and resolution" - Compute the x-axis data points from the XResolution and XOffset properties.
- "Custom" - Compute the x-axis data points using the CustomXData property.

Data Types: char | string

#### YDataMode — y-axis numbering mode

"Offset and resolution" (default) | "Span and resolution" | "Custom"

Specify the y-axis numbering mode:

- "Span and resolution" - Compute the y-axis data points from the YSpan and YResolution properties.
- "Offset and resolution" - Compute the y-axis data points from the YResolution and YOffset properties.
- "Custom" - Compute the y-axis data points using the CustomYData property.

Data Types: char | string

#### CustomXData — Custom data values for x-axis

[1 numberOfColumns] (default) | [min max] | monotonically increasing numeric vector

Specify custom values for the x-axis using a two-element numeric vector or a numeric vector with a finite number of elements.

If you specify a two-element vector, the numbers are used as the min and max values of the x-axis. If you specify a vector with more than two elements, the values must be monotonically increasing and the scope uses the first and last values of the vector as the minimum and maximum values, respectively. If you do not specify x-axis data limits, the scope uses the number of input columns as the maximum x-axis value.

Example: [5 156]

**Tunable:** Yes

#### Dependency

To enable this property, you must set XDataMode to "Custom".

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **CustomYData — Custom data values for y-axis**

[1 numberOfRows] (default) | [min max] | monotonically increasing numeric vector

Specify custom values for the y-axis using a two-element numeric vector or a numeric vector with a finite number of elements.

If you specify a two-element vector, the numbers are used as the min and max values of the y-axis. If you specify a vector with more than two elements, the values must be monotonically increasing and the scope uses the first and last values of the vector as minimum and maximum values, respectively. If you do not specify y-axis data limits, the scope uses the number of input rows as the maximum y-axis value.

Example: [-130 10]

**Tunable:** Yes

#### **Dependency**

To enable this property, you must set `YDataMode` to "Custom".

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **XOffset — Display offset of x-axis**

0 (default) | scalar

Specify the offset to display on the x-axis as a scalar.

**Tunable:** Yes

#### **Dependency**

To enable this property, you must set `XDataMode` to "Offset and resolution".

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **XResolution — Spacing of x-axis**

1 (default) | scalar

Specify the spacing of values along the x-axis as a scalar.

**Tunable:** Yes

#### **Dependency**

To enable this property, you must set `XDataMode` to "Offset and resolution".

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **YOffset — Display offset of y-axis**

0 (default) | scalar

Specify the offset to display on the y-axis as a scalar.

**Tunable:** No

#### **Dependency**

To enable this property, you must set `YDataMode` to "Offset and resolution".



Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **YResolution — Spacing of y-axis**

1 (default) | scalar

Specify the spacing of values along the y-axis as a scalar.

**Tunable:** No

#### **Dependency**

To enable this property, you must set `YDataMode` to "Offset and resolution" or "Span and resolution".

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **YSpan — Span of y-axis**

100 (default) | positive scalar

Specify the span of values along the y-axis as a scalar.

**Tunable:** No

#### **Dependency**

To enable this property, you must set `YDataMode` to "Span and resolution".

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

## **Visualization**

### **Name — Window name**

"Matrix Viewer" (default) | character vector | string scalar

Specify the name of the scope. This name appears as the title of the scope's figure window. To specify the title of plot, use the `Title` property.

Data Types: `char` | `string`

### **AxisOrigin — Starting location of plot**

"Upper left corner" (default) | "Lower left corner"

Specify the starting location of the plot. If you specify "Upper left corner", the plot starts in the top left corner of the axes and continues down.

Data Types: `char` | `string`

### **Position — Scope window position in pixels**

screen center (default) | [`left bottom width height`]

Specify, in pixels, the size and location of the scope window as a four-element vector of the form [`left bottom width height`]. By default, the scope window appears in the center of your screen with a width of 410 pixels and height of 300 pixels. The default values for this property may change depending on your screen resolution.

### **Title — Display title**

"" (default) | character vector | string scalar

Specify the title of the plot as a character vector or string. By default, there is no title.

**Tunable:** Yes

Data Types: char | string

**XLabel — x-axis label**

"" (default) | character vector | string

Specify the text for the scope to display below the x-axis. By default, the axes is unlabeled.

**Tunable:** Yes

Data Types: char | string

**YLabel — y-axis label**

"" (default) | character vector | string

Specify the text for the scope to display to the left of the y-axis. By default, the axes is unlabeled.

**Tunable:** Yes

Data Types: char | string

**ColorBarLabel — Color bar label**

"" (default) | character vector | string

Specify the text for the scope to display next to the color bar. By default, the color bar is unlabeled.

**Tunable:** Yes

Data Types: char | string

**Colormap — Color scheme**

"parula" (default) | colormap name | three-column matrix of RGB triplets

Color scheme for the colormap, specified as a predefined colormap name or a three-column matrix of RGB triplets.

For a list of acceptable colormap names, see `map`.

To use a custom colormap, specify a three-column matrix of RGB triplets. You can create the RGB matrix yourself, or you can call a predefined colormap function, such as `colormap`, to create the matrix.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | char | string

**ColorLimits — Color bar limits**

minimum and maximum matrix values (default) | two-element numeric vector [min max]

Specify the color bar limits as a two-element numeric vector [min max]. By default, limits are set as the minimum and maximum values of the input matrix.

Example: [1 80]

**Tunable:** Yes

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**ColorBarLocation — Color bar location**

"eastoutside" (default) | "northoutside" | "southoutside" | "westoutside"

Location of the color bar relative to the axes.

**Tunable:** Yes

Data Types: char | string

**ShowColorBar — Color bar visibility**

true (default) | false

Set this property to false to hide the color bar on the plot.

**ShowGrid — Grid visibility**

true (default) | false

Set this property to false to hide grid lines on the plot.

**Tunable:** Yes

**ShowTicks — Axes tick visibility**

true (default) | false

Set this property to false to hide the ticks on the x-axis and y-axis.

**Usage****Syntax**

scope(matrix)

**Description**

scope(matrix) displays the matrix by mapping matrix element values to a range of colors.

**Input Arguments****matrix — Matrix to visualize**

*n*-by-*m* numeric matrix

Specify an *n*-by-*m* numeric matrix to visualize.

Example: scope(rand(3,5))

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical

**Object Functions**

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named obj, use this syntax:

release(obj)

### Specific to `dsp.MatrixViewer`

`setCursorDataLabels` Customize data labels for cursor measurements

### Specific to Scopes

`show` Display scope window  
`hide` Hide scope window  
`isVisible` Determine visibility of scope

### Common to All System Objects

`step` Run System object algorithm  
`release` Release resources and allow changes to System object property values and input characteristics  
`reset` Reset internal states of System object

## Examples

### Generate Spectrogram with `dsp.MatrixViewer`

This example shows how to create a spectrogram of a quadratic chirp with the `dsp.MatrixViewer` System object.

Set up the sample rate and a chirp signal.

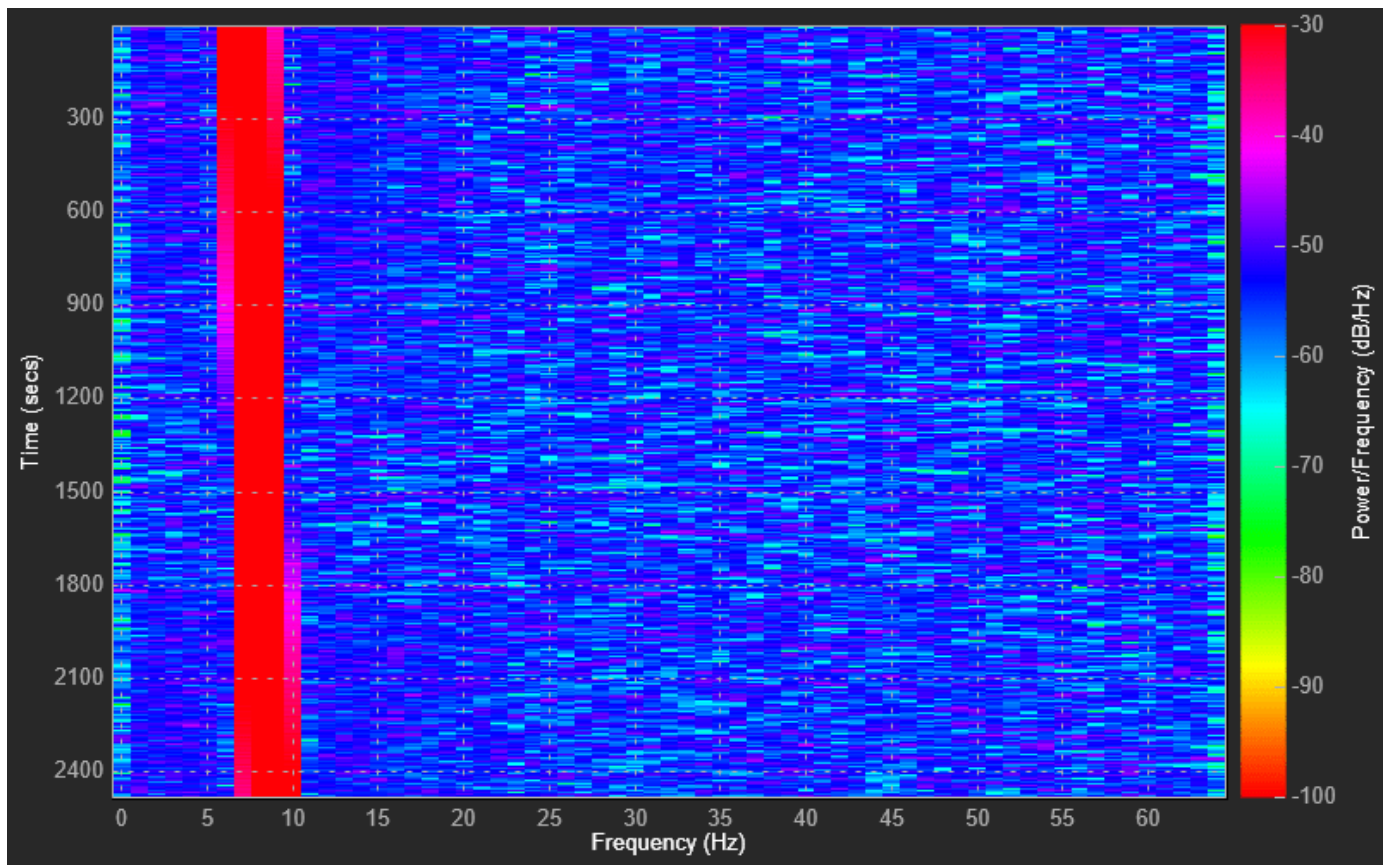
```
Fs = 233e3;  
frameSize = 20e3;  
chirp = dsp.Chirp("SampleRate",Fs,"SamplesPerFrame",frameSize,...  
    "InitialFrequency",11e3,"TargetFrequency",11e3+55e3,...  
    "Type","Quadratic");
```

Create a `dsp.MatrixViewer` scope. Set the axis labels, select a colormap, and set the limits of the colormap.

```
scope = dsp.MatrixViewer(...  
    "ColorBarLabel","Power/Frequency (dB/Hz)",...  
    "XLabel","Frequency (Hz)",...  
    "YLabel","Time (secs)",...  
    "Colormap","hsv",...  
    "ColorLimits",[-100,-30]);
```

Visualize the spectrogram of the chirp signal in the scope.

```
for idx = 1:50  
    y = chirp() + 0.05*randn(frameSize,1);  
    [~,~,~,Ps] = spectrogram(y,128,120,128,1e3);  
    val = 10*log10(abs(Ps)'+eps);  
    scope(val);  
end
```



## Version History

Introduced in R2019a

## See Also

[timescope](#) | [spectrumAnalyzer](#) | [dsp.LogicAnalyzer](#) | [dsp.DynamicFilterVisualizer](#)

# dsp.MedianFilter

**Package:** dsp

Median filter

## Description

The `dsp.MedianFilter` System object computes the moving median of the input signal along each channel, independently over time. The object uses the sliding window method to compute the moving median. In this method, a window of specified length is moved over each channel, sample by sample, and the object computes the median of the data in the window. For more details, see “Algorithms” on page 4-836.

To compute the moving median of the input:

- 1 Create the `dsp.MedianFilter` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
medFilt = dsp.MedianFilter
medFilt = dsp.MedianFilter(Len)
medFilt = dsp.MedianFilter(Name,Value)
```

### Description

`medFilt = dsp.MedianFilter` returns a median filter object, `medFilt`, using the default properties.

`medFilt = dsp.MedianFilter(Len)` sets the `WindowLength` property to `Len`.

`medFilt = dsp.MedianFilter(Name,Value)` specifies the `WindowLength` property using a `Name,Value` pair.

### Example:

```
movMin = dsp.MedianFilter('WindowLength',5);
```

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

### **WindowLength — Length of sliding window**

5 (default) | positive scalar integer

Length of the sliding window in samples, specified as a positive scalar integer.

## **Usage**

### **Syntax**

```
y = medFilt(x)
```

### **Description**

`y = medFilt(x)` computes the moving median of the input signal, `x`, using the sliding window method.

### **Input Arguments**

#### **x — Data input**

vector | matrix

Data input, specified as a vector or a matrix. If `x` is a matrix, each column is treated as an independent channel. The moving median is computed along each channel. The object accepts multichannel inputs, that is,  $m$ -by- $n$  size inputs, where  $m \geq 1$ , and  $n > 1$ .  $m$  is the number of samples in each frame (or channel), and  $n$  is the number of channels.

The object also accepts variable-size inputs. Once the object is locked, you can change the size of each input channel, but you cannot change the number of channels.

Data Types: `single` | `double`

### **Output Arguments**

#### **y — Filtered signal**

vector | matrix

Filtered signal, returned as a vector or a matrix. The size and data type of the output matches the size and data type of the input.

Data Types: `single` | `double`

## **Object Functions**

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## **Common to All System Objects**

`step`      Run System object algorithm

- release Release resources and allow changes to System object property values and input characteristics
- reset Reset internal states of System object

## Examples

### Remove High-Frequency Noise Using Median Filter

Filter high-frequency noise from a noisy sine wave signal using a median filter. Compare the performance of the median filter with an averaging filter.

#### Initialization

Set up a `dsp.MedianFilter` object, `medFilt`, and a `dsp.MovingAverage` object, `movavgWin`. These objects use the sliding window method with a window length of 7. Create a time scope for viewing the output.

```
Fs = 1000;
medFilt = dsp.MedianFilter(7);
movavgWin = dsp.MovingAverage(7);
scope = timescope('SampleRate',Fs,...
    'TimeSpanSource','Property',...
    'TimeSpanOverrunAction','Scroll',...
    'TimeSpan',1,'ShowGrid',true,...
    'YLimits',[-3 3],...
    'LayoutDimensions',[3 1],...
    'NumInputPorts',3);
scope.ActiveDisplay = 1;
scope.Title = 'Signal + Noise';
scope.ActiveDisplay = 2;
scope.Title = 'Moving Average Output (Window Length = 7)';
scope.ActiveDisplay = 3;
scope.Title = 'Median Filter Output (Window Length = 7)';

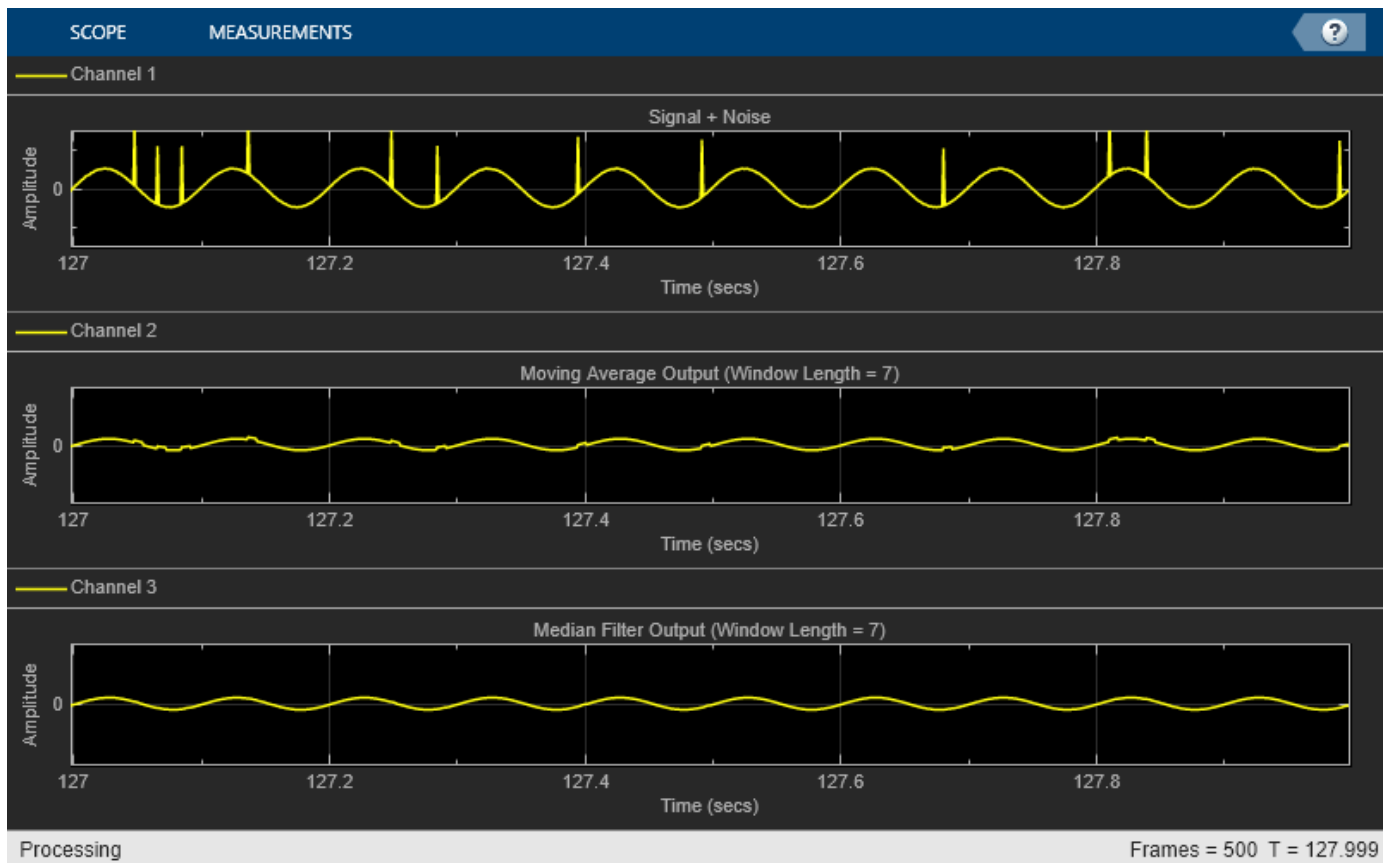
FrameLength = 256;
count = 1;
sine = dsp.SineWave('SampleRate',Fs,'Frequency',10,...
    'SamplesPerFrame',FrameLength);
```

#### Filter the Noisy Sine Wave

Generate a noisy sine wave signal with a frequency of 10 Hz. Apply the median filter and the moving average object to the signal. View the output on the time scope.

```
for i = 1:500
    hfn = 3 * (rand(FrameLength,1) < 0.02);
    x = sine() + 1e-2 * randn(FrameLength,1) + hfn;
    y1 = movavgWin(x);
    y2 = medFilt(x);
    scope(x,y1,y2);
end
```





The median filter removes the high-frequency noise more effectively than the moving average object does.

### Remove High-Frequency Noise from Gyroscope Data

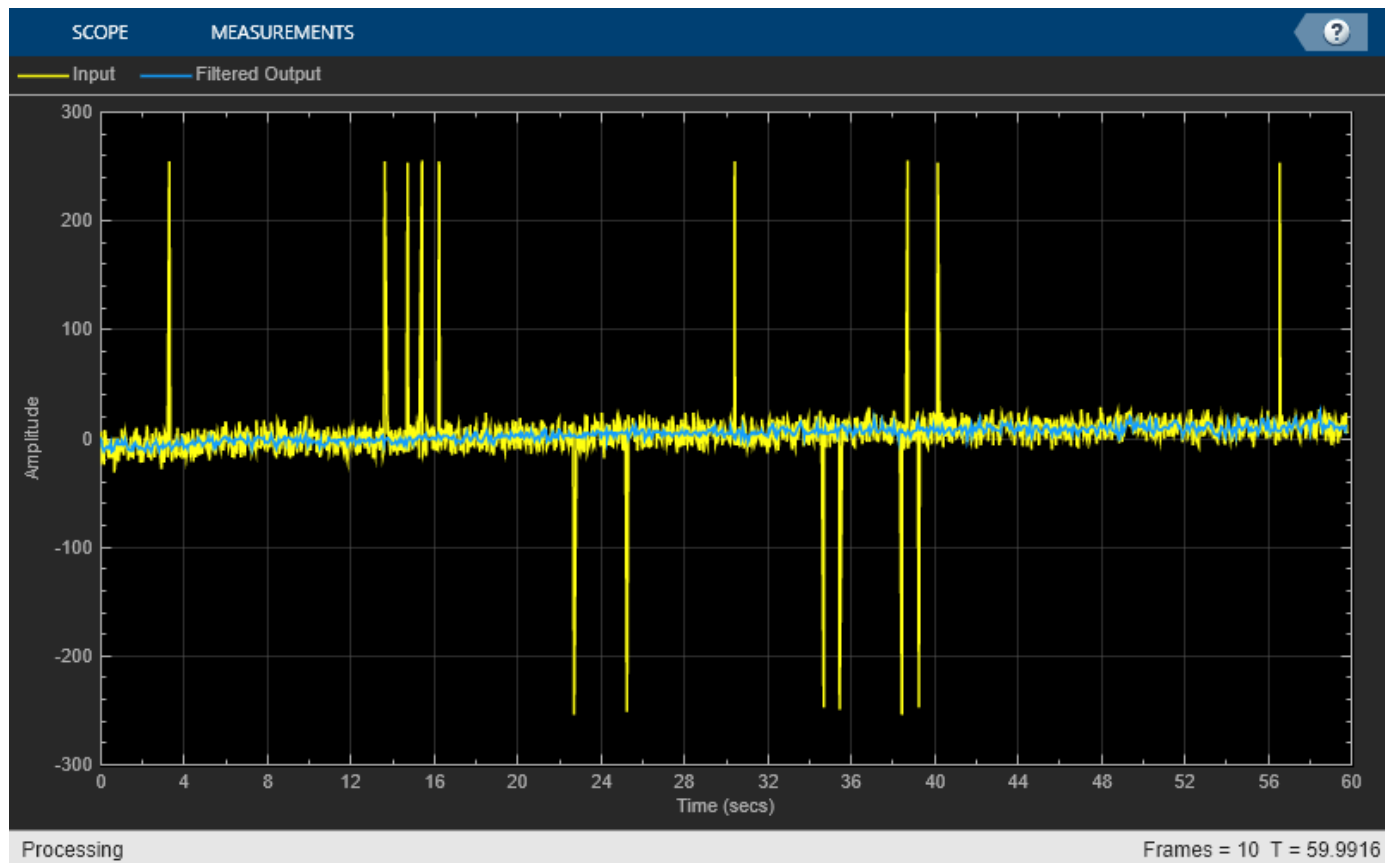
This example shows how to remove the high-frequency outliers from a streaming signal using the `dsp.MedianFilter` System object?

Use the `dsp.MatFileReader` System object to read the gyroscope MAT file. The gyroscope MAT file contains 3 columns of data, with each column containing 7140 samples. The three columns represent the X-axis, Y-axis, and Z-axis data from the gyroscope motion sensor. Choose a frame size of 714 samples so that each column of the data contains 10 frames. The `dsp.MedianFilter` System object uses a window length of 10. Create a `timescope` object to view the filtered output.

```
reader = dsp.MatFileReader('SamplesPerFrame',714,...
    'Filename','LSM9DS1gyroData73.mat',...
    'VariableName','data');
medFilt = dsp.MedianFilter(10);
scope = timescope('NumInputPorts',1,...
    'SampleRate',119,...
    'YLimits',[-300 300],...
    'ChannelNames',{'Input','Filtered Output'},...
    'TimeSpanSource','Property',...
    'TimeSpan',60,'ShowLegend',true);
```

Filter the gyroscope data using the `dsp.MedianFilter` System object. View the filtered Z-axis data in the time scope.

```
for i = 1:10
    gyroData = reader();
    filteredData = medFilt(gyroData);
    scope([gyroData(:,3),filteredData(:,3)]);
end
```



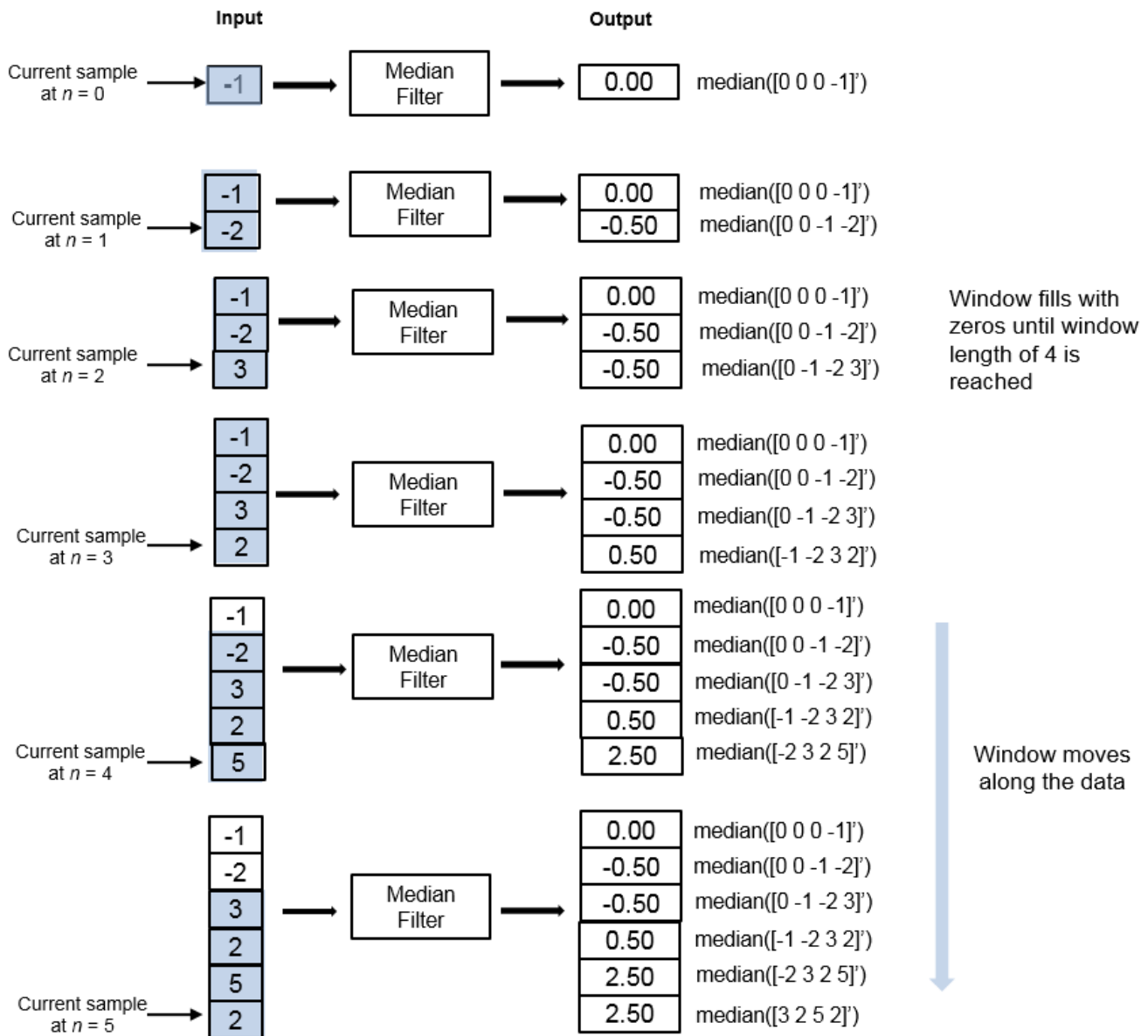
The original data contains several outliers. Zoom in on the data to confirm that the median filter removes all the outliers.

## Algorithms

### Sliding Window Method

In the sliding window method, the output for each input sample is the median of the current sample and the  $Len - 1$  previous samples.  $Len$  is the length of the window in samples. To compute the first  $Len - 1$  outputs, when the window does not have enough data yet, the algorithm fills the window with zeros. As an example, to compute the median value when the second input sample comes in, the algorithm fills the window with  $Len - 2$  zeros. The data vector,  $x$ , is then the two data samples followed by  $Len - 2$  zeros. This object performs median filtering on the input data over time.

Consider an example of computing the moving median of a streaming input data using the sliding window method. The algorithm uses a window length of 4. With each input sample that comes in, the window of length 4 moves along the data.



## Version History

Introduced in R2016b

## References

- [1] Bodenham, Dean. "Adaptive Filtering and Change Detection for Streaming Data." PH.D. Thesis. Imperial College, London, 2012.

### Extended Capabilities

#### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

This object supports C and C++ code generation.

### See Also

#### Objects

`dsp.MovingMaximum` | `dsp.MovingMinimum` | `dsp.MovingAverage` | `dsp.MovingRMS` | `dsp.MovingVariance` | `dsp.MovingStandardDeviation`

#### Blocks

Median Filter | Median | Moving Maximum | Moving Minimum | Moving Average | Moving RMS | Moving Variance | Moving Standard Deviation

#### Topics

“What Are Moving Statistics?”

“Streaming Signal Statistics”

“Remove High-Frequency Noise from Gyroscope Data”

# dsp.MovingAverage

**Package:** dsp

Moving average

## Description

The `dsp.MovingAverage` System object computes the moving average of the input signal along each channel, independently over time. The object uses either the sliding window method or the exponential weighting method to compute the moving average. In the sliding window method, a window of specified length is moved over the data, sample by sample, and the average is computed over the data in the window. In the exponential weighting method, the object multiplies the data samples with a set of weighting factors. The average is computed by summing the weighted data. For more details on these methods, see “Algorithms” on page 4-843.

To compute the moving average of the input:

- 1 Create the `dsp.MovingAverage` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
movAvg = dsp.MovingAverage
movAvg = dsp.MovingAverage(Len)
movAvg = dsp.MovingAverage(Len,Overlap)
movAvg = dsp.MovingAverage(Name,Value)
```

### Description

`movAvg = dsp.MovingAverage` returns a moving average object, `movAvg`, using the default properties.

`movAvg = dsp.MovingAverage(Len)` sets the `WindowLength` property to `Len`.

`movAvg = dsp.MovingAverage(Len,Overlap)` sets the `WindowLength` property to `Len` and the `OverlapLength` property to `Overlap`.

`movAvg = dsp.MovingAverage(Name,Value)` specifies additional properties using `Name,Value` pairs. Unspecified properties have default values.

```
Example: movAvg = dsp.MovingAverage('Method','Exponential
weighting','ForgettingFactor',0.9);
```

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see *System Design in MATLAB Using System Objects*.

### Method — Averaging method

'Sliding window' (default) | 'Exponential weighting'

Averaging method, specified as 'Sliding window' or 'Exponential weighting'.

- 'Sliding window' — A window of length specified by `SpecifyWindowLength` is moved over the input data along each channel. For every sample the window moves by, the object computes the average over the data in the window.
- 'Exponential weighting' — The object multiplies the samples with a set of weighting factors. The magnitude of the weighting factors decreases exponentially as the age of the data increases, never reaching zero. To compute the average, the algorithm sums the weighted data.

For more details on these methods, see “Algorithms” on page 4-843.

### SpecifyWindowLength — Specify window length

true (default) | false

Flag to specify a window length, specified as a scalar boolean.

- true — The length of the sliding window is equal to the value you specify in the `WindowLength` property.
- false — The length of the sliding window is infinite. In this mode, the average is computed using the current sample and all the past samples.

### Dependencies

This property applies when you set `Method` to 'Sliding window'.

### WindowLength — Length of the sliding window

4 (default) | positive scalar integer

Length of the sliding window in samples, specified as a positive scalar integer.

### Dependencies

This property applies when you set `Method` to 'Sliding window' and `SpecifyWindowLength` to true.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### OverlapLength — Overlap length between windows

WindowLength - 1 (default) | nonnegative integer

Overlap length between sliding windows, specified as a nonnegative integer. The value of overlap length varies in the range  $[0, \text{WindowLength} - 1]$ . If not specified, the overlap length is `WindowLength - 1`.

**Dependencies**

This property applies when you set `Method` to 'Sliding window' and `SpecifyWindowLength` to true.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**ForgettingFactor — Exponential weighting factor**

0.9 (default) | positive real scalar in the range (0,1]

Exponential weighting factor, specified as a positive real scalar in the range (0,1]. A forgetting factor of 0.9 gives more weight to the older data than does a forgetting factor of 0.1. A forgetting factor of 1.0 indicates infinite memory. All the past samples are given an equal weight.

Since this property is tunable, you can change its value even when the object is locked.

**Tunable:** Yes

**Dependencies**

This property applies when you set `Method` to 'Exponential weighting'.

Data Types: `single` | `double`

**Usage****Syntax**

$$y = \text{movAve}(x)$$
**Description**

$y = \text{movAve}(x)$  computes the moving average of the input signal,  $x$ , using either the sliding window method or exponential weighting method.

**Input Arguments****x — Data input**

vector | matrix

Data input, specified as a vector or a matrix. If  $x$  is a matrix, each column is treated as an independent channel. The moving average is computed along each channel.

The object accepts variable-size inputs. Once the object is locked, you can change the size of each input channel, but you cannot change the number of channels.

Data Types: `single` | `double`

**Output Arguments****y — Moving average**

vector | matrix

Moving average of the input signal, returned as a vector or a matrix.

When you input a signal of size  $m$ -by- $n$  to the object, and if you set `Method` to 'Sliding window' and `SpecifyWindowLength` to `true`, the output has an upper bound size of  $\text{ceil}(m/\text{hop size})$ -by- $n$ . Hop size is window length – overlap length. In other cases, the output has a size of  $m$ -by- $n$ .

When you generate code from this object, the variable-size behavior of the output in the generated code depends on the input frame length and whether the size of the input signal is fixed or variable. For more details, see “Code Generation” on page 4-845.

Data Types: `single` | `double`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Common to All System Objects

`step`     Run System object algorithm  
`release`   Release resources and allow changes to System object property values and input characteristics  
`reset`     Reset internal states of System object

## Examples

### Moving Average of Noisy Ramp Signal

Compute the moving average of a noisy ramp signal using the `dsp.MovingAverage` object.

#### Initialization

Set up `movavgWindow`, `movavgWindow_overlap`, and `movavgExp` objects. `movavgWindow` uses the sliding window method with a window length of 50 samples and a default overlap length of 49 samples, which is one sample less than the specified window length. `movavgWindow_overlap` uses a window length of 50 samples and an overlap length of 45 samples. `movavgExp` uses the exponentially weighting method with a forgetting factor of 0.95.

Create a time scope for viewing the output.

```
FrameLength = 1001;
Fs = 1000;
movavgWindow = dsp.MovingAverage(50);
movavgWindow_overlap = dsp.MovingAverage(50,45);
movavgExp = dsp.MovingAverage('Method','Exponential weighting',...
    'ForgettingFactor',0.95);
scope = timescope('SampleRate',[Fs, Fs, Fs/(50-45), Fs],...
    'TimeSpanOvrrunAction','Scroll',...
    'ShowGrid',true,...
    'YLimits',[-0.5 1.5]);
title = 'Moving Average';
scope.Title = title;
scope.ChannelNames = {'Original Signal',...
    'Sliding window of 50 samples with default overlap',...

```

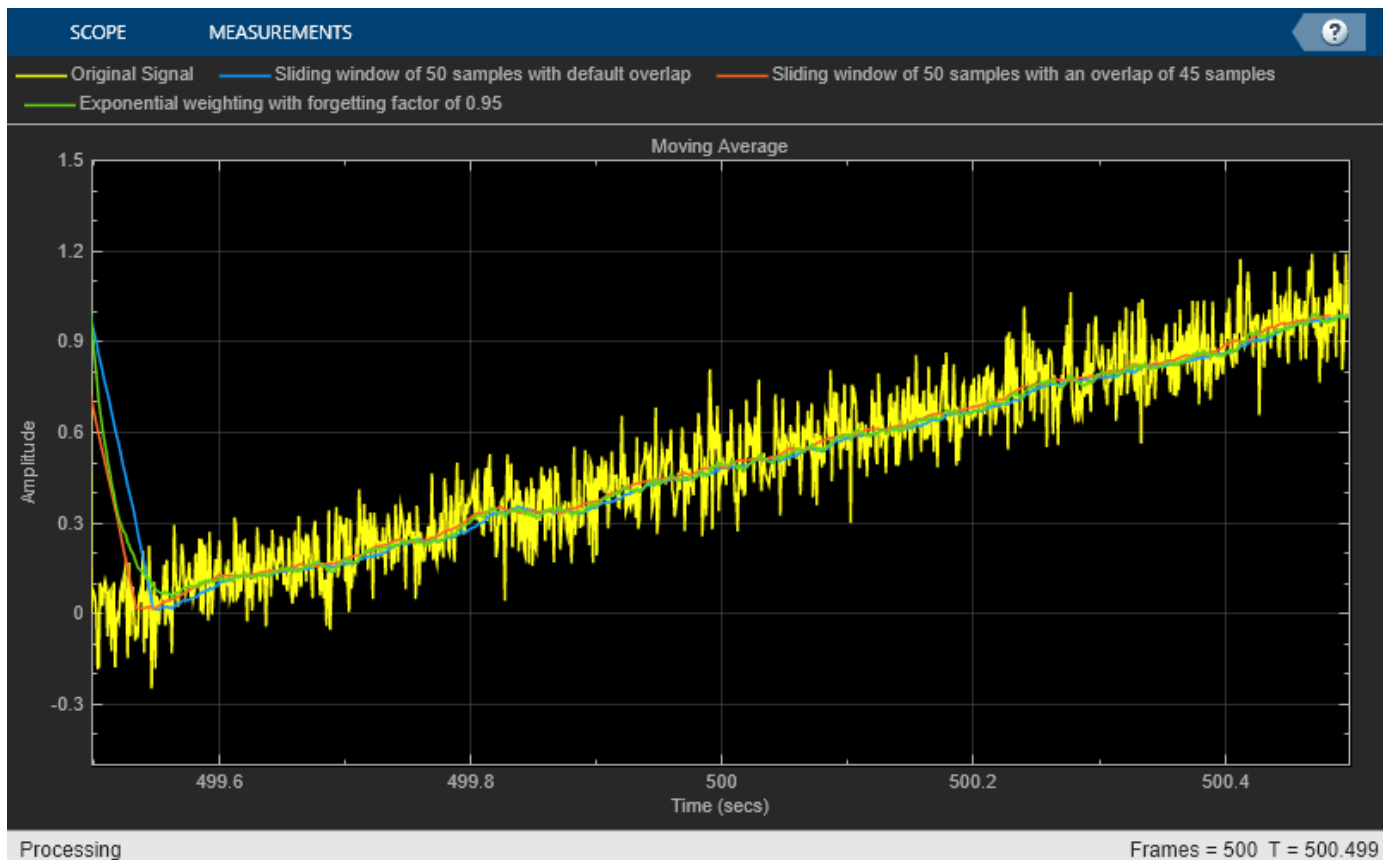


```
'Sliding window of 50 samples with an overlap of 45 samples',...
'Exponential weighting with forgetting factor of 0.95'};
```

### Compute the Average

Generate a ramp signal with an amplitude of 1 and a time span of 2 seconds. Apply the sliding window average and exponentially weighted average to the ramp. View the output in the time scope.

```
for i = 1:500
    t = (0:0.001:1)';
    unitstep = t>=0;
    ramp = t.*unitstep;
    x = ramp + 0.1 * randn(FrameLength,1);
    y1 = movavgWindow(x);
    y2 = movavgWindow_overlap(x);
    y3 = movavgExp(x);
    scope(x,y1,y2,y3);
end
```



## Algorithms

### Sliding Window Method

In the sliding window method, the output for each input sample is the average of the current sample and  $Len - 1$  previous samples.  $Len$  is the length of the window in samples. To compute the first output sample, the algorithm waits until it receives the hop size number of input samples. Hop size is defined

as window length - overlap length. Remaining samples in the window are considered to be zero. As an example, if the window length is 5 and the overlap length is 2, then the algorithm waits until it receives 3 samples of input to compute the first sample of the output. After generating the first output, it generates the subsequent output samples for every hop size number of input samples.

When you do not specify the window length, the algorithm chooses an infinite window length. In this mode, the output is the moving average of the current sample and all the previous samples in the channel.

For an example, see “Sliding Window Method and Exponential Weighting Method”.

### Exponential Weighting Method

In the exponential weighting method, the moving average is computed recursively using these formulas:

$$w_{N,\lambda} = \lambda w_{N-1,\lambda} + 1$$
$$\bar{x}_{N,\lambda} = \left(1 - \frac{1}{w_{N,\lambda}}\right) \bar{x}_{N-1,\lambda} + \left(\frac{1}{w_{N,\lambda}}\right) x_N$$

- $\bar{x}_{N,\lambda}$  — Moving average at the current sample
- $x_N$  — Current data input sample
- $\bar{x}_{N-1,\lambda}$  — Moving average at the previous sample
- $\lambda$  — Forgetting factor
- $w_{N,\lambda}$  — Weighting factor applied to the current data sample
- $\left(1 - \frac{1}{w_{N,\lambda}}\right) \bar{x}_{N-1,\lambda}$  — Effect of the previous data on the average

For the first sample, where  $N = 1$ , the algorithm chooses  $w_{N,\lambda} = 1$ . For the next sample, the weighting factor is updated and used to compute the average, as per the recursive equation. As the age of the data increases, the magnitude of the weighting factor decreases exponentially and never reaches zero. In other words, the recent data has more influence on the current average than the older data.

The value of the forgetting factor determines the rate of change of the weighting factors. A forgetting factor of 0.9 gives more weight to the older data than does a forgetting factor of 0.1. A forgetting factor of 1.0 indicates infinite memory. All the previous samples are given an equal weight.

For an example, see “Sliding Window Method and Exponential Weighting Method”.

## Version History

### Introduced in R2016b

#### R2022b: Change in variable-size behavior for output signal in generated code

Starting in R2022b, if you generate code from this object with the `Method` property set to 'Sliding window' and the `SpecifyWindowLength` property set to `true`, and if you input a signal with a frame length that is a multiple of the hop size (window length - overlap length), this object generates

a fixed-size output signal in the generated code . For more details, see “Code Generation” on page 4-845.

## References

- [1] Bodenham, Dean. “Adaptive Filtering and Change Detection for Streaming Data.” PH.D. Thesis. Imperial College, London, 2012.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- See “System Objects in MATLAB Code Generation” (MATLAB Coder).
- When you set `Method` to 'Sliding window' and `SpecifyWindowLength` to `true`, and generate code from this object, the variable-size behavior of the output depends on the input frame length and whether the size of the input signal is fixed or variable.

See this table for more details.

| Input signal  | Input size   | Output signal   |
|---------------|--------------|---|
| Fixed-size    | $m$ -by- $n$ | When the input frame length is a multiple of the hop size, the output signal has a fixed-size of $(m/\text{hop size})$ -by- $n$ .<br><br>When input frame length is not a multiple of the hop size, the output signal is variable-sized and has an upper bound of $\text{ceil}(m/\text{hop size})$ -by- $n$ . |
| Variable-size | $m$ -by- $n$ | Output is a variable-size signal.<br><br>Output has an upper bound size of $\text{ceil}(m/\text{hop size})$ -by- $n$ .  |

## See Also

### Objects

`dsp.MovingRMS` | `dsp.MovingMaximum` | `dsp.MovingMinimum` | `dsp.MovingStandardDeviation` | `dsp.MovingVariance` | `dsp.MedianFilter` | `powermeter`

### Blocks

Moving Average | Moving RMS | Moving Maximum | Moving Minimum | Moving Standard Deviation | Moving Variance | Median Filter

### Topics

“What Are Moving Statistics?”

“Sliding Window Method and Exponential Weighting Method”  
“How Is a Moving Average Filter Different from an FIR Filter?”  
“Measure Statistics of Streaming Signals”  
“Streaming Signal Statistics”

# dsp.MovingMaximum

**Package:** dsp

Moving maximum

## Description

The `dsp.MovingMaximum` System object determines the moving maximum of the input signal along each channel, independently over time. The object uses the sliding window method to determine the moving maximum. In this method, a window of specified length is moved over each channel, sample by sample, and the object determines the maximum of the data in the window. For more details, see “Algorithms” on page 4-850.

To determine the moving maximum of the input:

- 1 Create the `dsp.MovingMaximum` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
movMax = dsp.MovingMaximum
movMax = dsp.MovingMaximum(Len)
movMax = dsp.MovingMaximum(Name,Value)
```

### Description

`movMax = dsp.MovingMaximum` returns a moving maximum object, `movMax`, using the default properties.

`movMax = dsp.MovingMaximum(Len)` sets the `WindowLength` property to `Len`.

`movMax = dsp.MovingMaximum(Name,Value)` specifies additional properties using `Name, Value` pairs. Unspecified properties have default values.

Example: `movMax = dsp.MovingMaximum('SpecifyWindowLength',1,'WindowLength',10);`

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

**SpecifyWindowLength — Specify window length**`true (default) | false`

Flag to specify a window length, specified as a scalar boolean.

- `true` — The length of the sliding window is equal to the value you specify in the `WindowLength` property.
- `false` — The length of the sliding window is infinite. In this mode, the object determines the maximum of the current sample and all the past samples.

**WindowLength — Length of the sliding window**`4 (default) | positive scalar integer`

Length of the sliding window in samples, specified as a positive scalar integer.

**Dependencies**

This property applies when you set `SpecifyWindowLength` to `true`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**Usage****Syntax**

```
y = movMax(x)
```

**Description**

`y = movMax(x)` determines the moving maximum of the input signal, `x`, using the sliding window method.

**Input Arguments****x — Data input**`vector | matrix`

Data input, specified as a vector or a matrix. If `x` is a matrix, each column is treated as an independent channel. The moving maximum is determined along each channel. The object also accepts variable-size inputs. Once the object is locked, you can change the size of each input channel, but you cannot change the number of channels.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**Output Arguments****y — Moving maximum output**`vector | matrix`

Moving maximum of the input signal, returned as a vector or a matrix.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Common to All System Objects

|                      |  |
|----------------------|--|
| <code>step</code>    | Run System object algorithm  |
| <code>release</code> | Release resources and allow changes to System object property values and input characteristics |
| <code>reset</code>   | Reset internal states of System object   |

## Examples

### Moving Maximum of Sine Wave Signal

Compute the moving maximum of a sum of three sine waves with varying amplitude. Use a sliding window of length 30.

#### Initialization

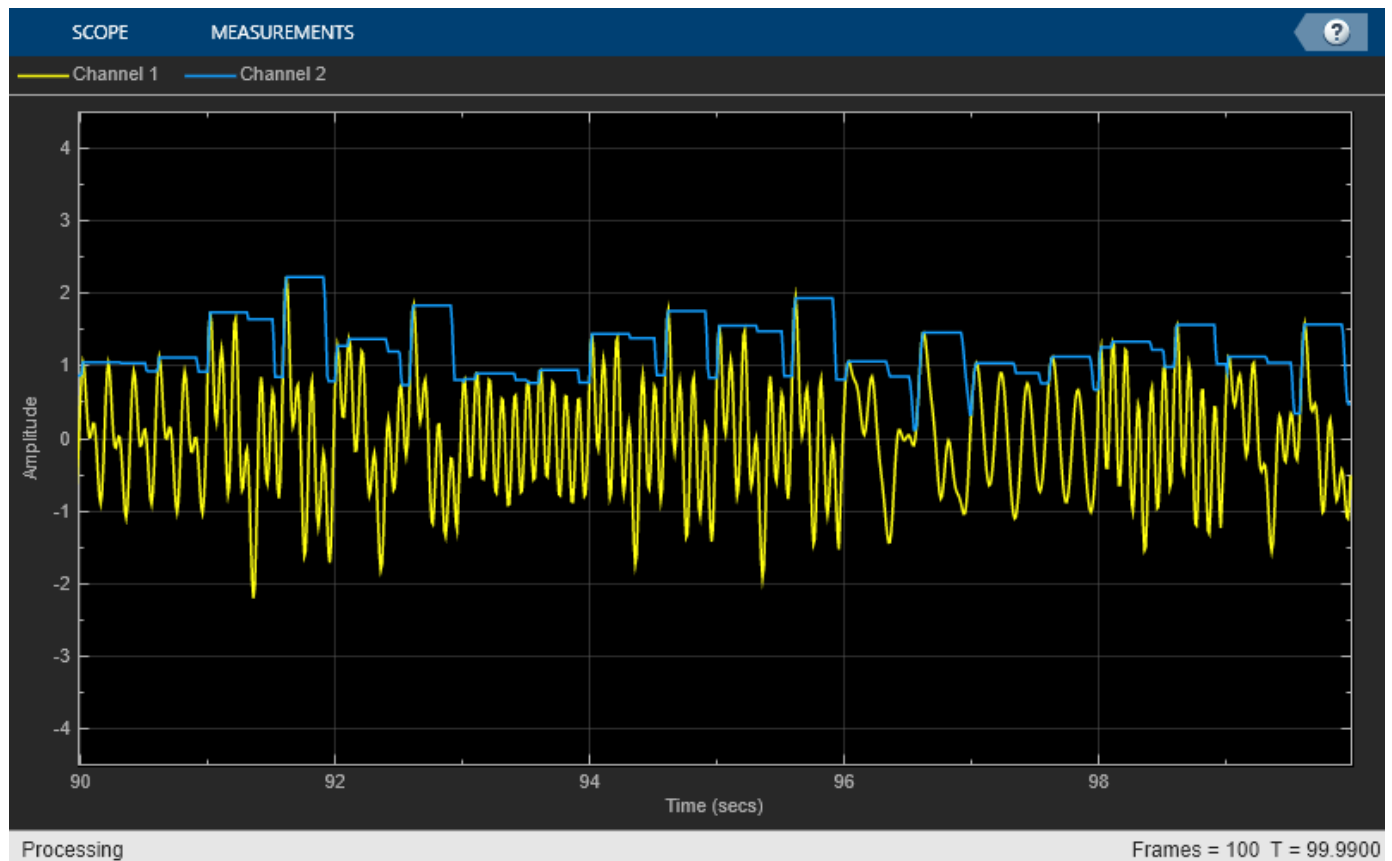
Set up an input signal that is a sum of three sine waves with frequencies at 2 Hz, 5 Hz, and 10 Hz. The sampling frequency is 100 Hz. Create a `dsp.MovingMaximum` object with a window length of 30. Create a time scope for viewing the output.

```
sin = dsp.SineWave('SampleRate',100,...
    'Frequency',[2 5 10],...
    'SamplesPerFrame',100);
movMax = dsp.MovingMaximum(30);
scope = timescope('SampleRate',100,...
    'TimeSpanOvrrunAction','Scroll',...
    'TimeSpanSource','Property',...
    'TimeSpan',10,'ShowGrid',true,...
    'YLimits',[-4.5 4.5]);
```

#### Compute the Moving Maximum

Each sine wave component of the input signal has a different amplitude that varies with the iteration. Use the `movMax` object to determine the maximum value of the current sample and the past 29 samples of the input signal.

```
for index = 1:100
    sin.Amplitude = rand(1,3);
    x = sum(sin(),2);
    xmax = movMax(x);
    scope([x,xmax])
end
```



## Algorithms

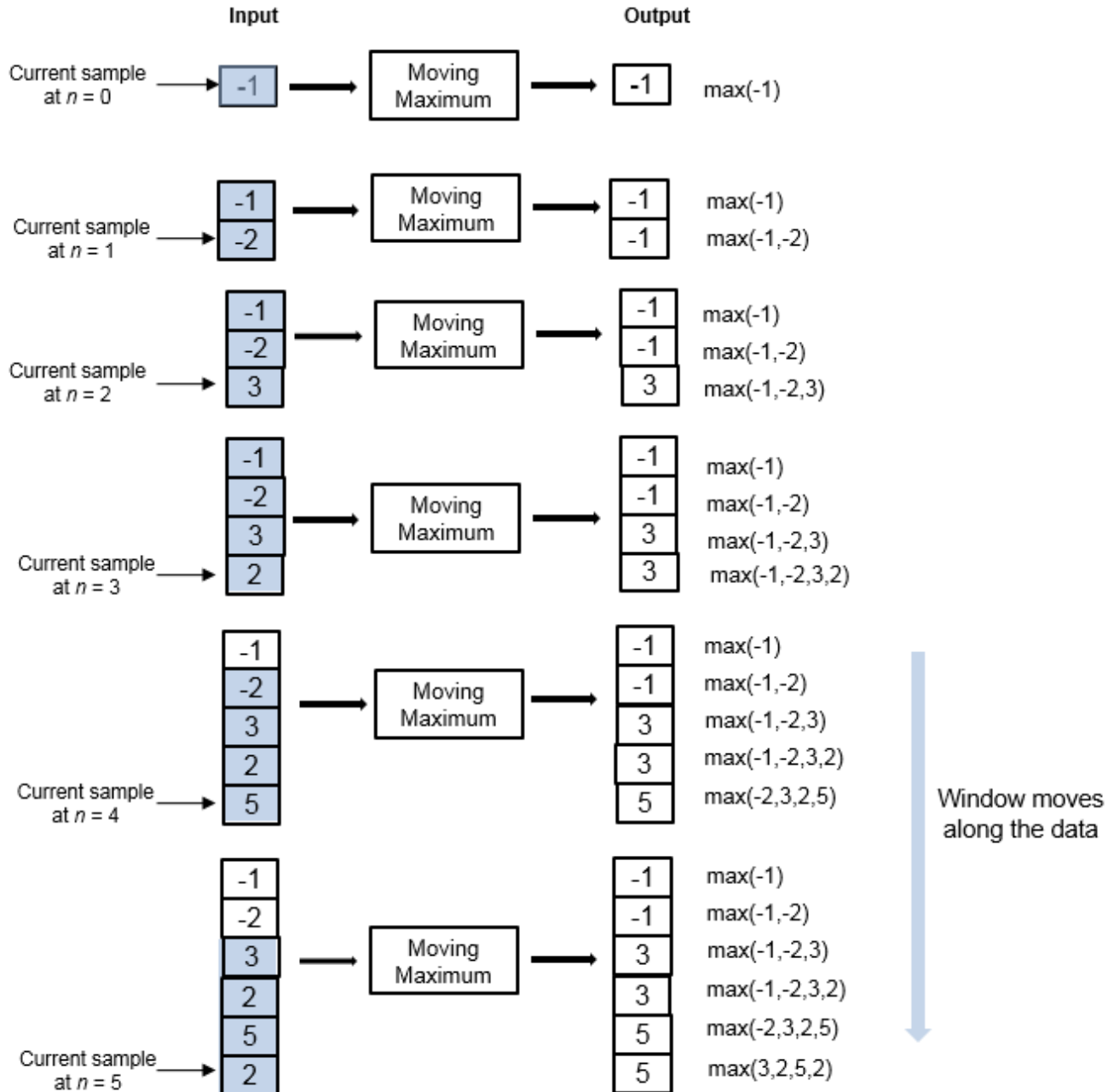
### Sliding Window Method

In the sliding window method, the output for each input sample is the maximum of the current sample and the  $Len - 1$  previous samples.  $Len$  is the length of the window in samples. When the algorithm computes the first  $Len - 1$  outputs, the length of the window is the length of the data that is available.

When you do not specify the window length, the algorithm chooses an infinite window length. In this mode, the output is the maximum of the current sample and all the previous samples in the channel.

Consider an example of computing the moving maximum of a streaming input data using the sliding window method. The algorithm uses a window length of 4. With each input sample that comes in, the window of length 4 moves along the data.





## Version History

Introduced in R2016b

## References

- [1] Bodenham, Dean. "Adaptive Filtering and Change Detection for Streaming Data." PH.D. Thesis. Imperial College, London, 2012.

### Extended Capabilities

#### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

### See Also

#### Objects

`dsp.MovingMinimum` | `dsp.MovingAverage` | `dsp.MovingRMS` |  
`dsp.MovingStandardDeviation` | `dsp.MovingVariance` | `dsp.MedianFilter` | `powermeter`

#### Blocks

Moving Maximum | Moving Minimum | Moving Average | Moving RMS | Moving Standard Deviation |  
Moving Variance | Median Filter

#### Topics

“Streaming Signal Statistics”  
“What Are Moving Statistics?”

# dsp.MovingMinimum

**Package:** dsp

Moving minimum

## Description

The `dsp.MovingMinimum` System object determines the moving minimum of the input signal along each channel, independently over time. The object uses the sliding window method to determine the moving minimum. In this method, a window of specified length is moved over each channel, sample by sample, and the object determines the minimum of the data in the window. For more details, see “Algorithms” on page 4-856.

To determine the moving minimum of the input:

- 1 Create the `dsp.MovingMinimum` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
movMin = dsp.MovingMinimum
movMin = dsp.MovingMinimum(Len)
movMin = dsp.MovingMinimum(Name,Value)
```

### Description

`movMin = dsp.MovingMinimum` returns a moving minimum object, `movMin`, using the default properties.

`movMin = dsp.MovingMinimum(Len)` sets the `WindowLength` property to `Len`.

`movMin = dsp.MovingMinimum(Name,Value)` specifies additional properties using `Name, Value` pairs. Unspecified properties have default values.

Example: `movMin = dsp.MovingMinimum('SpecifyWindowLength',1,'WindowLength',10);`

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

**SpecifyWindowLength — Specify window length**`true (default) | false`

Flag to specify a window length, specified as a scalar boolean.

- `true` — The length of the sliding window is equal to the value you specify in the `WindowLength` property.
- `false` — The length of the sliding window is infinite. In this mode, the object determines the minimum of the current sample and all the past samples.

**WindowLength — Length of the sliding window**`4 (default) | positive scalar integer`

Length of the sliding window in samples, specified as a positive scalar integer.

**Dependencies**

This property applies when you set `SpecifyWindowLength` to `true`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**Usage****Syntax**

```
y = movMin(x)
```

**Description**

`y = movMin(x)` determines the moving minimum of the input signal, `x`, using the sliding window method.

**Input Arguments****x — Data input**`vector | matrix`

Data input, specified as a vector or a matrix. If `x` is a matrix, each column is treated as an independent channel. The moving minimum is determined along each channel. The object also accepts variable-size inputs. Once the object is locked, you can change the size of each input channel, but you cannot change the number of channels.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**Output Arguments****y — Moving minimum output**`vector | matrix`

Moving minimum output, returned as a vector or a matrix.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Common to All System Objects

|                      |  |
|----------------------|--|
| <code>step</code>    | Run System object algorithm  |
| <code>release</code> | Release resources and allow changes to System object property values and input characteristics |
| <code>reset</code>   | Reset internal states of System object   |

## Examples

### Moving Minimum of Sine Wave Signal

Compute the moving minimum of a sum of three sine waves with varying amplitude. Use a sliding window of length 30.

#### Initialization

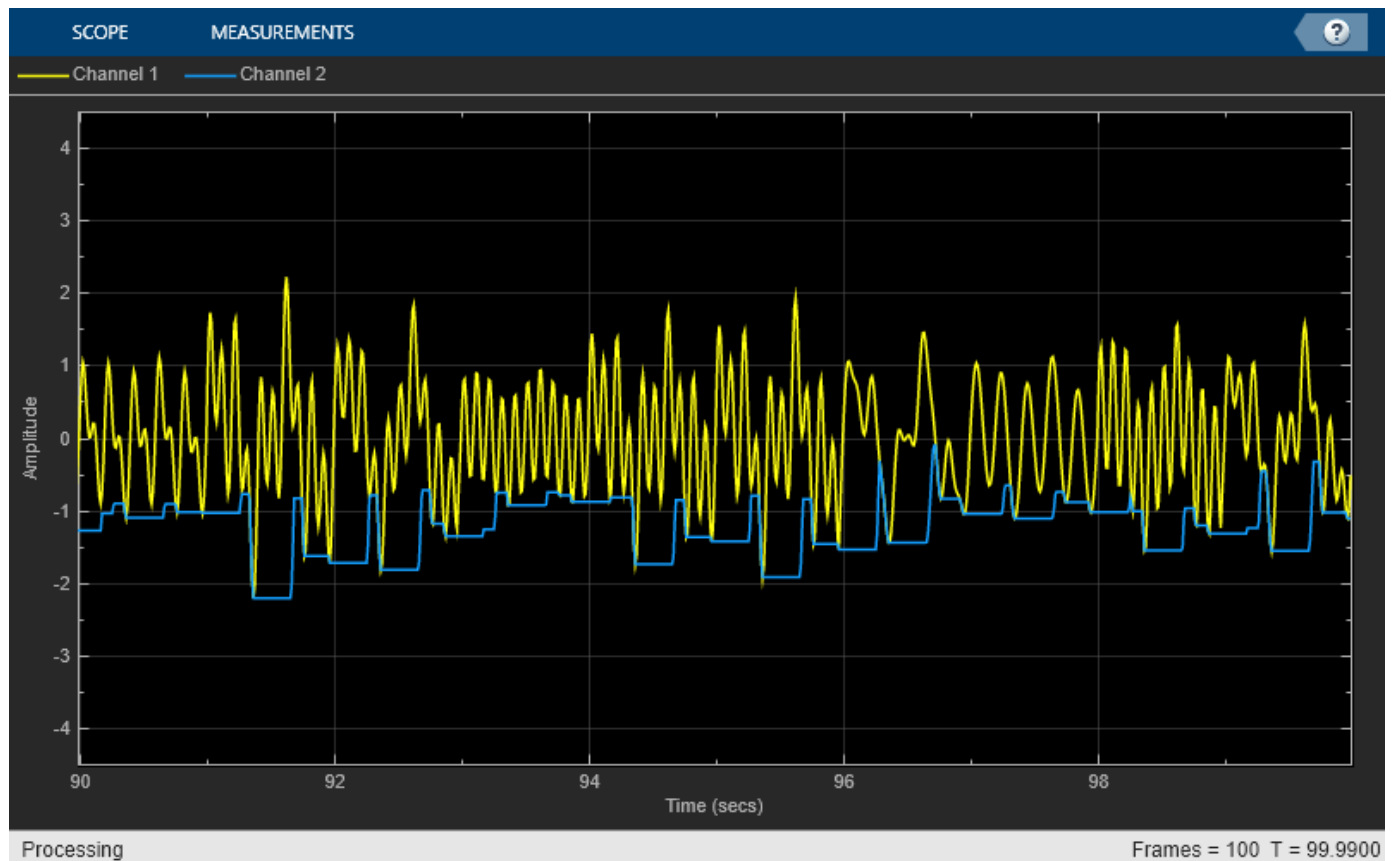
Set up an input signal that is a sum of three sine waves with frequencies at 2 Hz, 5 Hz, and 10 Hz. The sampling frequency is 100 Hz. Create a `dsp.MovingMinimum` object with a window length of 30. Create a time scope for viewing the output.

```
sin = dsp.SineWave('SampleRate',100,...
    'Frequency',[2 5 10],...
    'SamplesPerFrame',100);
movMin = dsp.MovingMinimum(30);
scope = timescope('SampleRate',100,...
    'TimeSpanSource','property','TimeSpan',10,...
    'TimeSpanOvverrunAction','Scroll',...
    'ShowGrid',true,'YLimits',[-4.5 4.5]);
```

#### Compute the Moving Minimum

Each sine wave component of the input signal has a different amplitude that varies with the iteration. Use the `movMin` object to determine the minimum value of the current sample and the past 29 samples of the input signal.

```
for index = 1:100
    sin.Amplitude = rand(1,3);
    x = sum(sin(),2);
    xmin = movMin(x);
    scope([x,xmin])
end
scope.AxesScaling = 'onceatstop';
```



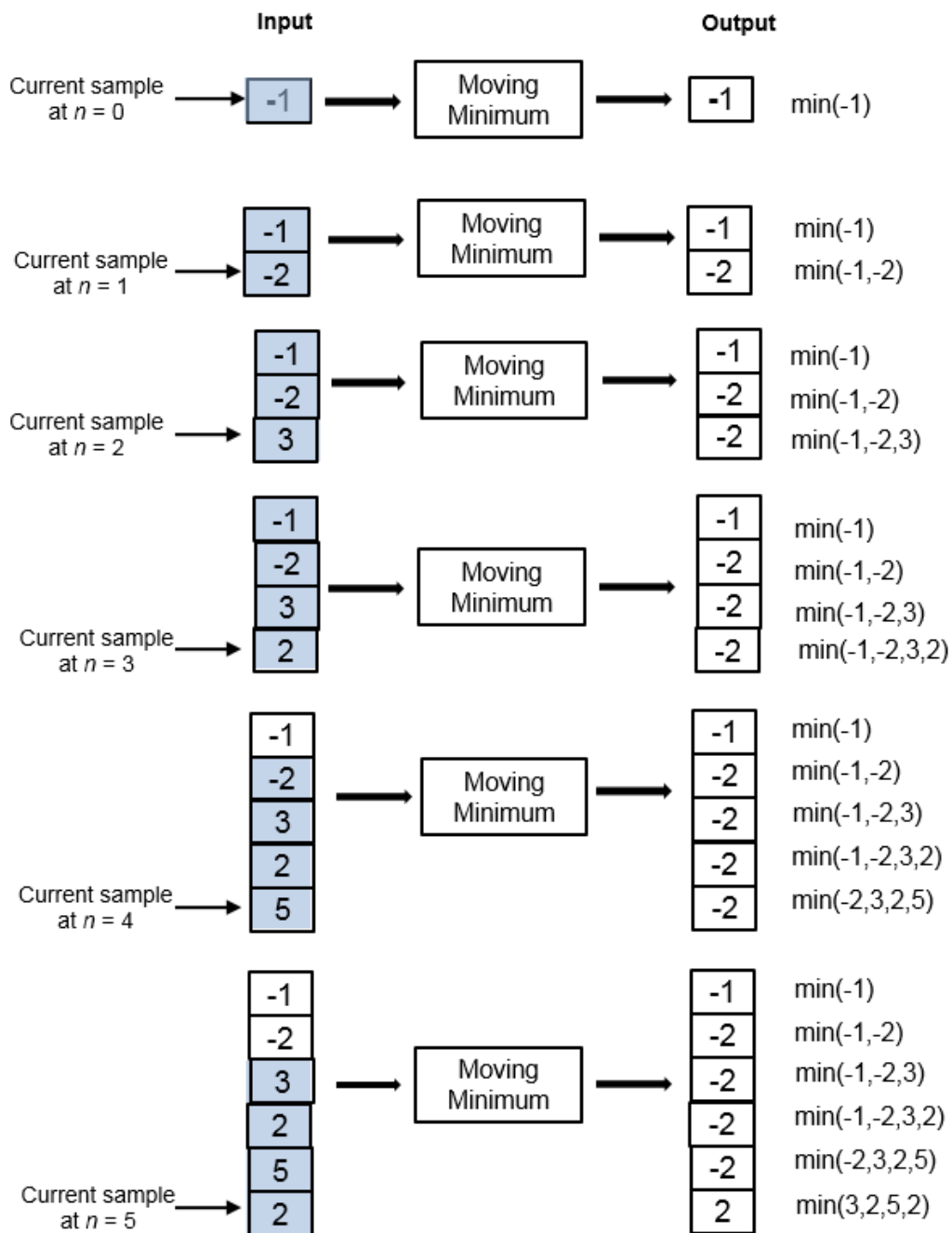
## Algorithms

### Sliding Window Method


In the sliding window method, the output for each input sample is the minimum of the current sample and the  $Len - 1$  previous samples.  $Len$  is the length of the window in samples. When the algorithm computes the first  $Len - 1$  outputs, the length of the window is the length of the data that is available.

When you do not specify the window length, the algorithm chooses an infinite window length. In this mode, the output is the minimum of the current sample and all the previous samples in the channel.

Consider an example of computing the moving minimum of a streaming input data using the sliding window method. The algorithm uses a window length of 4. With each input sample that comes in, the window of length 4 moves along the data.



Window moves  
along the data



## Version History

Introduced in R2016b

### References

[1] Bodenham, Dean. "Adaptive Filtering and Change Detection for Streaming Data." PH.D. Thesis. Imperial College, London, 2012.

### Extended Capabilities

#### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

### See Also

#### Objects

`dsp.MovingMaximum` | `dsp.MovingAverage` | `dsp.MovingRMS` |  
`dsp.MovingStandardDeviation` | `dsp.MovingVariance` | `dsp.MedianFilter`

#### Blocks

Moving Minimum | Minimum | Moving Maximum | Moving Average | Moving RMS | Moving Standard Deviation | Moving Variance | Median Filter

#### Topics

"Streaming Signal Statistics"  
"What Are Moving Statistics?"



# dsp.MovingRMS

**Package:** dsp

Moving root mean square

## Description

The `dsp.MovingRMS` System object computes the moving root mean square (RMS) of the input signal along each channel, independently over time. The object uses either the sliding window method or the exponential weighting method to compute the moving RMS. In the sliding window method, a window of specified length is moved over the data, sample by sample, and the RMS is computed over the data in the window. In the exponential weighting method, the object squares the data samples, multiplies them with a set of weighting factors, and sums the weighed data. The object then computes the RMS by taking the square root of the sum. For more details on these methods, see “Algorithms” on page 4-864.

To compute the moving RMS of the input:

- 1 Create the `dsp.MovingRMS` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
movRMS = dsp.MovingRMS
movRMS = dsp.MovingRMS(Len)
MovRMS = dsp.MovingRMS(Len,Overlap)
movRMS = dsp.MovingRMS(Name,Value)
```

### Description

`movRMS = dsp.MovingRMS` returns a moving RMS object, `movRMS`, using the default properties.

`movRMS = dsp.MovingRMS(Len)` sets the `WindowLength` property to `Len`.

`MovRMS = dsp.MovingRMS(Len,Overlap)` sets the `WindowLength` property to `Len` and the `OverlapLength` property to `Overlap`.

`movRMS = dsp.MovingRMS(Name,Value)` specifies additional properties using `Name, Value` pairs. Unspecified properties have default values.

```
Example: movRMS = dsp.MovingRMS('Method','Exponential
weighting','ForgettingFactor',0.9);
```

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

### Method — Moving RMS method

'Sliding window' (default) | 'Exponential weighting'

Moving RMS method, specified as 'Sliding window' or 'Exponential weighting'.

- 'Sliding window' — A window of length specified by `SpecifyWindowLength` is moved over the input data along each channel. For every sample the window moves by, the object computes the RMS over the data in the window.
- 'Exponential weighting' — The object multiplies the squares of the samples with a set of weighting factors. The magnitude of the weighting factors decreases exponentially as the age of the data increases, never reaching zero. To compute the RMS, the algorithm sums the weighted data, and takes a square root of the sum.

For more details on these methods, see “Algorithms” on page 4-864.

### SpecifyWindowLength — Specify window length

true (default) | false

Flag to specify a window length, specified as a scalar Boolean.

- `true` — The length of the sliding window is equal to the value you specify in the `WindowLength` property.
- `false` — The length of the sliding window is infinite. In this mode, the RMS is computed using the current sample and all past samples.

### Dependencies

This property applies when you set `Method` to 'Sliding window'.

### WindowLength — Length of the sliding window

4 (default) | positive scalar integer

Length of the sliding window in samples, specified as a positive scalar integer.

### Dependencies

This property applies when you set `Method` to 'Sliding window' and `SpecifyWindowLength` to `true`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### OverlapLength — Overlap length between windows

`WindowLength - 1` (default) | nonnegative integer

Overlap length between sliding windows, specified as a nonnegative integer. The value of overlap length varies in the range  $[0, \text{WindowLength} - 1]$ . If not specified, the overlap length is  $\text{WindowLength} - 1$ .

### Dependencies

This property applies when you set `Method` to 'Sliding window' and `SpecifyWindowLength` to `true`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### ForgettingFactor — Exponential weighting factor

0.9 (default) | positive real scalar in the range (0,1]

Exponential weighting factor, specified as a positive real scalar in the range (0,1].

A forgetting factor of 0.9 gives more weight to the older data than does a forgetting factor of 0.1. A forgetting factor of 1.0 indicates infinite memory. All the past samples are given an equal weight.

Since this property is tunable, you can change its value even when the object is locked.

**Tunable:** Yes

### Dependencies

This property applies when you set `Method` to 'Exponential weighting'.

Data Types: `single` | `double`

## Usage

### Syntax

`y = movRMS(x)`

### Description

`y = movRMS(x)` computes the moving RMS of the input signal, `x`, using either the sliding window method or exponential weighting method.

### Input Arguments

#### **x** — Data input

vector | matrix

Data input, specified as a vector or a matrix. If `x` is a matrix, each column is treated as an independent channel. The moving RMS is computed along each channel.

The object accepts variable-size inputs. Once the object is locked, you can change the size of each input channel, but you cannot change the number of channels.

Data Types: `single` | `double`

## Output Arguments

### **y** — Moving RMS

vector | matrix

Moving RMS of the input signal, returned as a vector or a matrix.

When you input a signal of size  $m$ -by- $n$  to the object, and if you set `Method` to 'Sliding window' and `SpecifyWindowLength` to `true`, the output has an upper bound size of `ceil(m/hop size)`-by- $n$ . Hop size is window length – overlap length. In other cases, the output has a size of  $m$ -by- $n$ .

When you generate code from this object, the variable-size behavior of the output in the generated code depends on the input frame length and whether the size of the input signal is fixed or variable. For more details, see “Code Generation” on page 4-867.

Data Types: `single` | `double`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Common to All System Objects

|                      |  |
|----------------------|--|
| <code>step</code>    | Run System object algorithm  |
| <code>release</code> | Release resources and allow changes to System object property values and input characteristics |
| <code>reset</code>   | Reset internal states of System object   |

## Examples

### Moving RMS of Noisy Square Wave Signal

Compute the moving RMS of a noisy square wave signal with varying amplitude using the `dsp.MovingRMS` object.

#### Initialization

Set up `movrmsWin`, `movrmsWin_overlap`, and `movrmsExp` objects. `movrmsWin` uses the sliding window method with a window length of 20 samples and a default overlap length of 19 samples, which is one sample less than the specified window length. `movrmsWin_overlap` uses a window length of 20 samples and an overlap length of 15 samples. `movrmsExp` uses the exponentially weighting method with a forgetting factor of 0.995.

Create a time scope for viewing the output.

```
FrameLength = 10;  
Fs = 100;  
movrmsWin = dsp.MovingRMS(20);  
movrmsWin_overlap = dsp.MovingRMS(20,15);  
movrmsExp = dsp.MovingRMS('Method','Exponential weighting',...  
    'ForgettingFactor',0.995);
```

```

scope = timescope('SampleRate',[Fs,Fs,Fs/(20-15),Fs],...
    'TimeSpanOvverrunAction','Scroll',...
    'TimeSpanSource','Property',...
    'TimeSpan',100,...
    'ShowGrid',true,...
    'YLimits',[-1.0 5.5]);
title = 'Moving RMS';
scope.Title = title;
scope.ChannelNames = {'Original Signal',...
    'Sliding window of 20 samples with default overlap',...
    'Sliding window of 20 samples with an overlap of 15 samples',...
    'Exponential weighting with forgetting factor of 0.995'};

```

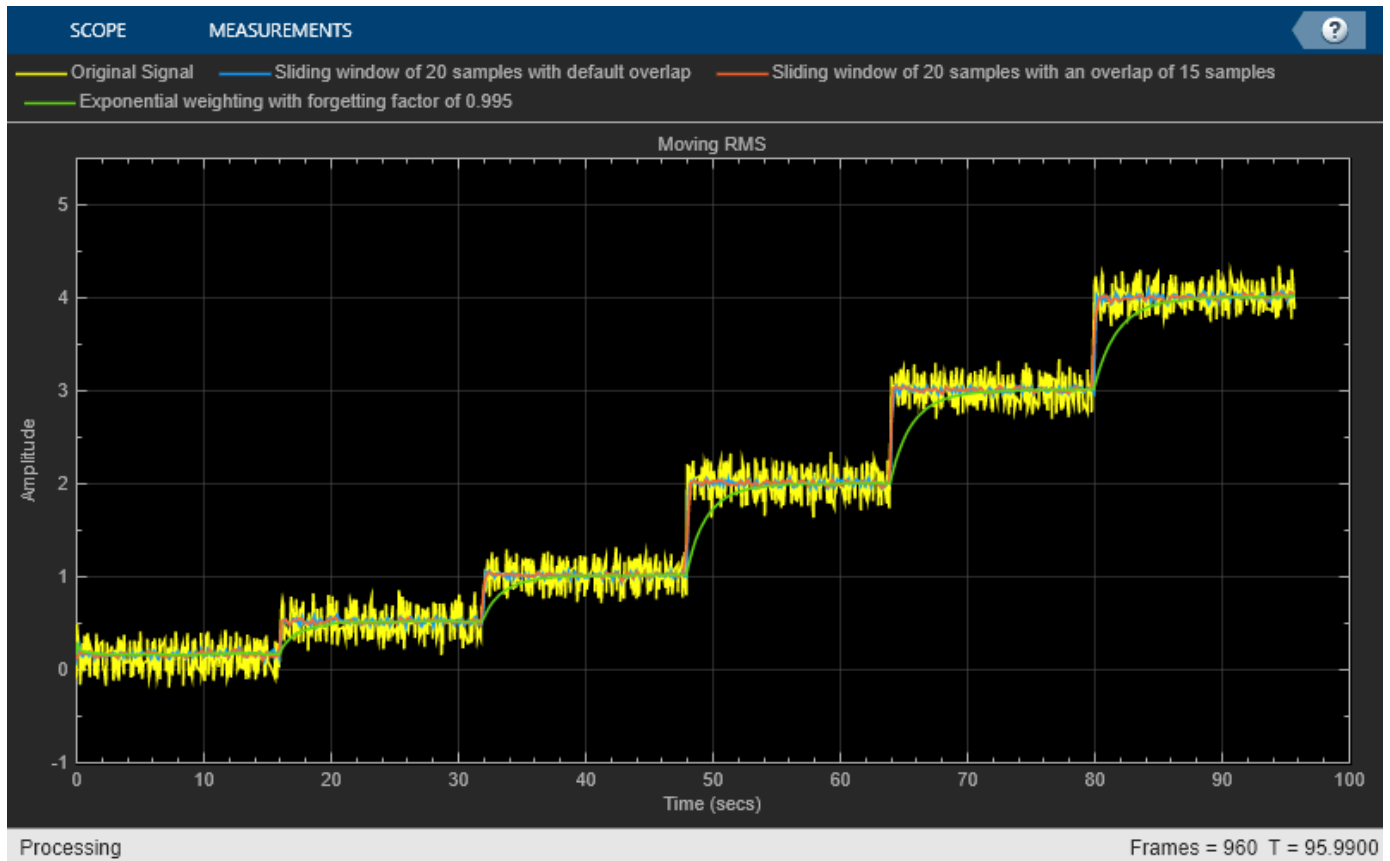
### Compute the RMS

Generate a noisy square wave signal. Vary the amplitude of the square wave after a given number of frames. Apply the sliding window method and the exponential weighting method to this signal. View the output in the time scope.

```

count = 1;
Vect = [1/8 1/2 1 2 3 4];
for index = 1:length(Vect)
    V = Vect(index);
    for i = 1:160
        x = V + 0.1 * randn(FrameLength,1);
        y1 = movrmsWin(x);
        y2 = movrmsWin_overlap(x);
        y3 = movrmsExp(x);
        scope(x,y1,y2,y3);
    end
end

```



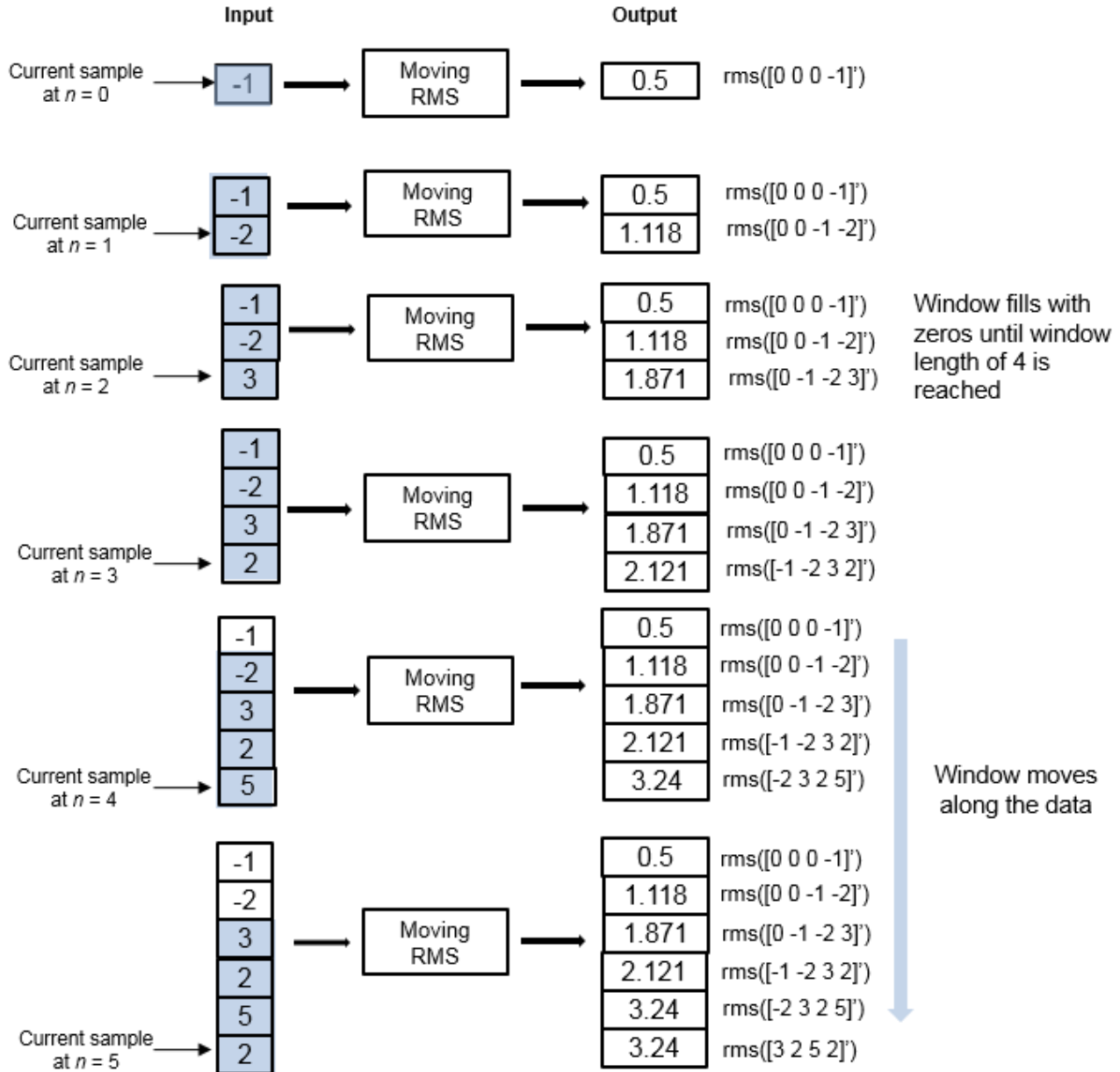
## Algorithms

### Sliding Window Method

In the sliding window method, the output for each input sample is the RMS of the current sample and  $Len - 1$  previous samples.  $Len$  is the length of the window in samples. To compute the first output sample, the algorithm waits until it receives the hop size number of input samples. Hop size is defined as window length - overlap length. Remaining samples in the window are considered to be zero. As an example, if the window length is 5 and the overlap length is 2, then the algorithm waits until it receives 3 samples of input to compute the first sample of the output. After generating the first output, it generates the subsequent output samples for every hop size number of input samples.

When you do not specify the window length, the algorithm chooses an infinite window length. In this mode, the output is the moving RMS of the current sample and all the previous samples in the channel.

Consider an example of computing the moving RMS of a streaming input data using the sliding window method. The algorithm uses a window length of 4 and an overlap length of 3. With each input sample that comes in, the window of length 4 moves along the data.



### Exponential Weighting Method

In the exponential weighting method, the moving RMS is computed recursively using these formulas:

$$w_{N,\lambda} = \lambda w_{N-1,\lambda} + 1$$

$$x_{rmsN,\lambda} = \sqrt{\left(1 - \frac{1}{w_{N,\lambda}}\right)x_{rmsN-1,\lambda} + \left(\frac{1}{w_{N,\lambda}}\right)x_N^2}$$

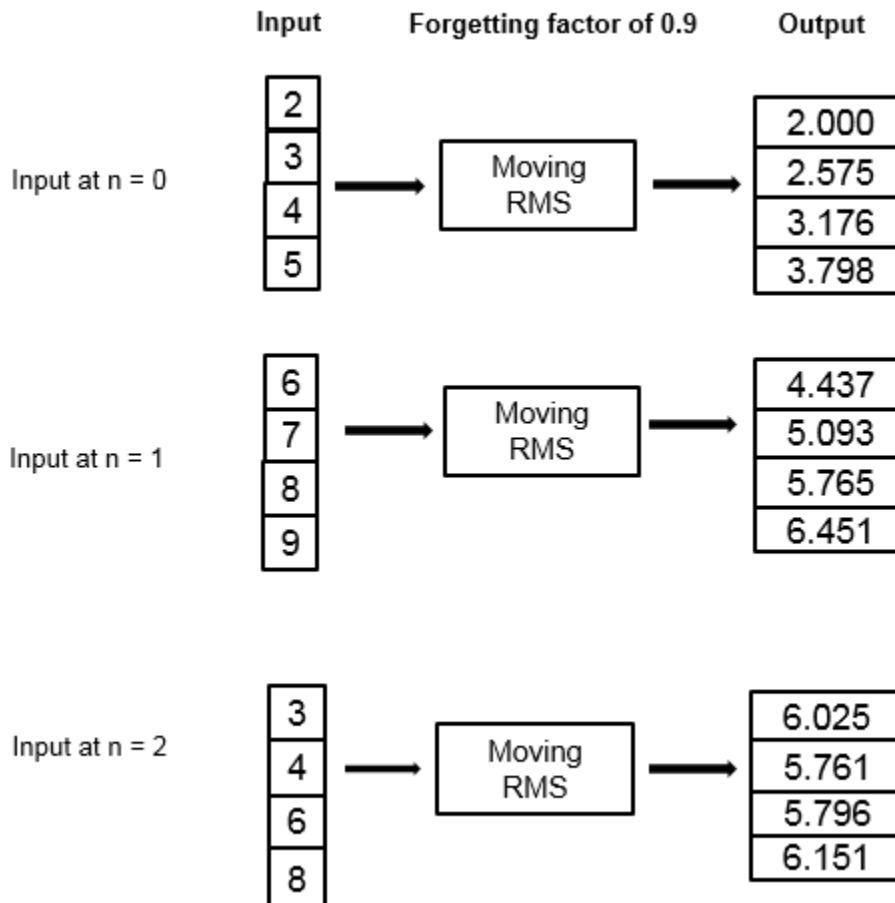
- $x_{rmsN,\lambda}$  – Moving RMS at the current sample
- $x_N^2$  – Square of the current input data sample

- $x_{rms_{N-1,\lambda}}$  — Moving RMS at the previous sample
- $\lambda$  — Forgetting factor
- $w_{N,\lambda}$  — Weighting factor applied to the current data sample
- $\left(1 - \frac{1}{w_{N,\lambda}}\right)x_{rms_{N-1,\lambda}}$  — Effect of the previous data on the RMS

For the first sample, where  $N = 1$ , the algorithm chooses  $w_{N,\lambda} = 1$ . For the next sample, the weighting factor is updated and used to compute the RMS, as per the recursive equation. As the age of the data increases, the magnitude of the weighting factor decreases exponentially and never reaches zero. In other words, the recent data has more influence on the current RMS than the older data.

The value of the forgetting factor determines the rate of change of the weighting factors. A forgetting factor of 0.9 gives more weight to the older data than does a forgetting factor of 0.1. A forgetting factor of 1.0 indicates infinite memory. All the previous samples are given an equal weight.

Here is an example of computing the moving RMS using the exponential weighting method. The forgetting factor is 0.9.





## Version History

Introduced in R2016b

### R2022b: Change in variable-size behavior for output signal in generated code

Starting in R2022b, if you generate code from this object with the `Method` property set to 'Sliding window' and the `SpecifyWindowLength` property set to `true`, and if you input a signal with a frame length that is a multiple of the hop size (window length – overlap length), this object generates a fixed-size output signal in the generated code. For more details, see “Code Generation” on page 4-867.

## References

[1] Bodenham, Dean. “Adaptive Filtering and Change Detection for Streaming Data.” PH.D. Thesis. Imperial College, London, 2012.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- See “System Objects in MATLAB Code Generation” (MATLAB Coder).
- When you set `Method` to 'Sliding window' and `SpecifyWindowLength` to `true`, and generate code from this object, the variable-size behavior of the output depends on the input frame length and whether the size of the input signal is fixed or variable.

See this table for more details.

| Input signal  | Input size   | Output signal   |
|---------------|--------------|---|
| Fixed-size    | $m$ -by- $n$ | When the input frame length is a multiple of the hop size, the output signal has a fixed-size of $(m/\text{hop size})$ -by- $n$ .<br><br>When input frame length is not a multiple of the hop size, the output signal is variable-sized and has an upper bound of $\text{ceil}(m/\text{hop size})$ -by- $n$ . |
| Variable-size | $m$ -by- $n$ | Output is a variable-size signal.<br><br>Output has an upper bound size of $\text{ceil}(m/\text{hop size})$ -by- $n$ .  |

## See Also

### Objects

`dsp.MovingAverage` | `dsp.MovingMaximum` | `dsp.MovingMinimum` |  
`dsp.MovingStandardDeviation` | `dsp.MovingVariance` | `dsp.MedianFilter`

### Blocks

Moving RMS | RMS | Moving Average | Moving Maximum | Moving Minimum | Moving Standard  
Deviation | Moving Variance | Median Filter

### Topics

“What Are Moving Statistics?”

“Streaming Signal Statistics”

“Sliding Window Method and Exponential Weighting Method”

“Energy Detection in the Time Domain”

# dsp.MovingStandardDeviation

**Package:** dsp

Moving standard deviation

## Description

The `dsp.MovingStandardDeviation` System object computes the moving standard deviation of the input signal along each channel, independently over time. The object uses either the sliding window method or the exponential weighting method to compute the moving standard deviation. In the sliding window method, a window of specified length is moved over the data, sample by sample, and the object computes the standard deviation over the data in the window. In the exponential weighting method, the object computes the exponentially weighted moving variance, and takes the square root. For more details on these methods, see “Algorithms” on page 4-874.

To compute the moving standard deviation of the input:

- 1 Create the `dsp.MovingStandardDeviation` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
MovStd = dsp.MovingStandardDeviation
MovStd = dsp.MovingStandardDeviation(Len)
MovStd = dsp.MovingStandardDeviation(Len,Overlap)
MovStd = dsp.MovingStandardDeviation(Name,Value)
```

### Description

`MovStd = dsp.MovingStandardDeviation` returns a moving standard deviation object, `MovStd`, using the default properties.

`MovStd = dsp.MovingStandardDeviation(Len)` sets the `WindowLength` property to `Len`.

`MovStd = dsp.MovingStandardDeviation(Len,Overlap)` sets the `WindowLength` property to `Len` and the `OverlapLength` property to `Overlap`.

`MovStd = dsp.MovingStandardDeviation(Name,Value)` specifies additional properties using `Name,Value` pairs. Unspecified properties have default values.

Example: `MovStd = dsp.MovingStandardDeviation('Method','Exponential weighting','ForgettingFactor',0.999);`

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see *System Design in MATLAB Using System Objects*.

### Method — Moving standard deviation method

'Sliding window' (default) | 'Exponential weighting'

- 'Sliding window' — A window of length specified by `SpecifyWindowLength` is moved over the input data along each channel. For every sample the window moves by, the object computes the standard deviation over the data in the window.
- 'Exponential weighting' — The object computes the exponentially weighted moving variance, and takes the square root.

For more details on these methods, see “Algorithms” on page 4-874.

### SpecifyWindowLength — Specify window length

true (default) | false

Flag to specify a window length, specified as a scalar boolean.

- true — The length of the sliding window is equal to the value you specify in the `WindowLength` property.
- false — The length of the sliding window is infinite. In this mode, the standard deviation is computed using the current sample and all the past samples.

### Dependencies

This property applies when you set `Method` to 'Sliding window'.

### WindowLength — Length of the sliding window

4 (default) | positive scalar integer

Length of the sliding window in samples, specified as a positive scalar integer.

### Dependencies

This property applies when you set `Method` to 'Sliding window' and `SpecifyWindowLength` to true.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### OverlapLength — Overlap length between windows

WindowLength - 1 (default) | nonnegative integer

Overlap length between sliding windows, specified as a nonnegative integer. The value of overlap length varies in the range  $[0, \text{WindowLength} - 1]$ . If not specified, the overlap length is `WindowLength - 1`.

**Dependencies**

This property applies when you set `Method` to 'Sliding window' and `SpecifyWindowLength` to `true`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**ForgettingFactor — Exponential weighting factor**

0.9 (default) | positive real scalar in the range (0,1]

Exponential weighting factor, specified as a positive real scalar in the range (0,1].

Since this property is tunable, you can change its value even when the object is locked.

**Tunable:** Yes

**Dependencies**

This property applies when you set `Method` to 'Exponential weighting'.

Data Types: `single` | `double`

**Usage****Syntax**

`y = movStd(x)`

**Description**

`y = movStd(x)` computes the moving standard deviation of the input signal, `x`, using either the sliding window method or exponential weighting method.

**Input Arguments****x — Data input**

vector | matrix

Data input, specified as a vector or a matrix. If `x` is a matrix, each column is treated as an independent channel. The moving standard deviation is computed along each channel.

The object accepts variable-size inputs. Once the object is locked, you can change the size of each input channel, but you cannot change the number of channels.

Data Types: `single` | `double`

**Output Arguments****y — Moving standard deviation**

vector | matrix

Moving standard deviation of the input signal, returned as a vector or a matrix.

When you input a signal of size  $m$ -by- $n$  to the object, and if you set `Method` to 'Sliding window' and `SpecifyWindowLength` to `true`, the output has an upper bound size of  $\text{ceil}(m/\text{hop size})$ -by- $n$ . Hop size is window length – overlap length. In other cases, the output has a size of  $m$ -by- $n$ .

When you generate code from this object, the variable-size behavior of the output in the generated code depends on the input frame length and whether the size of the input signal is fixed or variable. For more details, see “Code Generation” on page 4-877.

Data Types: `single` | `double`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Common to All System Objects

|                      |  |
|----------------------|--|
| <code>step</code>    | Run System object algorithm  |
| <code>release</code> | Release resources and allow changes to System object property values and input characteristics |
| <code>reset</code>   | Reset internal states of System object   |

## Examples

### Moving Standard Deviation of Noisy Square Wave Signal

Compute the moving standard deviation of a noisy square wave signal with varying amplitude using the `dsp.MovingStandardDeviation` object.

#### Initialization

Set up `movstdWindow`, `movstdWindow_overlap`, and `movstdExp` objects. `movstdWindow` uses the sliding window method with a window length of 800 samples and a default overlap length of 799 samples, which is one sample less than the specified window length. `movstdWindow_overlap` uses a window length of 800 samples and an overlap length of 700 samples. `movstdExp` uses the exponentially weighting method with a forgetting factor of 0.999.

Create a time scope for viewing the output.

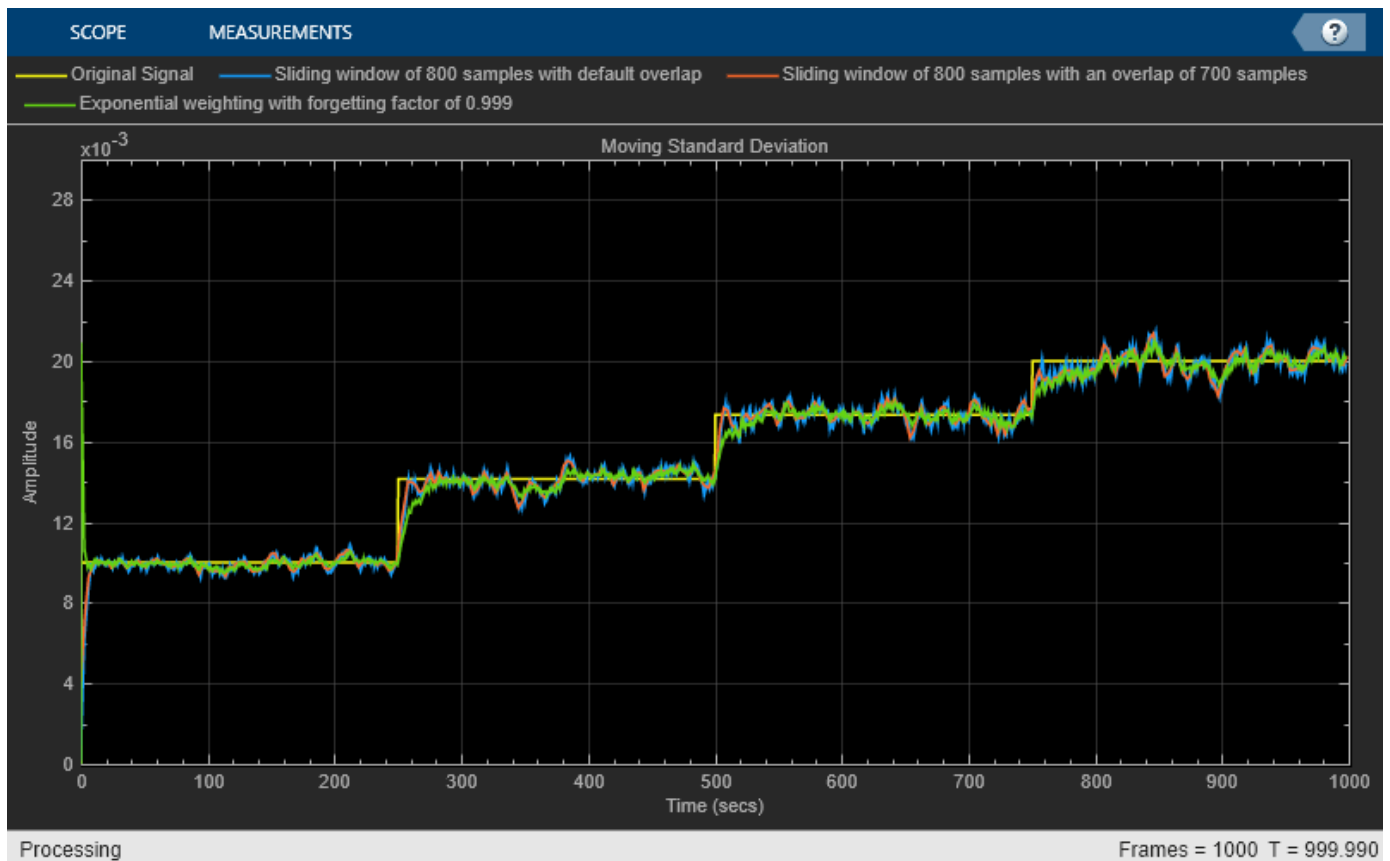
```
FrameLength = 100;
Fs = 100;
movstdWindow = dsp.MovingStandardDeviation(800);
movstdWindow_overlap = dsp.MovingStandardDeviation(800,700);
movstdExp = dsp.MovingStandardDeviation(...
    'Method','Exponential weighting',...
    'ForgettingFactor',0.999);
scope = timescope('SampleRate',[Fs,Fs,Fs/(800-700),Fs],...
    'TimeSpanOverrunAction','Scroll',...
    'TimeSpanSource','Property',...
    'TimeSpan',1000,...
    'ShowGrid',true,...
    'BufferLength',1e7,...
    'YLimits',[0 3e-2]);
title = 'Moving Standard Deviation';
scope.Title = title;
scope.ChannelNames = {'Original Signal',...}
```

```
'Sliding window of 800 samples with default overlap',...
'Sliding window of 800 samples with an overlap of 700 samples',...
'Exponential weighting with forgetting factor of 0.999'}];
```

### Compute the Standard Deviation

Generate a noisy square wave signal. Vary the amplitude of the square wave after a given number of frames. Apply the sliding window method and the exponential weighting method to this signal. The actual standard deviation is  $\sqrt{np}$ . The object uses this value while adding noise to the data. Compare the actual standard deviation with the computed standard deviation in the time scope.

```
count = 1;
noisepower = 1e-4 * [1 2 3 4];
for index = 1:length(noisepower)
    np = noisepower(index);
    yexp = sqrt(np)*ones(FrameLength,1);
    for i = 1:250
        x = sqrt(np) * randn(FrameLength,1);
        y1 = movstdWindow(x);
        y2 = movstdWindow_overlap(x);
        y3 = movstdExp(x);
        scope(yexp,y1,y2,y3);
    end
end
```



## Algorithms

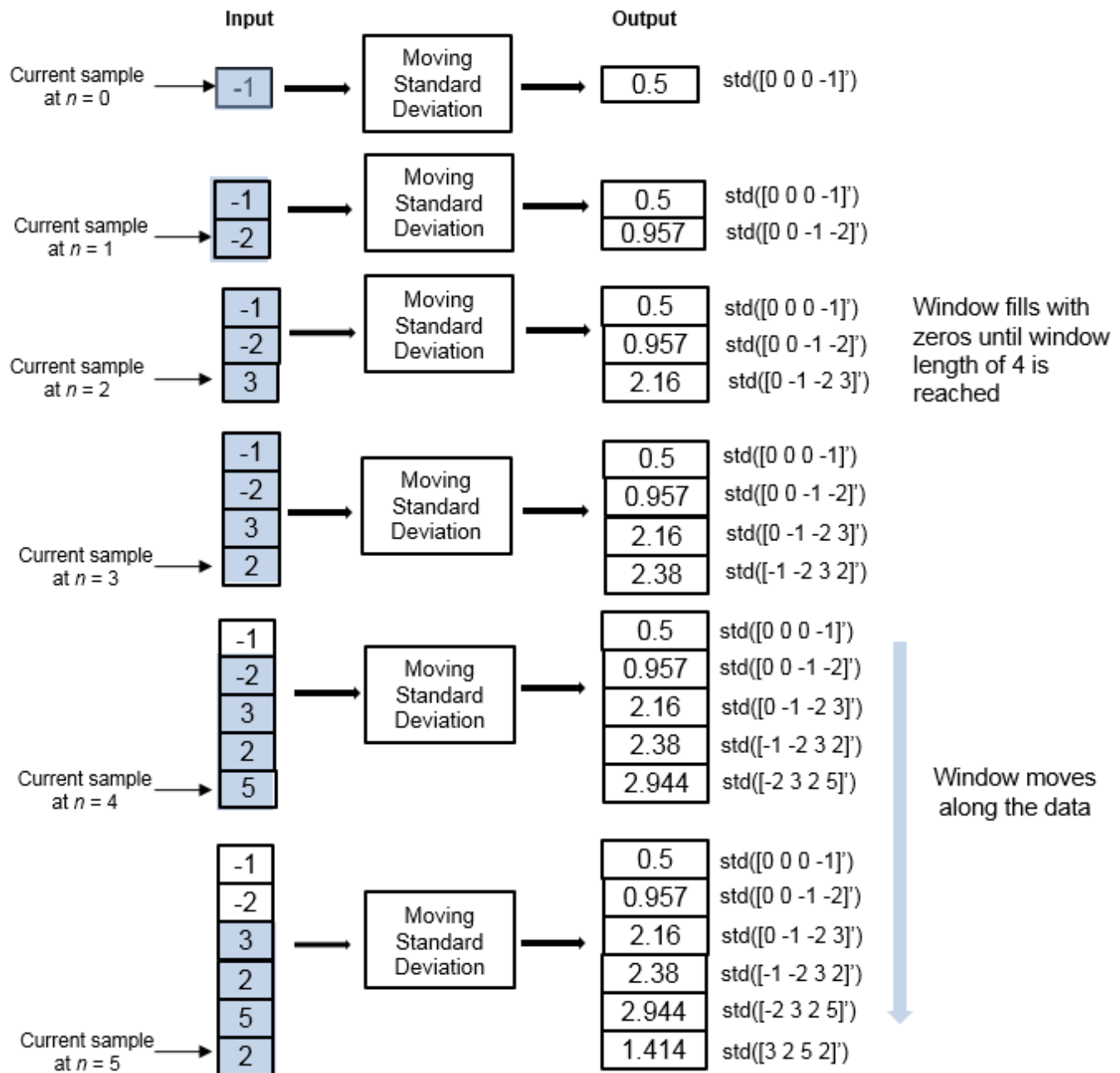
### Sliding Window Method

In the sliding window method, the output at the current sample is the standard deviation of the current sample with respect to the data in the window. To compute the first output sample, the algorithm waits until it receives the hop size number of input samples. Hop size is defined as window length - overlap length. Remaining samples in the window are considered to be zero. As an example, if the window length is 5 and the overlap length is 2, then the algorithm waits until it receives 3 samples of input to compute the first sample of the output. After generating the first output, it generates the subsequent output samples for every hop size number of input samples.

When you do not specify the window length, the algorithm chooses an infinite window length. In this mode, the output is the moving standard deviation of the current sample with respect to all the previous samples in the channel.

Consider an example of computing the moving standard deviation of a streaming input data using the sliding window method. The algorithm uses a window length of 4 and an overlap length of 3. With each input sample that comes in, the window of length 4 moves along the data.





### Exponential Weighting Method

In the exponential weighting method, the moving standard deviation is computed recursively using these formulas:

$$s_{N,\lambda} = \sqrt{\frac{1}{v_{N,\lambda}} \sum_{k=1}^N \lambda^{N-k} [x_k - \bar{x}_{N,\lambda}]^2}$$

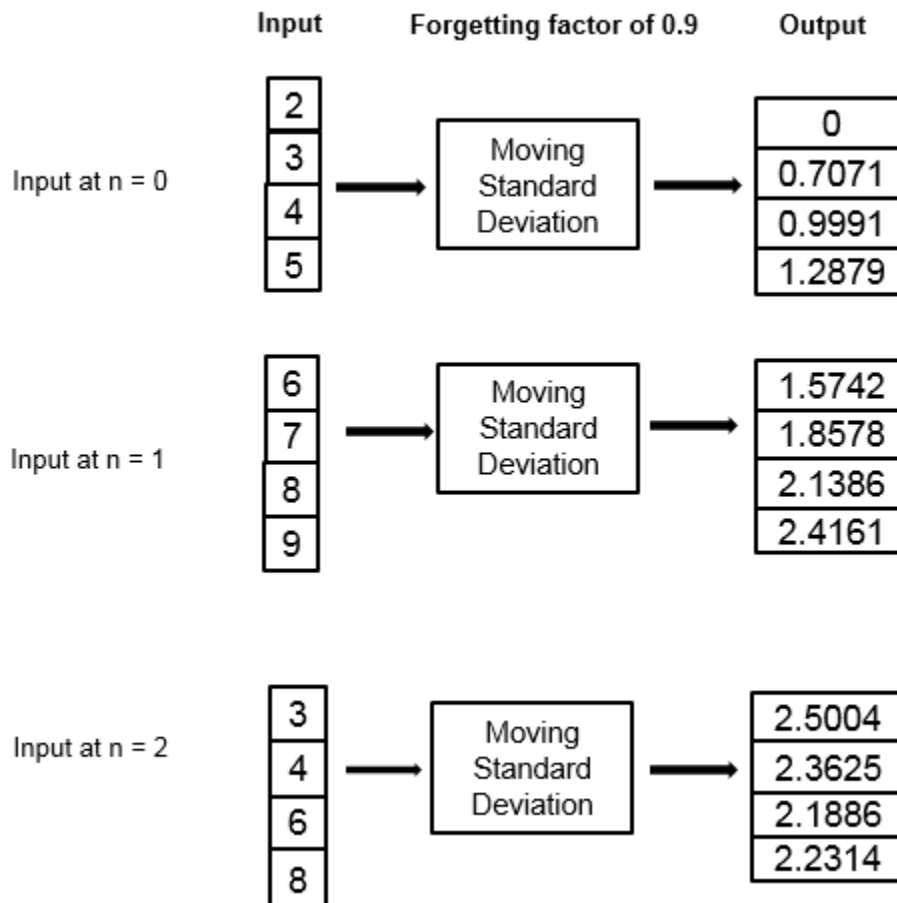
$$v_{N,\lambda} = \frac{2\lambda(1 - \lambda^{N-1})}{(1 - \lambda)(1 + \lambda)}$$

- $s_{N,\lambda}$  — Moving standard deviation of the current data sample with respect to the rest of the data.
- $[x_k - \bar{x}_{N,\lambda}]^2$  — Difference between each data sample and the average of the data, squared.
- $\sum_{k=1}^N \lambda^{N-k} [x_k - \bar{x}_{N,\lambda}]^2$  — Difference between each data sample and the average of the data, squared and multiplied with the forgetting factor. All the squared terms are added.
- $\frac{1}{v_{N,\lambda}}$  — Weighting factor applied to the sum.
- $\lambda$  — Forgetting factor.

As the age of the data increases, the magnitude of the weighting factor decreases exponentially and never reaches zero. In other words, the recent data has more influence on the current standard deviation than the older data.

The value of the forgetting factor determines the rate of change of the weighting factors. A forgetting factor of 0.9 gives more weight to the older data than does a forgetting factor of 0.1. A forgetting factor of 1.0 indicates infinite memory. All previous samples are given an equal weight.

Consider an example of computing the moving standard deviation using the exponential weighting method. The forgetting factor is 0.9.



## Version History

Introduced in R2016b

### R2022b: Change in variable-size behavior for output signal in generated code

Starting in R2022b, if you generate code from this object with the `Method` property set to 'Sliding window' and the `SpecifyWindowLength` property set to `true`, and if you input a signal with a frame length that is a multiple of the hop size (window length – overlap length), this object generates a fixed-size output signal in the generated code. For more details, see “Code Generation” on page 4-877.

## References

[1] Bodenham, Dean. “Adaptive Filtering and Change Detection for Streaming Data.” PH.D. Thesis. Imperial College, London, 2012.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- See “System Objects in MATLAB Code Generation” (MATLAB Coder).
- When you set `Method` to 'Sliding window' and `SpecifyWindowLength` to `true`, and generate code from this object, the variable-size behavior of the output depends on the input frame length and whether the size of the input signal is fixed or variable.

See this table for more details.

| Input signal  | Input size   | Output signal   |
|---------------|--------------|---|
| Fixed-size    | $m$ -by- $n$ | When the input frame length is a multiple of the hop size, the output signal has a fixed-size of $(m/\text{hop size})$ -by- $n$ .<br><br>When input frame length is not a multiple of the hop size, the output signal is variable-sized and has an upper bound of $\text{ceil}(m/\text{hop size})$ -by- $n$ . |
| Variable-size | $m$ -by- $n$ | Output is a variable-size signal.<br><br>Output has an upper bound size of $\text{ceil}(m/\text{hop size})$ -by- $n$ .  |

## **See Also**

### **Objects**

`dsp.MovingMaximum` | `dsp.MovingMinimum` | `dsp.MovingAverage` | `dsp.MovingRMS` | `dsp.MovingVariance` | `dsp.MedianFilter`

### **Blocks**

Moving Standard Deviation | Standard Deviation | Moving Maximum | Moving Minimum | Moving Average | Moving RMS | Moving Variance | Median Filter

### **Topics**

“What Are Moving Statistics?”

“Sliding Window Method and Exponential Weighting Method”

“Streaming Signal Statistics”

# dsp.MovingVariance

**Package:** dsp

Moving variance

## Description

The `dsp.MovingVariance` System object computes the moving variance of the input signal along each channel, independently over time. The object uses either the sliding window method or the exponential weighting method to compute the moving variance. In the sliding window method, a window of specified length is moved over the data, sample by sample, and the variance is computed over the data in the window. In the exponential weighting method, the object subtracts each sample of the data from the average, squares the difference, and multiplies the squared result with a weighting factor. The object then computes the variance by adding all the weighted data. For more details on these methods, see “Algorithms” on page 4-884.

To compute the moving variance of the input:

- 1 Create the `dsp.MovingVariance` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
MovVar = dsp.MovingVariance
MovVar = dsp.MovingVariance(Len)
MovVar = dsp.MovingVariance(Len,Overlap)
MovVar = dsp.MovingVariance(Name,Value)
```

### Description

`MovVar = dsp.MovingVariance` returns a moving variance object, `MovVar`, using the default properties.

`MovVar = dsp.MovingVariance(Len)` sets the `WindowLength` property to `Len`.

`MovVar = dsp.MovingVariance(Len,Overlap)` sets the `WindowLength` property to `Len` and the `OverlapLength` property to `Overlap`.

`MovVar = dsp.MovingVariance(Name,Value)` specifies additional properties using `Name, Value` pairs. Unspecified properties have default values.

```
Example: MovVar = dsp.MovingVariance('Method','Exponential
weighting','ForgettingFactor',0.9);
```

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see *System Design in MATLAB Using System Objects*.

### Method — Method to compute the variance

'Sliding window' (default) | 'Exponential weighting'

- 'Sliding window' — A window of length specified by `SpecifyWindowLength` is moved over the input data along each channel. For every sample the window moves by, the object computes the variance over the data in the window.
- 'Exponential weighting' — The object subtracts each sample of the data from the average, squares the difference, and multiplies the squared result with a weighting factor. The object then computes the variance by adding all the weighted data. The magnitude of the weighting factors decreases exponentially as the age of the data increases, never reaching zero.

For more details on these methods, see “Algorithms” on page 4-884.

### SpecifyWindowLength — Specify window length

true (default) | false

Flag to specify a window length, specified as a scalar Boolean.

- true — The length of the sliding window is equal to the value you specify in the `WindowLength` property.
- false — The length of the sliding window is infinite. In this mode, the variance is computed using the current sample and all past samples.

### Dependencies

This property applies when you set `Method` to 'Sliding window'.

### WindowLength — Length of the sliding window

4 (default) | positive scalar integer

Length of the sliding window in samples, specified as a positive scalar integer.

### Dependencies

This property applies when you set `Method` to 'Sliding window' and `SpecifyWindowLength` to true.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### OverlapLength — Overlap length between windows

WindowLength - 1 (default) | nonnegative integer

Overlap length between sliding windows, specified as a nonnegative integer. The value of overlap length varies in the range  $[0, \text{WindowLength} - 1]$ . If not specified, the overlap length is set to `WindowLength - 1`.

**Dependencies**

This property applies when you set `Method` to 'Sliding window' and `SpecifyWindowLength` to `true`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**ForgettingFactor — Exponential weighting factor**

0.9 (default) | positive real scalar in the range (0,1]

Exponential weighting factor, specified as a positive real scalar in the range (0,1].

A forgetting factor of 0.9 gives more weight to the older data than does a forgetting factor of 0.1. A forgetting factor of 1.0 indicates infinite memory. All the past samples are given an equal weight.

Since this property is tunable, you can change its value even when the object is locked.

**Tunable:** Yes

**Dependencies**

This property applies when you set `Method` to 'Exponential weighting'.

Data Types: `single` | `double`

**Usage****Syntax**

```
y = movVar(x)
```

**Description**

`y = movVar(x)` computes the moving variance of the input signal, `x`, using either the sliding window method or exponential weighting method.

**Input Arguments****x — Data input**

vector | matrix

Data input, specified as a vector or a matrix. If `x` is a matrix, each column is treated as an independent channel. The moving variance is computed along each channel.

The object accepts variable-size inputs. Once the object is locked, you can change the size of each input channel, but you cannot change the number of channels.

Data Types: `single` | `double`

**Output Arguments****y — Moving variance**

vector | matrix

Moving variance of the input signal, returned as a vector or a matrix.

When you input a signal of size  $m$ -by- $n$  to the object, and if you set `Method` to 'Sliding window' and `SpecifyWindowLength` to `true`, the output has an upper bound size of  $\text{ceil}(m/\text{hop size})$ -by- $n$ . Hop size is window length – overlap length. In other cases, the output has a size of  $m$ -by- $n$ .

When you generate code from this object, the variable-size behavior of the output in the generated code depends on the input frame length and whether the size of the input signal is fixed or variable. For more details, see “Code Generation” on page 4-888.

Data Types: `single` | `double`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Common to All System Objects

`step` Run System object algorithm  
`release` Release resources and allow changes to System object property values and input characteristics  
`reset` Reset internal states of System object

## Examples

### Moving Variance of Noisy Square Wave Signal

Compute the moving variance of a noisy square wave signal with varying amplitude using the `dsp.MovingVariance` object.

#### Initialization

Set up `movvarWindow`, `movvarWindow_overlap`, and `movvarExp` objects. `movvarWindow` uses the sliding window method with a window length of 800 samples and a default overlap length of 799 samples, which is 1 sample less than the specified window length. `movvarWindow_overlap` uses a window length of 800 samples and an overlap length of 700 samples. `movvarExp` uses the exponentially weighting method with a forgetting factor of 0.999.

Create a time scope for viewing the output.

```
FrameLength = 100;  
Fs = 100;  
movvarWindow = dsp.MovingVariance(800);  
movvarWindow_overlap = dsp.MovingVariance(800,700);  
movvarExp = dsp.MovingVariance('Method','Exponential weighting',...  
    'ForgettingFactor',0.999);  
scope = timescope('SampleRate',[Fs,Fs,Fs/(800-700),Fs],...  
    'TimeSpanOverrunAction','Scroll',...  
    'TimeSpanSource','Property',...  
    'TimeSpan',1000,...  
    'ShowGrid',true,...  
    'BufferLength',1e7,...  
    'YLimits',[0 7e-4]);
```



```

title = 'Moving Variance';
scope.Title = title;
scope.ChannelNames = {'Original Signal',...
    'Sliding window of 800 samples with default overlap',...
    'Sliding window of 800 samples with an overlap of 700 samples',...
    'Exponential weighting with forgetting factor of 0.999'};

```

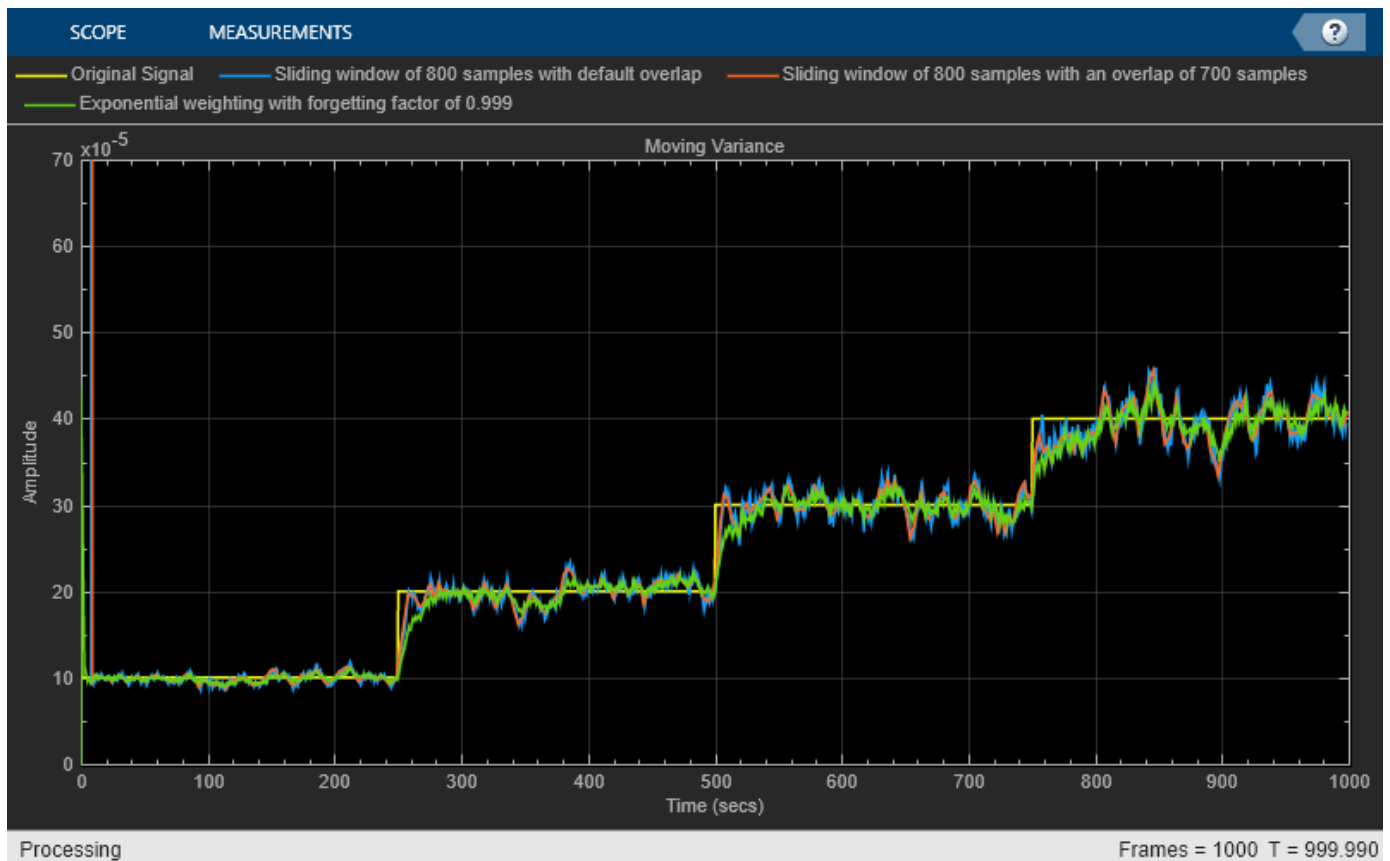
### Compute the Variance

Generate a noisy square wave signal. Vary the amplitude of the square wave after a given number of frames. Apply the sliding window method and the exponentially weighting method on this signal. The actual variance is  $np$ . This value is used while adding noise to the data. Compare the actual variance with the computed variances on the time scope.

```

count = 1;
noisepower = 1e-4 * [1 2 3 4];
for index = 1:length(noisepower)
    np = noisepower(index);
    yexp = np*ones(FrameLength,1);
    for i = 1:250
        x = 1 + sqrt(np) * randn(FrameLength,1);
        y1 = movvarWindow(x);
        y2 = movvarWindow_overlap(x);
        y3 = movvarExp(x);
        scope(yexp,y1,y2,y3);
    end
end
end

```



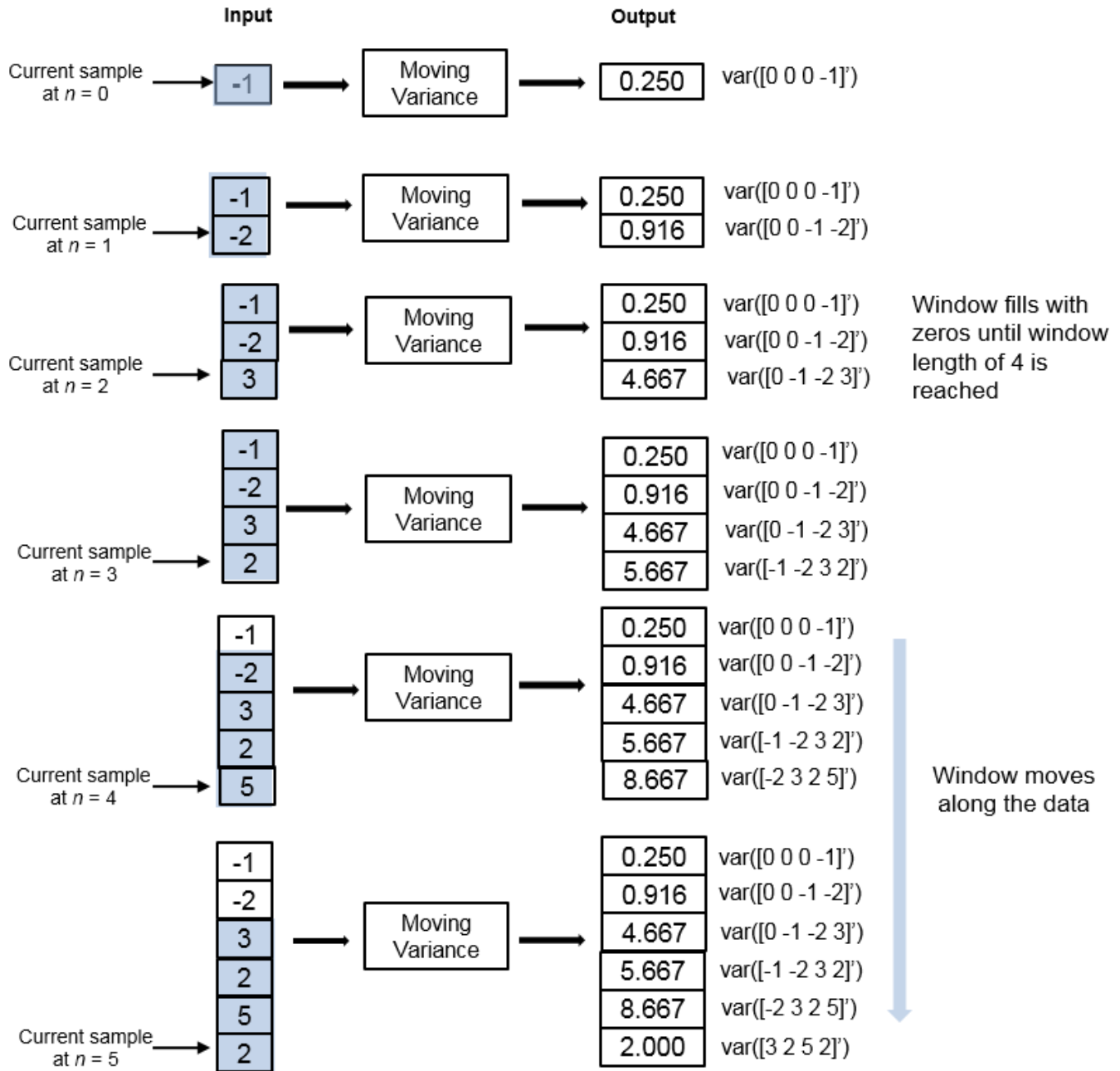
## Algorithms

### Sliding Window Method

In the sliding window method, the output at the current sample is the variance of the current sample with respect to the data in the window. To compute the first output sample, the algorithm waits until it receives the hop size number of input samples. Hop size is defined as window length - overlap length. Remaining samples in the window are considered to be zero. As an example, if the window length is 5 and the overlap length is 2, then the algorithm waits until it receives 3 samples of input to compute the first sample of the output. After generating the first output, it generates the subsequent output samples for every hop size number of input samples.

When you do not specify the window length, the algorithm chooses an infinite window length. In this mode, the output is the moving variance of the current sample with respect to all previous samples in the channel.

Consider an example of computing the moving variance of a streaming input data using the sliding window method. The algorithm uses a window length of 4 and an overlap length of 3. With each input sample that comes in, the window of length 4 moves along the data.



### Exponential Weighting Method

In the exponential weighting method, the moving variance is computed recursively using these formulas:

$$s^2_{N,\lambda} = \frac{1}{v_{N,\lambda}} \sum_{k=1}^N \lambda^{N-k} [x_k - \bar{x}_{N,\lambda}]^2$$

$$v_{N,\lambda} = \frac{2\lambda(1 - \lambda^{N-1})}{(1 - \lambda)(1 + \lambda)}$$

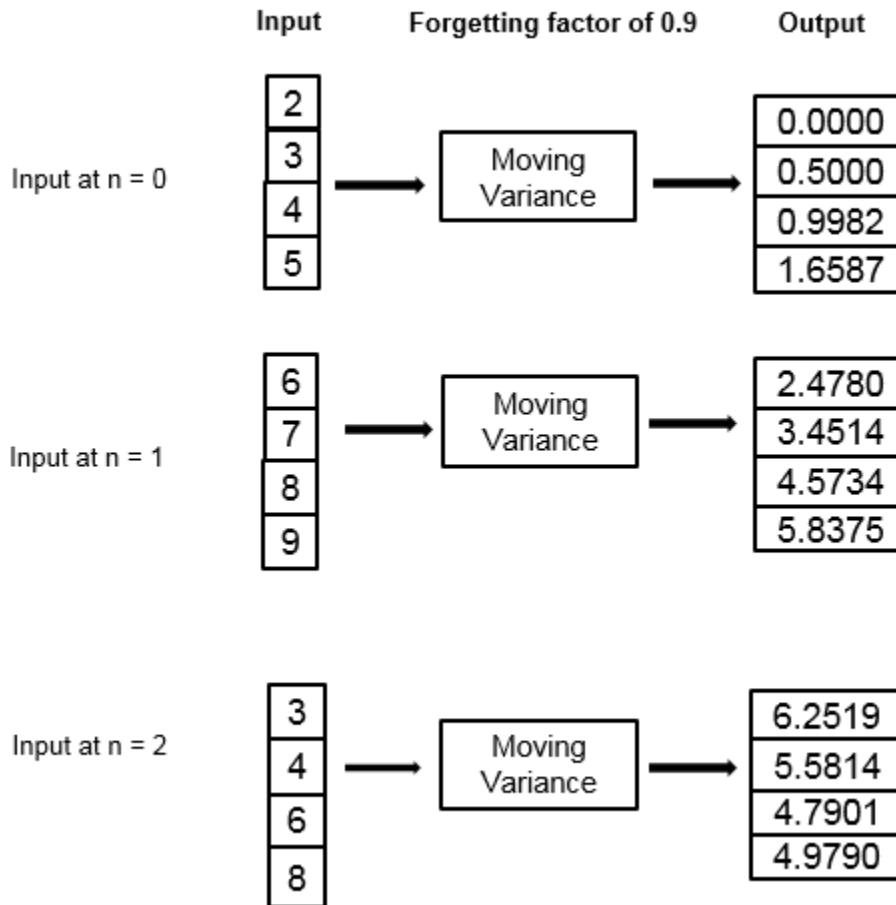
To compute the moving variance, the algorithm implements these equations recursively.

- $s_{N,\lambda}^2$  — Moving variance of the current data sample with respect to the rest of the data in the channel.
- $\bar{x}_{N,\lambda}$  — Moving average at the current sample. For details on computing the moving average, see `dsp.MovingAverage`.
- $[x_k - \bar{x}_{N,\lambda}]^2$  — Difference between each data sample and the average of the data, squared.
- $\sum_{k=1}^N \lambda^{N-k} [x_k - \bar{x}_{N,\lambda}]^2$  — Difference between each data sample and the average of the data, squared and multiplied with the forgetting factor. All the squared terms are added.
- $\frac{1}{v_{N,\lambda}}$  — Weighting factor applied to the sum.
- $\lambda$  — Forgetting factor you can specify through the `ForgettingFactor` property.

As the age of the data increases, the magnitude of the weighting factor decreases exponentially, and never reaches zero. In other words, the recent data has more influence on the current variance, than the older data.

The value of the forgetting factor determines the rate of change of the weighting factors. A forgetting factor of 0.9 gives more weight to the older data than does a forgetting factor of 0.1. A forgetting factor of 1.0 indicates infinite memory. All the past samples are given an equal weight.

Consider an example of computing the moving variance using the exponential weighting method. The forgetting factor is 0.9.



## Version History

Introduced in R2016b

### R2022b: Change in variable-size behavior for output signal in generated code

Starting in R2022b, if you generate code from this object with the Method property set to 'Sliding window' and the SpecifyWindowLength property set to true, and if you input a signal with a frame length that is a multiple of the hop size (window length – overlap length), this object generates a fixed-size output signal in the generated code. For more details, see “Code Generation” on page 4-888.

## References

- [1] Bodenham, Dean. “Adaptive Filtering and Change Detection for Streaming Data.” PH.D. Thesis. Imperial College, London, 2012.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- See “System Objects in MATLAB Code Generation” (MATLAB Coder).
- When you set `Method` to 'Sliding window' and `SpecifyWindowLength` to true, and generate code from this object, the variable-size behavior of the output depends on the input frame length and whether the size of the input signal is fixed or variable.

See this table for more details.

| Input signal  | Input size   | Output signal   |
|---------------|--------------|---|
| Fixed-size    | $m$ -by- $n$ | When the input frame length is a multiple of the hop size, the output signal has a fixed-size of $(m/\text{hop size})$ -by- $n$ .<br><br>When input frame length is not a multiple of the hop size, the output signal is variable-sized and has an upper bound of $\text{ceil}(m/\text{hop size})$ -by- $n$ . |
| Variable-size | $m$ -by- $n$ | Output is a variable-size signal.<br><br>Output has an upper bound size of $\text{ceil}(m/\text{hop size})$ -by- $n$ .  |

## See Also

### Objects

`dsp.MovingMaximum` | `dsp.MovingMinimum` | `dsp.MovingAverage` | `dsp.MovingRMS` | `dsp.MovingStandardDeviation` | `dsp.MedianFilter`

### Blocks

Moving Variance | Variance | Moving Maximum | Moving Minimum | Moving Average | Moving RMS | Moving Standard Deviation | Median Filter

### Topics

“What Are Moving Statistics?”

“Sliding Window Method and Exponential Weighting Method”

“Streaming Signal Statistics”

# dsp.NCO

**Package:** dsp

Generate real or complex sinusoidal signals

## Description

The numerically controlled oscillator, or NCO object generates real or complex sinusoidal signals. The amplitude of the generated signal is always 1.

To generate real or complex sinusoidal signals:

- 1 Create the `dsp.NCO` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
nco = dsp.NCO
nco = dsp.NCO(Name, Value)
```

### Description

`nco = dsp.NCO` returns an NCO System object, `nco`, that generates a multichannel real or complex sinusoidal signal, with independent frequency and phase in each output channel.

`nco = dsp.NCO(Name, Value)` returns an NCO System object, `nco`, with each specified property set to the specified value.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

### PhaseIncrementSource — Source of phase increment

'Input port' (default) | 'Property'

Specify the source of the phase increment as 'Property' or 'Input port'.

### PhaseIncrement — Phase increment

100 (default) | scalar | vector

Specify the phase increment as an integer-valued scalar or vector.

**Dependencies**

This property applies only when you set the “PhaseIncrementSource” on page 4-0 property to 'Property'.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**PhaseOffsetSource — Source of phase offset**

'Property' (default) | 'Input port'

Specify the source of the phase offset as 'Property' or 'Input port'.

**PhaseOffset — Phase offset**

0 (default) | scalar | vector

Specify the phase offset as an integer-valued scalar or vector.

**Dependencies**

This property applies only when you set the “PhaseOffsetSource” on page 4-0 property to 'Property'.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**Dither — Enable adding internal dithering to NCO algorithm**

`true` (default) | `false`

Set this property to `true` to add internal dithering to the NCO algorithm. Dithering is added using the PN Sequence Generator.

**NumDitherBits — Number of dither bits**

4 (default) | positive integer

Specify the number of dither bits as a positive integer.

**Dependencies**

This property applies only when you set the `Dither` property to `true`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**PhaseQuantization — Enable quantization of accumulated phase**

`true` (default) | `false`

Set this property to `true` to enable quantization of the accumulated phase.

**NumQuantizerAccumulatorBits — Number of quantizer accumulator bits**

12 (default) | integer

Specify the number of quantizer accumulator bits as an integer scalar greater than 2 and less than the accumulator word length (“CustomAccumulatorDataType” on page 4-0 ). This property determines the number of entries in the lookup table of sine values.

**Dependencies**

This property applies only when you set the “PhaseQuantization” on page 4-0 property to `true`.



Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **PhaseQuantizationErrorOutputPort — Enable output of phase quantization error**

`false` (default) | `true`

Set this property to `true` to output the phase quantization error.

#### **Dependencies**

This property applies only when you set the “PhaseQuantization” on page 4-0 property to `true`.

### **Waveform — Type of output signal**

`'Sine'` (default) | `'Cosine'` | `'Complex exponential'` | `'Sine and cosine'`

Specify the type of the output signal.

### **SamplesPerFrame — Number of output samples per frame**

`1` (default) | positive integer

Specify the number of samples per frame of the output signal. When the `PhaseOffsetSource` property is `'Input port'`, and the “PhaseIncrementSource” on page 4-0 property is `'Property'`, the number of rows or frame size of the phase offset input determines the number of samples per frame of the output signal. When you set both the “PhaseOffsetSource” on page 4-0 and “PhaseIncrementSource” on page 4-0 properties to `'Input port'`, the number of rows in the inputs must be 1, and the samples per frame of the output signal is 1.

#### **Dependencies**

This property applies only when you set the “PhaseOffsetSource” on page 4-0 property to `'Property'`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

### **OutputDataType — Output data type**

`'Custom'` (default) | `'double'` | `'single'`

Specify the output data type as `'double'`, `'single'` or `'Custom'`. When you select `'Custom'`, you must also set the “CustomOutputDataType” on page 4-0 property.

## **Fixed-Point Properties**

### **RoundingMethod — Rounding method for fixed-point operations**

`'Floor'` (default)

This constant property has a value `'Floor'`.

### **OverflowAction — Overflow action for fixed-point operations**

`'Wrap'` (default)

This constant property has a value `'Wrap'`.

### **AccumulatorDataType — Accumulator word and fraction lengths**

`'Custom'` (default)

This constant property has a value `'Custom'`.

**CustomAccumulatorDataType — Accumulator word and fraction lengths**

numericitytype([],16) (default) | numericitytype

Specify the accumulator fixed-point type as an unscaled numericitytype object with a Signedness of Auto.

**CustomOutputDataType — Output word and fraction lengths**

numericitytype([],16,14) (default) | numericitytype

Specify the output fixed-point type as a scaled numericitytype object with a Signedness of Auto.

**Dependencies**

This property applies only when you set the “OutputDataType” on page 4-0 property to Custom.

**Usage****Syntax**

```
Y = nco()  
Y = nco(phInc)  
Y = nco(OFFSET)  
Y = nco(phInc,OFFSET)  
[Y,cosine] = nco(____)  
[Y,qErr] = nco(____)
```

**Description**

`Y = nco()` returns a sinusoidal signal when the `PhaseIncrementSource` and the `PhaseOffsetSource` properties are both set to 'Property'.

`Y = nco(phInc)` returns a sinusoidal signal, `Y`, generated by the NCO with the specified phase increment, `phInc`.

`Y = nco(OFFSET)` returns a sinusoidal signal, `Y`, with phase offset, `OFFSET`, when the `PhaseOffsetSource` property is set to 'Input port'.

`Y = nco(phInc,OFFSET)` returns a sinusoidal signal, `Y`, with phase increment, `phInc`, and phase offset, `OFFSET`, when the `PhaseIncrementSource` and the `PhaseOffsetSource` properties are both 'Input port'. `phInc` and `OFFSET` must both be row vectors of the same length, where the length determines the number of channels in the output signal.

`[Y,cosine] = nco(____)` returns a sinusoidal signal, `Y`, and a cosinusoidal signal, `cosine`, when the `Waveform` property is set to 'Sine and cosine'. This syntax can include any of the input arguments in previous syntaxes.

`[Y,qErr] = nco(____)` returns a sinusoidal signal, `Y`, and output quantization error, `qErr`, when the `PhaseQuantization` and the `PhaseQuantizationErrorOutputPort` properties are both true.

**Input Arguments****phInc — Phase increment**

scalar | row vector

Phase increment, specified as a scalar or a row vector, where each element corresponds to a separate channel.

When both `PhaseIncrementSource` and `PhaseOffsetSource` properties are set to 'Input port', the two inputs, `phInc` and `OFFSET` must have the same number of channels.

#### Dependencies

This property applies only when the `PhaseIncrementSource` property is set to 'Input port'.

Data Types: `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

#### OFFSET — Phase offset

row vector | matrix

Phase offset, specified as a row vector or a matrix. The number of rows of `OFFSET` determines the number of samples per frame of the output signal. The number of columns of `OFFSET` determines the number of channels of the output signal.

When both `PhaseIncrementSource` and `PhaseOffsetSource` properties are set to 'Input port', `phInc` and `OFFSET` must have the same number of channels.

#### Dependencies

This property applies only when the `PhaseOffsetSource` property is set to 'Input port'.

Data Types: `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

#### Output Arguments

##### Y — NCO output

vector | matrix

NCO output, returned as a vector or a matrix. The number of rows in the output signal is determined by:

- `SamplesPerFrame` property -- When `PhaseOffsetSource` is set to 'Property'
- `OFFSET` input argument -- When `PhaseOffsetSource` is set to 'Input port'

The number of channels in the output signal is determined by the number of channels in the phase offset and the phase increment signals, which must be equal.

The data type of the output is determined by the `OutputDataType` property.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

##### cosine — Cosinusoidal signal

vector | matrix

Cosinusoidal signal, returned as a vector or a matrix. The `cosine` output signal has the same size and data type as the sinusoidal signal, `Y`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

##### qErr — Output quantization error

vector | matrix

Output quantization error, returned as a vector or a matrix. The `qErr` output signal has the same size as the sinusoidal signal, `Y`.

**Dependencies**

This output is only available when both the `PhaseQuantization` and the `PhaseQuantizationErrorOutputPort` properties are set to `true`.

Data Types: `fi`

**Object Functions**

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

**Specific to `dsp.NCO`**

`info` Characteristic information about generated signal

**Common to All System Objects**

`step` Run System object algorithm

`release` Release resources and allow changes to System object property values and input characteristics

`reset` Reset internal states of System object

**Examples****Design NCO Source**

Design an NCO source according to given specifications.

```
df = 0.05; % Frequency resolution = 0.05 Hz
minSFDR = 96; % Spurious free dynamic range >= 96 dB
Ts = 1/8000; % Sample period = 1/8000 sec
dphi = pi/2; % Desired phase offset = pi/2;
```

Calculate number of accumulator bits required for the given frequency resolution.

```
Nacc = ceil(log2(1/(df*Ts)));
```

Actual frequency resolution achieved.

```
actdf = 1/(Ts*2^Nacc);
```

Calculate number of quantized accumulator bits required from the SFDR requirement

```
Nqacc = ceil((minSFDR-12)/6);
```

Calculate the phase offset

```
ph0ffset = 2^Nacc*dphi/(2*pi);
```

Design the NCO source.

```

nco = dsp.NCO('PhaseOffset', phOffset,...
    'NumDitherBits', 4, ...
    'NumQuantizerAccumulatorBits', Nqacc,...
    'SamplesPerFrame', 1/Ts, ...
    'CustomAccumulatorDataType', numerictype([],Nacc));
san = spectrumAnalyzer('SampleRate', 1/Ts,...
    'Method', 'welch', 'AveragingMethod', 'exponential',...
    'ForgettingFactor', 0.3,...
    'PlotAsTwoSidedSpectrum', false);

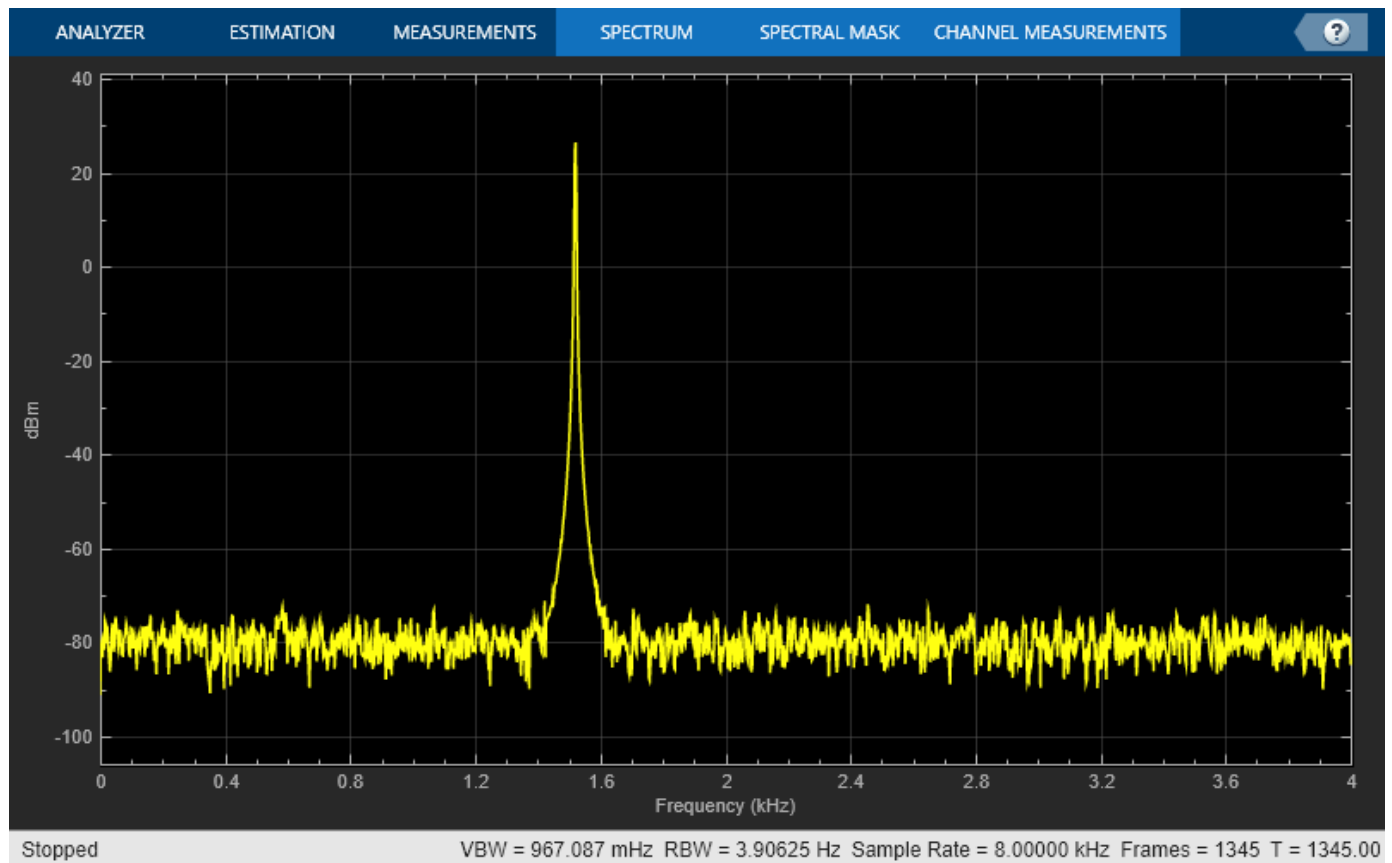
```

View the output of the NCO source on a spectrum analyzer. Change the output frequency in the middle of the simulation from 510 Hz to 1520 Hz.

```

tic;
while toc < 10
    if toc < 5
        F0 = 510;
    else
        F0 = 1520;
    end
    % Calculate the phase increment
    phIncr = int32(round(F0*Ts*2^Nacc));
    y = nco(phIncr);
    san(y)
end
release(san)

```



### Obtain the Characteristic Information of the NCO Object

The characteristic information of the NCO object is defined by the following fields:

- NumPointsLUT — Number of data points in the lookup table.
- SineLUTSize — Quarter-wave sine lookup table size in bytes.
- TheoreticalSFDR — Theoretical spurious free dynamic range (SFDR) in dBc.
- FrequencyResolution — Frequency resolution of the NCO.

To obtain the above characteristics for a specific NCO object, call the `info` function on the object.

```
nco = dsp.NCO
```

```
nco =  
    dsp.NCO with properties:
```

```
        PhaseIncrementSource: 'Input port'  
        PhaseOffsetSource: 'Property'  
        PhaseOffset: 0  
        Dither: true  
        NumDitherBits: 4  
        PhaseQuantization: true  
        NumQuantizerAccumulatorBits: 12  
        PhaseQuantizationErrorOutputPort: false
```

```

        Waveform: 'Sine'
    SamplesPerFrame: 1
    OutputDataType: 'Custom'

```

Show all properties

```
info(nco)
```

```
ans = struct with fields:
    NumPointsLUT: 1025
    SineLUTSize: 2050
    TheoreticalSFDR: 84
    FrequencyResolution: 1.5259e-05

```

The fields and their corresponding values change depending on the settings of the object. For instance, if the `PhaseQuantization` property is set to `false`, the `TheoreticalSFDR` field does not appear.

```
nco.PhaseQuantization = false;
info(nco)
```

```
ans = struct with fields:
    NumPointsLUT: 16385
    SineLUTSize: 32770
    FrequencyResolution: 1.5259e-05

```

## Algorithms

This object implements the algorithm, inputs, and outputs described on the NCO block reference page. The object properties correspond to the block properties, except there is no object property that corresponds to the **Sample time** block parameter. The objects assumes a sample time of one second.

## Version History

Introduced in R2012a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

## See Also

### Functions

info

**Objects**

dsp.SineWave



# dsp.NotchPeakFilter

**Package:** dsp

Second-order tunable notching and peaking IIR filter

## Description

The `NotchPeakFilter` object filters each channel of the input using IIR filter implementation.

To filter each channel of the input:

- 1 Create the `dsp.NotchPeakFilter` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
npFilter = dsp.NotchPeakFilter
npFilter = dsp.NotchPeakFilter('Specification','Quality factor and center
frequency')
npFilter = dsp.NotchPeakFilter('Specification','Coefficients')
npFilter = dsp.NotchPeakFilter(Name,Value)
```

### Description

`npFilter = dsp.NotchPeakFilter` returns a second-order notching and peaking IIR filter that independently filters each channel of the input over time, using a specified center frequency and 3 dB bandwidth.

`npFilter = dsp.NotchPeakFilter('Specification','Quality factor and center frequency')` specifies the quality factor (Q factor) of the notch or peak filter instead of the 3 dB bandwidth.

`npFilter = dsp.NotchPeakFilter('Specification','Coefficients')` specifies the coefficient values that affect bandwidth and center frequency directly, rather than specifying the design parameters in Hz. This removes the trigonometry calculations involved when the properties are tuned.

`npFilter = dsp.NotchPeakFilter(Name,Value)` returns a notch filter with each specified property name set to the specified value. Unspecified properties have default values.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

**Specification — Filter specification**

'Bandwidth and center frequency' (default) | 'Quality factor and center frequency' | 'Coefficients'

Set the specification as 'Bandwidth and center frequency', 'Quality factor and center frequency', or 'Coefficients'.

**Bandwidth — 3 dB bandwidth**

2205 (default) | positive scalar

Specify the filter's 3 dB bandwidth as a finite positive numeric scalar in Hz. The value must be a scalar between 0 and half the sample rate.

**Tunable:** Yes

**Dependencies**

This property is applicable only if Specification is 'Bandwidth and center frequency'.

Data Types: single | double

**CenterFrequency — Notch or peak center frequency**

11025 (default) | positive scalar

Specify the filter's center frequency (for both the notch and the peak) as a finite positive numeric scalar in Hz. The value must be a scalar between 0 and half the sample rate.

**Tunable:** Yes

**Dependencies**

This property is applicable only if Specification is set to 'Bandwidth and center frequency' or 'Quality factor and center frequency'.

Data Types: single | double

**QualityFactor — Quality factor for notch or peak filter**

5 (default) | positive scalar

Specify the quality factor (Q factor) for both the notch and the peak filters. The Q factor is defined as the center frequency divided by the bandwidth. A higher Q factor corresponds to a narrower notch or peak. The Q factor should be a scalar value greater than 0.

**Tunable:** Yes

**Dependencies**

This property is applicable only if Specification is set to 'Quality factor and center frequency'.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fi

**SampleRate — Sample rate of input**

44100 (default) | positive scalar

Specify the sample rate of the input in Hz as a finite numeric scalar.

Data Types: `single` | `double`

**BandwidthCoefficient — Bandwidth coefficient**

0.72654 (default) | real scalar in the range [-1 1]

Specify the value that determines the filter's 3 dB bandwidth as a finite numeric scalar in the range [-1 1]. The value -1 corresponds to the maximum 3 dB bandwidth ( $\text{SampleRate}/4$ ), and 1 corresponds to the minimum 3 dB bandwidth (0 Hz, an allpass filter).

**Tunable:** Yes

**Dependencies**

This property is only applicable if `Specification` is set to `'Coefficients'`.

Data Types: `single` | `double`

**CenterFrequencyCoefficient — Center frequency coefficient**

0 (default) | real scalar in the range [-1 1]

Specify the coefficient that determines the filter's center frequency as a finite numeric scalar in the range [-1 1]. The value -1 corresponds to the minimum center frequency (0 Hz), and 1 corresponds to the maximum center frequency ( $\text{SampleRate}/2$  Hz). The default is 0, which corresponds to  $\text{SampleRate}/4$  Hz.

**Tunable:** Yes

**Dependencies**

This property is only applicable if `Specification` is set to `'Coefficients'`.

Data Types: `single` | `double`

**Usage****Syntax**

```
Y = npFilter(x)
[Yn,Yp] = npFilter(x)
```

**Description**

`Y = npFilter(x)` filters each channel (column) of the input signal, `x`, to produce the notch filter output, `Y`.

`[Yn,Yp] = npFilter(x)` filters each channel of the input signal, `x`, to produce the notch filter output, `Yn`, and peak filter output, `Yp`.

**Input Arguments****x — Input signal**

vector | matrix

Input signal, specified as a vector or a matrix.

Data Types: `single` | `double`

### Output Arguments

#### **Yn — Notch filter output**

vector | matrix

Notch filter output, returned as a vector or a matrix.

Data Types: `single` | `double`

#### **Yp — Peak filter output**

vector | matrix

Peak filter output, returned as a vector or a matrix.

Data Types: `single` | `double`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Specific to `dsp.NotchPeakFilter`

|                                 |                                |
|---------------------------------|--------------------------------|
| <code>getBandwidth</code>       | Get 3 dB bandwidth             |
| <code>getCenterFrequency</code> | Get center frequency           |
| <code>getOctaveBandwidth</code> | Bandwidth in number of octaves |
| <code>getQualityFactor</code>   | Get quality factor             |
| <code>tf</code>                 | Transfer function              |

## Common to All System Objects

|                      |  |
|----------------------|--|
| <code>step</code>    | Run System object algorithm  |
| <code>release</code> | Release resources and allow changes to System object property values and input characteristics |
| <code>reset</code>   | Reset internal states of System object   |

## Examples

### Notch Filter

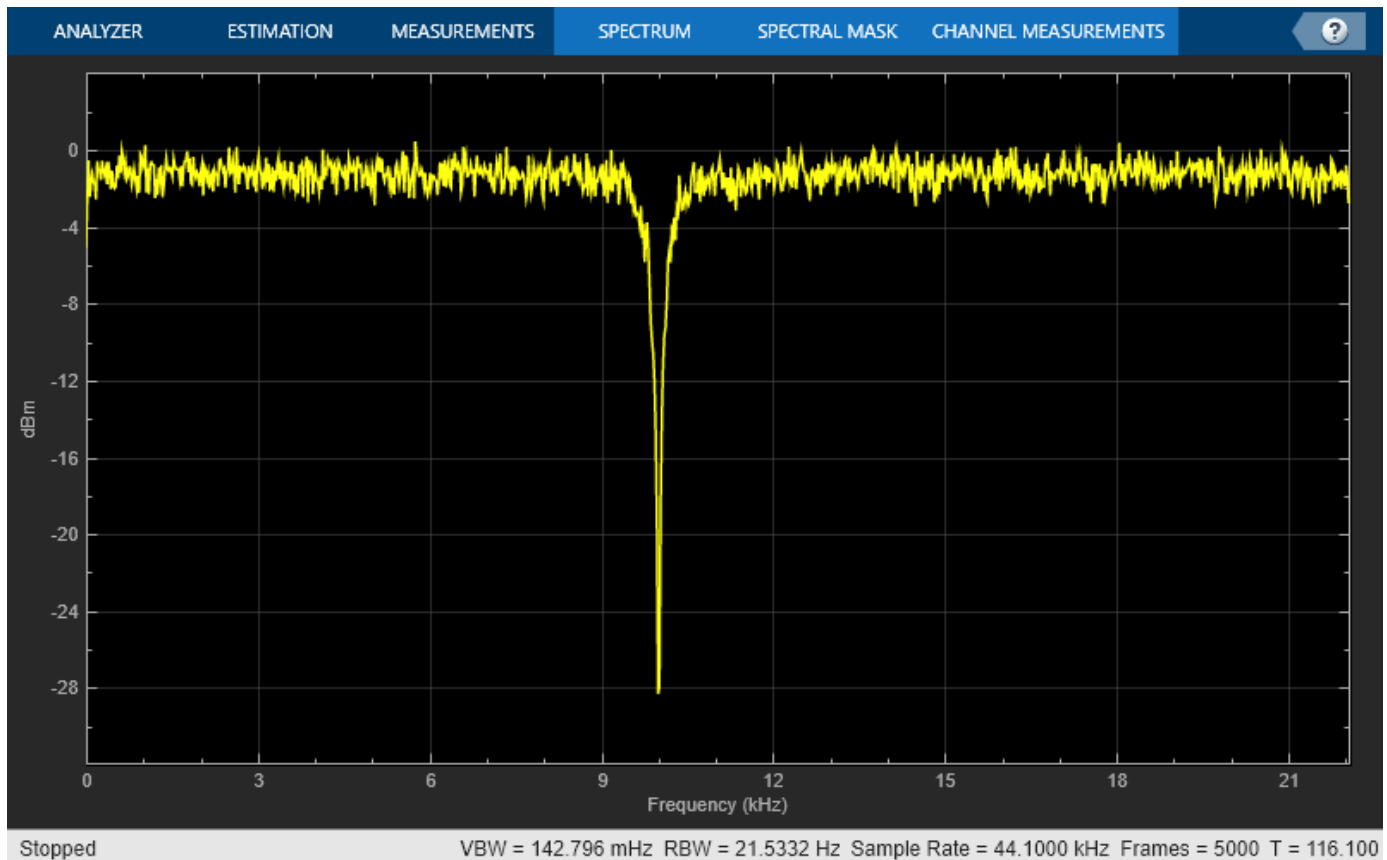
This example shows how to use `dsp.NotchPeakFilter` as a notch filter with center frequency of 5000 Hz and a 3 dB bandwidth of 500 Hz.

```
npFilter = dsp.NotchPeakFilter('CenterFrequency',5000,'Bandwidth',500);
sa = spectrumAnalyzer('SampleRate',44100,'AveragingMethod','exponential',...
    'ForgettingFactor',0.96,'PlotAsTwoSidedSpectrum',false);
for i=1:5000
    y = npFilter(randn(1024,1));
    sa(y);
```

```

if (i==2500)
    % Tune center frequency to 10000
    npFilter.CenterFrequency = 10000;
end
end
release(npFilter)
release(sa)

```



### Get 3 dB Bandwidth of Notch Peak Filter

Create a `dsp.NotchPeakFilter` object with the `Specification` property set to 'Quality factor and center frequency'. The default quality factor  $Q$  is 5, and the center frequency  $F_c$  is 11,025 Hz.

```

np = dsp.NotchPeakFilter('Specification',...
    'Quality factor and center frequency')

```

```

np =
    dsp.NotchPeakFilter with properties:

```

```

    Specification: 'Quality factor and center frequency'
    QualityFactor: 5
    CenterFrequency: 11025
    SampleRate: 44100

```

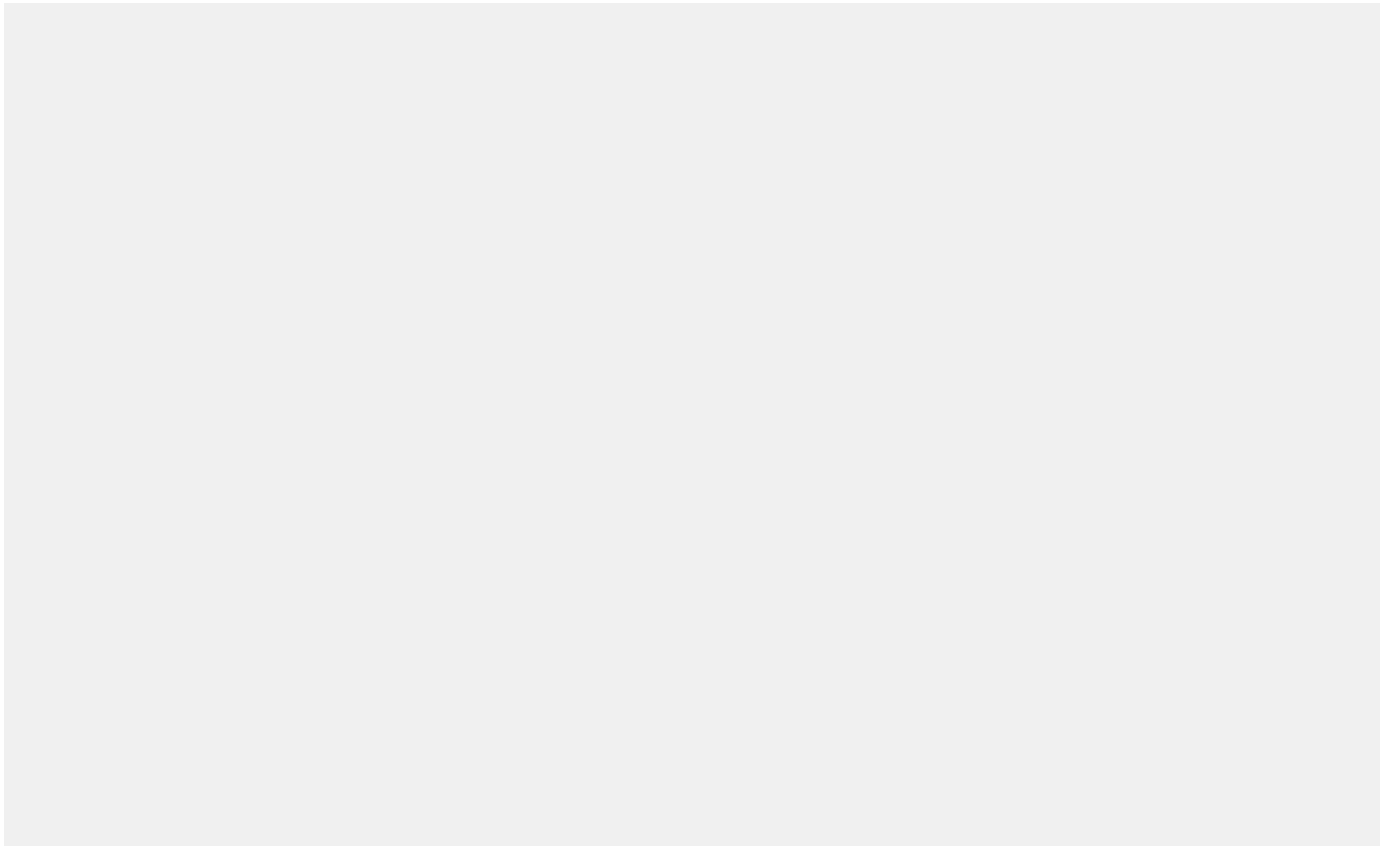
Compute the 3 dB bandwidth of the notch peak filter using the `getBandwidth` function. The bandwidth is computed as the ratio of the center frequency and the quality factor,  $\frac{F_c}{Q}$ .

```
getBandwidth(np)
```

```
ans = 2205
```

Visualize the filter response using `fvtool`.

```
fvtool(np)
```



### Get Center Frequency of Notch Peak Filter

Create a `dsp.NotchPeakFilter` object with the `Specification` property set to `'Coefficients'`.

```
np = dsp.NotchPeakFilter('Specification','Coefficients')
```

```
np =  
  dsp.NotchPeakFilter with properties:  
      Specification: 'Coefficients'  
  BandwidthCoefficient: 0.7265  
  CenterFrequencyCoefficient: 0
```

SampleRate: 44100

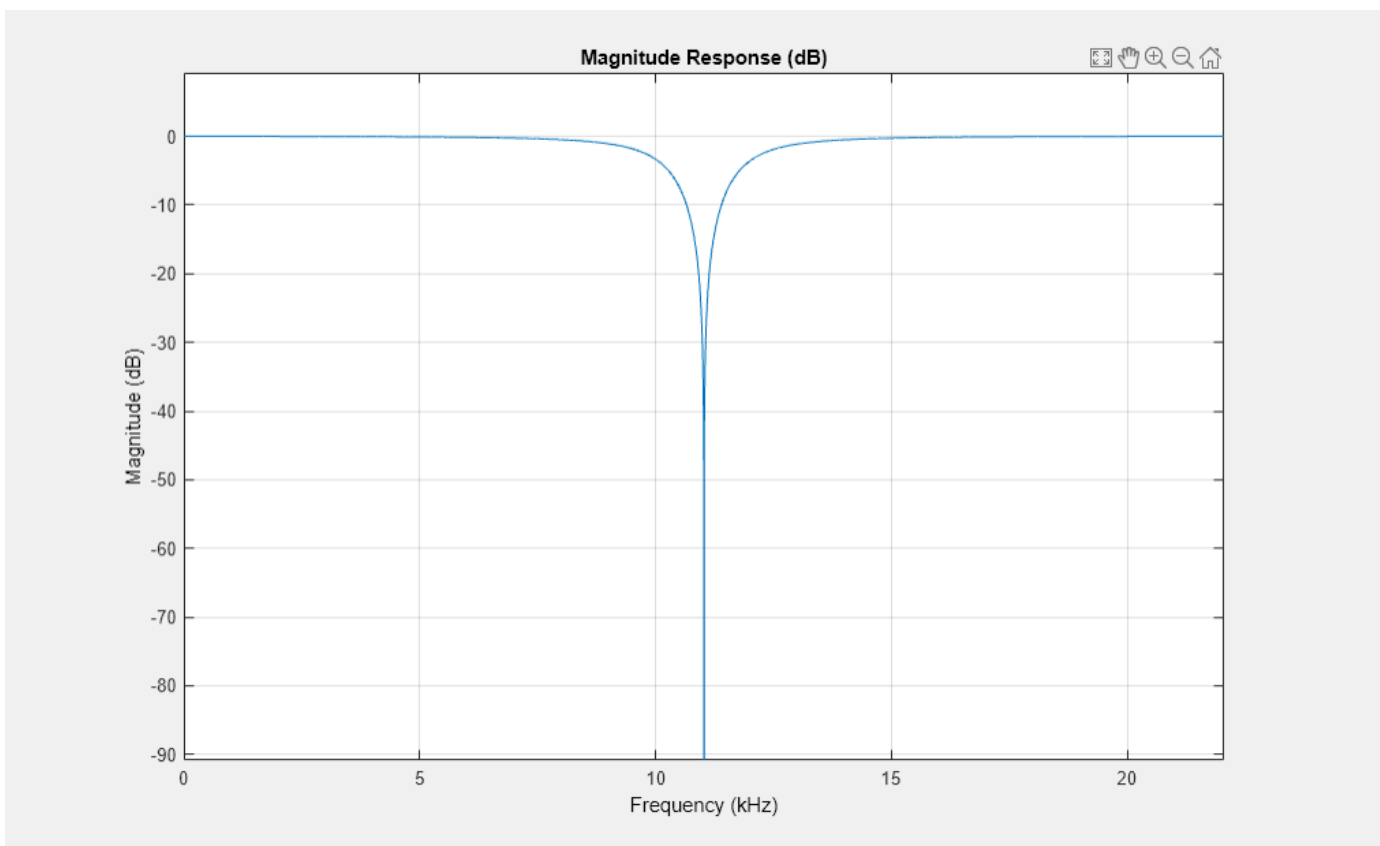
Determine the center frequency of the notch peak filter using the `getCenterFrequency` function. When the `Specification` is set to `'Coefficients'`, the center frequency is determined from the `CenterFrequencyCoefficient` value and the sample rate.

```
getCenterFrequency(np)
```

```
ans = 11025
```

Visualize the filter response using `fvtool`.

```
fvtool(np)
```



### Get Octave Bandwidth of Notch Peak Filter

Create a `dsp.NotchPeakFilter` object in the default configuration.

```
np = dsp.NotchPeakFilter
```

```
np =  
dsp.NotchPeakFilter with properties:
```

```
Specification: 'Bandwidth and center frequency'
```

```
Bandwidth: 2205  
CenterFrequency: 11025  
SampleRate: 44100
```

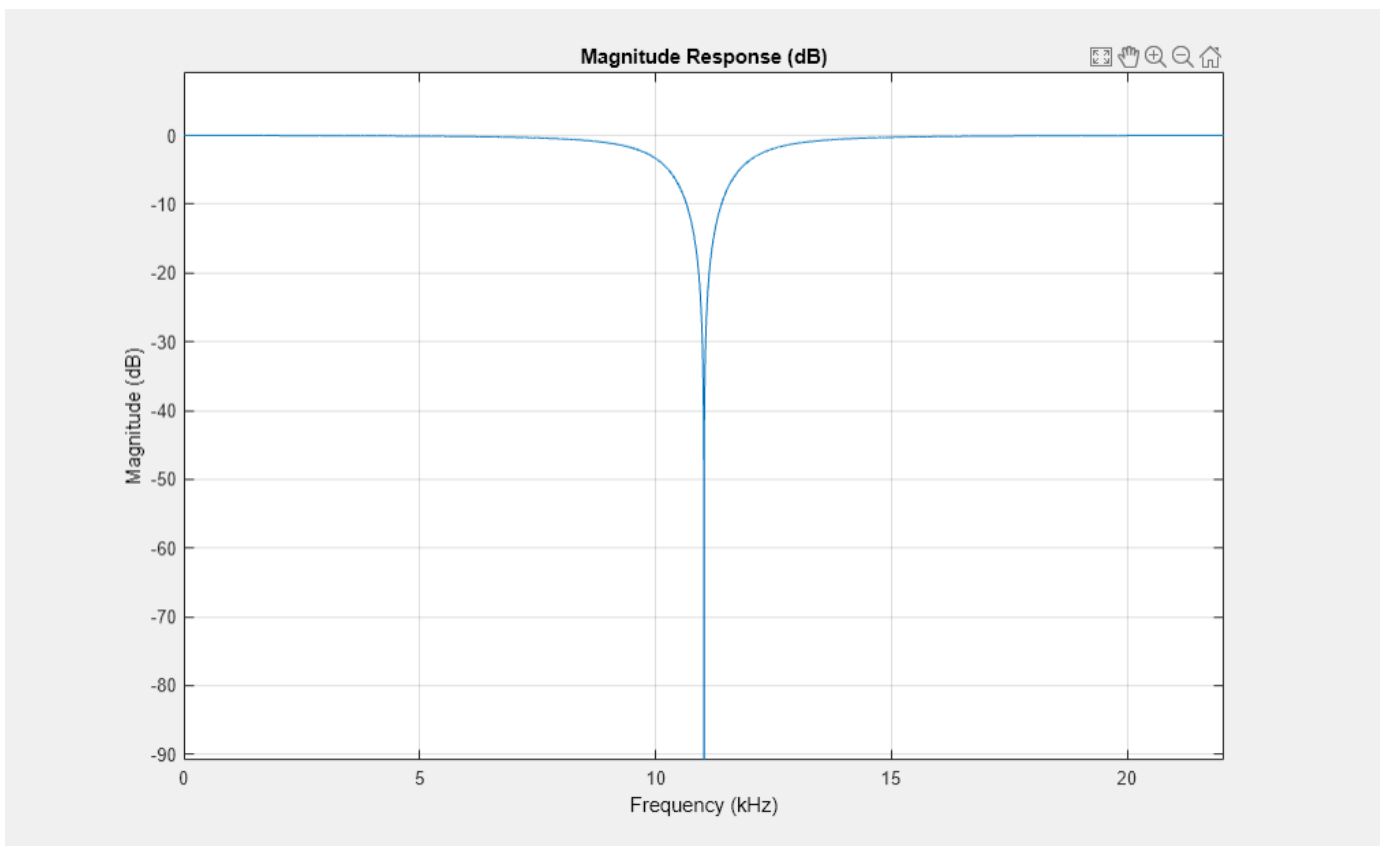
Determine the octave bandwidth of the filter using the `getOctaveBandwidth` function.

```
getOctaveBandwidth(np)
```

```
ans = 0.2881
```

Visualize the filter response using `fvtool`.

```
fvtool(np)
```



### Compute Quality Factor of Notch Peak Filter

Create a `dsp.NotchPeakFilter` object in the default configuration, where the `Specification` property is set to 'Bandwidth and center frequency'.

```
np = dsp.NotchPeakFilter
```

```
np =  
    dsp.NotchPeakFilter with properties:
```



```

Specification: 'Bandwidth and center frequency'
Bandwidth: 2205
CenterFrequency: 11025
SampleRate: 44100

```

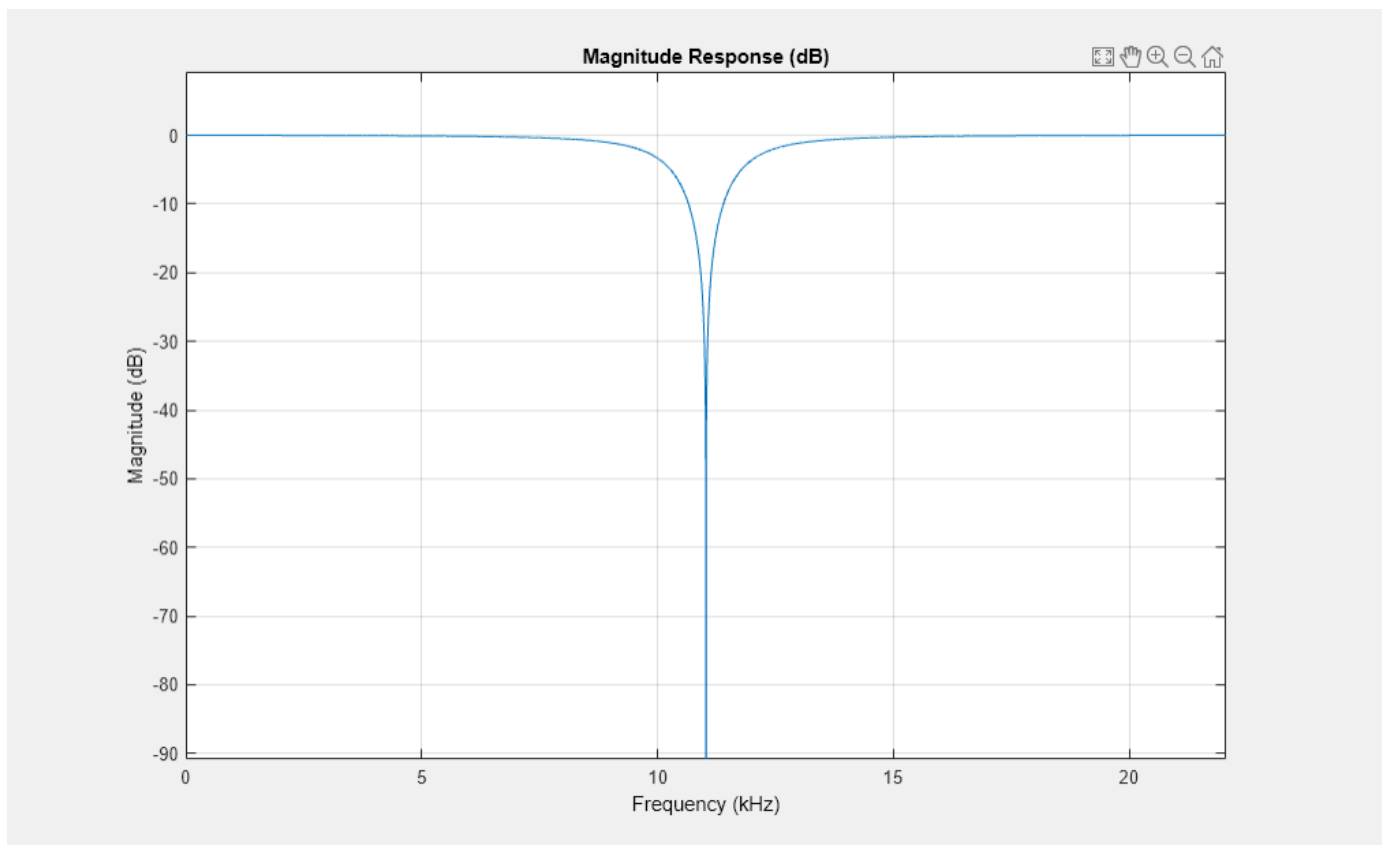
Determine the quality factor of the filter using the `getQualityFactor` function. The quality factor is given by the ratio of the center frequency to the bandwidth.

```
getQualityFactor(np)
```

```
ans = 5
```

Visualize the filter response using `fvtool`.

```
fvtool(np)
```



### Determine Transfer Function of Notch Peak Filter

Create a `dsp.NotchPeakFilter` System object™. Obtain the coefficients of the transfer function corresponding to the notch and peak filters.

```

notchpeak = dsp.NotchPeakFilter;
[Bnotch,Anotch,Bpeak,Apeak] = tf(notchpeak)

```

Bnotch = 1×3

0.8633 -0.0000 0.8633

Anotch = 1×3

1.0000 -0.0000 0.7265

Bpeak = 1×3

0.1367 0 -0.1367

Apeak = 1×3

1.0000 -0.0000 0.7265

Bnotch and Anotch are the vectors of numerator and denominator coefficients for the equivalent transfer function corresponding to the notch filter. Bpeak and Apeak are the vectors of numerator and denominator coefficients for the equivalent transfer function corresponding to the peak filter.

## Algorithms

The design equation for the peak filter is:

$$H(z) = (1 - b) \frac{1 - z^{-2}}{1 - 2b \cos \omega_0 z^{-1} + (2b - 1)z^{-2}}$$

The design equation for the notch filter is:

$$H(z) = b \frac{1 - 2 \cos \omega_0 z^{-1} + z^{-2}}{1 - 2b \cos \omega_0 z^{-1} + (2b - 1)z^{-2}}$$

with

$$b = \frac{1}{1 + \tan(\Delta\omega/2)}$$

where  $\omega_0 = 2\pi f_0/f_s$  is the center frequency in radians/sample ( $f_0$  is the center frequency in Hz and  $f_s$  is the sampling frequency in Hz).  $\Delta\omega = 2\pi\Delta f/f_s$  is the 3 dB bandwidth in radians/sample ( $\Delta f$  is the 3 dB bandwidth in Hz). Note that the two filters are complementary:

$$H_{\text{notch}}(z) + H_{\text{peak}}(z) = 1$$

they can be written as:

$$H_{\text{peak}}(z) = \frac{1}{2}[1 - A(z)]$$

$$H_{\text{notch}}(z) = \frac{1}{2}[1 + A(z)]$$

where  $A(z)$  is a 2<sup>nd</sup> order allpass filter.

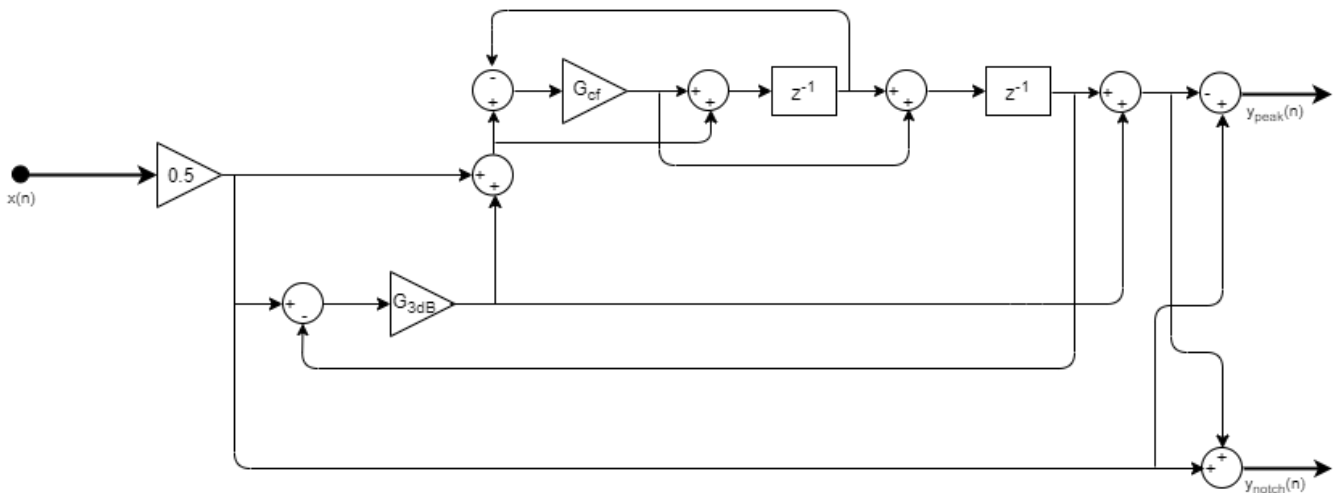
$$A(z) = \frac{a_2 + a_1z^{-1} + z^{-2}}{1 + a_1z^{-1} + a_2z^{-2}}$$

and

$$a_1 = -2b\cos\omega_0$$

$$a_2 = 2b - 1$$

The filter is implemented as follows:



where

$$G_{3\text{dB}} = a_2 = 2b - 1$$

$$G_{\text{cf}} = \frac{a_1 - a_1a_2}{1 - a_2^2} = -\cos\omega_0$$

Notice that  $G_{\text{cf}}$  depends only on the center frequency, and  $G_{3\text{dB}}$  depends only on the 3 dB bandwidth.

## Version History

Introduced in R2014a

## References

- [1] Orfanidis, Sophocles J. *Introduction to Signal Processing*. Upper Saddle River, NJ: Prentice-Hall, 1996.

## See Also

### Functions

iirnotch | iirpeak | getBandwidth | getCenterFrequency | getOctaveBandwidth |  
getQualityFactor | tf

### Objects

dsp.SOSFilter

### Blocks

Notch-Peak Filter

# dsp.PeakFinder

**Package:** dsp

(Removed) Identify peak values in input signal

---

**Note** The `dsp.PeakFinder` System object™ has been removed. To determine the local maxima, use the `findpeaks` function. To determine the local minima, use the `islocalmin` function and find the signal values corresponding to the local minima indices that the function determines. For more details, see “Compatibility Considerations”.

---

## Description

The `dsp.PeakFinder` System object counts the number of peak values (maxima, minima, or both) in each column of the real-valued input signal. To qualify as a peak, a point has to be larger (or smaller) than both of its neighboring points. The end points are not considered as peak values. The object can also output the indices and values of the peaks, and a binary array that indicates whether a peak is a maxima or a minima.

To output the peak indices and peak values, set “PeakIndicesOutputPort” on page 4-0 and “PeakValuesOutputPort” on page 4-0 to `true`, respectively. In addition, you can determine which of the peak values is a maxima or a minima using the polarity matrix. The polarity matrix is a logical array in which a 1 indicates a maxima, and a 0 indicates a minima. To view the polarity matrix, set “PeakType” on page 4-0 to 'Maxima and Minima' and access the fourth output.

Use the “MaximumPeakCount” on page 4-0 property to specify how many peak values to look for in each input signal. The object stops searching the input signal once this maximum number of peak values has been found.

If you set “IgnoreSmallPeaks” on page 4-0 to `true`, the object no longer detects low amplitude peaks and ignores noise within a threshold value that you define. In this mode, the current value is a maximum if  $(current - previous) > threshold$  and  $(current - next) > threshold$ . The current value is a minimum if  $(current - previous) < -threshold$  and  $(current - next) < -threshold$ .

To determine the peak values in an input signal:

- 1 Create the `dsp.PeakFinder` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
pkFind = dsp.PeakFinder
pkFind = dsp.PeakFinder(Name,Value)
```

## Description

`pkFind = dsp.PeakFinder` creates a peak finder System object that identifies the peak values (maxima, minima, or both) in an input signal.

`pkFind = dsp.PeakFinder(Name, Value)` creates a peak finder System object with each specified property set to the specified value. Enclose each property name in single quotes.

Example: `delay = dsp.PeakFinder('PeakType','Maxima');`

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see *System Design in MATLAB Using System Objects*.

### PeakType — Type of peaks to identify

'Maxima and Minima' (default) | 'Maxima' | 'Minima'

Type of peaks to identify, specified as one of the following:

- 'Maxima' -- If "IgnoreSmallPeaks" on page 4-0 is set to `true`, the current value is identified as a maximum if  $(current - previous) > threshold$  and  $(current - next) > threshold$ . Specify *threshold* in "PeakThreshold" on page 4-0 property. If `IgnoreSmallPeaks` is set to `false`, the current value must be larger than both its neighboring points.
- 'Minima' -- If `IgnoreSmallPeaks` is set to `true`, the current value is identified as a minimum if  $(current - previous) < -threshold$  and  $(current - next) < -threshold$ . If `IgnoreSmallPeaks` is set to `false`, the current value must be smaller than both its neighboring points.
- 'Maxima and Minima' -- The object identifies both the maxima and the minima points.

### PeakIndicesOutputPort — Enable output of peak indices

`false` (default) | `true`

Enable output of the peak indices, specified as either:

- `true` -- The object returns the indices of the input signal peak values.
- `false` -- The object does not return the indices of the input signal peak values.

### PeakValuesOutputPort — Enable output of peak values

`false` (default) | `true`

Enable output of the peak values, specified as either:

- `true` -- The object returns the peak values of the input signal.
- `false` -- The object does not return the peak values of the input signal.

### MaximumPeakCount — Maximum number of peak values to identify

10 (default) | integer greater than or equal to 1

Maximum number of peak values to identify in each input signal, specified as an integer greater than or equal to 1. The object stops searching the input signal for peaks once it identifies the maximum number.

Example: 5

Example: 50

### IgnoreSmallPeaks — Ignore peaks below a threshold

false (default) | true

Ignore peaks below a threshold, specified as either:

- **false** -- The object identifies the current value as a peak, if the current value is larger (or smaller) than both of its neighboring points.
- **true** -- The object identifies the current value as a maximum if  $(current - previous) > threshold$  and  $(current - next) > threshold$ . The object identifies the current value as a minimum if  $(current - previous) < -threshold$  and  $(current - next) < -threshold$ .

### PeakThreshold — Threshold below which peaks are ignored

0 (default) | positive real scalar | vector

Threshold below which peaks are ignored, specified as a real scalar greater than or equal to 0 or a vector with all elements greater than or equal to 0. The length of the vector must be equal to the number of channels.

This property identifies the current input value to be a maximum if  $(current\ input\ value - previous\ input\ value) > threshold$  and  $(current\ input\ value - next\ input\ value) > threshold$ . The current value is a minimum if  $(current\ input\ value - previous\ input\ value) < -threshold$  and  $(current\ input\ value - next\ input\ value) < -threshold$ .

Example: 0.2

Example: [0.3 2.4]

Example: [3; 0.4]

### Dependencies

This property applies only when you set IgnoreSmallPeaks to true.

### Fixed-Point Properties

#### RoundingMethod — Rounding method

'Floor' (default)

Rounding method, specified as 'Floor', which is equivalent to truncation. The setting rounds the result of a calculation to the closest representable number in the direction of negative infinity.

#### OverflowAction — Overflow action

'Wrap' (default) | 'Saturate'

The overflow action for fixed-point operations, specified as one of the following:

- **'Wrap'** -- The object wraps the result of its fixed-point operations.
- **'Saturate'** -- The object saturates the result of its fixed-point operations.

For more details on overflow actions, see overflow mode for fixed-point operations.

## Usage

### Syntax

```
cnt = pkFind(input)
[cnt,idx] = pkFind(input)
[ ___,val] = pkFind(input)
[ ___,pol] = pkFind(input)
```

### Description

`cnt = pkFind(input)` returns the number of peak values (minima, maxima, or both) in the input signal. Each column of the input is treated as a separate channel.

`[cnt,idx] = pkFind(input)` returns the number of peak values, `cnt`, and peak indices, `idx`, in the input signal.

To access the peak indices output, set the `PeakIndicesOutputPort` property to `true`.

```
pkFind = dsp.PeakFinder('PeakType','Maxima', ...
'PeakIndicesOutputPort',true);
...
[cnt,idx] = pkFind(input);
```

`[ ___,val] = pkFind(input)` returns the peak values `val` in the input signal.

To access the peak values output, set the `PeakValuesOutputPort` property to `true`.

```
pkFind = dsp.PeakFinder('PeakType','Maxima', ...
'PeakValuesOutputPort',true);
...
[cnt,idx,val] = pkFind(input);
```

`[ ___,pol] = pkFind(input)` returns the peak value polarity `pol` in input signal. The polarity is 1 for maxima and 0 for minima.

To access the polarity output, set the `PeakType` property to `'Maxima and Minima'`, and the `PeakIndicesOutputPort` property to `true`.

```
pkFind = dsp.PeakFinder('PeakType','Maxima and Minima', ...
'PeakIndicesOutputPort',true);
...
[cnt,idx,val,pol] = pkFind(input);
```

### Input Arguments

#### **input** — Data input

column vector of at least three rows | matrix with at least three rows

Data input whose peak values are detected by the object, specified as a vector or matrix containing at least three rows.

Example: `[9 6 10 3 5 5 0 12; 9 6 1 13 4 1 0 12]'`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`



## Output Arguments

### **cnt** — Number of peak values

scalar | row vector

Number of peak values, returned as a scalar or a row vector of length equal to the number of channels in the data input.

Example: [4 3]

Data Types: uint32

### **idx** — Indices of the peak values

column vector | matrix

Indices of the peak values in the input signal, returned as a column vector or matrix. The size of the peak indices output is the same as that of the input. All nonzero elements represent peak indices.

Example: [1 2 3 6 0 0 0 0 0 0; 2 3 6 0 0 0 0 0 0]'

#### Dependencies

To enable this output, set the `PeakIndicesOutputPort` property to `true`.

Data Types: uint32

### **val** — Peak values

column vector | matrix

Peak values in the input signal, returned as a column vector or matrix.

Example: [6 10 3 0 0 0 0 0 0 0; 1 13 0 0 0 0 0 0 0 0]'

#### Dependencies

To access the peak values output, set the `PeakValuesOutputPort` property to `true`.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fi

### **pol** — Peak value polarity

column vector | matrix

Peak value polarity in input signal, returned as a column vector or matrix of logical 1s and 0s. The polarity is 1 for maxima and 0 for minima.

Example: [0 1 0 0 0 0 0 0 0 0; 0 1 0 0 0 0 0 0 0 0]'

Data Types: logical

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Common to All System Objects

step     Run System object algorithm  
release   Release resources and allow changes to System object property values and input characteristics  
reset     Reset internal states of System object

## Examples

### Determine Local Maxima and Minima

Determine whether each value of an input signal is a local maximum or minimum.

```
pkFind = dsp.PeakFinder;  
pkFind.PeakIndicesOutputPort = true;  
pkFind.PeakValuesOutputPort = true;
```

```
x1 = [9 6 10 3 4 5 0 12]';
```

Find the peaks of each input [prev; cur; next]: {[9;6;10],[6;10;3],...}

```
[cnt1, idx1, val1, pol1] = pkFind(x1)
```

```
cnt1 = uint32
```

```
5
```

```
idx1 = 10x1 uint32 column vector
```

```
1  
2  
3  
5  
6  
0  
0  
0  
0  
0
```

```
val1 = 10x1
```

```
6  
10  
3  
5  
0  
0  
0  
0  
0  
0
```

```
pol1 = 10x1 logical array
```

```
0  
1  
0  
1  
0
```

```
0
0
0
0
0
```

## Version History

### Introduced in R2012a

#### R2021b: dsp.PeakFinder System object has been removed

*Errors starting in R2021b*

The dsp.PeakFinder System object has been removed. To determine the local maxima, use the findpeaks function. To determine the local minima, use the islocalmin function and find the signal values corresponding to the local minima indices that the function determines. For more details, see the **Update Code** section.

#### Update Code

This table shows typical usage of the System object and explains how to update existing code to use the equivalent function.

| Discouraged Usage   | Recommended Replacement   |
|---|---|
| <p><b>Find Local Maxima</b></p> <pre>x = [9 6 10 3 4 5 0 12]'; pf = dsp.PeakFinder('PeakType','Maxima', ..     'PeakValuesOutputPort', true); [cnt,val] = pf(x)</pre> <p><b>Find Local Minima</b></p> <pre>x = [9 6 10 3 4 5 0 12]'; pf = dsp.PeakFinder('PeakType','Minima', ..     'PeakValuesOutputPort', true); [cnt,val] = pf(x)</pre> <p>If you are using a release prior to R2016b, replace pf(x) with step(pf,x).</p> | <p><b>Find Local Maxima</b></p> <pre>x = [9 6 10 3 4 5 0 12]'; val = findpeaks(x) cnt = numel(val)</pre> <p><b>Find Local Minima</b></p> <pre>x = [9 6 10 3 4 5 0 12]'; tf = islocalmin(x); val = x(tf) cnt = sum(tf)</pre> |

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

## **See Also**

### **Functions**

findpeaks | islocalmin

### **Blocks**

Peak Finder

# dsp.PeakToPeak

**Package:** dsp

(Removed) Peak-to-peak value

---

**Note** dsp.PeakToPeak has been removed. Use the peak2peak function instead. For more information, see “Compatibility Considerations”.

---

## Description

The dsp.PeakToPeak System object computes the peak-to-peak value of an input.

To obtain the peak-to-peak value:

- 1 Create the dsp.PeakToPeak object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

## Creation

### Syntax

```
ptp = dsp.PeakToPeak
ptp = dsp.PeakToPeak(Name, Value)
```

### Description

ptp = dsp.PeakToPeak creates a peak-to-peak System object, ptp, that computes the difference between the maximum and minimum value in an input or a sequence of inputs.

ptp = dsp.PeakToPeak(Name, Value) returns a peak-to-peak System object with each specified property set to the specified value. Enclose each property name in single quotes. Unspecified properties have default values.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the release function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

### RunningPeakToPeak — Calculation over successive calls to object algorithm

false (default) | true

Set this property to `true` to enable the calculation of the peak-to-peak difference over successive calls to the algorithm.

**Dimension — Dimension to operate along**

'Column' (default) | 'All' | 'Row' | 'Custom'

Specify the dimension along which to calculate the peak-to-peak ratio as 'All', 'Row', 'Column', or 'Custom'. If you set this property to 'Custom', specify the dimension using the `CustomDimension` property.

**Dependencies**

This property applies when the `RunningPeakToPeak` property is `false`.

**CustomDimension — Dimension to operate along**

1 (default) | positive integer

Specify the dimension as a positive integer along which the peak-to-peak difference is computed. The value of this property cannot exceed the number of dimensions in the input signal.

**Dependencies**

This property applies when the `Dimension` property is 'Custom'.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**ResetInputPort — Enables resetting in running peak-to-peak mode**

`false` (default) | `true`

Set this property to `true` to enable resetting the running peak-to-peak. When the property is set to `true`, a reset input must be specified to the call of object algorithm to reset the running peak-to-peak difference.

**Dependencies**

This property applies when the `RunningPeakToPeak` property is `true`.

**ResetCondition — Reset condition for running peak-to-peak mode**

'Non-zero' (default) | 'Rising edge' | 'Falling edge' | 'Either edge'

Specify the event to reset the running peak-to-peak as one of 'Rising edge', 'Falling edge', 'Either edge', or 'Non-zero'.

**Dependencies**

This property applies when the `ResetInputPort` property is `true`.

## Usage

### Syntax

```
y = ptp(x)
y = ptp(x,r)
```

**Description**

`y = ptp(x)` computes the peak-to-peak value, `y`, of the floating-point input vector `x`.

$y = \text{ptp}(x, r)$  computes the peak-to-peak value of the input elements over successive calls to the object algorithm. The object optionally resets its state based on the reset input signal,  $r$ , and the value of the `ResetCondition` property. To enable reset, set both the `RunningPeakToPeak` and the `ResetInputPort` properties to `true`.

### Input Arguments

#### **x** — Data input

vector | matrix

Data input, specified as a vector or a matrix. If  $x$  is a matrix, each column is treated as an independent channel. The peak-to-peak value is computed along each channel.

Data Types: `single` | `double`

#### **r** — Reset signal

scalar

Reset signal, specified as a scalar value. The reset signal resets the object state based on the reset input signal and the value of the `ResetCondition` property.

### Dependencies

To enable this signal, set both the `RunningPeakToPeak` and the `ResetInputPort` properties to `true`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `logical`

### Output Arguments

#### **y** — Peak-to-peak value

scalar | vector | matrix

Peak-to-peak value of the input signal, returned as a scalar, vector, or matrix. If `RunningPeakToPeak` is set to:

- `false` -- The object computes the peak-to-peak value of each input channel. If the input is a column vector, the output is a scalar. If the input is a multichannel signal, the output signal is a 1-by- $N$  vector, where  $N$  is the number of input channels.
- `true` -- The object computes the running peak-to-peak value of the signal. The size of the output signal matches the size of the input signal.

When the `RunningPeakToPeak` property is `true`,  $y$  corresponds to the peak-to-peak value of the input elements over successive calls to the object algorithm.

Data Types: `single` | `double`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Common to All System Objects

|         |  |
|---------|--|
| step    | Run System object algorithm  |
| release | Release resources and allow changes to System object property values and input characteristics |
| reset   | Reset internal states of System object   |

## Examples

### Peak-to-Peak Value of Vector

**Note:** If you are using R2016a or an earlier release, replace each call to the object with the equivalent step syntax. For example, `obj(x)` becomes `step(obj,x)`.

Determine the peak-to-peak value for a vector input.

```
in = (1:10)';  
ptp = dsp.PeakToPeak;  
y = ptp(in)  
  
y = 9
```

## Version History

### Introduced in R2012a

### **R2021b: dsp.PeakToPeak System object will be removed**

*Warns starting in R2021b*

`dsp.PeakToPeak` System object will be removed in a future release. Use the `peak2peak` function instead.

### **R2023a: dsp.PeakToPeak System object has been removed**

*Errors starting in R2023a*

The `dsp.PeakToPeak` System object has been removed. Use the `peak2peak` function instead.

### Update Code

This table shows how to update existing code to use the `peak2peak` function.



| Discouraged Usage  | Recommended Replacement   |
|--|---|
| <p><b>Peak-to-Peak Value of Vector Input</b></p> <pre>in = (1:10)'; ptp = dsp.PeakToPeak; y = ptp(in)</pre> <p><i>y = 1×1</i></p> <p>9</p> <p><b>Peak-to-Peak Value of Matrix Input</b></p> <p>Set the Dimension property of the object to 'All' to compute the peak-to-peak values across both dimensions.</p> <pre>in = magic(4); ptp = dsp.PeakToPeak; ptp.Dimension = 'All'; y = ptp(in)</pre> <p><i>y = 1×1</i></p> <p>15</p> | <p><b>Peak-to-Peak Value of Vector Input</b></p> <pre>yfn = peak2peak(in)</pre> <p><i>yfn = 1×1</i></p> <p>9</p> <p><b>Peak-to-Peak Value of Matrix Input</b></p> <pre>yfn = peak2peak(in, 'all')</pre> <p><i>yfn = 1×1</i></p> <p>15</p> |

## References

[1] *IEEE Standard on Transitions, Pulses, and Related Waveforms*, IEEE Standard 181, 2003.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

## See Also

### Functions

peak2peak

### Objects

dsp.PeakToRMS

# dsp.PeakToRMS

**Package:** dsp

Peak-to-root-mean-square value of vector

## Description

The `dsp.PeakToRMS` System object calculates the peak-to-root-mean-square ratio of a vector.

To compute the peak-to-root-mean-square ratio:

- 1 Create the `dsp.PeakToRMS` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
ptr = dsp.PeakToRMS  
ptr = dsp.PeakToRMS(Name, Value)
```

### Description

`ptr = dsp.PeakToRMS` creates a peak-to-root-mean-square System object, `ptr`, that returns the ratio of the maximum magnitude (peak) to the root-mean-square (RMS) value in an input or a sequence of inputs.

`ptr = dsp.PeakToRMS(Name, Value)` returns an peak-to-root-mean-square System object, `ptr`, with each specified property set to the specified value. Enclose each property name in single quotes. Unspecified properties have default values.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

### RunningPeakToRMS — Calculation over successive calls to object algorithm

`false` (default) | `true`

Set this property to `true` to enable the calculation of the peak-to-RMS ratio over successive calls to the object algorithm.

**Dimension — Dimension to operate along**

'Column' (default) | 'All' | 'Row' | 'Custom'

Specify the dimension along which to calculate the peak-to-RMS ratio as one of 'All', 'Row', 'Column', or 'Custom'. If you set this property to 'Custom', specify the dimension using the CustomDimension property.

**Dependencies**

This property applies when the RunningPeakToRMS property is false.

**CustomDimension — Numerical dimension to operate along**

1 (default) | positive integer

Specify the dimension as a positive integer along which the peak-to-RMS ratio is computed. The value of this property cannot exceed the number of dimensions in the input signal.

**Dependencies**

This property applies when the Dimension property is 'Custom'.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**DecibelScaledOutput — Report output in decibels (dB)**

true (default) | false

Set this property to true to enable output in dB. Set this property to false to report output as a ratio.

**ResetInputPort — Enables resetting in running peak-to-RMS mode**

false (default) | true

Set this property to true to enable resetting. When the property is set to true, a reset input must be specified in the call of object algorithm to reset the running peak-to-RMS ratio.

**Dependencies**

This property applies when the RunningPeakToRMS property is true.

**ResetCondition — Reset condition for running peak-to-RMS mode**

'Non-zero' (default) | 'Rising edge' | 'Falling edge' | 'Either edge'

Specify the event to reset the running peak-to-RMS as 'Rising edge', 'Falling edge', 'Either edge', or 'Non-zero'.

**Dependencies**

This property applies when the ResetInputPort property is true.

**Usage****Syntax**

```
y = ptr(x)
y = ptr(x,r)
```

### Description

`y = ptr(x)` computes the peak-to-RMS ratio, `y`, of the floating-point input vector `x`.

`y = ptr(x, r)` computes the peak-to-RMS ratio of the input elements over successive calls to the object algorithm. The object optionally resets its state based on the reset input signal, `r`, and the value of the `ResetCondition` property. To enable reset, set both the `RunningPeakToRMS` and the `ResetInputPort` properties to `true`.

### Input Arguments

#### **x** — Data input

vector | matrix

Data input, specified as a vector or a matrix. If `x` is a matrix, each column is treated as an independent channel. The peak-to-RMS value is computed along each channel.

Data Types: `single` | `double`

#### **r** — Reset signal

scalar

Reset signal, specified as a scalar value. The reset signal resets the object state based on the reset input signal and the value of the `ResetCondition` property.

### Dependencies

To enable this signal, set both the `RunningPeakToRMS` and the `ResetInputPort` properties to `true`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `logical`

### Output Arguments

#### **y** — Peak-to-RMS value

scalar | vector | matrix

Peak-to-RMS value of the input signal, returned as a scalar, vector, or matrix. If `RunningPeakToRMS` is set to:

- `false` -- The object computes the peak-to-RMS value of each input channel. If the input is a column vector, the output is a scalar. If the input is a multichannel signal, the output signal is a 1-by- $N$  vector, where  $N$  is the number of input channels.
- `true` -- The object computes the running peak-to-RMS value of the signal. The size of the output signal matches the size of the input signal.

When the `RunningPeakToRMS` property is `true`, `y` corresponds to the peak-to-RMS ratio of the input elements over successive calls to the object algorithm.

Data Types: `single` | `double`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Common to All System Objects

```
step      Run System object algorithm
release   Release resources and allow changes to System object property values and input
           characteristics
reset     Reset internal states of System object
```

## Examples

### Peak-to-RMS Ratio of Vector Input

Determine the peak-to-RMS ratio of a vector input.

```
in = (1:10)';
ptr = dsp.PeakToRMS;
y = ptr(in)
```

```
y = 1.6116
```

### Peak-to-RMS Ratio of Matrix Input

Determine the peak-to-RMS ratio of a matrix input.

```
in = magic(4);
ptr = dsp.PeakToRMS;
ptr.Dimension = 'All';
y = ptr(in)
```

```
y = 1.6547
```

## More About

### Peak-magnitude-to-RMS Level

The peak-magnitude-to-RMS level is

$$\frac{\|X\|_{\infty}}{\sqrt{\frac{1}{N} \sum_{n=1}^N |X_n|^2}},$$

where the  $l$ -infinity norm and RMS values are computed along the specified dimension.

## Version History

Introduced in R2012a

## **References**

[1] *IEEE Standard on Transitions, Pulses, and Related Waveforms*, IEEE Standard 181, 2003.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

# dsp.PhaseExtractor

**Package:** dsp

Extract the unwrapped phase of a complex input

## Description

The `dsp.PhaseExtractor` System object extracts the unwrapped phase of a real or a complex input.

To extract the unwrapped phase of a signal input:

- 1 Create the `dsp.PhaseExtractor` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
phase = dsp.PhaseExtractor
phase = dsp.PhaseExtractor(Name, Value)
```

### Description

`phase = dsp.PhaseExtractor` returns a phase extractor System object that extracts the unwrapped phase of an input signal.

`phase = dsp.PhaseExtractor(Name, Value)` returns a phase extractor System object with the specified property name set to the specified value.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

### **TreatFramesIndependently** — Unwrap phase only within the frame

`false` (default) | `true`

Specify if the phase is to be unwrapped only within the frame, as a logical scalar.

When you set this property to:

- `false` -- The object returns the unwrapped phase while ignoring boundaries between input frames.
- `true` -- The object treats each frame of input data independently, and resets the initial cumulative unwrapped phase value to zero each time a new input frame is received.

### Usage

### Syntax

```
p = phase(input)
```

### Description

`p = phase(input)` extracts the unwrapped phase, `p`, of the input signal. Each column of the input signal is treated as a separate channel. The System object unwraps the phase of each channel of the input signal independently over time.

### Input Arguments

#### **input** — Data input

vector | matrix

Data input, specified as a vector or a matrix. This object supports variable-size input signals. That is, you can change the input frame size (number of rows) even after calling the algorithm. However, the number of channels (number of columns) must remain constant.

Data Types: `single` | `double`

### Output Arguments

#### **p** — Unwrapped phase

vector | matrix

Unwrapped phase of the input, returned as a vector or a matrix. The size and data type of the unwrapped phase output match the size and data type of the input signal.

Data Types: `single` | `double`

### Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

### Common to All System Objects

|                      |  |
|----------------------|--|
| <code>step</code>    | Run System object algorithm  |
| <code>release</code> | Release resources and allow changes to System object property values and input characteristics |
| <code>reset</code>   | Reset internal states of System object   |



## Examples

### Plot Unwrapped Phase Of a Sine Wave

Create a `dsp.SineWave` System object™. Specify that the object generates an exponential output with a complex exponent.

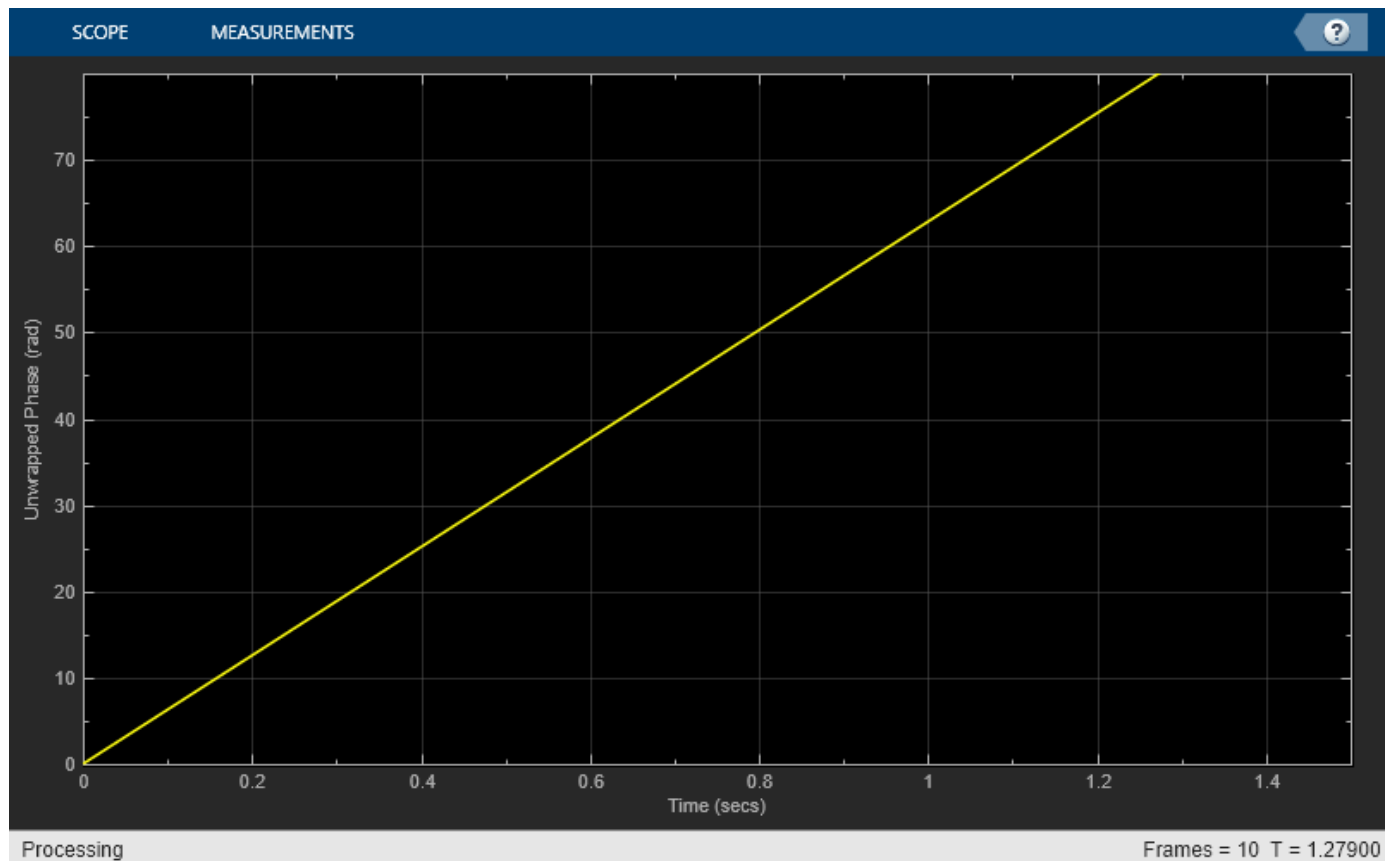
```
sine = dsp.SineWave('Frequency',10,...  
    'ComplexOutput',true,'SamplesPerFrame',128);
```

Create a `dsp.PhaseExtractor` System object?. Specify that the object ignores frame boundaries when returning the unwrapped phase.

```
phase = dsp.PhaseExtractor('TreatFramesIndependently',false);
```

Extract the unwrapped phase of a sine wave. Plot the phase versus time using a `timescope` System object.

```
timeplot = timescope('PlotType','Line','SampleRate',1000,...  
    'TimeSpanSource','Property','TimeSpan',1.5,'YLimits',[0 80],...  
    'ShowGrid',true,...  
    'YLabel','Unwrapped Phase (rad)');  
for ii = 1:10  
    sineOutput = sine();  
    phaseOutput = phase(sineOutput);  
    timeplot(phaseOutput)  
end
```



### Plot Phase Response of Third-Order IIR Filter

Create a `dsp.TransferFunctionEstimator` System object™.

```
tfe = dsp.TransferFunctionEstimator('FrequencyRange','centered');
```

Create a `dsp.PhaseExtractor` System object™. Specify that the object must treat each frame of data independently.

```
phase = dsp.PhaseExtractor('TreatFramesIndependently',true);
```

Create a `dsp.IIRFilter` System object™. Compute the transfer function of a third-order IIR filter. Use the `butter` function to generate coefficients for the filter.

```
[b,a] = butter(3,.3);
iir = dsp.IIRFilter('Numerator',b,'Denominator',a);
```

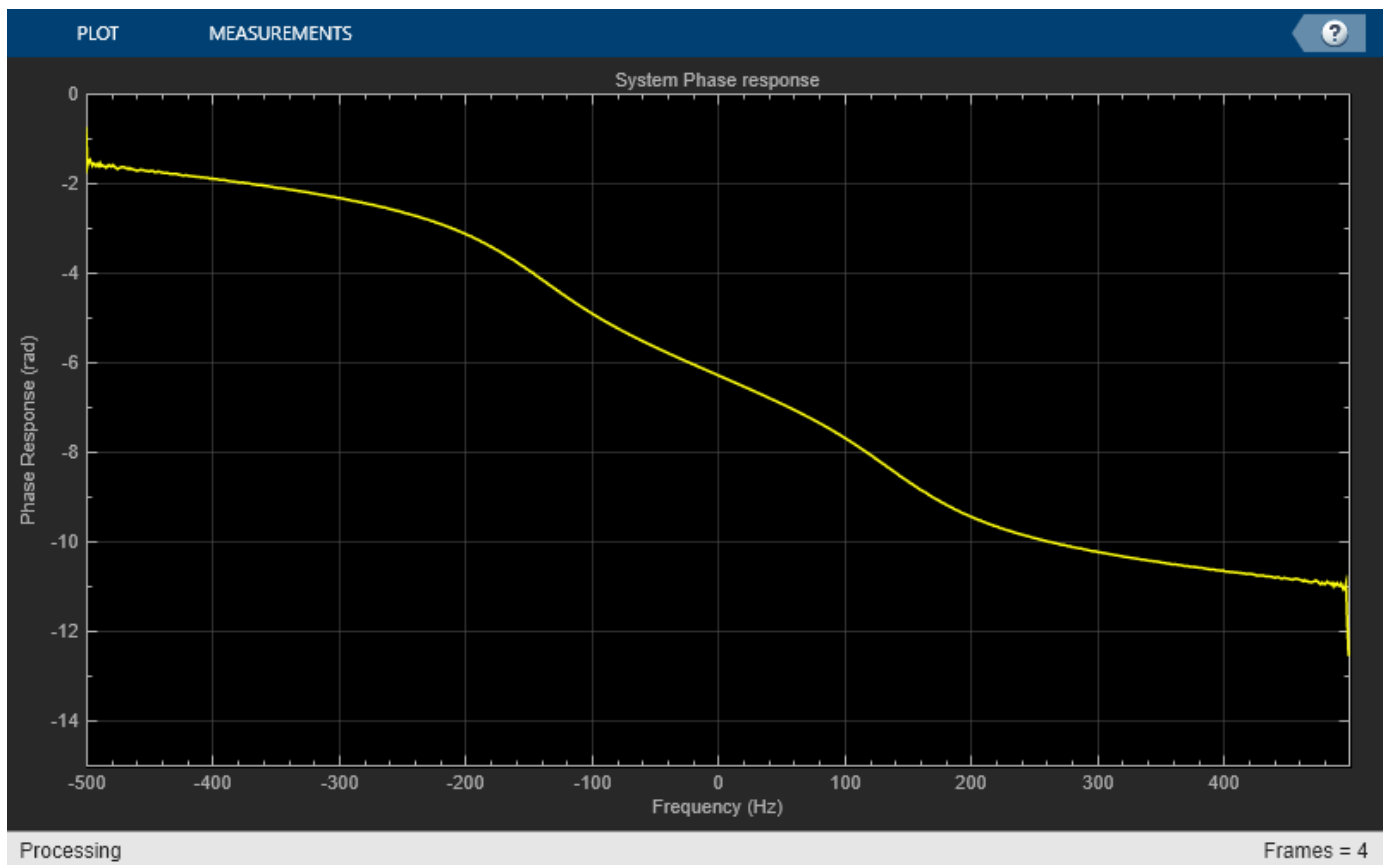
Extract the phase response of the transfer function. Plot using a `dsp.ArrayPlot` System object™.

```
sampleRate = 1e3;
phaseplot = dsp.ArrayPlot('PlotType','Line','XOffset',-sampleRate/2,...
    'YLimits',[-15 0],...
    'YLabel','Phase Response (rad)',...
    'XLabel','Frequency (Hz)',...
    'Title','System Phase response');
```

```

for ii = 1:100
    % Generate input
    input = 0.05*randn(1000,1);
    % Pass through IIR filter
    filterOutput = iir(input);
    % Estimate transfer function
    transferFunction = tfe(input,filterOutput);
    % Plot transfer function phase
    phaseOutput = phase(transferFunction);
    phaseplot(phaseOutput);
end

```



## Algorithms

Consider an input frame of length  $N$ :

$$\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{pmatrix}$$

The object acts on this frame and produces this output:

$$\begin{pmatrix} \Phi_1 \\ \Phi_2 \\ \vdots \\ \Phi_N \end{pmatrix}$$

where:

$$\Phi_i = \Phi_{i-1} + \text{angle}(x_{i-1}^* x_i)$$

Here,  $i$  runs from 1 to  $N$ . The `angle` function returns the phase angle in radians.

If the input signal consists of multiple frames:

- If you set `TreatFramesIndependently` to `true`, the object treats each frame independently. Therefore, in each frame, the object calculates the phase using the preceding formula where:
  - $\Phi_0$  is 0.
  - $x_0$  is 1.
- If you set `TreatFramesIndependently` to `false`, the object ignores boundaries between frames. Therefore, in each frame, the `step` method calculates the phase using the preceding formula where:
  - $\Phi_0$  is the last unwrapped phase from the previous frame.
  - $x_0$  is the last sample from the previous frame.

## Version History

Introduced in R2014b

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

## See Also

### Blocks

Phase Extractor

# dsp.PhaseUnwrapper

**Package:** dsp

Unwrap signal phase

## Description

The `dsp.PhaseUnwrapper` System object unwraps the phase of the input signal specified in radians.

To unwrap the signal phase input:

- 1 Create the `dsp.PhaseUnwrapper` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
phUnwrap = dsp.PhaseUnwrapper
phUnwrap = dsp.PhaseUnwrapper(Name, Value)
```

### Description

`phUnwrap = dsp.PhaseUnwrapper` returns a phase unwrapper System object that adds or subtracts appropriate multiples of  $2\pi$  to each input element to remove phase discontinuities (unwrap).

`phUnwrap = dsp.PhaseUnwrapper(Name, Value)` returns a phase unwrapper System object with each specified property set to the specified value.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

### **InterFrameUnwrap — Enable unwrapping of phase discontinuities between successive frames**

true (default) | false

Set this property to `false` to unwrap phase discontinuities only within the frame. Set this property to `true` to also unwrap phase discontinuities between successive frames.

**Tolerance — Jump size as true phase discontinuity**

3.1416 (default) | real scalar

Specify the jump size that the phase unwrapper recognizes as a true phase discontinuity. The default is set to  $\pi$  (rather than a smaller value) to avoid altering legitimate signal features. To increase the phase wrapper sensitivity, set the **Tolerance** property to a value slightly less than  $\pi$ .

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**Usage****Syntax**

```
output = phUnwrap(input)
```

**Description**

`output = phUnwrap(input)` unwraps the phase of the input signal. This is done by adding or subtracting appropriate multiples of  $2\pi$  to each input element to remove phase discontinuities (unwrap). Each column of the input signal is treated as a separate channel.

**Input Arguments****input — Data input**

vector | matrix

Data input, specified as a vector or a matrix. The phase of the input signal should be in radians.

Data Types: single | double

**Output Arguments****output — Unwrapped phase**

vector | matrix

Unwrapped phase of the input, returned as a vector or a matrix. The size and data type of the unwrapped phase output match the size and data type of the input signal.

Data Types: single | double

**Object Functions**

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

**Common to All System Objects**

|         |  |
|---------|--|
| step    | Run System object algorithm  |
| release | Release resources and allow changes to System object property values and input characteristics |
| reset   | Reset internal states of System object   |

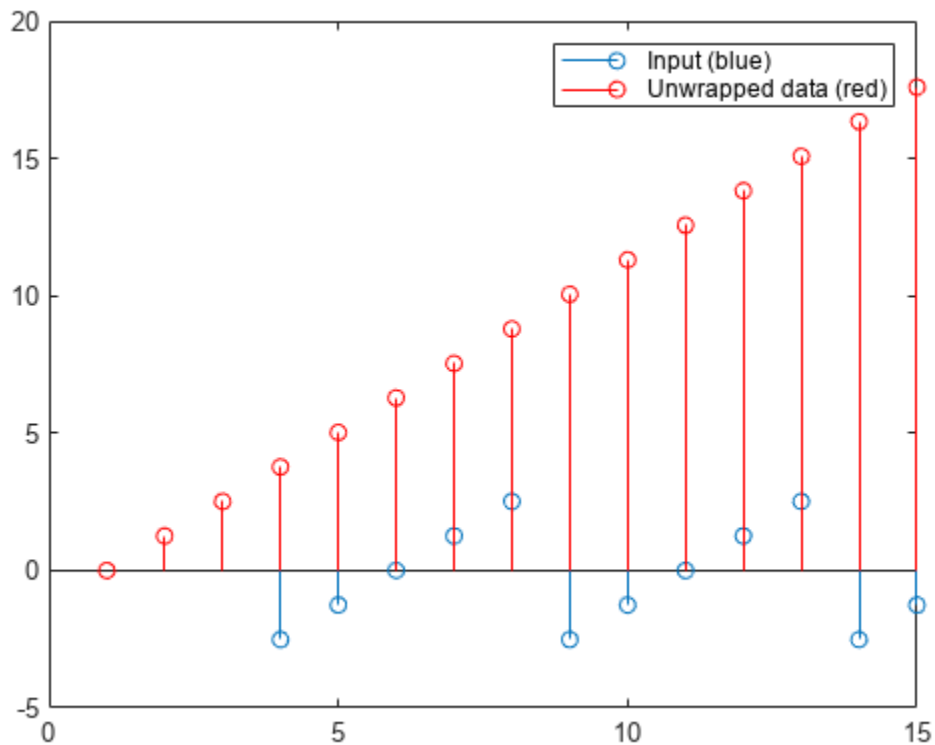
## Examples

### Unwrap input phase data

```
phUnwrap = dsp.PhaseUnwrapper;
p = [0 2/5 4/5 -4/5 -2/5 0 2/5 4/5 -4/5 -2/5 0 2/5 ...
     4/5 -4/5, -2/5]*pi;
y = phUnwrap(p');
figure,stem(p); hold
```

Current plot held

```
stem(y, 'r');
legend('Input (blue)', 'Unwrapped data (red)');
hold off;
```



## Algorithms

This object implements the algorithm, inputs, and outputs described on the Unwrap block reference page. The object properties correspond to the Simulink block parameters.

## Version History

Introduced in R2012a

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

### **See Also**

unwrap



# dsp.PulseMetrics

**Package:** dsp

(Removed) Pulse metrics of bilevel waveforms

---

**Note** `dsp.PulseMetrics` has been removed. Use functions from “Pulse and Transition Metrics” instead. Functions in this set include `dutycycle`, `midcross`, `pulseperiod`, `pulsesep`, and `pulsewidth` among others. For more information, see “Compatibility Considerations”.

---

## Description

The `dsp.PulseMetrics` object computes rise times, fall times, pulse widths, and cycle metrics including pulse period, pulse separation, and duty cycle for bilevel waveforms.

To obtain pulse metrics for a bilevel waveform:

- 1 Create the `dsp.PulseMetrics` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
pm = dsp.PulseMetrics
pm = dsp.PulseMetrics(Name,Value)
```

### Description

`pm = dsp.PulseMetrics` creates a pulse metrics System object, `pm`. The object computes the rise time, fall time, and width of a pulse. `dsp.PulseMetrics` also computes cycle metrics such as pulse separations, periods, and duty cycles. Because a pulse contains two transitions, the object contains a superset of the capability defined in `dsp.TransitionMetrics`.

`pm = dsp.PulseMetrics(Name,Value)` returns a `PulseMetrics` System object, `pm`, with each specified property set to the specified value.

### Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

**CycleOutputPort — Enable cycle metrics**`false` (default) | `true`

If `CycleOutputPort` is `true`, cycle metrics are reported for each pulse period.

**MaximumRecordLength — Maximum samples to preserve**`1000` (default) | positive integer

Maximum samples to preserve between calls to the algorithm. This property requires a positive integer that specifies the maximum number of samples to save between calls to the algorithm. When the number of samples to be saved exceeds this length, the oldest excess samples are discarded.

**Tunable:** Yes

**Dependencies**

This property applies when the `RunningMetrics` property is `true`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**PercentReferenceLevels — Percent reference levels**`[10 50 90]` (default) | three-element row vector

Lower-, middle-, and upper-percent reference levels. This property contains a three-element numeric row vector that contains the lower-, middle-, and upper-percent reference levels. These reference levels are used as an offset between the lower and upper states of the waveform when computing the duration of each transition.

Data Types: `double`

**PercentStateLevelTolerance — Tolerance of state level**`2` (default) | positive scalar

Tolerance of the state level (in percent). This property requires a scalar that specifies the maximum deviation from either the low or high state before it is considered to be outside that state. The tolerance is expressed as a percentage of the waveform amplitude.

Data Types: `double`

**Polarity — Polarity of pulse to extract**`'positive'` (default) | `'negative'`

Polarity of pulse to extract. This property specifies the type of pulse to extract by the polarity of the leading transition. Valid values for this property are `'positive'` or `'negative'`.

**PostshootOutputPort — Enable posttransition aberration metrics**`false` (default) | `true`

Enable posttransition aberration metrics. If this property is set to `true`, overshoot and undershoot metrics are reported for a region defined immediately after each transition. The posttransition aberration region is defined as the waveform interval that begins at the end of each transition and whose duration is the value of `PostshootSeekFactor` times the computed transition duration. If a complete subsequent transition is detected before the interval is over, the region is truncated at the start of the subsequent transition. The metrics are computed for each transition that has a complete posttransition aberration region.

**PostshootSeekFactor — Postshoot seek factor**

3 (default) | positive scalar

Corresponds to the duration of time to search for the overshoot and undershoot metrics immediately following each transition. The duration is expressed as a factor of the duration of the transition.

**Tunable:** Yes**Dependencies**

This property is enabled only when the `PostshootOutputPort` property is set to `true`.

Data Types: `double`**PreshootOutputPort — Enable pretransition aberration metrics**`false` (default) | `true`

Enable pretransition aberration metrics. If `PreshootOutputPort` is set to `true`, overshoot and undershoot metrics are reported for a region defined immediately before each transition. The pretransition aberration region is defined as the waveform interval that ends at the start of each transition and whose duration is `PreshootSeekFactor` times the computed transition duration.

**PreshootSeekFactor — Preshoot seek factor**

3 (default) | positive scalar

Corresponds to the duration of time to search for the overshoot and undershoot metrics immediately preceding each transition. The duration is expressed as a factor of the duration of the transition.

**Tunable:** Yes**Dependencies**

This property is enabled only when the `PreshootOutputPort` property is set to `true`.

Data Types: `double`**RunningMetrics — Enable metrics**`false` (default) | `true`

Enable metrics over all calls to the algorithm. If `RunningMetrics` is set to `false`, metrics are computed for each call to the algorithm independently. If `RunningMetrics` is set to `true`, metrics are computed across subsequent calls to the algorithm. If there are not enough samples to compute metrics associated with the last transition, posttransition aberration region, or settling seek duration in the current record, the object defers reporting all transition, aberration, and settling metrics associated with the last transition until a subsequent call to the algorithm is made with enough data to compute all enabled metrics for that transition.

**SampleRate — Sampling rate**

1 (default) | positive scalar

Sampling rate of uniformly-sampled signal. Specify the sample rate in hertz as a positive scalar. This property is used to construct the internal time values that correspond to the input sample values. Time values start with zero.

**Dependencies**

This property applies when the `TimeInputPort` property is set to `false`.

Data Types: `double`

### **SettlingOutputPort — Enable settling metrics**

`false` (default) | `true`

Enable settling metrics. If the `SettlingOutputPort` property is set to `true`, settling metrics are reported for each transition. The region used to compute the settling metrics starts at the midcrossing and lasts until the `SettlingSeekDuration` has elapsed. If an intervening transition occurs, or the signal has not settled within the `PercentStateLevelTolerance` of the final level, `NaN` is returned for each metric. If there are not enough samples after the last transition to complete the `SettlingSeekDuration`, no metrics are reported for the last transition. The metrics are reported for the transition the next time the object algorithm is called if the `RunningMetrics` property is set to `true`.

### **SettlingSeekDuration — Duration of time**

`0.02` (default) | positive scalar

Duration of time over which to search for settling. This property value is a scalar that specifies the amount of time to inspect from the mid-reference level crossing (in seconds). If the transition has not yet settled, or a subsequent complete transition is detected within this duration, the object reports `NaN` for all settling metrics.

**Tunable:** Yes

#### **Dependencies**

This property applies only when you set the `SettlingOutputPort` property to `true`.

Data Types: `double`

### **StateLevels — State levels**

`[0 2.3]` (default) | two-element row vector

Low- and high-state levels. This property is a two-element numeric row vector that contains the low- and high-state levels. These state levels correspond to the nominal logic low and high levels of the pulse waveform.

**Tunable:** Yes

Data Types: `double`

### **StateLevelsSource — State-level computation**

`'Property'` (default) | `'Auto'`

Auto or manual state-level computation. If `StateLevelsSource` is set to `'Auto'`, the first record sent to the object is sent to `dsp.StateLevels` with the default settings to determine the state levels of the incoming waveform. If this property is set to `'Property'`, the object uses the values the user specifies in the `StateLevels` property.

### **TimeInputPort — Specify sample instants**

`false` (default) | `true`

Add input to specify sample instants. Set `TimeInputPort` to `true` to enable an additional real input column vector to the object algorithm to specify the sample instants that correspond to the sample values. If this property is `false`, the sample instants are built internally. The sample instants start at zero and increment by the reciprocal of the `SampleRate` property for subsequent samples. The

sample instants continue to increment if the `RunningMetrics` property is set to `true` and no intervening calls to the `reset` or `release` methods are encountered.

### TransitionOutputPort — Enable transition metrics

`false` (default) | `true`

Enable transition metrics. If the `TransitionOutputPort` property is set to `true`, transition metrics are reported for the initial and final transitions of each pulse.

## Usage

### Syntax

```
pulse = pm(x)
[pulse,cycle] = pm(x)
[pulse,transition] = pm(x)
[pulse,preshoot] = pm(x)
[pulse,postshoot] = pm(x)
[pulse,settling] = pm(x)
[pulse,cycle,transition,preshoot,postshoot,settling] = pm(x)
[ ___ ] = pm(x,T)
```

### Description

`pulse = pm(x)` returns a structure array, `pulse`, whose fields contain real-valued column vectors. The number of rows of each field corresponds to the number of complete pulses found in the real-valued column vector input, `x`.

`[pulse,cycle] = pm(x)` returns a structure array, `cycle`, whose fields contain real-valued column vectors when you set the `CycleOutputPort` property to `true`. The number of rows of each field corresponds to the number of complete pulse periods found in the real-valued column vector input, `x`.

`[pulse,transition] = pm(x)` returns a structure array, `transition`, when you set the `TransitionOutputPort` property to `true`. The fields of `transition` contain real-valued matrices with two columns that correspond to the metrics of the first and second transitions. The number of rows corresponds to the number of pulses found in the input waveform.

`[pulse,preshoot] = pm(x)` returns a structure array, `preshoot`, when you set the `PreshootOutputPort` property to `true`. The fields of `preshoot` contain real-valued two-column matrices whose row length corresponds to the number of transitions found in the input waveform. The field names are identical to those of the `postshoot` structure array.

`[pulse,postshoot] = pm(x)` returns a structure, `postshoot`, when you set the `PostshootOutputPort` property to `true`. The fields of `postshoot` contain real-valued two-column matrices whose row length corresponds to the number of transitions found in the input waveform.

`[pulse,settling] = pm(x)` returns a structure array, `settling`, when you set the `SettlingOutputPort` property to `true`. The fields of `settling` correspond to the settling metrics for each transition. Each field is a column vector whose elements correspond to the individual settling durations, levels, and instants.

`[pulse,cycle,transition,preshoot,postshoot,settling] = pm(x)` which returns the `pulse`, `cycle`, `transition`, `preshoot`, `postshoot`, and `settling` structures when the

`CycleOutputPort`, `PreshootOutputPort`, `PostshootPort`, and `SettlingOutputPort` properties are `true`. You may enable or disable any combination of output ports. However, the output arguments are defined in the order shown here.

[ \_\_\_ ] = `pm`(`x`, `T`) calculates the above metrics with respect to a sampled signal, whose sample values, `x`, and sample instants, `T`, are real-valued column vectors of the same length. The additional input `T` applies only when you set the `TimeInputPort` property to `true`.

### Input Arguments

#### **x** — Input signal

column vector

Input signal, specified as a real-valued column vector.

Data Types: `double`

#### **T** — Sampling instants

column vector

Sampling instants, specified as a real-valued column vector. Set `TimeInputPort` to `true` to enable an additional real input column vector to the object algorithm to specify the sample instants that correspond to the sample values. If `TimeInputPort` is `false`, the sample instants are built internally. The sample instants start at zero and increment by the reciprocal of the `SampleRate` property for subsequent samples. The sample instants continue to increment if the `RunningMetrics` property is set to `true` and no intervening calls to the `reset` or `release` methods are encountered.

### Dependencies

This input is applicable when you set the `TimeInputPort` property to `true`.

Data Types: `double`

### Output Arguments

#### **pu**lse — Complete pulses

structure

Complete pulses, returned as a structure whose fields contain real-valued column vectors. The number of rows of each field corresponds to the number of complete pulses found in the real-valued column vector input, `x`. Each pulse starts with a transition of the polarity specified by the `Polarity` property and ends with a transition of the opposite polarity.

The `pu`lse output contains the following fields:

- `PositiveCross` — Instants where the positive-going transitions cross the mid-reference level of each pulse
- `NegativeCross` — Instants where the negative-going transitions cross the mid-reference level of each pulse
- `Width` — Absolute difference between `PositiveCross` and `NegativeCross` of each pulse
- `RiseTime` — Duration between the linearly interpolated instants when the positive-going (rising) transition of each pulse crosses the lower- and upper-reference levels
- `FallTime` — Duration between the linearly interpolated instants when the negative-going (falling) transition of each pulse crosses the upper- and lower-reference levels

Data Types: `struct`

### **cycle — Complete pulse periods**

structure

Complete pulse periods, returned as a structure whose fields contain real-valued column vectors. This structure can only be returned when you set the `CycleOutputPort` property to `true`. The number of rows of each field corresponds to the number of complete pulse periods found in the real-valued column vector input, `x`. You need at least three consecutive alternating polarity transitions that start and end with the same polarity as the value of the `Polarity` property if you want to compute cycle metrics. If the last transition found in the input `x` does not match the polarity of the `Polarity` property, the pulse separation, period, frequency, and duty cycle are not reported for the last pulse. If the `RunningMetrics` property is set to `true` when this occurs, all pulse, cycle, transition, preshoot, postshoot, and settling metrics associated with the last pulse are deferred until a subsequent call to the algorithm detects the next transition.

The `cycle` output contains the following fields:

- **Period** — Duration between the first transition of the current pulse and the first transition of the next pulse.
- **Frequency** — Reciprocal of the period.
- **Separation** — Durations between the mid-reference level crossings of the second transition of each pulse and the first transition of the next pulse.
- **Width** — Durations between the mid-reference level crossings of the first and second transitions of each pulse. This is equivalent to the width parameter of the `pulse` structure.
- **DutyCycle** — Ratio of the width to the period for each pulse.

Data Types: `struct`

### **transition — Transition metrics**

structure

Transition metrics, returned as a structure array. This structure can only be returned when you set the `TransitionOutputPort` property to `true`. The fields of `transition` contain real-valued matrices with two columns, which correspond to the metrics of the first and second transitions. The number of rows corresponds to the number of pulses found in the input waveform.

The `transition` output contains the following fields:

- **Duration** — Amount of time between the interpolated instants where the transition crosses the lower- and upper-reference levels
- **SlewRate** — Ratio of absolute difference between the upper and lower reference levels to the transition duration
- **MiddleCross** — Linearly interpolated instant in time where the transition first crosses the mid-reference level
- **LowerCross** — Linearly interpolated instant where the signal crosses the lower-reference level
- **UpperCross** — Linearly interpolated instant where the signal crosses the upper-reference level

Data Types: `struct`

### **preshoot — Preshoot metrics**

structure

Preshoot metrics, returned as a structure array. This structure can only be returned when you set the `PreshootOutputPort` property to `true`. The fields of `preshoot` contain real-valued two-column matrices whose row length corresponds to the number of transitions found in the input waveform.

The `preshoot` output contains the following fields:

- `Overshoot` — Overshoot of the region of interest expressed as a percentage of the waveform amplitude
- `Undershoot` — Undershoot of the region of interest expressed as a percentage of the waveform amplitude
- `OvershootLevel` — Level of the overshoot
- `UndershootLevel` — Level of the undershoot
- `OvershootInstant` — Instant that corresponds to the overshoot
- `UndershootInstant` — Instant that corresponds to the undershoot

Data Types: `struct`

### **postshoot — Postshoot metrics**

structure

Postshoot metrics, returned as a structure array. This structure can only be returned when you set the `PostshootOutputPort` property to `true`. The fields of `postshoot` contain real-valued two-column matrices whose row length corresponds to the number of transitions found in the input waveform.

The `postshoot` output contains the following fields:

- `Overshoot` — Overshoot of the region of interest expressed as a percentage of the waveform amplitude
- `Undershoot` — Undershoot of the region of interest expressed as a percentage of the waveform amplitude
- `OvershootLevel` — Level of the overshoot
- `UndershootLevel` — Level of the undershoot
- `OvershootInstant` — Instant that corresponds to the overshoot
- `UndershootInstant` — Instant that corresponds to the undershoot

Data Types: `struct`

### **settling — Settling metrics**

structure

Settling metrics for each transition, returned as a structure array. This structure can only be returned when you set the `SettlingOutputPort` property to `true`. The fields of `settling` correspond to the settling metrics for each transition. Each field is a column vector whose elements correspond to the individual settling durations, levels, and instants.

The `settling` output contains the following fields:

- `Duration` — Amount of time from when the signal crosses the mid-reference level to the time where the signal enters and remains within the specified `PercentStateLevelTolerance` of the waveform amplitude over the specified settling seek duration



- **Instant** — Instant in time where the signal enters and remains within the specified tolerance
- **Level** — Level of the waveform where it enters and remains within the specified tolerance

Data Types: `struct`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Specific to `dsp.PulseMetrics`

`plot` (To be removed) Plot pulse signal and metrics

## Common to All System Objects

`step` Run System object algorithm

`release` Release resources and allow changes to System object property values and input characteristics

`reset` Reset internal states of System object

## Examples

### Width, Period, and Duty Cycle

Determine the width, period, and duty cycle of a 5 V pulse sampled at 4 MHz.

```
load('pulseex.mat','x','t');
```

Construct the `dsp.PulseMetrics` object. Set the `TimeInputPort` property to `true` to specify the sampling instants as an input. Set the `CycleOutputPort` property to `true` to obtain metrics for each pulse. Because the input is a 5 V pulse, set the `StateLevels` property to `[0 5]`.

```
pm = dsp.PulseMetrics('TimeInputPort',true, ...
                    'CycleOutputPort', true, ...
                    'StateLevels',[0 5])
```

```
pm =
  dsp.PulseMetrics with properties:
        Polarity: 'Positive'
  StateLevelsSource: 'Property'
        StateLevels: [0 5]
  PercentStateLevelTolerance: 2
  PercentReferenceLevels: [10 50 90]
        RunningMetrics: false
        TimeInputPort: true
        CycleOutputPort: true
  TransitionOutputPort: false
  PreshootOutputPort: false
  PostshootOutputPort: false
  SettlingOutputPort: false
```

Call the object to compute the cycle metrics and plot the result.

```
[pulse,cycle] = pm(x,t);  
plot(pm)  
text(t(2),-0.5,['Duty Cycle: ',num2str(cycle.DutyCycle)]);
```

## Version History

### Introduced in R2012a

#### **R2021b: dsp.PulseMetrics System object will be removed**

*Warns starting in R2021b*

`dsp.PulseMetrics` System object will be removed in a future release. Use functions from “Pulse and Transition Metrics” instead. Functions in this set include `dutycycle`, `midcross`, `pulseperiod`, `pulsesep`, and `pulsewidth` among others.

#### **R2023a: dsp.PulseMetrics System object has been removed**

*Errors starting in R2023a*

The `dsp.PulseMetrics` System object has been removed. Use functions from “Pulse and Transition Metrics” instead. This set includes functions such as `dutycycle`, `midcross`, `pulseperiod`, `pulsesep`, and `pulsewidth` among others.

### Update Code

This table shows how to update existing code to use these functions.

| Discouraged Usage   | Recommended Replacement  |
|---|--|
| <pre>load('clockx.mat','x','t'); pm = dsp.PulseMetrics(SampleRate=4e6,...     TransitionOutputPort=true,...     StateLevelsSource='Auto',...     CycleOutputPort=true); [pulse, cycle, transition] = pm(x)  pulse =      struct with fields:          PositiveCross: [4x1 double]         NegativeCross: [4x1 double]         Width: [4x1 double]         RiseTime: [4x1 double]         FallTime: [4x1 double]  cycle =      struct with fields:          Period: [4x1 double]         Frequency: [4x1 double]         Separation: [4x1 double]         Width: [4x1 double]         DutyCycle: [4x1 double]  transition =      struct with fields:          Duration: [4x2 double]         Polarity: [4x2 double]         SlewRate: [4x2 double]         MiddleCross: [4x2 double]         LowerCross: [4x2 double]         UpperCross: [4x2 double]</pre> | <p>MATLAB code using the replacement functions is shown below.</p>   |
| <p><b>Duty Cycle</b></p> <p>Duty cycle is the ratio of the width to the period.</p> <pre>dutycycleObj = cycle.Width./cycle.Period  dutycycleObj =      0.4999     0.4999     0.4994     0.4997</pre>  | <p><b>Duty Cycle</b></p> <pre>dutycyclefn = dutycycle(x,t)  dutycyclefn =      0.4999     0.4999     0.4994     0.4997</pre> |

| Discouraged Usage  | Recommended Replacement  |
|--|--|
| <p><b>Middle Cross</b></p> <pre>middlecross0bj = transition.MiddleCross middlecross0bj =     1.0e-04 *         0.0237    0.0487         0.0737    0.0987         0.1237    0.1487         0.1737    0.1987</pre> | <p><b>Middle Cross</b></p> <pre>middlecrossfn = midcross(x,t) middlecrossfn =     1.0e-04 *         0.0237         0.0487         0.0737         0.0987         0.1237         0.1487         0.1737         0.1987         0.2237</pre> |
| <p><b>Pulse Period</b></p> <pre>period0bj = cycle.Period period0bj =     1.0e-05 *         0.4999         0.5002         0.5000         0.5000</pre>   | <p><b>Pulse Period</b></p> <pre>periodfn = pulseperiod(x,t) periodfn =     1.0e-05 *         0.4999         0.5002         0.5000         0.5000</pre>   |
| <p><b>Pulse Separation</b></p> <pre>sep0bj = cycle.Separation sep0bj =     1.0e-05 *         0.2500         0.2502         0.2503         0.2501</pre>   | <p><b>Pulse Separation</b></p> <pre>sepfm = pulsesep(x,t) sepfm =     1.0e-05 *         0.2500         0.2502         0.2503         0.2501</pre>  |
| <p><b>Pulse Width</b></p> <pre>pulse.Width pulse.Width ans =     1.0e-05 *         0.2499         0.2501         0.2497         0.2499</pre>   | <p><b>Pulse Width</b></p> <pre>widths = pulsewidth(x,t) widths =     1.0e-05 *         0.2499         0.2501         0.2497         0.2499</pre>   |

| Discouraged Usage   | Recommended Replacement  |
|---|--|
| <p><b>Plot the pulse</b></p> <p><code>plot(pm)</code></p> | <p><b>Plot the pulse</b></p> <p>To plot the individual pulse metrics, call the metrics functions without an output argument.</p> <p><code>dutycycle(x,t)</code></p> <p><code>midcross(x,t)</code></p> <p><code>pulseperiod(x,t)</code></p> <p><code>pulsesep(x,t)</code></p> <p><code>pulsewidth(x,t)</code></p> |

## References

[1] *IEEE Standard on Transitions, Pulses, and Related Waveforms*, IEEE Standard 181, 2003.

## See Also

### Functions

`dutycycle` | `midcross` | `pulseperiod` | `pulsesep` | `pulsewidth`

### Topics

“Pulse and Transition Metrics”

## **dsp.RLSFilter**

**Package:** dsp

Compute output, error and coefficients using recursive least squares (RLS) algorithm

### **Description**

The `dsp.RLSFilter` System object filters each channel of the input using RLS filter implementations.

To filter each channel of the input:

- 1 Create the `dsp.RLSFilter` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

### **Creation**

#### **Syntax**

```
rlsFilt = dsp.RLSFilter  
rlsFilt = dsp.RLSFilter(len)  
rlsFilt = dsp.RLSFilter(Name,Value)
```

#### **Description**

`rlsFilt = dsp.RLSFilter` returns an adaptive RLS filter System object, `rlsFilt`. This System object computes the filtered output, filter error, and the filter weights for a given input and desired signal using the RLS algorithm.

`rlsFilt = dsp.RLSFilter(len)` returns an RLS filter System object, `rlsFilt`. This System object has the `Length` property set to `len`.

`rlsFilt = dsp.RLSFilter(Name,Value)` returns an RLS filter System object with each specified property set to the specified value. Enclose each property name in single quotes. Unspecified properties have default values.

#### **Properties**

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

**Method — Method to calculate filter coefficients**

Conventional RLS (default) | Householder RLS | Sliding-window RLS | Householder sliding-window RLS | QR decomposition

You can specify the method used to calculate filter coefficients as Conventional RLS [1] [2], Householder RLS [3] [4], Sliding-window RLS [5][1][2], Householder sliding-window RLS [4], or QR decomposition [1] [2]. This property is nontunable.

**Length — Length of filter coefficients vector**

32 (default) | positive integer

Specify the length of the RLS filter coefficients vector as a scalar positive integer value. This property is nontunable.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**SlidingWindowBlockLength — Width of sliding window**

48 (default) | positive integer

Specify the width of the sliding window as a scalar positive integer value greater than or equal to the Length property value. This property is nontunable.

**Dependencies**

This property applies only when the Method property is set to Sliding-window RLS or Householder sliding-window RLS.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**ForgettingFactor — RLS forgetting factor**

1 (default) | positive scalar

Specify the RLS forgetting factor as a scalar positive numeric value less than or equal to 1. Setting this property value to 1 denotes infinite memory, while adapting to find the new filter.

**Tunable:** Yes

Data Types: single | double

**InitialCoefficients — Initial coefficients of filter**

0 (default) | scalar | vector

Specify the initial values of the FIR adaptive filter coefficients as a scalar or a vector of length equal to the Length property value.

**Tunable:** Yes

Data Types: single | double

**InitialInverseCovariance — Initial inverse covariance**

1000 (default) | scalar | square matrix

Specify the initial values of the inverse covariance matrix of the input signal. This property must be either a scalar or a square matrix, with each dimension equal to the Length property value. If you set a scalar value, the InverseCovariance property is initialized to a diagonal matrix with diagonal elements equal to that scalar value.

**Tunable:** Yes

**Dependencies**

This property applies only when the `Method` property is set to `Conventional RLS` or `Sliding-window RLS`.

Data Types: `single` | `double`

**InitialSquareRootInverseCovariance — Initial square root inverse covariance**

`sqrt(1000)` (default) | `scalar` | `square matrix`

Specify the initial values of the square root inverse covariance matrix of the input signal. This property must be either a scalar or a square matrix with each dimension equal to the `Length` property value. If you set a scalar value, the `SquareRootInverseCovariance` property is initialized to a diagonal matrix with diagonal elements equal to that scalar value.

**Tunable:** Yes

**Dependencies**

This property applies only when the `Method` property is set to `Householder RLS` or `Householder sliding-window RLS`.

Data Types: `single` | `double`

**InitialSquareRootCovariance — Initial square root covariance**

`sqrt(1/1000)` (default) | `scalar` | `square matrix`

Specify the initial values of the square root covariance matrix of the input signal. This property must be either a scalar or a square matrix with each dimension equal to the `Length` property value. If you set a scalar value, the `SquareRootCovariance` property is initialized to a diagonal matrix with diagonal elements equal to the scalar value.

**Tunable:** Yes

**Dependencies**

This property applies only when the `Method` property is set to `QR-decomposition RLS`.

Data Types: `single` | `double`

**LockCoefficients — Lock coefficient updates**

`false` (default) | `true`

Specify whether the filter coefficient values should be locked. When you set this property to `true`, the filter coefficients are not updated and their values remain the same. The default value is `false` (filter coefficients continuously updated).

**Tunable:** Yes

**Usage****Syntax**

```
y = rlsFilt(x,d)  
[y,e] = rlsFilt(x,d)
```



## Description

$y = \text{rlsFilt}(x, d)$  recursively adapts the reference input,  $x$ , to match the desired signal,  $d$ , using the System object, `rlsFilt`. The desired signal,  $d$ , is the signal desired plus some undesired noise.

$[y, e] = \text{rlsFilt}(x, d)$  shows the output of the RLS filter along with the error,  $e$ , between the reference input and the desired signal. The filter adapts its coefficients until the error  $e$  is minimized. You can access these coefficients by accessing the `Coefficients` property of the object. This can be done only after calling the object. For example, to access the optimized coefficients of the `rlsFilt` filter, call `rlsFilt.Coefficients` after you pass the input and desired signal to the object.

## Input Arguments

### **x** — Data input

scalar | column vector

The signal to be filtered by the RLS filter. The input,  $x$ , and the desired signal,  $d$ , must have the same size and data type.

The input can be a variable-size signal. You can change the number of elements in the column vector even when the object is locked. The System object locks when you call the object to run its algorithm.

Data Types: `single` | `double`  
Complex Number Support: Yes

### **d** — Desired signal

scalar | column vector

The RLS filter adapts its coefficients to minimize the error,  $e$ , and converge the input signal  $x$  to the desired signal  $d$  as closely as possible.

The input,  $x$ , and the desired signal,  $d$ , must have the same size and data type.

The desired signal,  $d$ , can be a variable-size signal. You can change the number of elements in the column vector even when the object is locked. The System object locks when you call the object to run its algorithm.

Data Types: `single` | `double`  
Complex Number Support: Yes

## Output Arguments

### **y** — Filtered output

scalar | column vector

Filtered output, returned as a scalar or a column vector. The object adapts its filter coefficients to converge the input signal  $x$  to match the desired signal  $d$ . The filter outputs the converged signal.

Data Types: `single` | `double`

### **e** — Difference between output and desired signal

scalar | column vector

Difference between the output signal  $y$  and the desired signal  $d$ , returned as a scalar or a column vector. The objective of the RLS filter is to minimize this error. The object adapts its coefficients to

converge toward optimal filter coefficients that produce an output signal that matches closely with the desired signal. For more details on how  $e$  is computed, see “Algorithms” on page 4-964. To access the RLS filter coefficients, call `rlsFilt.Coefficients` after you pass the input and desired signal to the object.

Data Types: `single` | `double`  
Complex Number Support: Yes

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Specific to `dsp.RLSFilter`

`mseSim` Estimated mean squared error for adaptive filters

## Common to All System Objects

`step` Run System object algorithm  
`release` Release resources and allow changes to System object property values and input characteristics  
`reset` Reset internal states of System object

## Examples

### System Identification of FIR Filter Using RLS Filter

Use a recursive least squares (RLS) filter to identify an unknown system modeled with a lowpass FIR filter. Compare the frequency responses of the unknown and estimated systems.

#### Initialization

Create a `dsp.FIRFilter` object that represents the system to be identified. Pass the signal `x` to the FIR filter. The output of the unknown system is the desired signal `d`, which is the sum of the output of the unknown system (FIR filter) and an additive noise signal `n`.

```
filt = dsp.FIRFilter('Numerator',fir1(10,.25));  
x = randn(1000,1);  
n = 0.01*randn(1000,1);  
d = filt(x) + n;
```

#### Adaptive Filter

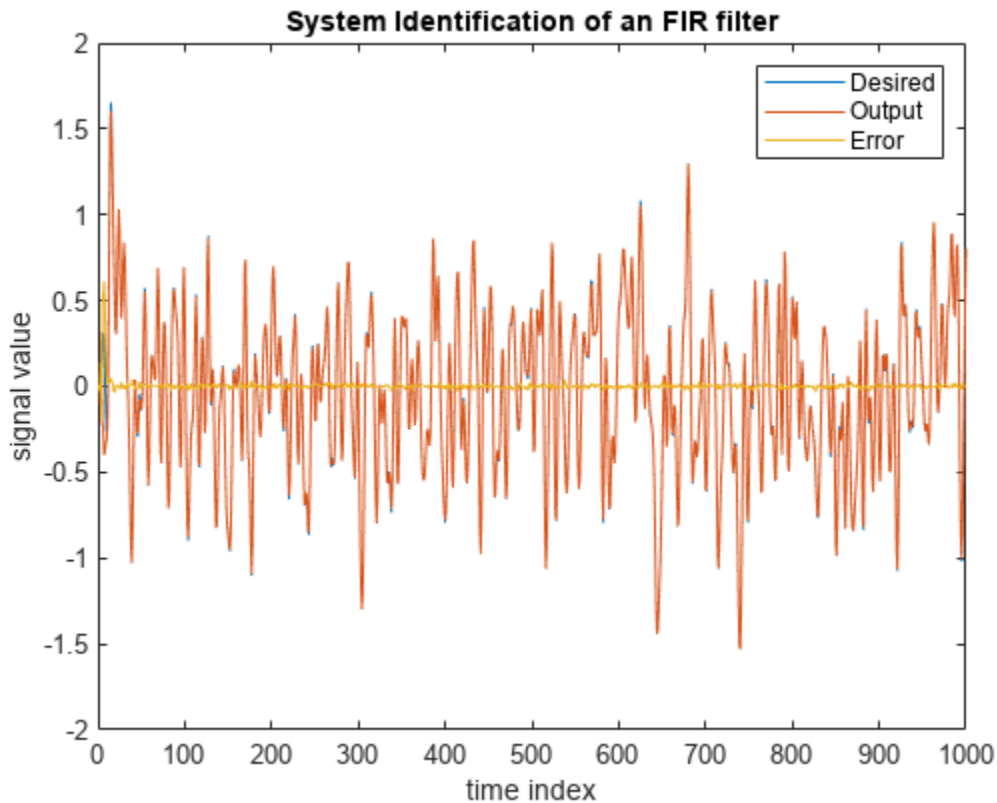
Create a `dsp.RLSFilter` object to create an RLS filter. Set the length of the filter to 11 taps and the forgetting factor to 0.98. Pass the primary input signal `x` and the desired signal `d` to the RLS filter. The output `y` of the adaptive filter is the signal converged to the desired signal `d` thereby minimizing the error `e` between the two signals.

```
rls = dsp.RLSFilter(11, 'ForgettingFactor', 0.98);  
[y,e] = rls(x,d);  
w = rls.Coefficients;
```

## Plot the results

The output signal matches the desired signal, making the error between the two close to zero.

```
plot(1:1000, [d,y,e]);
title('System Identification of an FIR filter');
legend('Desired', 'Output', 'Error');
xlabel('time index');
ylabel('signal value');
```

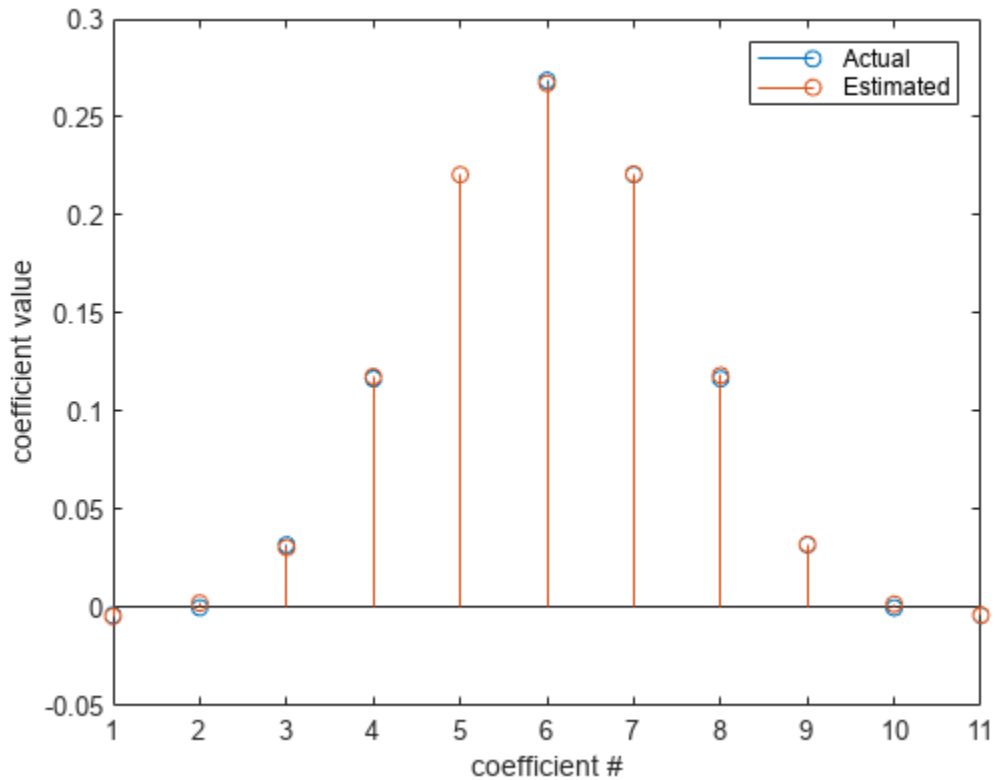


## Compare the weights

The weights vector  $w$  represents the coefficients of the RLS filter that is adapted to resemble the unknown system (FIR filter). To confirm the convergence, compare the numerator of the FIR filter and the estimated weights of the RLS filter.

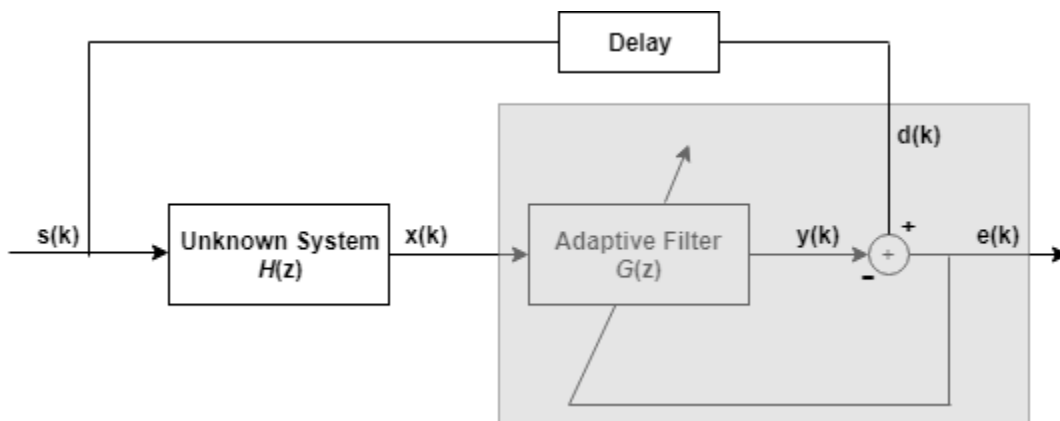
The estimated filter weights closely match the actual filter weights, confirming the results seen in the previous signal plot.

```
stem([filt.Numerator; w].');
legend('Actual', 'Estimated');
xlabel('coefficient #');
ylabel('coefficient value');
```



### Inverse System Identification Using RLS Algorithm

This example demonstrates the RLS adaptive algorithm using the inverse system identification model shown here.



Cascading the adaptive filter with an unknown filter causes the adaptive filter to converge to a solution that is the inverse of the unknown system.

If the transfer function of the unknown system and the adaptive filter are  $H(z)$  and  $G(z)$ , respectively, the error measured between the desired signal and the signal from the cascaded system reaches its

minimum when  $G(z) \times H(z) = 1$ . For this relation to be true,  $G(z)$  must equal  $1/H(z)$ , the inverse of the transfer function of the unknown system.

To demonstrate that this is true, create a signal  $s$  to input to the cascaded filter pair.

```
s = randn(3000,1);
```

In the cascaded filters case, the unknown filter results in a delay in the signal arriving at the summation point after both filters. To prevent the adaptive filter from trying to adapt to a signal it has not yet seen (equivalent to predicting the future), delay the desired signal by 12 samples, which is the order of the unknown system.

Generally, you do not know the order of the system you are trying to identify. In that case, delay the desired signal by number of samples equal to half the order of the adaptive filter. Delaying the input requires prepending 12 zero-value samples to the input  $s$ .

```
delay = zeros(12,1);
d = [delay; s(1:2988)]; % Concatenate the delay and the signal.
```

You have to keep the desired signal vector  $d$  the same length as  $x$ , so adjust the signal element count to allow for the delay samples.

Although not generally the case, for this example you know the order of the unknown filter, so add a delay equal to the order of the unknown filter.

For the unknown system, use a lowpass, 12th-order FIR filter.

```
filt = dsp.FIRFilter;
filt.Numerator = fir1(12,0.55,'low');
```

Filtering  $s$  provides the input data signal for the adaptive algorithm function.

```
x = filt(s);
```

To use the RLS algorithm, create a `dsp.RLSFilter` object and set its `Length`, `ForgettingFactor`, and `InitialInverseCovariance` properties.

For more information about the input conditions to prepare the RLS algorithm object, refer to `dsp.RLSFilter`.

```
p0 = 2 * eye(13);
lambda = 0.99;
rls = dsp.RLSFilter(13,'ForgettingFactor',lambda,...
    'InitialInverseCovariance',p0);
```

This example seeks to develop an inverse solution, you need to be careful about which signal carries the data and which is the desired signal.

Earlier examples of adaptive filters use the filtered noise as the desired signal. In this case, the filtered noise ( $x$ ) carries the unknown system's information. With Gaussian distribution and variance of 1, the unfiltered noise  $d$  is the desired signal. The code to run this adaptive filter is:

```
[y,e] = rls(x,d);
```

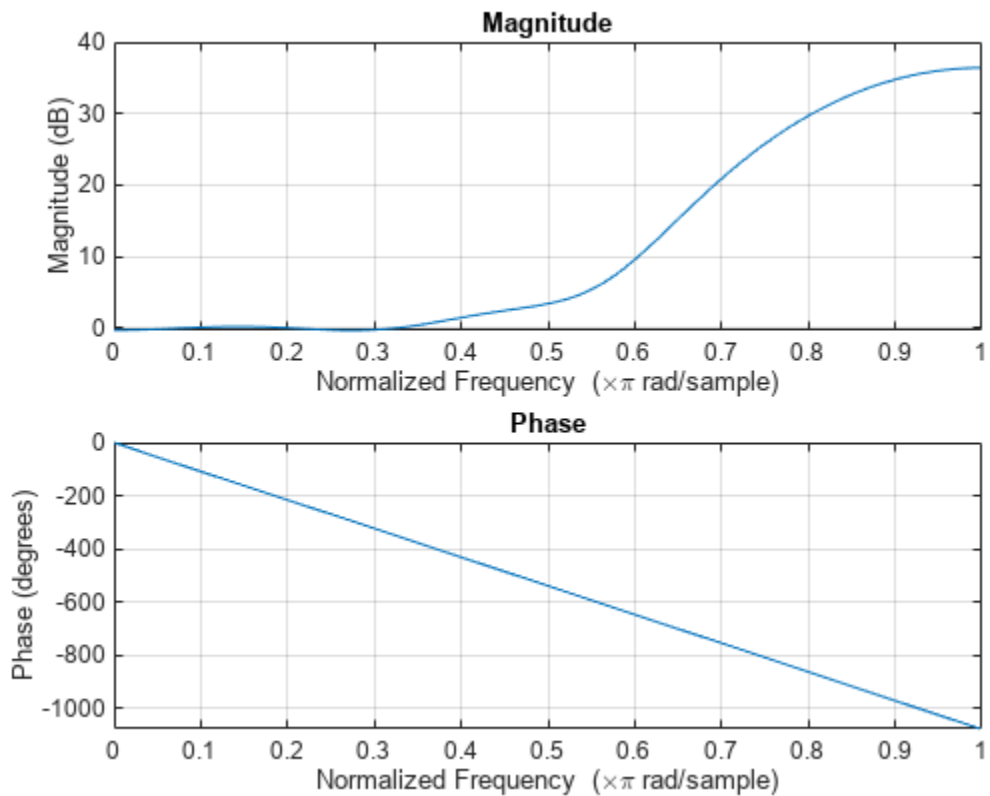
where  $y$  returns the filtered output and  $e$  contains the error signal as the filter adapts to find the inverse of the unknown system.

Obtain the estimated coefficients of the RLS filter.

```
b = rls.Coefficients;
```

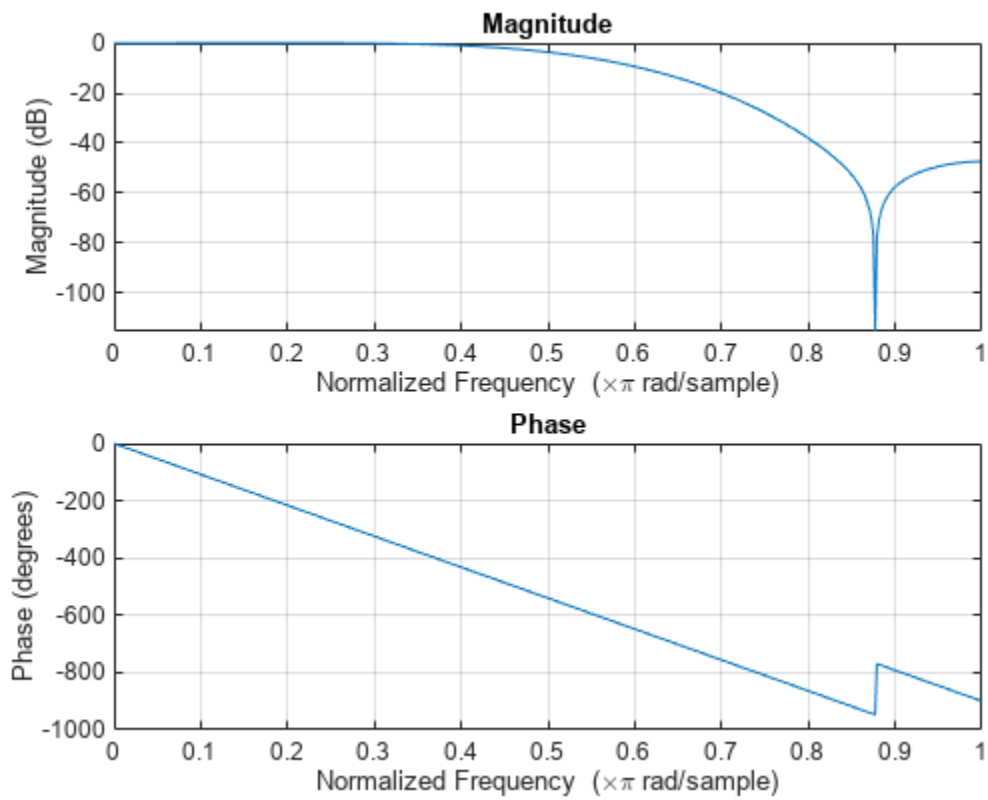
View the frequency response of the adapted RLS filter (inverse system,  $G(z)$ ) using `freqz`. The inverse system looks like a highpass filter with linear phase.

```
freqz(b,1)
```



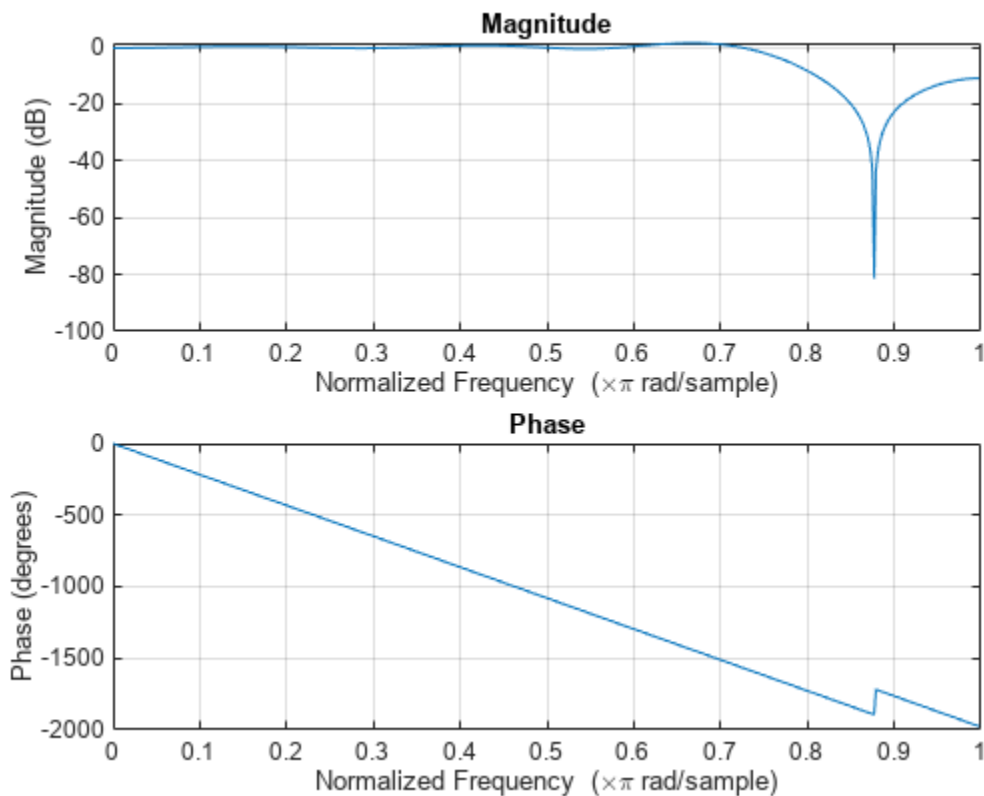
View the frequency response of the unknown system,  $H(z)$ . The response is that of a lowpass filter with a cutoff frequency of 0.55.

```
freqz(filt.Numerator,1)
```



The result of the cascade of the unknown system and the adapted filter is a compensated system with an extended cutoff frequency of 0.8.

```
overallCoeffs = conv(filt.Numerator,b);  
freqz(overallCoeffs,1)
```



### Cancel Noise Using RLS Filter

Cancel additive noise  $n$  added to an unknown system using an RLS filter. The RLS filter adapts its coefficients until its transfer function matches the transfer function of the unknown system as closely as possible. The difference between the output of the adaptive filter and the output of the unknown system is the error signal  $e$ , which represents the additive white noise. Minimizing this error signal is the objective of the adaptive filter.

#### Initialization

Create a `dsp.FIRFilter` System object™ to represent the unknown system. Create a `dsp.RLSFilter` object and set the length to 11 taps. Set the method to 'Householder RLS'. Create a sine wave to represent the noise added to the unknown system. View the signals in a time scope.

```
FrameSize = 100;
NIter = 10;

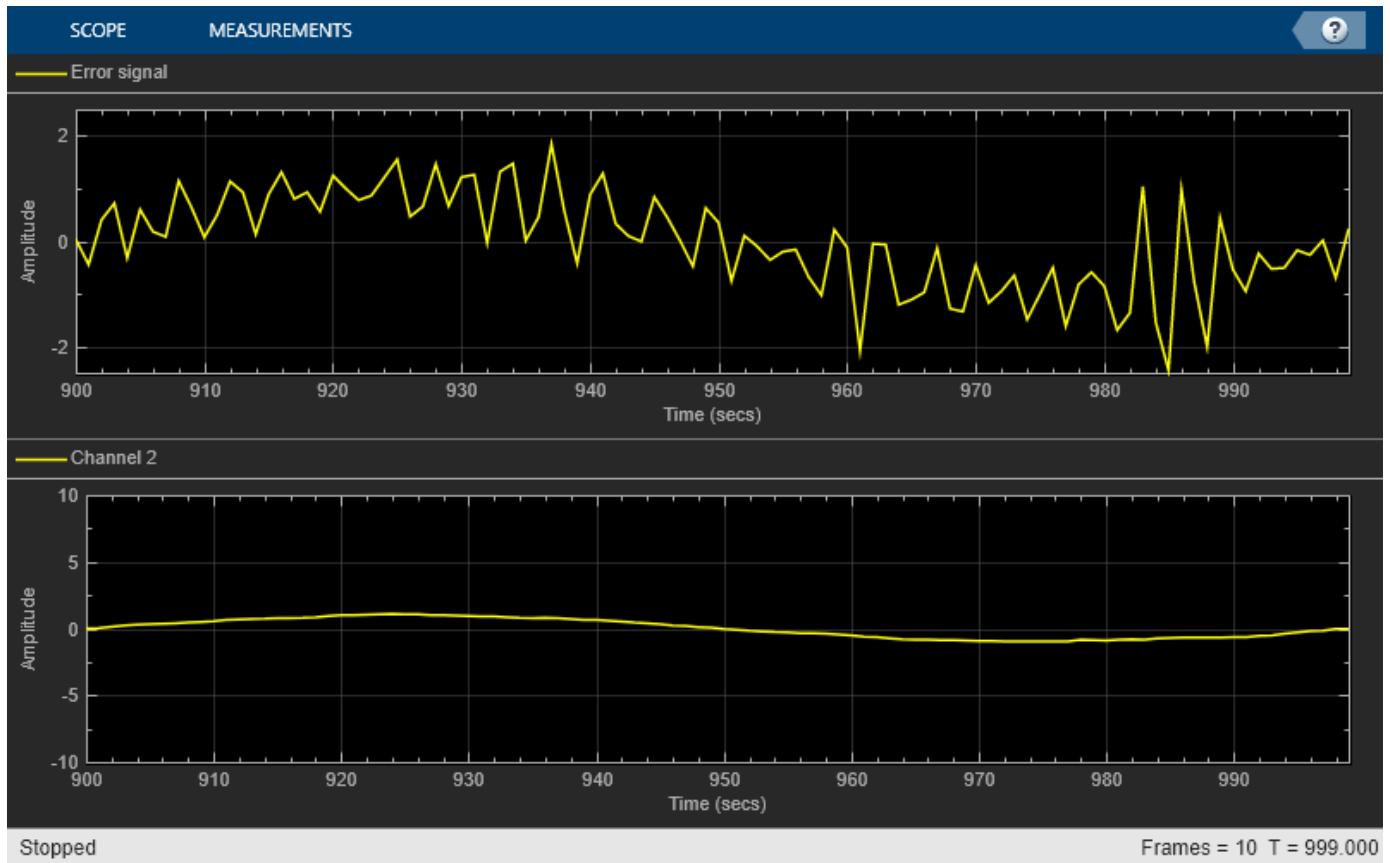
rls = dsp.RLSFilter('Length',11,...
    'Method','Householder RLS');
filt = dsp.FIRFilter('Numerator',...
    fir1(10,[.5,.75]));

sinewave = dsp.SineWave('Frequency',0.01,...
```



```
    'SampleRate',1,...
    'SamplesPerFrame',FrameSize);
scope = timescope('LayoutDimensions',[2 1],...
    'NumInputPorts',2, ...
    'TimeUnits','Seconds',...
    'YLimits',[-2.5 2.5], ...
    'BufferLength',2*FrameSize*NIter,...
    'ActiveDisplay',1,...
    'ShowLegend',true,...
    'ChannelNames',{'Noisy signal'},...
    'ActiveDisplay',2,...
    'ShowLegend',true,...
    'ChannelNames',{'Error signal'});

for k = 1:NIter
    x = randn(FrameSize,1);
    d = filt(x) + sinewave();
    [y,e] = rls(x,d);
    w = rls.Coefficients;
    scope(d,e)
end
release(scope)
```



## Algorithms

The `dsp.RLSFilter` System object, when `Conventional` RLS is selected, recursively computes the least squares estimate (RLS) of the FIR filter weights. The System object estimates the filter weights or coefficients, needed to convert the input signal into the desired signal. The input signal can be a scalar or a column vector. The desired signal must have the same data type, complexity, and dimensions as the input signal. The corresponding RLS filter is expressed in matrix form as  $\mathbf{P}(n)$  :

$$\mathbf{k}(n) = \frac{\lambda^{-1}\mathbf{P}(n-1)\mathbf{u}(n)}{1 + \lambda^{-1}\mathbf{u}^H(n)\mathbf{P}(n-1)\mathbf{u}(n)}$$

$$y(n) = \mathbf{w}^T(n-1)\mathbf{u}(n)$$

$$e(n) = d(n) - y(n)$$

$$\mathbf{w}(n) = \mathbf{w}(n-1) + \mathbf{k}^*(n)e(n)$$

$$\mathbf{P}(n) = \lambda^{-1}\mathbf{P}(n-1) - \lambda^{-1}\mathbf{k}(n)\mathbf{u}^H(n)\mathbf{P}(n-1)$$

where  $\lambda^{-1}$  denotes the reciprocal of the exponential weighting factor. The variables are as follows:

| Variable        | Description                                      |
|-----------------|--|
| $n$             | The current time index                           |
| $\mathbf{u}(n)$ | The vector of buffered input samples at step $n$ |

| Variable          | Description   |
|-------------------|---|
| $\mathbf{P}(n)$   | The conjugate of the inverse correlation matrix at step $n$ |
| $\mathbf{k}(n)$   | The gain vector at step $n$                                 |
| $\mathbf{k}^*(n)$ | Complex conjugate of $\mathbf{k}$                           |
| $\mathbf{w}(n)$   | The vector of filter tap estimates at step $n$              |
| $y(n)$            | The filtered output at step $n$                             |
| $e(n)$            | The estimation error at step $n$                            |
| $d(n)$            | The desired response at step $n$                            |
| $\lambda$         | The forgetting factor                                       |

$\mathbf{u}$ ,  $\mathbf{w}$ , and  $\mathbf{k}$  are all column vectors.

## Version History

Introduced in R2013a

## References

- [1] M Hayes, *Statistical Digital Signal Processing and Modeling*, New York: Wiley, 1996.
- [2] S. Haykin, *Adaptive Filter Theory*, 4th Edition, Upper Saddle River, NJ: Prentice Hall, 2002 .
- [3] A.A. Rontogiannis and S. Theodoridis, "Inverse factorization adaptive least-squares algorithms," *Signal Processing*, vol. 52, no. 1, pp. 35-47, July 1996.
- [4] S.C. Douglas, "Numerically-robust  $O(N^2)$  RLS algorithms using least-squares prewhitening," Proc. IEEE Int. Conf. on Acoustics, Speech, and Signal Processing, Istanbul, Turkey, vol. I, pp. 412-415, June 2000.
- [5] A. H. Sayed, *Fundamentals of Adaptive Filtering*, Hoboken, NJ: John Wiley & Sons, 2003.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

## See Also

### Objects

`dsp.FIRFilter` | `dsp.LMSFilter` | `dsp.AffineProjectionFilter`

### Topics

"Adaptive Noise Cancellation Using RLS Adaptive Filtering"

# **dsp.SampleRateConverter**

**Package:** dsp

Multistage sample rate converter

## **Description**

The `SampleRateConverter` System object converts the sample rate of an incoming signal.

To convert the sample rate of a signal:

- 1 Create the `dsp.SampleRateConverter` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## **Creation**

### **Syntax**

```
src = dsp.SampleRateConverter  
src = dsp.SampleRateConverter(Name, Value)
```

### **Description**

`src = dsp.SampleRateConverter` creates a multistage FIR sample rate converter System object, `src`, that converts the sample rate of each channel of an input signal.

`src = dsp.SampleRateConverter(Name, Value)` returns a multistage FIR sample rate converter System object, `src`, with properties and options specified by one or more `Name, Value` pair arguments.

### **Properties**

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

#### **Bandwidth — Two-sided bandwidth of interest**

40000 (default) | positive scalar

Specify the two-sided bandwidth of interest (after rate conversion) as a positive scalar expressed in Hz. This property sets the two-sided bandwidth of the information-carrying portion of the signal that you wish to retain.

**InputSampleRate — Sample rate of input signal**

192000 (default) | positive scalar

Specify the sample rate of the input signal as a positive scalar expressed in Hz. The input sample rate must be greater than the bandwidth of interest.

**OutputRateTolerance — Maximum allowed tolerance for output sample rate**

0 (default) | positive scalar

Specify the maximum allowed tolerance for the sample rate of the output signal as a positive scalar between 0 and 1.

The output rate tolerance allows for a simpler design in many cases. The actual output sample rate varies but is within the specified range. For example, if `OutputRateTolerance` is specified as `0.01`, then the actual output sample rate is in the range given by `OutputSampleRate ± 1%`.

**OutputSampleRate — Sample rate of output signal**

44100 (default) | positive scalar

Specify the sample rate of the output signal as a positive scalar expressed in Hz. The output sample rate must be greater than the bandwidth of interest.

**StopbandAttenuation — Minimum dB attenuation for aliased components**

80 (default) | positive scalar

Specify the stopband attenuation as a positive scalar expressed in dB. This property is the minimum amount by which any aliasing involved in the process is attenuated.

**Usage****Syntax**

$$y = \text{src}(x)$$
**Description**

`y = src(x)` designs one or more multirate FIR filters and then uses the filters to convert the rate of each channel (column) of the real or complex input signal `x` to the output sampling rate.

**Input Arguments****x — Input signal**

vector | matrix

Input signal, specified as a vector or a matrix of size  $P$ -by- $Q$ . The number of input rows  $P$  can be arbitrary and does not have to be a multiple of the overall decimation factor. You can determine the decimation factor using the `getRateChangeFactors` function. The object treats each column of `x` as a separate channel.

This object supports variable-size input signals, that is, the frame length (number of rows) of the signal can change even when the object is locked. However, the number of channels (columns) must remain constant.

Data Types: `single` | `double`

## Output Arguments

### **y** — Resampled signal

vector | matrix

Resampled signal, returned as a vector or matrix.

When the input is of size  $P$ -by- $Q$ , the output signal has an upper bound size of  $\text{ceil}(P/R)$ -by- $Q$ , where  $R$  is the ratio of the input sample rate and the output sample rate. The number of channels (columns) does not change.

Data Types: `single` | `double`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Specific to `dsp.SampleRateConverter`

|                                    |  |
|------------------------------------|--|
| <code>getActualOutputRate</code>   | Get actual output rate                           |
| <code>getFilters</code>            | Obtain single-stage filters                      |
| <code>getRateChangeFactors</code>  | Get overall interpolation and decimation factors |
| <code>visualizeFilterStages</code> | Visualize filter stages                          |

## Filter Analysis

|                    |   |
|--------------------|---|
| <code>cost</code>  | Implementation cost of the sample rate converter      |
| <code>freqz</code> | Frequency response of the multirate multistage filter |
| <code>info</code>  | Display information about sample rate converter       |

## Common to All System Objects

|                      |  |
|----------------------|--|
| <code>step</code>    | Run System object algorithm  |
| <code>release</code> | Release resources and allow changes to System object property values and input characteristics |
| <code>reset</code>   | Reset internal states of System object   |

## Examples

### Convert Sample Rate of Audio Signal

Convert the sample rate of an audio signal from 44.1 kHz (CD quality) to 96 kHz (DVD quality).

**Note:** The `dsp.AudioFileWriter` System object™ is not supported in MATLAB Online.

```
FsIn = 44.1e3;  
FsOut = 96e3;
```

```
SRC = dsp.SampleRateConverter('Bandwidth',40e3,...  
    'InputSampleRate',FsIn,'OutputSampleRate',FsOut);
```

```
[L,M] = getRateChangeFactors(SRC);
FrameSize = 10*M;
```

```
AR = dsp.AudioFileReader('guitar10min.ogg', ...
    'SamplesPerFrame',FrameSize);
AW = dsp.AudioFileWriter('guitar10min_96k.wav', ...
    'SampleRate',FsOut);
```

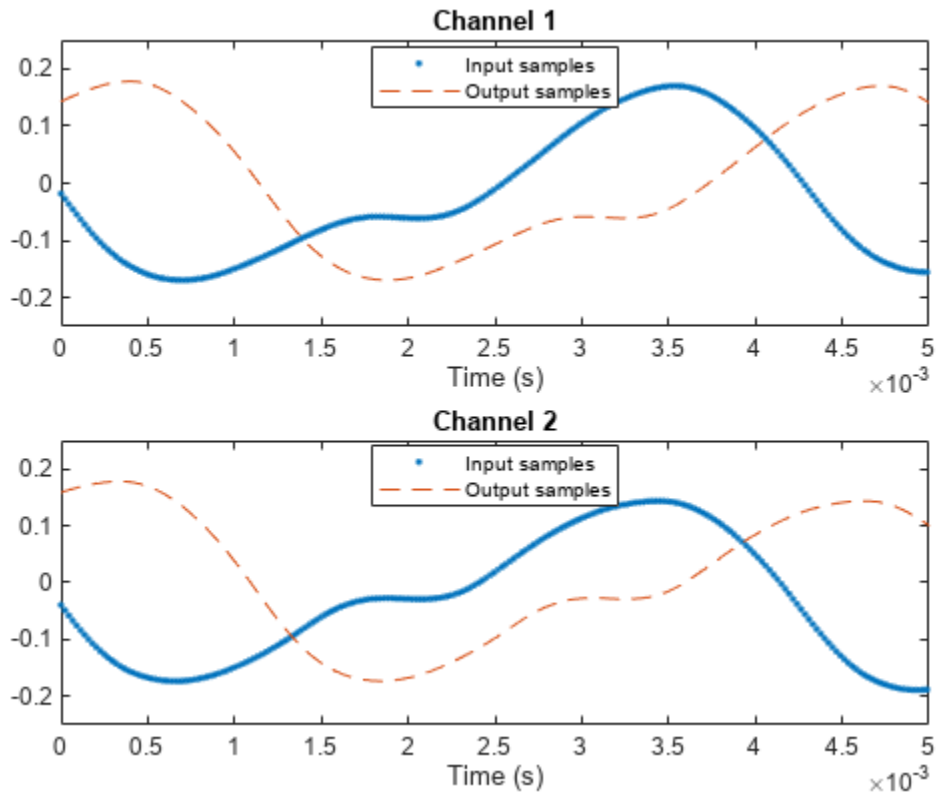
Convert 100 frames of audio signal. Release all objects.

```
for n = 1:100
    x = AR();
    y = SRC(x);
    AW(y);
end
```

```
release(AR);
release(AW);
release(SRC);
```

Plot the input and output signals. The latency of the sample rate converter introduces a delay in the output signal.

```
tx = (0:length(x)-1)./FsIn;
ty = (0:length(y)-1)./FsOut;
figure
subplot(2,1,1)
plot(tx,x(:,1),'.')
hold on
plot(ty,y(:,1),'--')
ylim([-0.25 0.25]);
xlim([0 0.005])
xlabel('Time (s)')
legend('Input samples','Output samples','Location','best')
title('Channel 1')
subplot(2,1,2)
plot(tx,x(:,2),'.')
hold on
plot(ty,y(:,2),'--')
ylim([-0.25 0.25]);
xlim([0 0.005])
xlabel('Time (s)')
legend('Input samples','Output samples','Location','best')
title('Channel 2')
```



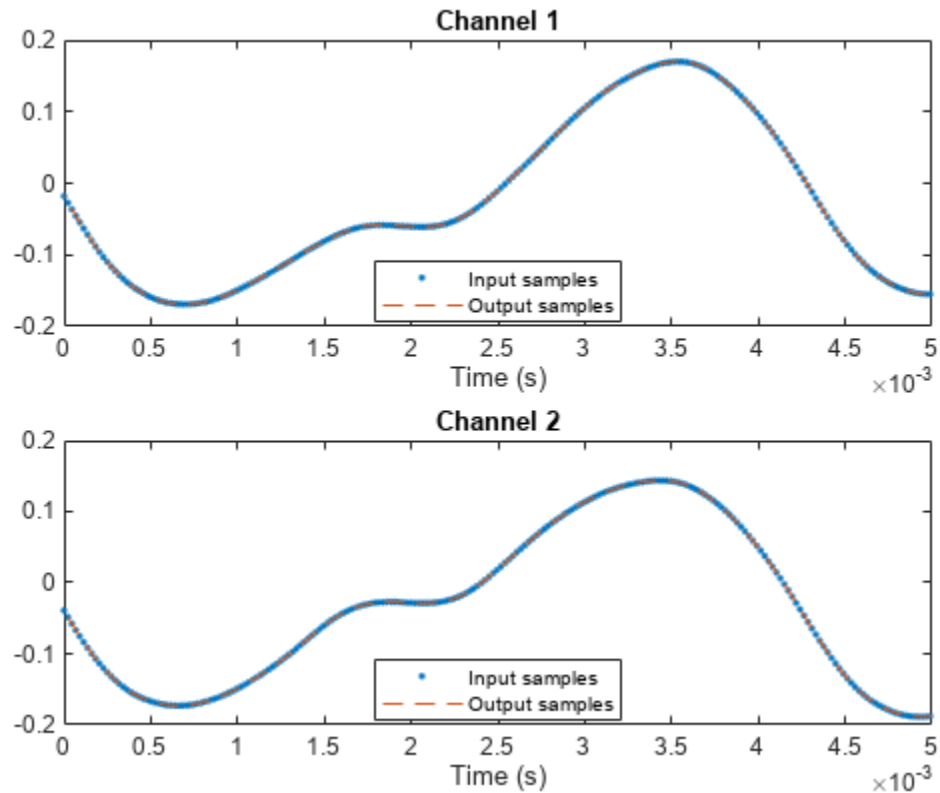
Use the `outputDelay` function to determine this delay value. To account for this delay, shift the output by this delay value.

```
[delay,~,~] = outputDelay(SRC,Fc=0)

delay = 0.0012

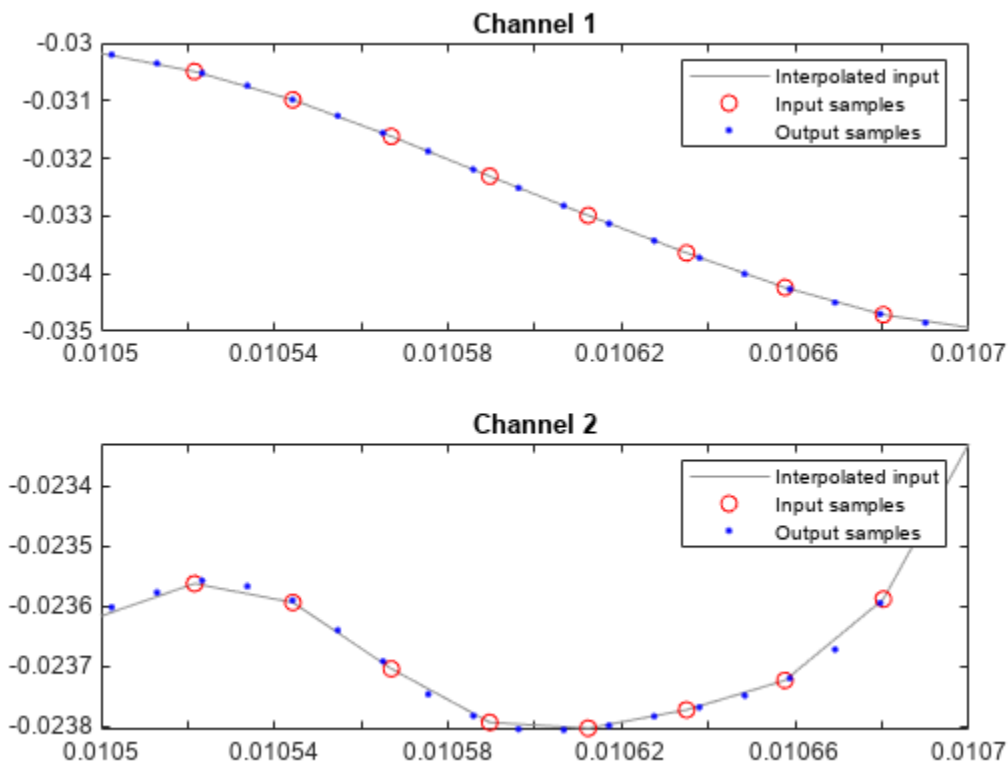
tx = (0:length(x)-1)./FsIn;
ty = (0:length(y)-1)./FsOut;
figure
subplot(2,1,1)
plot(tx,x(:,1),'.')
hold on
plot(ty-delay,y(:,1),'--')
xlim([0 0.005])
xlabel('Time (s)')
legend('Input samples','Output samples','Location','best')
title('Channel 1')
subplot(2,1,2)
plot(tx,x(:,2),'.')
hold on
plot(ty-delay,y(:,2),'--')
xlim([0 0.005])
xlabel('Time (s)')
legend('Input samples','Output samples','Location','best')
title('Channel 2')
```





Zoom in to see the difference in sample rates.

```
figure
subplot(2,1,1)
plot(tx,x(:,1),Color=[0.6 0.6 0.6])
hold on
plot(tx,x(:,1),'ro')
plot(ty-delay,y(:,1),'b.')
xlim([0.0105 0.0107])
legend('Interpolated input','Input samples','Output samples')
title('Channel 1')
subplot(2,1,2)
plot(tx,x(:,2),Color=[0.6 0.6 0.6])
hold on
plot(tx,x(:,2),'ro')
plot(ty-delay,y(:,2),'b.')
xlim([0.0105 0.0107])
legend('Interpolated input','Input samples','Output samples')
title('Channel 2')
```



### Convert Sample Rate of Sinusoid

Create a multistage sample rate converter with default properties. The converter converts from 192 kHz to 44.1 kHz in three stages.

```
src = dsp.SampleRateConverter;
```

Use `src` to convert the sample rate of a noisy sinusoid. The sinusoid has a frequency of 20 kHz and is sampled for 0.1 s.

```
f = 20e3;
```

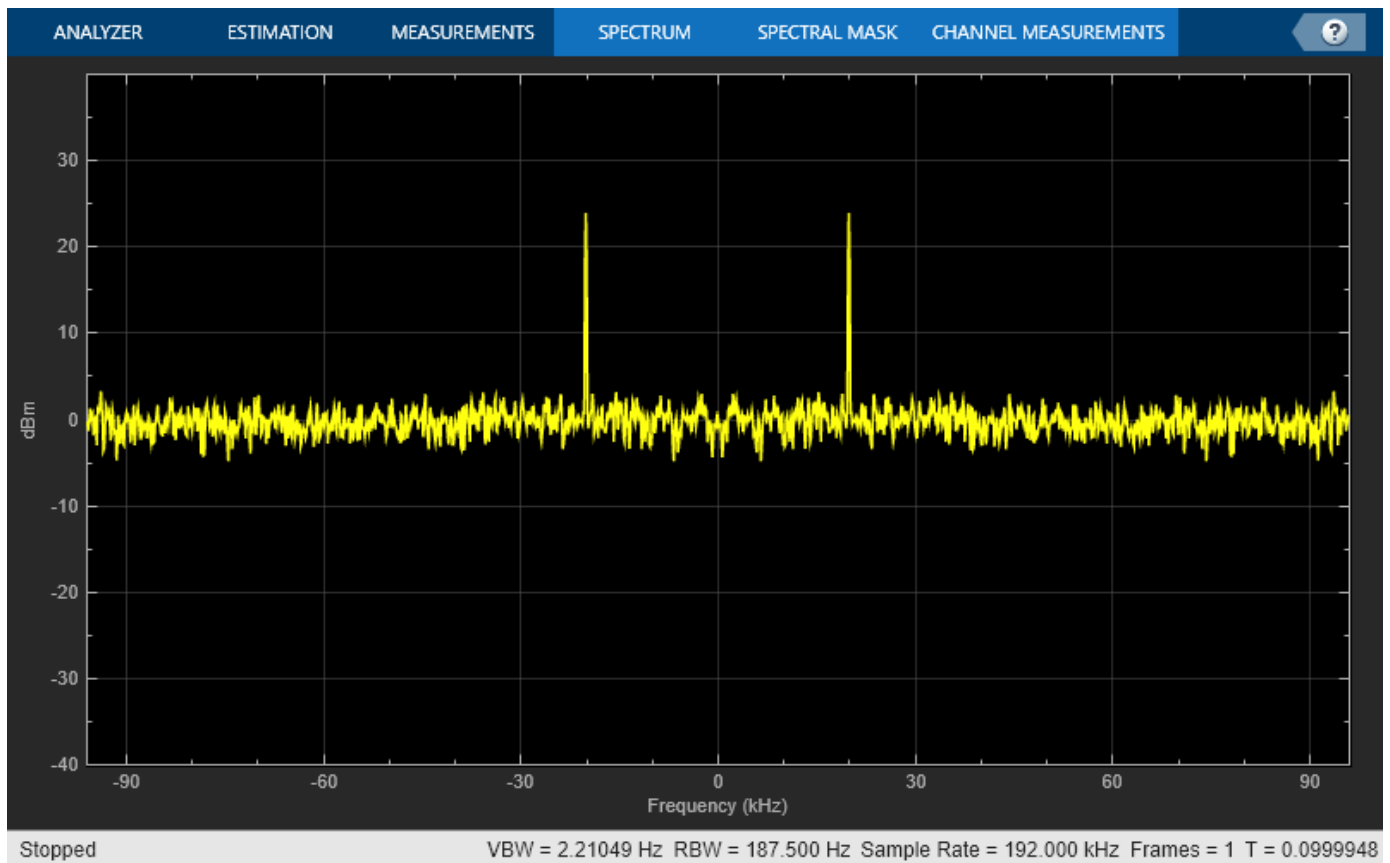
```
FsIn = src.InputSampleRate;  
FsOut = src.OutputSampleRate;
```

```
t1 = (0:1/FsIn:0.1-1/FsIn)';
```

```
sIn = sin(2*pi*f*t1) + randn(size(t1));
```

Estimate the power spectral density of the input.

```
hsa = spectrumAnalyzer('SampleRate',FsIn,...  
    'Method','welch','YLimits',[-40 40]);  
hsa(sIn)  
release(hsa)
```

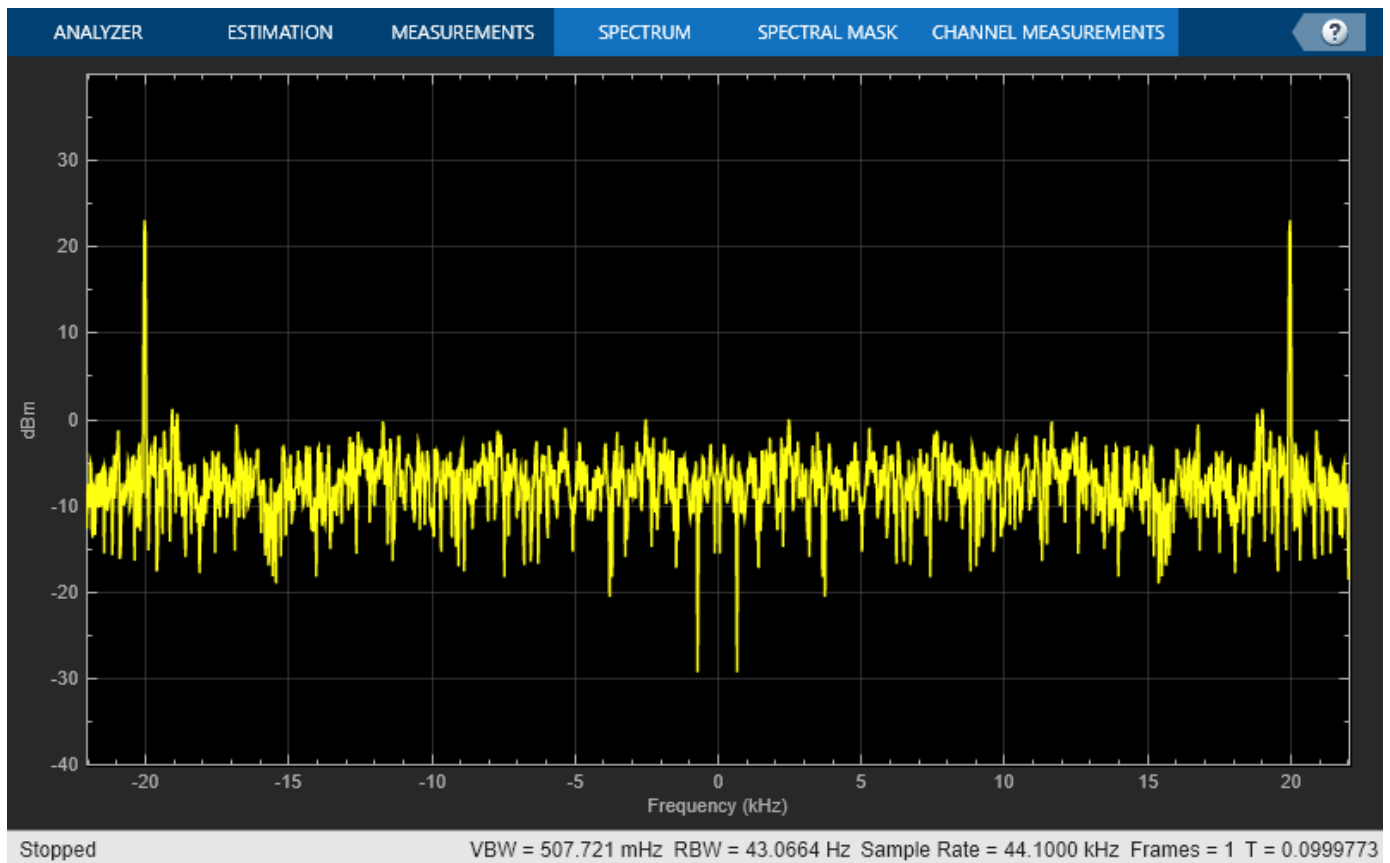


Convert the sample rate of the signal. Estimate the power spectral density of the output.

```
s0out = src(sIn);
```

```
hsb = spectrumAnalyzer('SampleRate',FsOut,...  
    'Method','welch','YLimits',[-40 40]);
```

```
hsb(s0out)  
release(hsb)
```



### Tolerance Cost in Sample Rate Conversion

A signal output from an A/D converter is sampled at 98.304 MHz. The signal has a bandwidth of 20 MHz. Reduce the sample rate of the signal to 22 MHz, which is the bandwidth of 802.11 channels. Make the conversion exactly and then redo it with an output rate tolerance of 1%.

```
SRC1 = dsp.SampleRateConverter('Bandwidth',20e6, ...
    'InputSampleRate',98.304e6,'OutputSampleRate',22e6, ...
    'OutputRateTolerance',0);
SRC2 = dsp.SampleRateConverter('Bandwidth',20e6, ...
    'InputSampleRate',98.304e6,'OutputSampleRate',22e6, ...
    'OutputRateTolerance',0.01);
```

Use the `cost` method to determine the cost of each sample rate conversion. The zero-tolerance process requires more than 500 times as many coefficients as the 1% process.

```
c1 = cost(SRC1)

c1 = struct with fields:
    NumCoefficients: 84779
    NumStates: 133
    MultiplicationsPerInputSample: 27.0422
    AdditionsPerInputSample: 26.0684
```

```

c2 = cost(SRC2)
c2 = struct with fields:
    NumCoefficients: 150
    NumStates: 127
    MultiplicationsPerInputSample: 22.6667
    AdditionsPerInputSample: 22.1111

```

Find the integer upsampling and downsampling factors used in each conversion.

```

[L1,M1] = getRateChangeFactors(SRC1)
L1 = 1375
M1 = 6144
[L2,M2] = getRateChangeFactors(SRC2)
L2 = 2
M2 = 9

```

Compute the actual sample rate of the output signal when the sample rate conversion has a tolerance of 1%.

```

getActualOutputRate(SRC2)
ans = 2.1845e+07

```

### Reset a Sample Rate Converter

Create a multistage sample rate converter with default properties, corresponding to the combined three filter stages used to convert from 192 kHz to 44.1 kHz. Determine its overall decimation and interpolation factors.

```

src = dsp.SampleRateConverter;
[L,M] = getRateChangeFactors(src);

```

Create a two-channel random signal. Specify a number of samples equal to the decimation factor. Call the object twice on the signal.

```

x = randn(M,2);
y1 = src(x);
y2 = src(x);
no = all(y2==y1)
no = 1x2 logical array
    0    0

```

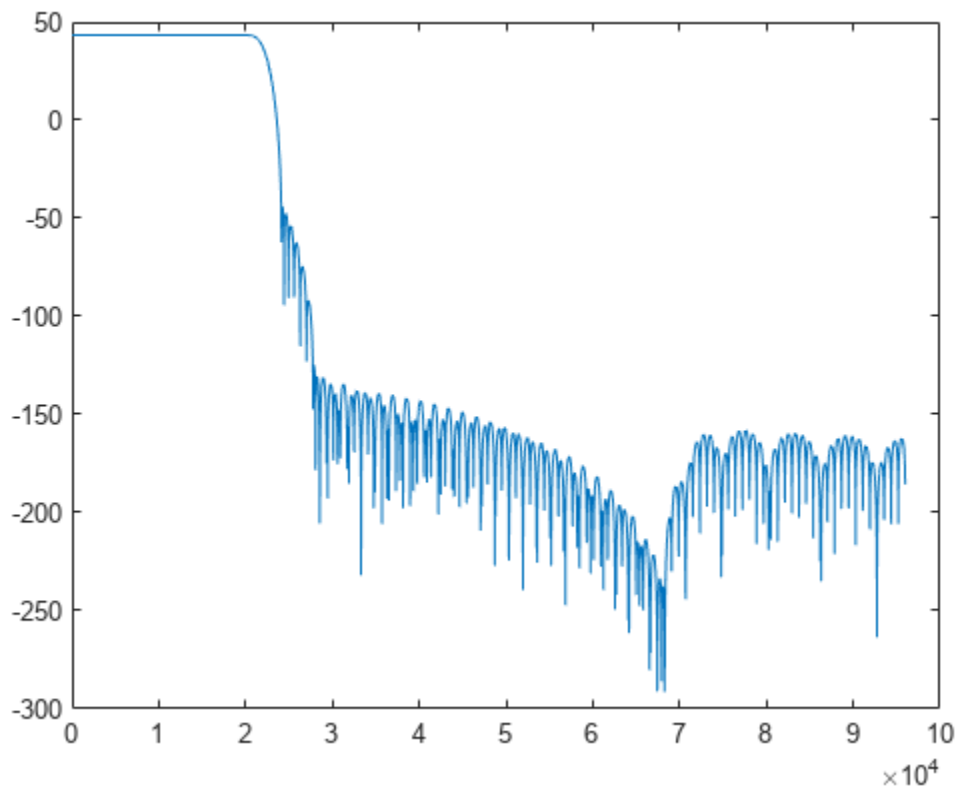
The output is different because the internal states of `src` have changed. Use `reset` to reset the converter and call the converter again. Verify that the output is unchanged.

```
reset(src)
y3 = src(x);
yes = all(y3==y1)
yes = 1x2 logical array
     1     1
```

### Frequency Response of Default Converter

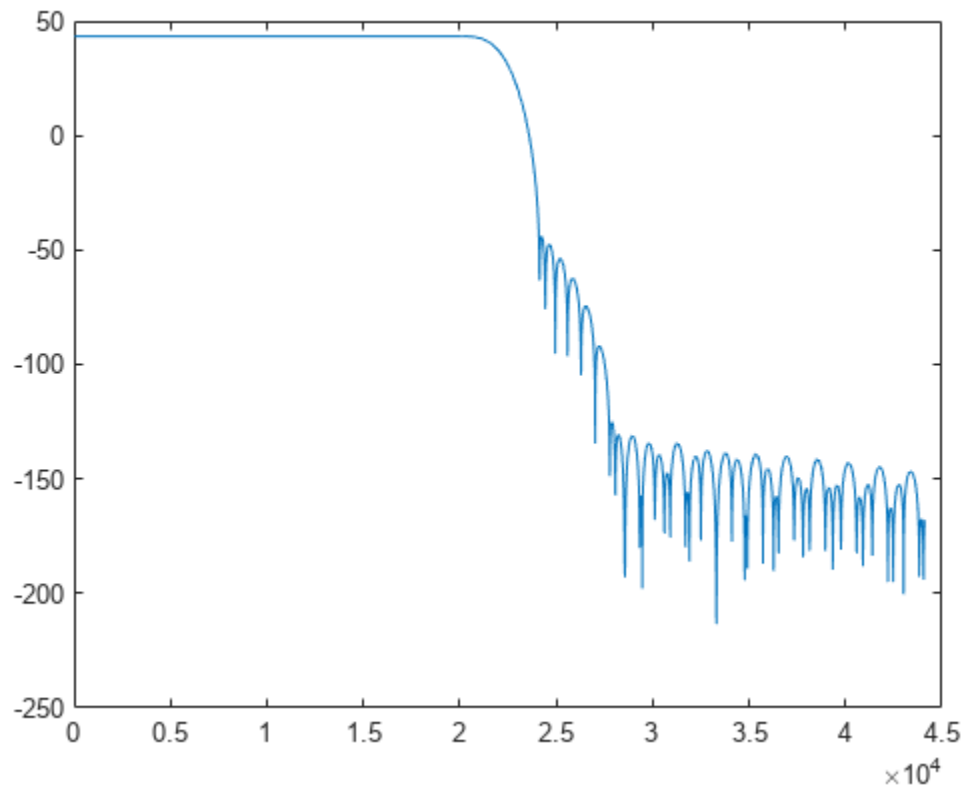
Create a multistage sample rate converter with default properties, corresponding to the combined three filter stages used to convert from 192 kHz to 44.1 kHz. Compute and display the frequency response.

```
src = dsp.SampleRateConverter;
[H,f] = freqz(src);
plot(f,20*log10(abs(H)))
```



Compute and display the frequency response over the range between 20 Hz and 44.1 kHz.

```
f = 20:10:44.1e3;
[H,f] = freqz(src,f);
plot(f,20*log10(abs(H)))
```



### Single-Stage Filters

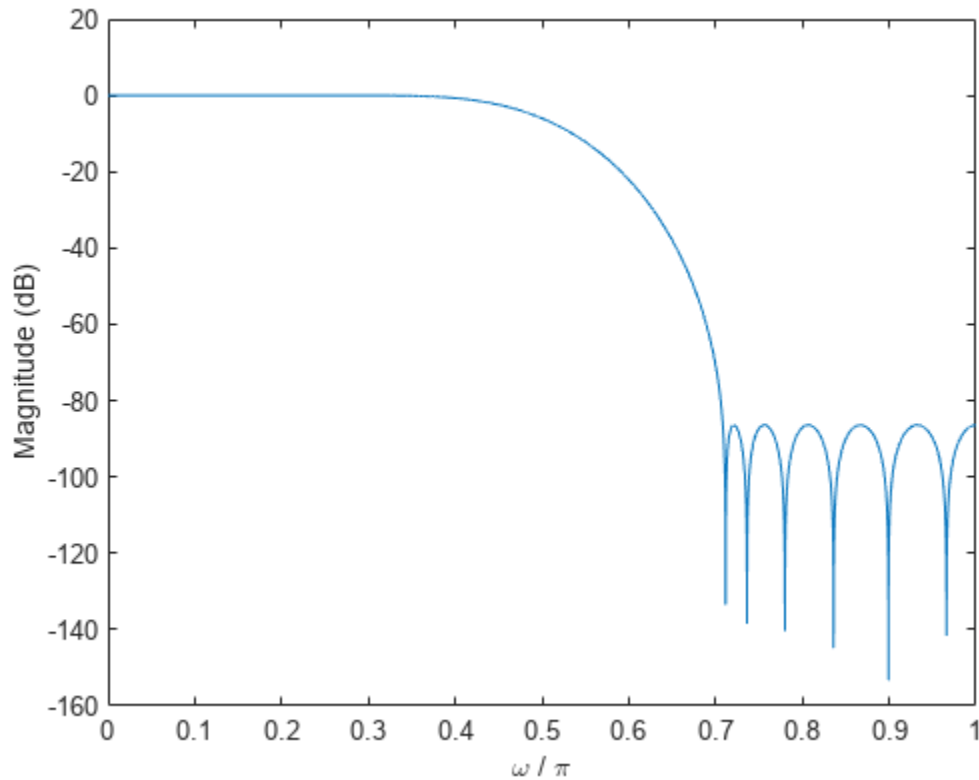
Create `src`, a multistage sample rate converter with default properties. `src` converts between 192 kHz and 44.1 kHz. Find the individual filters that are cascaded together to perform the conversion.

```
src = dsp.SampleRateConverter;
c = getFilters(src);
```

Visualize the frequency response of the decimator used in the first stage of the process.

```
m = c.Stage1;

[h,w] = freqz(m);
plot(w/pi,20*log10(abs(h)))
xlabel('\omega / \pi')
ylabel('Magnitude (dB)')
```



### Default Multistage Sample Rate Converter

Create a multistage sample rate converter with default properties, corresponding to the combined three filter stages used to convert from 192 kHz to 44.1 kHz.

```
src = dsp.SampleRateConverter
```

```
src =  
  dsp.SampleRateConverter with properties:
```

```
    InputSampleRate: 192000  
    OutputSampleRate: 44100  
    OutputRateTolerance: 0  
    Bandwidth: 40000  
    StopbandAttenuation: 80
```

Display information about the design.

```
info(src)
```

```
ans =  
  'Overall Interpolation Factor    : 147  
  Overall Decimation Factor      : 640  
  Number of Filters              : 3
```



```

Multiplications per Input Sample: 27.667188
Number of Coefficients           : 8631
Filters:
  Filter 1:
    dsp.FIRDecimator      - Decimation Factor   : 2
  Filter 2:
    dsp.FIRDecimator      - Decimation Factor   : 2
  Filter 3:
    dsp.FIRRateConverter  - Interpolation Factor: 147
                        - Decimation Factor    : 160

```

### Output Sample Rate with Given Tolerance

Get the actual output sample rate for conversion between 192 kHz and 44.1 kHz when given a tolerance of 1%.

```

src = dsp.SampleRateConverter;
src.OutputRateTolerance = 0.01;
FsOut = getActualOutputRate(src)

```

```

FsOut = 4.4308e+04

```

### Default Resampling Factors

Create `src`, a multistage sample rate converter with default properties. `src` combines three filter stages to convert from 192 kHz to 44.1 kHz. Determine its overall interpolation and decimation factors.

```

src = dsp.SampleRateConverter;
[L,M] = getRateChangeFactors(src)

```

```

L = 147

```

```

M = 640

```

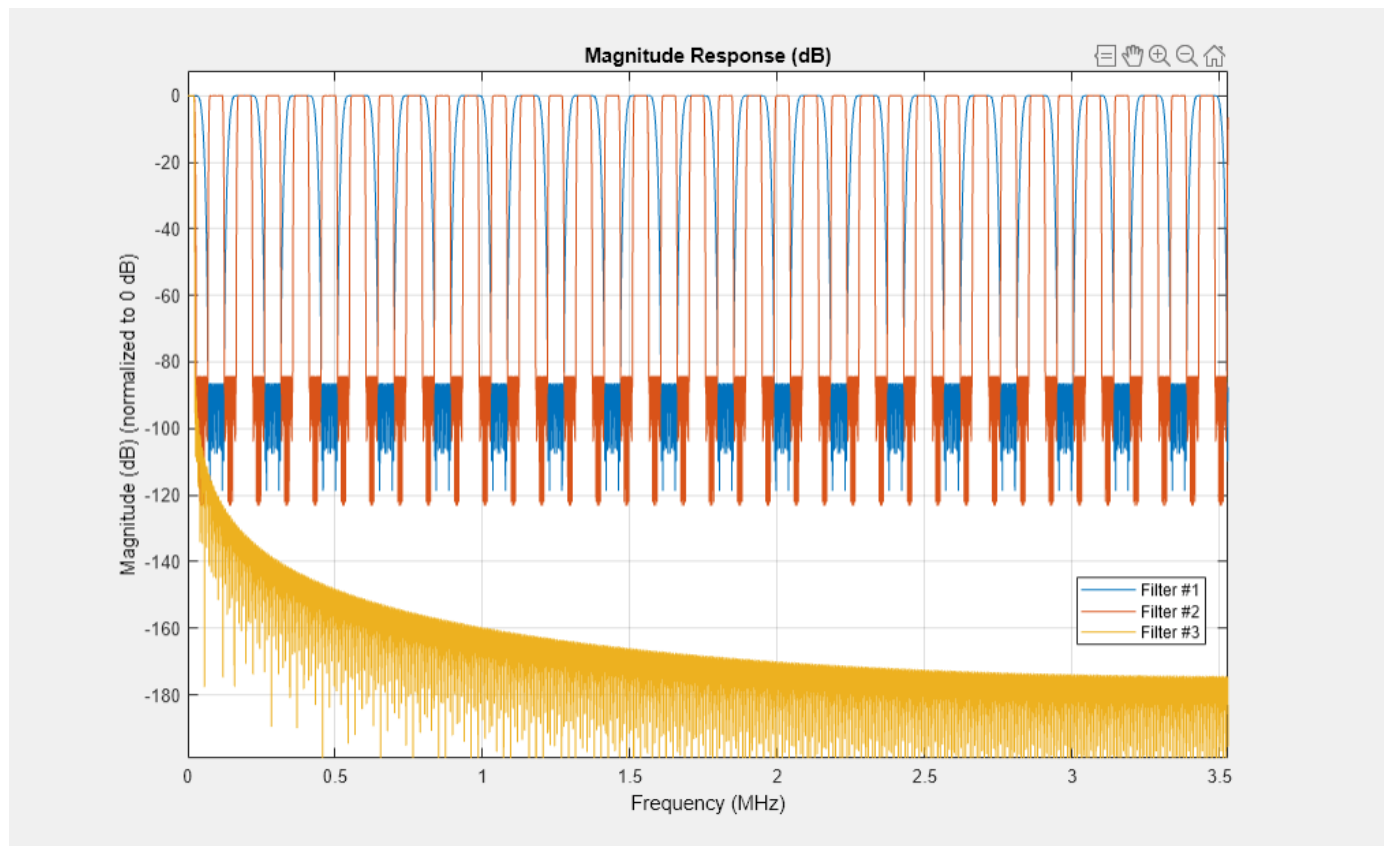
### Sample Rate Converter Stages

Create a multistage sample rate converter with default properties, corresponding to the combined three filter stages used to convert from 192 kHz to 44.1 kHz. Visualize the stages.

```

src = dsp.SampleRateConverter;
visualizeFilterStages(src)

```



## Algorithms

- The general multistage sample rate converter performs a multistage decimation, a single-stage sample rate conversion, and a multistage interpolation. Actual designs include at most two of these steps.
- The procedure automatically determines the optimal number of decimation or interpolation stages. In special cases, the algorithm can perform a decimation or interpolation in a single stage.
- The algorithm always attempts to start by reducing the sample rate. This decreases the amount of computation required. The decimation step ensures that the intermediate sample rate does not go below the bandwidth of interest and that the block does not filter out any information.
- Each individual stage uses halfband or Nyquist filters to minimize the number of nonzero coefficients.
- The algorithm allows transition-band aliasing because it decreases the implementation cost. It keeps the signal within the bandwidth of interest alias free up to the value specified by the `StopbandAttenuation` property.

## Version History

Introduced in R2014b

**R2023a: Support for arbitrary input frame length**

This object supports an input signal with an arbitrary frame length, so the input frame length does not have to be a multiple of the decimation factor.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- See “System Objects in MATLAB Code Generation” (MATLAB Coder).
- The `getRateChangeFactors` function supports C and C++ code generation.
- This object supports SIMD code generation using Intel AVX2 technology under these conditions:
  - For upsampling, the ratio of output sample rate to input sample rate must be an integer.
  - For downsampling, the ratio of input sample rate to output sample rate must be an integer.
  - Input signal has a data type of `single` or `double`.

The SIMD technology significantly improves the performance of the generated code.

- In the code you generate from this object, the resampled signal output is a fixed-size signal if the input signal is fixed size and is a multiple of the decimation factor. If the input is not a multiple of the decimation factor or if the input is a variable-size signal, then the resampled signal output is of variable-size.

## See Also

### Functions

`cost` | `freqz` | `getActualOutputRate` | `getFilters` | `info` | `visualizeFilterStages` | `getRateChangeFactors`

### Objects

`dsp.FarrowRateConverter`

### Topics

“Efficient Sample Rate Conversion Between Arbitrary Factors”

“Multirate Filtering in MATLAB and Simulink”

“Design of Decimators and Interpolators”

“Digital Up and Down Conversion for Family Radio Service in MATLAB”

## dsp.ScalarQuantizerDecoder

**Package:** dsp

(Removed) Convert each index value into quantized output value

---

**Note** `dsp.ScalarQuantizerDecoder` has been removed.

---

### Description

The `dsp.ScalarQuantizerDecoder` object converts each index value into a quantized output value. The specified codebook defines the set of all possible quantized output values or codewords. Input index values less than 0 are set to 0 and index values greater  $N - 1$  are set to  $N - 1$ .  $N$  is the length of the codebook vector.

To convert an index value into a quantized output value:

- 1 Create the `dsp.ScalarQuantizerDecoder` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

### Creation

#### Syntax

```
sqdec = dsp.ScalarQuantizerDecoder  
sqdec = dsp.ScalarQuantizerDecoder(Name,Value)
```

#### Description

`sqdec = dsp.ScalarQuantizerDecoder` returns a scalar quantizer decoder System object, `sqdec`, that transforms zero-based input index values into quantized output values.

`sqdec = dsp.ScalarQuantizerDecoder(Name,Value)` returns a scalar quantizer decoder object, `sqdec`, with each specified property set to the specified value.

### Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

#### CodebookSource — How to specify codebook values

'Property' (default) | 'Input port'

Specify how to determine the codebook values as 'Property' or 'Input port'.

### Codebook — Codebook

1:10 (default) | vector

Specify the codebook as a vector of quantized output values that correspond to each index value.

**Tunable:** Yes

### Dependencies

This property applies when you set CodebookSource to 'Property'.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### OutputDataType — Data type of codebook and quantized output

'double' | 'single' | 'Custom' | 'Same as input'

Specify the data type of the codebook and quantized output values as 'Same as input', 'double', 'single', or 'Custom'.

### Dependencies

This property applies when you set CodebookSource to 'Property'.

### Fixed-Point Properties

#### CustomOutputDataType — Output word and fraction lengths

numerictype(true,16) (default) | numerictype

Specify the output fixed-point type as a signed or unsigned numerictype object.

### Dependencies

This property applies only when you set the OutputDataType property to Custom.

## Usage

### Syntax

```
Q = sqdec(I)
Q = sqdec(I,C)
```

### Description

$Q = \text{sqdec}(I)$  returns the quantized output values  $Q$  corresponding to the input indices  $I$ .

$Q = \text{sqdec}(I,C)$  uses input  $C$  as the codebook values when you set the CodebookSource property to Input port. The data type of  $C$  can be double, single, or fixed-point. The output  $Q$  has the same data type as the codebook input  $C$ .

### Input Arguments

#### I — Input indices

vector | matrix

Input indices, specified as a vector or a matrix.

Data Types: `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

### **C — Codebook values**

vector

Codebook, specified as a vector of quantized output values that correspond to each index value.

#### **Dependencies**

This property applies when you set `CodebookSource` to `'Input port'`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **Output Arguments**

#### **Q — Quantized output values**

vector | matrix

Quantized output values, returned as a vector or a matrix of the same size as the input, `I`. When codebook is specified as a property, the output data type is determined by the `OutputDataType` property. When the codebook is specified as an input, the output data type is same as the data type of the codebook input.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fi`

## **Object Functions**

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## **Common to All System Objects**

|                      |  |
|----------------------|--|
| <code>step</code>    | Run System object algorithm  |
| <code>release</code> | Release resources and allow changes to System object property values and input characteristics |
| <code>reset</code>   | Reset internal states of System object   |

## **Examples**

### **Quantize Signal**

Given a codebook and index values as inputs, determine the corresponding output quantized values.

```
codebook = single([-2.1655 -1.3238 -0.7365 -0.2249 0.2726, ...  
    0.7844 1.3610 2.1599]);  
indices = uint8([1 3 5 7 6 4 2 0]);  
sqdec = dsp.ScalarQuantizerDecoder;  
sqdec.CodebookSource = 'Input port';  
qout = sqdec(indices,codebook)
```

*qout = 1x8 single row vector*

-1.3238 -0.2249 0.7844 2.1599 1.3610 0.2726 -0.7365 -2.1655

## Algorithms

This object implements the algorithm, inputs, and outputs described on the Scalar Quantizer Decoder block reference page. The object properties correspond to the block parameters, except there is no object property that directly corresponds to the **Action for out of range index value** block parameter. The object sets any index values less than 0 to 0 and any index values greater than or equal to  $N$  to  $N - 1$ .

## Version History

### Introduced in R2012a

### **R2021b: dsp.ScalarQuantizerDecoder System object will be removed**

*Warns starting in R2021b*

`dsp.ScalarQuantizerDecoder` System object will be removed in a future release.

### **R2023a: dsp.ScalarQuantizerDecoder System object has been removed**

*Errors starting in R2023a*

The `dsp.ScalarQuantizerDecoder` System object has been removed.

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

# dsp.ScalarQuantizerEncoder

**Package:** dsp

(Removed) Associate input value with index value of quantization region

---

**Note** dsp.ScalarQuantizerEncoder has been removed.

---

## Description

The dsp.ScalarQuantizerEncoder object encodes each input value by associating that value with the index value of the quantization region. Then, the object outputs the index of the associated region.

To encode an input value by associating it with an index value of the quantization region:

- 1 Create the dsp.ScalarQuantizerEncoder object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
sqenc = dsp.ScalarQuantizerEncoder  
sqenc = dsp.ScalarQuantizerEncoder(Name,Value)
```

### Description

sqenc = dsp.ScalarQuantizerEncoder returns a scalar quantizer encoder System object, sqenc. This object maps each input value to a quantization region by comparing the input value to the user-specified boundary points.

sqenc = dsp.ScalarQuantizerEncoder(Name,Value) returns a scalar quantizer encoder object, sqenc, with each specified property set to the specified value.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

### BoundaryPointsSource — Source of boundary points

'Property' (default) | 'Input port'



Specify how to determine the boundary points and codebook values as 'Property' or 'Input port'.

### Partitioning — Quantizer is bounded or unbounded

'Bounded' (default) | 'Unbounded'

Specify the quantizer as 'Bounded' or 'Unbounded'.

### BoundaryPoints — Boundary points of quantizer regions

1:10 (default) | vector

Specify the boundary points of quantizer regions as a vector. The vector values must be in ascending order. Let  $[p_0 \ p_1 \ p_2 \ p_3 \ \dots \ p_N]$  denote the boundary points property in the quantizer. If the quantizer is bounded, the object uses this property to specify  $[p_0 \ p_1 \ p_2 \ p_3 \ \dots \ p_N]$ . If the quantizer is unbounded, the object uses this property to specify  $[p_1 \ p_2 \ p_3 \ \dots \ p_{(N-1)}]$  and sets  $p_0 = -\text{Inf}$  and  $p_N = +\text{Inf}$ .

**Tunable:** Yes

### Dependencies

This property applies when you set the BoundaryPointsSource property to Property.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### SearchMethod — Find quantizer index by linear or binary search

'Linear' (default) | 'Binary'

Specify whether to find the appropriate quantizer index using a linear search or a binary search as one of 'Linear' or 'Binary'. The computational cost of the linear search method is of the order  $P$  and the computational cost of the binary search method is of the order

$$\log_2(P)$$

where  $P$  is the number of boundary points.

### TiebreakerRule — Behavior when input equals boundary point

'Choose the lower index' (default) | 'Choose the higher index'

Specify whether the input value is assigned to the lower indexed region or higher indexed region when the input value equals boundary point by selecting 'Choose the lower index' or 'Choose the higher index'.

### CodewordOutputPort — Enable output of codeword value

false (default) | true

Set this property to true to output the codeword values that correspond to each index value.

### QuantizationErrorOutputPort — Enable output of quantization error

false (default) | true

Set this property to true to output the quantization error for each input value. The quantization error is the difference between the input value and the quantized output value.

### Codebook — Codebook

1.5:9.5 (default) | vector

Specify the codebook as a vector of quantized output values that correspond to each region. If the `Partitioning` property is 'Bounded' and the boundary points vector has length `N`, you must set this property to a vector of length `N-1`. If the `Partitioning` property is 'Unbounded' and the boundary points vector has length `N`, you must set this property to a vector of length `N+1`.

**Tunable:** Yes

**Dependencies**

This property applies when you set the `BoundaryPointsSource` property to 'Property' and either the `CodewordOutputPort` property or the `QuantizationErrorOutputPort` property is true.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**ClippingStatusOutputPort — Enable output of clipping status**

`false` (default) | `true`

Set this property to `true` to output the clipping status. The output is a 1 when an input value is outside the range defined by the `BoundaryPoints` property. When the value is inside the range, the exception output is a 0.

**Dependencies**

This property applies when you set the `Partitioning` property to 'Bounded'.

**OutputIndexDataType — Data type of the index output**

`int32` (default) | `int8` | `int16` | `uint8` | `uint16` | `uint32`

Specify the data type of the index output from the object as: `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`.

**Fixed-Point Properties**

**RoundingMethod — Rounding method for fixed-point operations**

'Floor' (default) | 'Ceiling' | 'Convergent' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'

Specify the rounding method.

**OverflowAction — Overflow action for fixed-point operations**

'Wrap' (default) | 'Saturate'

Specify the overflow action.

## Usage

### Syntax

```
index = sqenc(input)
[index] = sqenc(input,bpoints)
[index] = sqenc(input,bpoints,cBook)
[index,codeword] = sqenc(____)
[____,qerr] = sqenc(____)
[____,cStatus] = sqenc(____)
```

## Description

`index = sqenc(input)` returns the index of the quantization region to which the `input` belongs. The input data, boundary points, codebook values, quantized output values, and the quantization error must have the same data type whenever they are present.

`[index] = sqenc(input,bpoints)` uses input `bpoints` as the boundary points when the `BoundaryPointsSource` property is `Input port`.

`[index] = sqenc(input,bpoints,cBook)` uses input `bpoints` as the boundary points and input `cBook` as the codebook when the `BoundaryPointsSource` property is `'Input port'` and either the `CodewordOutputPort` property or the `QuantizationErrorOutputPort` property is `true`.

`[index,codeword] = sqenc( ___ )` outputs the codeword values that corresponds to each index value when the `CodewordOutputPort` property is `true`.

`[ ___ ,qerr] = sqenc( ___ )` outputs the quantization error `qerr` for each input value when the `QuantizationErrorOutputPort` property is `true`.

`[ ___ ,cStatus] = sqenc( ___ )` also returns output `cStatus` as the clipping status output port for each input value when the `Partitioning` property is `'Bounded'` and the `ClippingStatusOutputPort` property is `true`. If an input value is outside the range defined by the `BoundaryPoints` property, `cStatus` is `true`. If an input value is inside the range, `cStatus` is `false`.

## Input Arguments

### **input** — Input data

vector | matrix

Input data, specified as a vector or a matrix. If the input is fixed point, it must be signed fixed point with power-of-two slope and zero bias.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `fi`

### **bpoints** — Boundary points

vector

Boundary points of the quantizer regions, specified as a vector. The vector values must be in ascending order. Let `[p0 p1 p2 p3 ... pN]` denote the boundary points in the quantizer. If the quantizer is bounded, the object uses this input to specify `[p0 p1 p2 p3 ... pN]`. If the quantizer is unbounded, the object uses this input to specify `[p1 p2 p3 ... p(N-1)]` and sets `p0 = -Inf` and `pN = +Inf`.

## Dependencies

This input applies when you set the `BoundaryPointsSource` property to `Input port`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **cBook** — Code book

vector

Codebook, specified as a vector of quantized output values that correspond to each region. If the `Partitioning` property is `'Bounded'` and the boundary points vector has length  $N$ , this input must be a vector of length  $N-1$ . If the `Partitioning` property is `'Unbounded'` and the boundary points vector has length  $N$ , this input must be a vector of length  $N+1$ .

### Dependencies

This input applies when you set the `BoundaryPointsSource` property is 'Input port' and either the `CodewordOutputPort` property or the `QuantizationErrorOutputPort` property is true.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### Output Arguments

#### **index** — Index of quantization region

vector | matrix

Index of the quantization region to which the input belongs, returned as a vector or a matrix of the same size as the input.

Data Types: `int32`

#### **codeword** — Codeword values

vector | matrix

Codeword values that correspond to each index value, returned as a vector or a matrix of the same size as the input.

### Dependencies

This output is available when the `CodewordOutputPort` property is set to true.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `fi`

#### **qerr** — Quantization error

vector | matrix

Quantization error for each input value, returned as a vector or a matrix of the same size as the input.

### Dependencies

This output is available when the `QuantizationErrorOutputPort` property is set to true.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `fi`

#### **cStatus** — Clipping status

vector | matrix

Clipping status for each input value, returned as a vector or a matrix. If an input value is outside the range defined by the `BoundaryPoints` property, `cStatus` is true. If an input value is inside the range, `cStatus` is false.

### Dependencies

This output is available when the `Partitioning` property is 'Bounded' and the `ClippingStatusOutputPort` property is set to true.

Data Types: `logical`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Common to All System Objects

|         |  |
|---------|--|
| step    | Run System object algorithm  |
| release | Release resources and allow changes to System object property values and input characteristics |
| reset   | Reset internal states of System object   |

## Examples

### Quantize Varying Fractional Inputs

Quantize the varying fractional inputs between zero and five to the closest integers, and then plot the results.

```
sqenc = dsp.ScalarQuantizerEncoder;
sqenc.BoundaryPoints = [-.001 .499 1.499 ...
    2.499 3.499 4.499 5.001];
sqenc.CodewordOutputPort = true;
sqenc.Codebook = [0 1 2 3 4 5];
input = (0:0.02:5)';
[index, quantizedValue] = sqenc(input);
plot(1:length(input), [input quantizedValue]);
```

## Algorithms

This object implements the algorithm, inputs, and outputs described on the Scalar Quantizer Encoder block reference page. The object properties correspond to the block parameters.

## Version History

**Introduced in R2012a**

**R2021b: dsp.ScalarQuantizerEncoder System object will be removed**

*Warns starting in R2021b*

dsp.ScalarQuantizerEncoder System object will be removed in a future release.

**R2023a: dsp.ScalarQuantizerEncoder System object has been removed**

*Errors starting in R2023a*

The dsp.ScalarQuantizerEncoder System object has been removed.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

## dsp.SignalSink

**Package:** dsp

Log simulation data in buffer

### Description

The `dsp.SignalSink` System object logs MATLAB simulation data. This object accepts any numeric data type.

To log MATLAB simulation data :

- 1 Create the `dsp.SignalSink` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

### Creation

#### Syntax

```
ss = dsp.SignalSink  
ss = dsp.SignalSink(Name,Value)
```

#### Description

`ss = dsp.SignalSink` returns a signal sink, `ss`, that logs 2-D input data in the object.

`ss = dsp.SignalSink(Name,Value)` returns a signal sink, `ss`, with each specified property set to the specified value. Enclose each property name in single quotes. Unspecified properties have default values.

### Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

#### **BufferLength — Maximum number of input frames to log**

`inf` (default) | positive integer

Specify the maximum number of frames to log. The object always preserves the most recent data in the buffer. When you specify a buffer length that is greater than the input length, the object pads the end of the logged data with zeros. To capture all input data without extra padding, set the `BufferLength` property to `inf`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

### **Decimation — Decimation factor**

1 (default) | positive integer

Setting this property to any positive integer  $d$  causes the signal sink to write data at every  $d$ th sample.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **FrameHandlingMode — Output dimensionality for frame-based inputs**

2-D array (concatenate) (default) | 3-D array (separate)

Set the dimension of the output array for frame-based inputs as 2-D array (concatenate) or 3-D array (separate). Concatenation occurs along the first dimension for 2-D array (concatenate).

### **Buffer — Logged data (read only)**

matrix

This property is read-only.

The signal sink writes simulation data into a buffer. Specify the maximum length of the buffer with the `BufferLength` property.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

## **Usage**

### **Syntax**

`ss(x)`

### **Description**

`ss(x)` buffers the signal  $x$ . The buffer may be accessed at any time from the `Buffer` property of `ss`.

### **Input Arguments**

#### **x — Data input**

vector | matrix

Data input, specified as a vector or a matrix.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `fi`

## **Object Functions**

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

`release(obj)`

## Common to All System Objects

step     Run System object algorithm  
release   Release resources and allow changes to System object property values and input characteristics  
reset     Reset internal states of System object

## Examples

### Log Input Data

```
hlog = dsp.SignalSink;  
for i=1:10  
    y = sin(i);  
    hlog(y);  
end  
log = hlog.Buffer;  
display(log)
```

```
log = 10×1
```

```
    0.8415  
    0.9093  
    0.1411  
   -0.7568  
   -0.9589  
   -0.2794  
    0.6570  
    0.9894  
    0.4121  
   -0.5440
```

## Algorithms

This object implements the algorithm, inputs, and outputs described on the To Workspace block reference page. The object properties correspond to the block properties, except the object always generates fixed-point output for fixed-point input.

## Version History

Introduced in R2012b

## See Also

### Objects

dsp.SignalSource



# dsp.SignalSource

**Package:** dsp

Import variable from workspace

## Description

The `SignalSource` object imports a variable from the MATLAB workspace.

To import a variable from the MATLAB workspace:

- 1 Create the `dsp.SignalSource` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
src = dsp.SignalSource
src = dsp.SignalSource(signal,spf)
src = dsp.SignalSource(Name,Value)
```

### Description

`src = dsp.SignalSource` returns a signal source System object, `src`, that outputs the variable, specified by the `Signal` property, one sample or frame at a time.

`src = dsp.SignalSource(signal,spf)` returns a signal source object, `src`, with the `Signal` property set to `signal` and the “SamplesPerFrame” on page 4-0 property set to `spf`.

`src = dsp.SignalSource(Name,Value)` returns a signal source object, `src`, with each specified property set to the specified value.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

### Signal — Variable or expression containing the signal

[1:10]' (default) | vector | matrix

Specify the name of the workspace variable from which to import the signal, or a valid expression specifying the signal.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `fi`

### **SamplesPerFrame — Number of samples per output frame**

1 (default) | positive scalar

Specify the number of samples to buffer into each output frame. This property must be 1 when you specify a 3-D array in the `Signal` property.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **SignalEndAction — Action after final signal values are generated**

'Set to zero' (default) | 'Hold final value' | 'Cyclic repetition'

Specify the output after all of the specified signal samples have been generated as one of 'Set to zero', 'Hold final value', or 'Cyclic repetition'.

## **Usage**

### **Syntax**

```
Y = src()
```

### **Description**

`Y = src()` outputs one sample or frame of data from each column of the imported signal. The imported signal is the variable or expression you specify for the `Signal` property of the `SignalSource` System object, `src`.

### **Output Arguments**

#### **Y — Output data**

scalar | vector | matrix

One sample or one frame of data from each column of the imported signal, returned as a scalar, vector, or matrix. The number of columns in the output signal matches the number of columns in the imported signal, `Signal`. The number of rows in the output signal matches the value specified in the `SamplesPerFrame` property.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `fi`

## **Object Functions**

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Specific to dsp.SignalSource

isDone End-of-file status for signal reader object

## Common to All System Objects

step Run System object algorithm

release Release resources and allow changes to System object property values and input characteristics

reset Reset internal states of System object

## Examples

### Create Signal Source

Create a signal source to output one sample at a time.

```
src1 = dsp.SignalSource;
src1.Signal = randn(1024,1);
y1 = zeros(1024,1);
idx = 1;
while(~isDone(src1))
    y1(idx) = src1();
    idx = idx + 1;
end
```

Create a signal source to output vectors.

```
src2 = dsp.SignalSource(randn(1024,1),128);
y2 = src2(); % y2 is a 128-by-1 frame of samples
```

## Algorithms

This object implements the algorithm, inputs, and outputs described on the Signal From Workspace block reference page. The object properties correspond to the block parameters, except the System object does not have properties that correspond to the **Sample time** or **Warn when frame size does not evenly divide input length** block parameters.

## Version History

Introduced in R2012b

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

## **See Also**

### **Functions**

isDone

### **Objects**

dsp.SignalSink

# dsp.SineWave

**Package:** dsp

Generate discrete sine wave

## Description

The `dsp.SineWave` System object generates a real or complex, multichannel sinusoidal signal with independent amplitude, frequency, and phase in each output channel.

For both real and complex sinusoids, the “Amplitude” on page 4-0 , “Frequency” on page 4-0 , and “PhaseOffset” on page 4-0 properties can be scalars or length- $N$  vectors, where  $N$  is the number of channels in the output. When you specify at least one of these properties as a length- $N$  vector, scalar values specified for the other properties are applied to each of the  $N$  channels.

To generate a discrete-time sinusoidal signal:

- 1 Create the `dsp.SineWave` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
sine = dsp.SineWave
sine = dsp.SineWave(Name,Value)
sine = dsp.SineWave(amp,freq,phase,Name,Value)
```

### Description

`sine = dsp.SineWave` creates a sine wave object that generates a real-valued sinusoid with an amplitude of 1, a frequency of 100 Hz, and a phase offset of 0. By default, the sine wave object generates only one sample.

`sine = dsp.SineWave(Name,Value)` creates a sine wave object with each specified property set to the specified value. Enclose each property name in single quotes.

Example: `sine = dsp.SineWave('Amplitude',2);`

`sine = dsp.SineWave(amp,freq,phase,Name,Value)` creates a sine wave object with the “Amplitude” on page 4-0 property set to `amp`, “Frequency” on page 4-0 property set to `freq`, “PhaseOffset” on page 4-0 property set to `phase`, and any other specified properties set to the specified values.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see *System Design in MATLAB Using System Objects*.

### Amplitude — Amplitude of sine wave

1 (default) | scalar | vector

Amplitude of the sine wave, specified as one of the following:

- scalar -- A scalar applies to all channels.
- vector -- A length- $N$  vector contains the amplitudes of the sine waves in each of the  $N$  output channels. The vector length must be the same as that specified for the “Frequency” on page 4-0 and “PhaseOffset” on page 4-0 properties.

**Tunable:** Yes

### Dependencies

This property is tunable only when you set “Method” on page 4-0 to either 'Trigonometric function' or 'Differential'.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### Frequency — Frequency of sine wave

100 (default) | scalar | vector

Frequency of the sine wave in Hz, specified as one of the following:

- scalar -- A scalar applies to all channels.
- vector -- A length- $N$  vector contains the frequencies of the sine waves in each of the  $N$  output channels. The vector length must be the same as that specified for the “Amplitude” on page 4-0 and “PhaseOffset” on page 4-0 properties.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### PhaseOffset — Phase offset of sine wave

0 (default) | scalar | vector

Phase offset of the sine wave in radians, specified as one of the following:

- scalar -- A scalar applies to all channels.
- vector -- A length- $N$  vector contains the phase offsets of the sine waves in each of the  $N$  output channels. The vector length must be the same as that specified for the “Amplitude” on page 4-0 and “Frequency” on page 4-0 properties.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### ComplexOutput — Flag that indicates whether waveform is real or complex

false (default) | true

Flag that indicates whether the waveform is real or complex, specified as either:

- `false` -- The waveform output is real.
- `true` -- The waveform output is complex.

#### Method — Method used to generate sinusoids

'Trigonometric function' (default) | 'Table lookup' | 'Differential'

Method used to generate sinusoids, specified as one of the following:

- 'Trigonometric function' -- The object computes the sinusoid by sampling the continuous-time function.
- 'Table lookup' -- The object precomputes the unique samples of every output sinusoid at the start of the simulation, and recalls the samples from memory as needed.
- 'Differential' -- The object uses an incremental algorithm. This algorithm computes the output samples based on the output values computed at the previous sample time and precomputed update terms.

#### TableOptimization — Optimize table of sine values for speed or memory

'Speed' (default) | 'Memory'

Optimize table of sine values for speed or memory, specified as either:

- 'Speed' -- The table contains  $k$  elements, where  $k$  is the number of input samples in one full period of the sine wave. The period of each sinusoid must be an integer multiple of  $1/F_s$ , where  $F_s$  is the value of the "SampleRate" on page 4-0 property value. That is, each element of the "Frequency" on page 4-0 property must be of the form  $F_s/m$ , where  $m$  is an integer greater than 1.
- 'Memory' -- The table contains  $k/4$  elements.

#### Dependencies

This property applies only when you set the Method property to 'Table lookup'.

#### SampleRate — Sample rate of output signal

1000 (default) | positive scalar

Sample rate of output signal in Hz, specified as a positive scalar.

Example: 44100

Example: 22050

#### SamplesPerFrame — Number of samples per frame

1 (default) | positive integer

Number of consecutive samples from each sinusoid to buffer into the output frame, specified as a positive integer.

Example: 1000

Example: 5000

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

#### OutputDataType — Data type of the sine wave output

'double' (default) | 'single' | 'Custom'

Data type of the sine wave output, specified as 'double', 'single', or 'Custom'.

### Fixed-Point Properties

#### CustomOutputDataType — Output word and fraction lengths

`numericType([],16)` (default) | `numericType([],32,30)`

Output word and fraction lengths, specified as an autosigned numeric type with a word length of 16.

Example: `numericType([],32,30)`

Example: `numericType([],16,15)`

### Dependencies

This property applies only when you set the “Method” on page 4-0 property to 'Table lookup' and the “OutputDataType” on page 4-0 property to 'Custom'.

## Usage

### Syntax

```
sineOut = sine()
```

### Description

`sineOut = sine()` creates the sine wave output, `sineOut`.

### Output Arguments

#### sineOut — Sine wave output

vector | matrix

Sine wave output, returned as a vector or matrix. The “SamplesPerFrame” on page 4-0 property determines the number of rows in the output matrix. If the “Frequency” on page 4-0 or the “PhaseOffset” on page 4-0 property is a vector, the length of the vector determines the number of columns (channels) in the output matrix. If the Frequency or the PhaseOffset properties is a scalar, then the number of channels in the output matrix is 1.

The “OutputDataType” on page 4-0 property sets the data type of the output.

Data Types: `single` | `double` | `fi`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Common to All System Objects

`step` Run System object algorithm

`release` Release resources and allow changes to System object property values and input characteristics



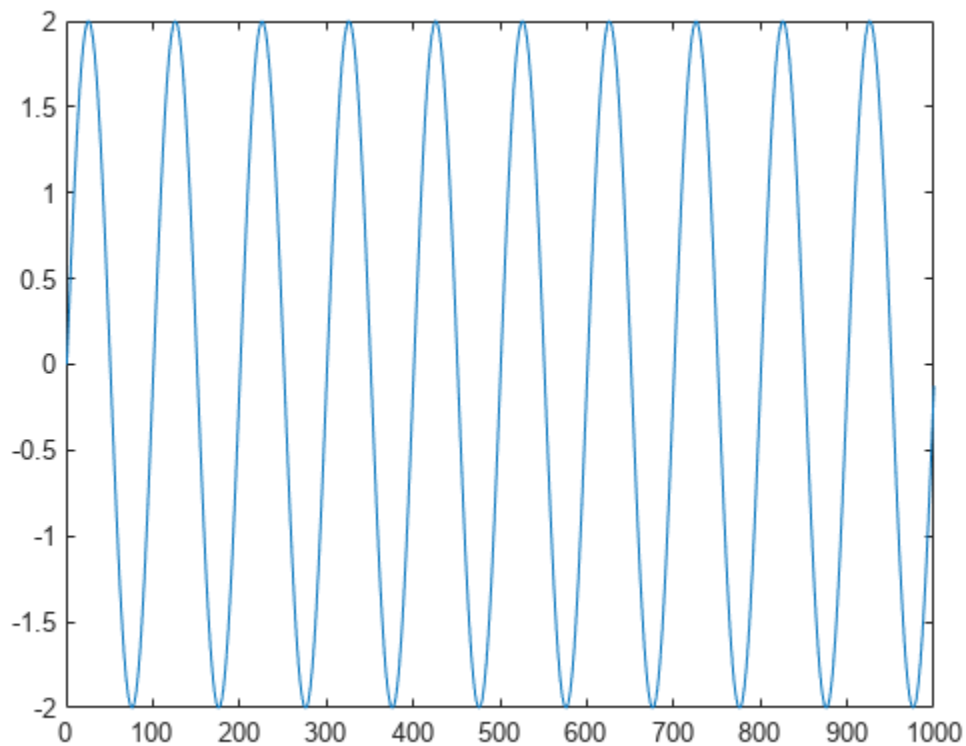
reset     Reset internal states of System object

## Examples

### Generate a Sine Wave Signal

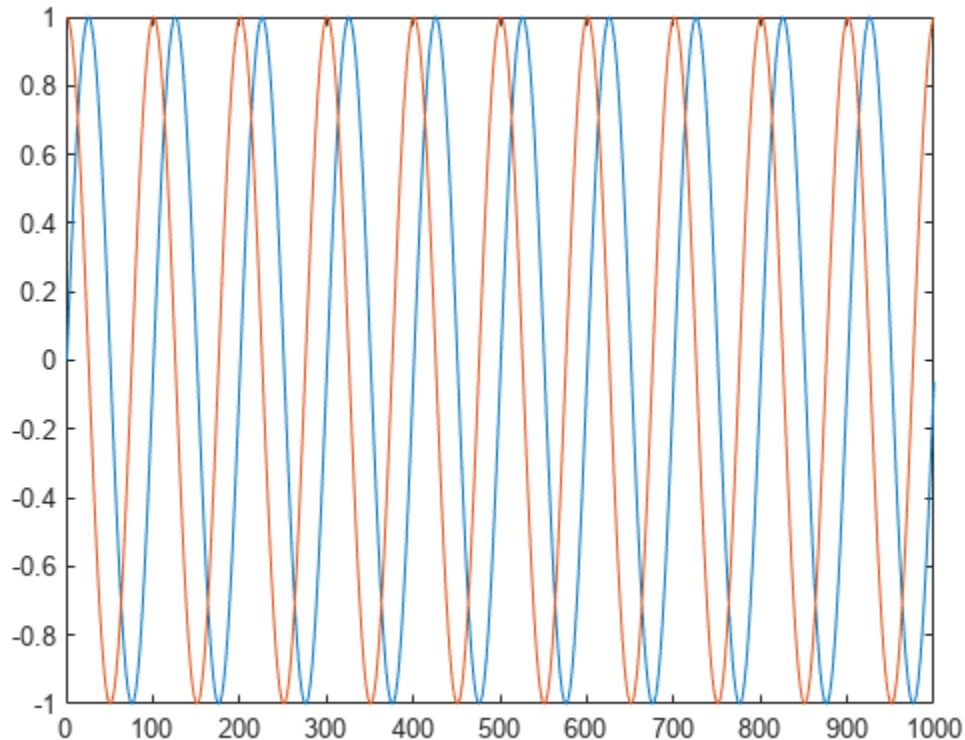
Generate a sine wave with an amplitude of 2, frequency of 10 Hz, and an initial phase of 0.

```
sine1 = dsp.SineWave(2,10);  
sine1.SamplesPerFrame = 1000;  
y = sine1();  
plot(y)
```



Generate two sine waves offset by a phase of  $\pi/2$  radians.

```
sine2 = dsp.SineWave;  
sine2.Frequency = 10;  
sine2.PhaseOffset = [0 pi/2];  
sine2.SamplesPerFrame = 1000;  
y = sine2();  
plot(y)
```



### Filter Frames of a Noisy Sine Wave Signal in MATLAB

This example shows how to lowpass filter a noisy signal in MATLAB® and visualize the original and filtered signals using a spectrum analyzer. For a Simulink® version of this example, see “Filter Frames of a Noisy Sine Wave Signal in Simulink”.

#### Specify Signal Source

The input signal is the sum of two sine waves with frequencies of 1 kHz and 10 kHz. The sampling frequency is 44.1 kHz.

```
Sine1 = dsp.SineWave('Frequency',1e3,'SampleRate',44.1e3);  
Sine2 = dsp.SineWave('Frequency',10e3,'SampleRate',44.1e3);
```

#### Create Lowpass Filter

The lowpass FIR filter, `dsp.LowpassFilter`, designs a minimum-order FIR lowpass filter using the generalized Remez FIR filter design algorithm. Set the passband frequency to 5000 Hz and the stopband frequency to 8000 Hz. The passband ripple is 0.1 dB and the stopband attenuation is 80 dB.

```
FIRLowPass = dsp.LowpassFilter('PassbandFrequency',5000,...  
    'StopbandFrequency',8000);
```

## Create Spectrum Analyzer

Set up the spectrum analyzer to compare the power spectra of the original and filtered signals. The spectrum units are dBm.

```
SpecAna = spectrumAnalyzer('PlotAsTwoSidedSpectrum',false,...
    'SampleRate',Sine1.SampleRate,...
    'ShowLegend',true, ...
    'YLimits',[-145,45]);

SpecAna.ChannelNames = {'Original noisy signal',...
    'Lowpass filtered signal'};
```

## Specify Samples per Frame

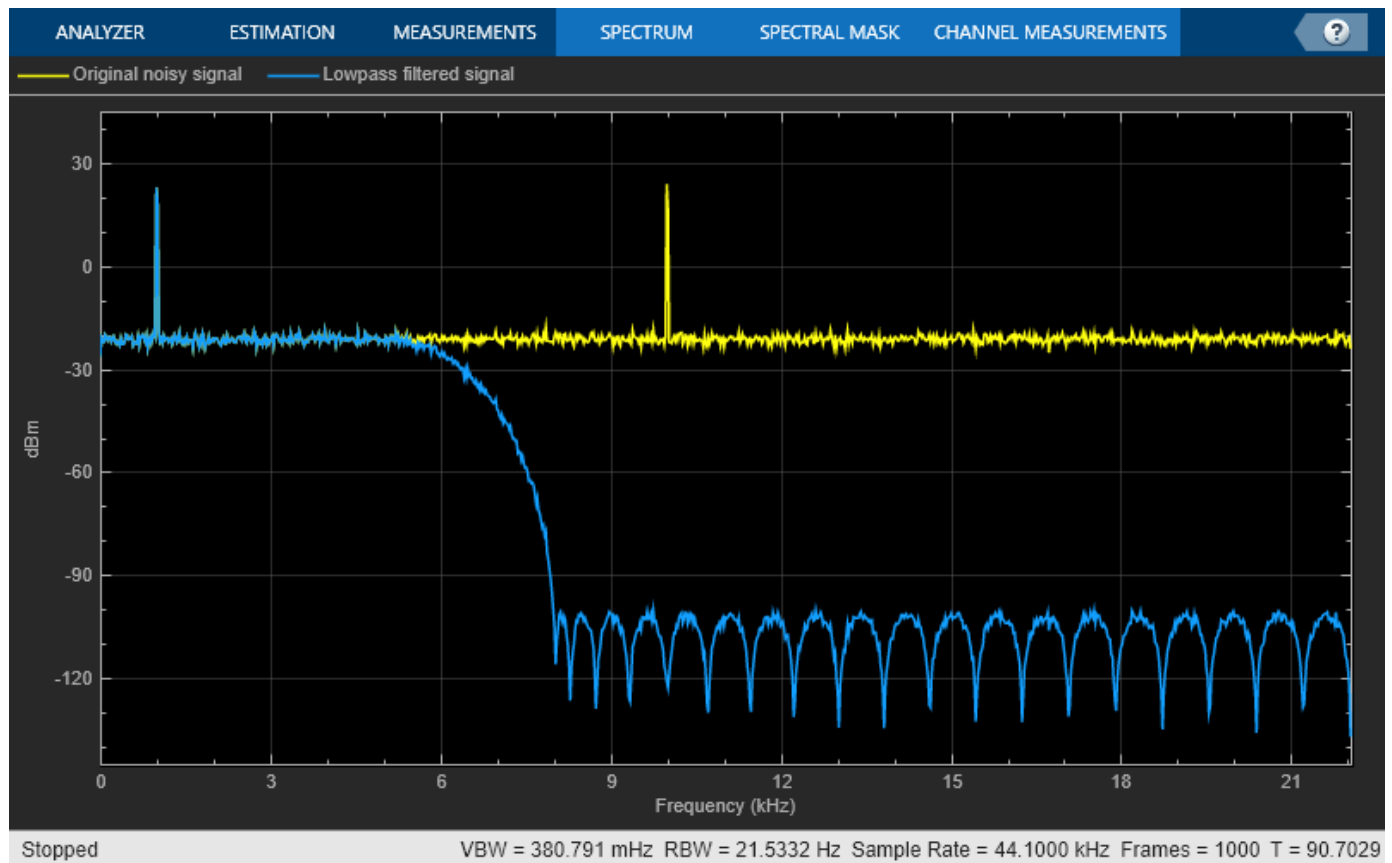
This example uses frame-based processing, where data is processed one frame at a time. Each frame of data contains sequential samples from an independent channel. Frame-based processing is advantageous for many signal processing applications because you can process multiple samples at once. By buffering your data into frames and processing multisample frames of data, you can improve the computational time of your signal processing algorithms. Set the number of samples per frame to 4000.

```
Sine1.SamplesPerFrame = 4000;
Sine2.SamplesPerFrame = 4000;
```

## Filter the Noisy Sine Wave Signal

Add zero-mean white Gaussian noise with a standard deviation of 0.1 to the sum of sine waves. Filter the result using the FIR filter. While running the simulation, the spectrum analyzer shows that frequencies above 8000 Hz in the source signal are attenuated. The resulting signal maintains the peak at 1 kHz because it falls in the passband of the lowpass filter.

```
for i = 1 : 1000
    x = Sine1()+Sine2()+0.1.*randn(Sine1.SamplesPerFrame,1);
    y = FIRLowPass(x);
    SpecAna(x,y);
end
release(SpecAna)
```



### Bandpass Filtering of Sinusoids

Bandpass filter a discrete-time sine wave signal which consists of three sinusoids at frequencies, 1 kHz, 10 kHz, and 15 kHz.

Design an FIR Equiripple bandpass filter by first creating a bandpass filter design specifications object, and then designing a filter using these specifications.

#### Design Bandpass Filter

Create a bandpass filter design specifications object using `fdesign.bandpass`.

```
bandpassSpecs = fdesign.bandpass('Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2', ...
    1/4,3/8,5/8,6/8,60,1,60);
```

List the available design methods for this object.

```
designmethods(bandpassSpecs)
```

Design Methods for class `fdesign.bandpass (Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2)`:

```
butter
cheby1
cheby2
```

```

ellip
equiripple
kaiserwin

```

To design an Equiripple filter, pick 'equiripple'.

```
bpFilter = design(bandpassSpecs, 'equiripple', 'Systemobject', true)
```

```

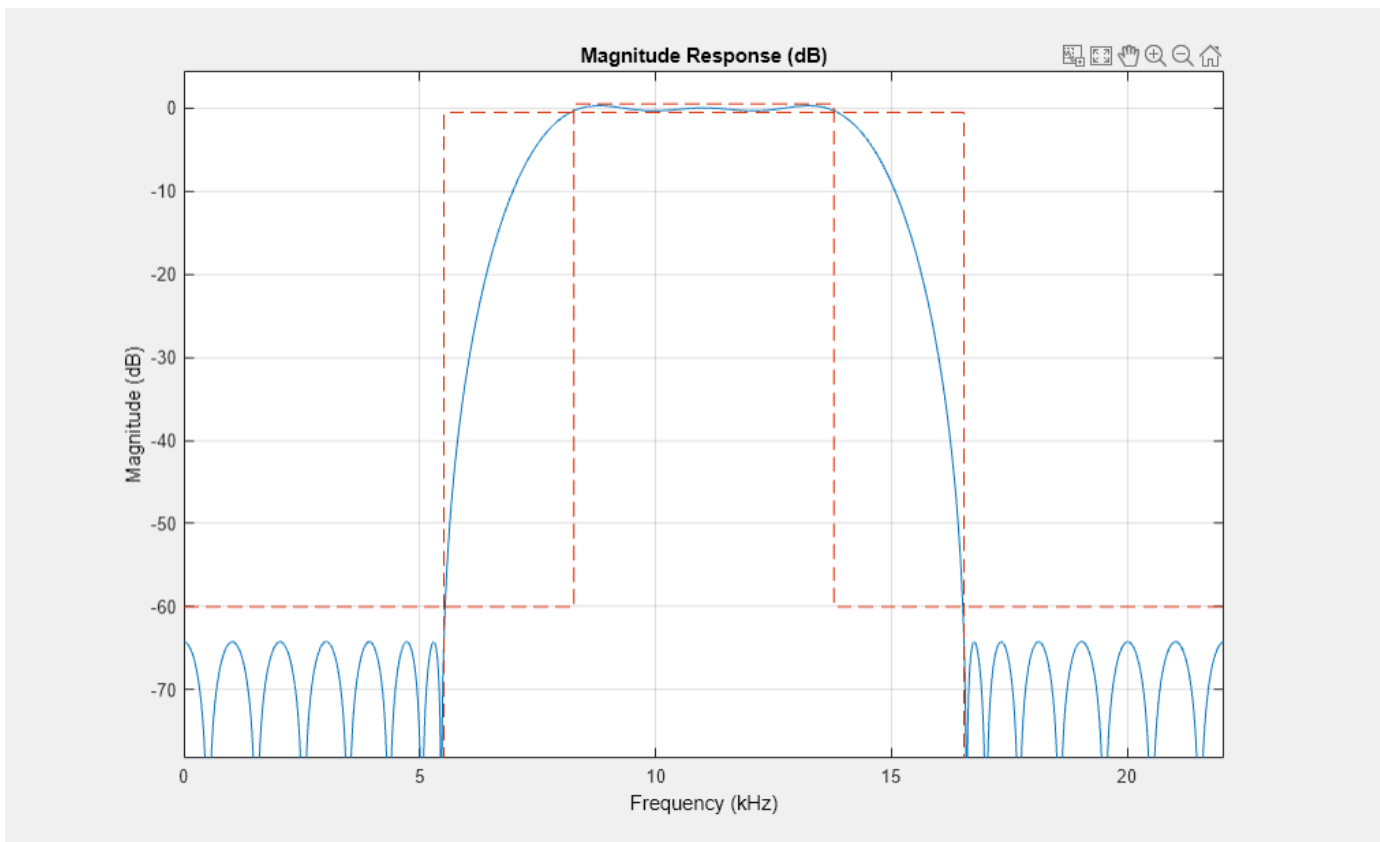
bpFilter =
  dsp.FIRFilter with properties:
    Structure: 'Direct form'
    NumeratorSource: 'Property'
    Numerator: [-0.0043 -3.0812e-15 0.0136 3.7820e-15 -0.0180 -4.2321e-15 7.1634e-04 4.09e-04]
    InitialConditions: 0

```

Show all properties

Visualize the frequency response of the designed filter.

```
fvtool(bpFilter, 'Fs', 44100)
```



### Create Sinusoidal Signal

Create a signal that is a sum of three sinusoids with frequencies at 1 kHz, 10 kHz, and 15 kHz. Initialize Spectrum Analyzer to view the original signal and the filtered signal.

```
Sine1 = dsp.SineWave('Frequency',1e3,'SampleRate',44.1e3,'SamplesPerFrame',4000);
Sine2 = dsp.SineWave('Frequency',10e3,'SampleRate',44.1e3,'SamplesPerFrame',4000);
Sine3 = dsp.SineWave('Frequency',15e3,'SampleRate',44.1e3,'SamplesPerFrame',4000);

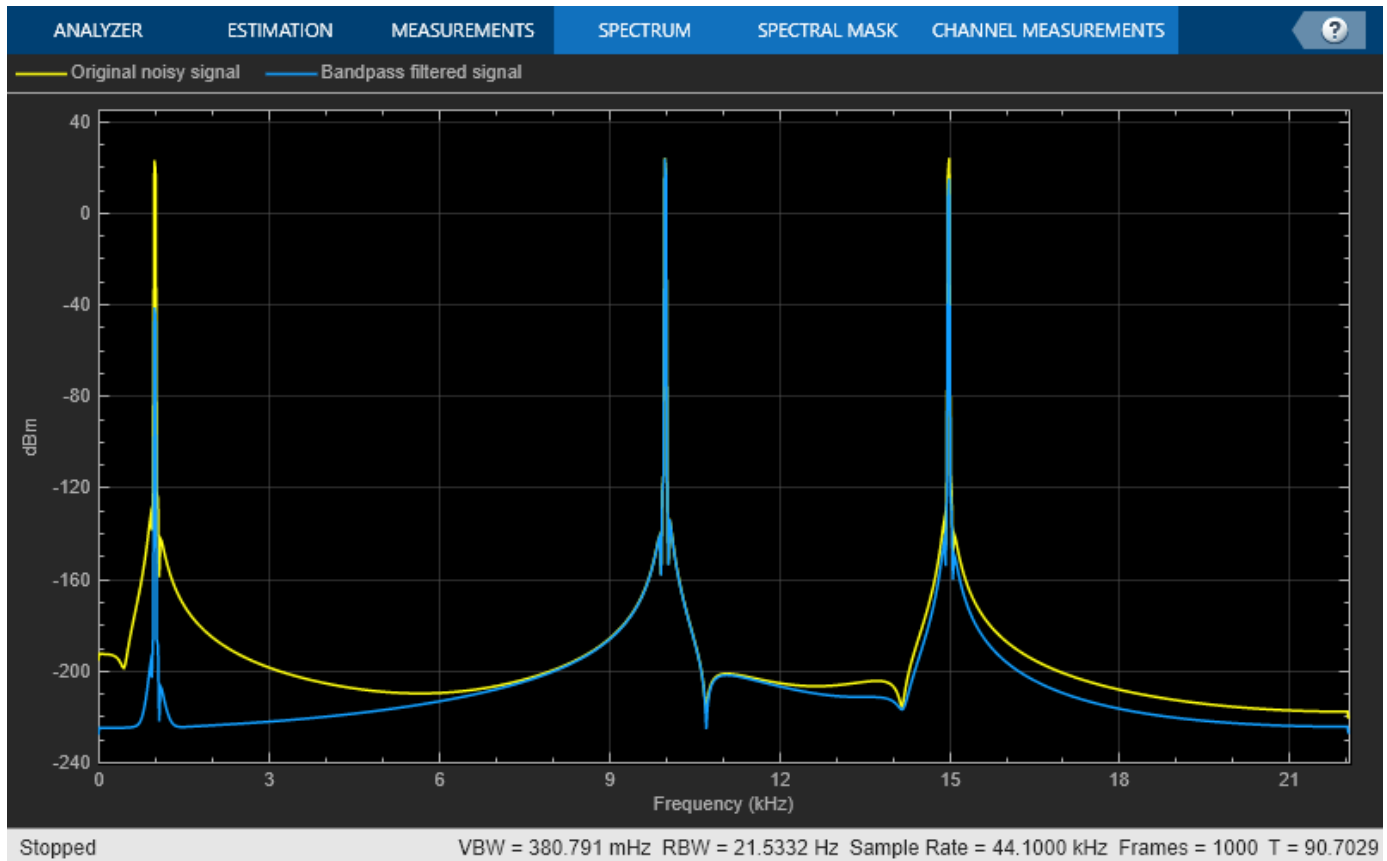
SpecAna = spectrumAnalyzer('PlotAsTwoSidedSpectrum',false, ...
    'SampleRate',Sine1.SampleRate, ...
    'ShowLegend',true, ...
    'YLimits',[-240,45]);

SpecAna.ChannelNames = {'Original noisy signal','Bandpass filtered signal'};
```

### **Filter Sinusoidal Signal**

Filter the sinusoidal signal using the bandpass filter that has been designed. View the original signal and the filtered signal in the Spectrum Analyzer. The tone at 1 kHz is filtered out and attenuated. The tone at 10 kHz is unaffected, and the tone at 15 kHz is mildly attenuated because it appears in the transition band of the filter.

```
for i = 1 : 1000
    x = Sine1()+Sine2()+Sine3();
    y = bpFilter(x);
    SpecAna(x,y);
end
release(SpecAna)
```



## More About

### Sinusoid

A real-valued, discrete-time sinusoid is defined as:

$$y(n) = A \sin(2\pi f n + \phi)$$

where  $A$  is the amplitude,  $f$  is the frequency in Hz, and  $\phi$  is the initial phase, or phase offset, in radians.

A complex sinusoid is defined as:

$$y(n) = A e^{j(2\pi f n + \phi)}$$

## Algorithms

### Trigonometric Function

The trigonometric function method computes the sinusoid in the  $i$ th channel,  $y_i$ , by sampling the continuous function

$$y_i = A_i \sin(2\pi f_i t + \phi_i) \text{ (real)}$$

or

$$y_i = A_i e^{j(2\pi f_i t + \phi_i)} \text{ (complex)}$$

with a period of  $T_s$ , where you specify  $T_s$  in the sample time.

At each sample time, the algorithm evaluates the sine function at the appropriate time value *within the first cycle* of the sinusoid. By constraining trigonometric evaluations to the first cycle of each sinusoid, the algorithm avoids the imprecision of computing the sine of very large numbers. This constraint also eliminates the possibility of discontinuity during extended operations, when an absolute time variable might overflow. This method therefore avoids the memory demands of the table lookup method at the expense of many more floating-point operations.

### Table Lookup

The table lookup method precomputes the *unique* samples of every output sinusoid at the start of the simulation, and recalls the samples from memory as needed. Because a table of finite length can only be constructed when all output sequences repeat, the method requires that the period of every sinusoid in the output be evenly divisible by the sample period. That is,  $1/(f_i T_s) = k_i$  must be an integer value for every channel  $i = 1, 2, \dots, N$ .

When the algorithm optimizes the table of sine values for **Speed**, the table constructed for each channel contains  $k_i$  elements. When the optimization is for **Memory**, the table constructed for each channel contains  $k_i/4$  elements.

For long output sequences, the table lookup method requires far fewer floating-point operations than any of the other methods. However, the method can demand considerably more memory, especially for high sample rates (long tables). This method is recommended for models that are intended to emulate or generate code for DSP hardware, which need to be optimized for execution speed.

---

**Note** The lookup table for this object is constructed from double-precision floating-point values. When you use the **Table Lookup** computation mode, the maximum amount of precision you can achieve in your output is 53 bits. Setting the word length of the output data type to values greater than 53 bits does not improve the precision of your output.

---

### Differential

The differential method uses an incremental algorithm. This algorithm computes the output samples based on the output values computed at the previous sample time (and precomputed update terms) by making use of the following identities.

$$\sin(t + T_s) = \sin(t)\cos(T_s) + \cos(t)\sin(T_s)$$

$$\cos(t + T_s) = \cos(t)\cos(T_s) - \sin(t)\sin(T_s)$$

The update equations for the sinusoid in the  $i$ th channel,  $y_i$ , can therefore be written in matrix form as

$$\begin{bmatrix} \sin\{2\pi f_i(t + T_s) + \phi_i\} \\ \cos\{2\pi f_i(t + T_s) + \phi_i\} \end{bmatrix} = \begin{bmatrix} \cos(2\pi f_i T_s) & \sin(2\pi f_i T_s) \\ -\sin(2\pi f_i T_s) & \cos(2\pi f_i T_s) \end{bmatrix} \begin{bmatrix} \sin(2\pi f_i t + \phi_i) \\ \cos(2\pi f_i t + \phi_i) \end{bmatrix}$$

where you specify  $T_s$  in the sample time. Since  $T_s$  is constant, the right-hand matrix is a constant and can be computed once at the start of the simulation. The value of  $A_i \sin[2\pi f_i(t + T_s) + \phi_i]$  is then



computed from the values of  $\sin(2\pi f_i t + \phi_i)$  and  $\cos(2\pi f_i t + \phi_i)$  by a simple matrix multiplication at each time step.

This mode offers reduced computational load, but is subject to drift over time due to cumulative quantization error. Because the method is not contingent on an absolute time value, there is no danger of discontinuity during extended operations, when an absolute time variable might overflow.

## Version History

Introduced in R2012a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This object has no tunable properties for code generation.
- See “System Objects in MATLAB Code Generation” (MATLAB Coder).

## See Also

### Objects

`dsp.Chirp` | `dsp.NCO` | `dsp.ColoredNoise`

### Topics

“Introduction to Streaming Signal Processing in MATLAB”

“Filter Frames of a Noisy Sine Wave Signal in MATLAB”

“Estimate the Power Spectrum in MATLAB”

# **dsp.SOSFilter**

**Package:** dsp

Second-order section biquadratic IIR filter structures

## **Description**

The `dsp.SOSFilter` System object implements an IIR filter structure using second-order sections (SOS).

To implement an IIR filter structure using SOS:

- 1 Create the `dsp.SOSFilter` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## **Creation**

### **Syntax**

```
sos = dsp.SOSFilter
sos = dsp.SOSFilter(num,den)
sos = dsp.SOSFilter(Name,Value)
```

### **Description**

`sos = dsp.SOSFilter` returns a biquadratic IIR filter System object, `sos`, which independently filters each channel (column) of the input over time using a specified biquadratic structure.

`sos = dsp.SOSFilter(num,den)` returns a biquadratic filter object with the `Numerator` property set to `num` and the `Denominator` property set to `den`.

`sos = dsp.SOSFilter(Name,Value)` returns a biquadratic filter object with each property set to the specified value. Enclose each property name in single quotes.

Example: `sos = dsp.SOSFilter('CoefficientSource','Input port')`

### **Properties**

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

**Structure – Filter structure**

'Direct form II transposed' (default) | 'Direct form I' | 'Direct form I transposed' | 'Direct form II'

Filter structure, specified as one of 'Direct form I', 'Direct form I transposed', 'Direct form II', or 'Direct form II transposed'.

**CoefficientSource – Source of filter coefficients**

'Property' (default) | 'Input port'

Source of the filter coefficients, specified as one of the following:

- 'Property' -- The filter coefficients are specified through the Numerator, Denominator, and ScaleValues properties.
- 'Input port' -- The numerator coefficients, denominator coefficients, and the scale values are specified as inputs to the object while running the algorithm. For more details, see “Usage” on page 4-1016.

**Numerator – Numerator coefficients of filter**

[0.0975 0.195 0.0975] (default) |  $N$ -by-3 matrix

Numerator coefficients of the filter, specified as an  $N$ -by-3 matrix, where  $N$  is the number of biquadratic sections.

The size of this property cannot be modified once you have run the System object algorithm. However, the coefficient values can change as the property is tunable.

**Tunable:** Yes

**Dependencies**

To enable this property, set the CoefficientSource property to 'Property'.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**Denominator – Denominator coefficients of filter**

[1 -0.9428 0.3333] (default) |  $N$ -by-3 matrix

Denominator coefficients of the filter, specified as an  $N$ -by-3 matrix, where  $N$  is the number of biquadratic sections.

The leading denominator coefficient is always assumed to be 1. If any other value is specified in the first column, the object ignores this value and treats it as 1.

The size of this property cannot be modified once you step through the algorithm. However, the denominator values can be modified as the property is tunable.

**Tunable:** Yes

**Dependencies**

To enable this property, set CoefficientSource property to 'Property'.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**HasScaleValues – Specify if filter has scale values for each section**

false (default) | true

Specify if the filter has scale values for each section. When set to `true`, using the `ScaleValues` property, you can specify the scale values to be applied before and after each section of the biquadratic filter.

**ScaleValues — Scale values for each biquad second-order section**

[1 1] (default) | vector

Scale values to apply before and after each section of a biquadratic filter, specified as a vector. The length of the `ScaleValues` vector must be  $N + 1$ , where  $N$  is the number of biquadratic sections. If you set this property to a scalar value, the scalar value is used as the gain value only before the first filter section. The remaining gain values are set to 1. If you set this property to a vector of  $N + 1$  values, each value is used for a separate section of the filter.

**Tunable:** Yes

**Dependencies**

This property applies only when you set the `CoefficientSource` property to `'Property'` and `HasScaleValues` property to `true`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**Fixed-Point Properties****RoundingMethod — Rounding method for fixed-point operations**

'Floor' (default) | 'Ceiling' | 'Convergent' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'

Rounding method for fixed-point operations, specified as one of the following:

- 'Floor'
- 'Ceiling'
- 'Convergent'
- 'Nearest'
- 'Round'
- 'Simplest'
- 'Zero'

For more details, see Rounding Modes.

**OverflowAction — Overflow action for fixed-point operations**

Wrap (default) | Saturate

Overflow action for fixed-point operations, specified as one of the following:

- 'Wrap' -- The object wraps the result of its fixed-point operations.
- 'Saturate' -- The object saturates the result of its fixed-point operations.

For more details on overflow actions, see Overflow Handling for fixed-point operations.

**SectionInputDataType — Section input word- and fraction-length designations**

'Same as input' (default) | `numericType` object

Section input word- and fraction-length designations, specified as either `'Same as input'` or a `numericType` object.

When specified as a `numericType` object, the data type must be signed fixed point with a power-of-two slope and zero bias.

#### **Dependencies**

This property applies only when you set the `HasScaleValues` property to `true`.

#### **SectionOutputDataType — Section output word- and fraction-length designations**

'Same as section input' (default) | `numericType` object

Section output word- and fraction-length designations, specified as either 'Same as section input' or a `numericType` object.

When specified as a `numericType` object, the data type must be signed fixed point with a power-of-two slope and zero bias.

#### **Dependencies**

This property applies only when you set the `HasScaleValues` property to `true`.

#### **NumeratorDataType — Numerator coefficients word- and fraction-length designations**

'Same word length as input' (default) | `numericType` object

Numerator coefficients word- and fraction-length designations, specified as either 'Same word length as input' or as a `numericType` object.

When specified as a `numericType` object, the data type must be signed fixed point with a power-of-two slope and zero bias. If not specified, the fraction length is determined based on the numerator coefficient values to give the best possible precision.

#### **Dependencies**

This property applies only when you set the `CoefficientSource` property to 'Property'.

#### **DenominatorDataType — Denominator coefficients word- and fraction-length designations**

'Same word length as input' (default) | `numericType` object

Denominator coefficients word- and fraction-length designations, specified as either 'Same word length as input' or as a `numericType` object.

When specified as a `numericType` object, the data type must be signed fixed point with a power-of-two slope and zero bias. If not specified, the fraction length is determined based on the denominator coefficient values to give the best possible precision.

#### **Dependencies**

This property applies only when you set the `CoefficientSource` property to 'Property'.

#### **ScaleValuesDataType — Scale values word- and fraction-length designations**

'Same word length as input' (default) | `numericType` object

Scale values word- and fraction-length designations, specified as either 'Same word length as input' or as a `numericType` object.

When specified as a `numericType` object, the data type must be signed fixed point with a power-of-two slope and zero bias. If not specified, the fraction length is determined based on the scale values to give the best possible precision.

**Dependencies**

This property applies only when you set the `CoefficientSource` property to 'Property' and `HasScaleValues` property to `true`.

**MultiplicandDataType — Multiplicand word- and fraction-length designations**

'Same as output' (default) | `numerictype` object

Multiplicand word- and fraction-length designations, specified as either 'Same as output' or as a `numerictype` object.

When specified as a `numerictype` object, the data type must be signed fixed point with a power-of-two slope and zero bias.

**Dependencies**

This property applies only when you set the `Structure` property to 'Direct form I transposed'.

**StateDataType — State word- and fraction-length designations**

'Full precision' (default) | `numerictype` object

State word- and fraction-length designations, specified as either 'Full precision' or as a `numerictype` object.

When specified as a `numerictype` object, the data type must be signed fixed point with a power-of-two slope and zero bias.

**Dependencies**

This property applies only when you set the `Structure` property to 'Direct form II'.

**DenominatorAccumulatorDataType — Denominator accumulator word- and fraction-length designations**

`numerictype(1,64,48)` (default) | `numerictype` object

Denominator accumulator word- and fraction-length designations, specified as a `numerictype` object.

**OutputDataType — Output word- and fraction-length designations**

'Full precision' (default) | `numerictype` object

Output word- and fraction-length designations, specified as either 'Full precision' or as a `numerictype` object.

When specified as a `numerictype` object, the data type must be signed fixed point with a power-of-two slope and zero bias.

**Usage****Syntax**

```
y = sos(x)
y = sos(x, num, den)
y = sos(x, num, den, g)
```

## Description

$y = \text{sos}(x)$  filters the input signal  $x$  and outputs the filtered values  $y$ . The sos filter object filters each channel (column) of the input signal independently over successive calls to the algorithm.

This syntax is valid only when the `CoefficientSource` property is set to 'Property'.

$y = \text{sos}(x, \text{num}, \text{den})$  filters the input using `num` as the numerator coefficients and `den` as the denominator coefficients of the sos filter.

This syntax is valid only when the `CoefficientSource` property is set to 'Input port' and `HasScaleValues` property is set to `false`.

$y = \text{sos}(x, \text{num}, \text{den}, g)$  specifies the scale values `g` of the sos filter.

This syntax is valid only when the `CoefficientSource` property is set to 'Input port' and `HasScaleValues` property is set to `true`.

## Input Arguments

### **x** — Data input

vector | matrix

Data input, specified as a vector or a matrix.

This object also accepts variable-size inputs. Once you have run the System object algorithm, you can change the size of each input channel, but you cannot change the number of channels.

If the input is fixed-point, it must be signed fixed point with a power-of-two slope and zero bias. When the fraction length is not specified, the object determines the fraction length based on the input data to give the best possible precision.

The data type of all inputs must be the same.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `fi`

### **num** — Numerator coefficients

$N$ -by-3 matrix

Numerator coefficients, specified as an  $N$ -by-3 matrix, where  $N$  is the number of biquadratic sections.

Once you step through the algorithm, the size of this input cannot be modified. However, the numerator coefficient values can be modified as the input is tunable.

If `num` is fixed-point, it must be signed fixed point with a power-of-two slope and zero bias. When the fraction length is not specified, the object determines the fraction length based on the numerator coefficient values to give the best possible precision.

The data type of all inputs must be the same.

The size and complexity of the `num` and `den` inputs must be the same.

**Tunable:** Yes

### Dependencies

This input applies only when you set the `CoefficientSource` property to 'Input port'.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `fi`

**den — Denominator coefficients**

*N*-by-3 matrix

Denominator coefficients of the filter, specified as an *N*-by-3 matrix, where *N* is the number of biquadratic sections.

The leading denominator coefficient is always assumed to be 1. If any other value is specified in the first column, the object ignores this value and treats it as 1.

The size of this input cannot be modified once you step through the algorithm. However, the denominator values can be modified as the input is tunable.

If **den** is fixed-point, it must be signed fixed point with a power-of-two slope and zero bias. When the fraction length is not specified, the object determines the fraction length based on the denominator coefficient values to give the best possible precision.

The data type of all inputs must be the same.

The size and complexity of the **num** and **den** inputs must be the same.

**Tunable:** Yes

**Dependencies**

This input applies only when you set the `CoefficientSource` property to `'Input port'`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `fi`

**g — Scale values for each biquad second-order section**

1-by-(*N* + 1) vector

Scale values of the biquadratic filter, specified as a 1-by-(*N*+1) vector, where *N* is the number of biquadratic filter sections.

If **g** is fixed-point, it must be signed fixed point with a power-of-two slope and zero bias. When the fraction length is not specified, the object determines the fraction length based on the scale values to give the best possible precision.

The data type of all inputs must be the same.

**Tunable:** Yes

**Dependencies**

This input applies only when you set the `CoefficientSource` property to `'Input port'` and `HasScaleValues` property to `true`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `fi`

**Output Arguments****y — Filtered output**

vector | matrix

Filtered output, returned as a vector or a matrix. The size and complexity of the output signal matches that of the input signal.



The data type of the output is determined based on the value of the `OutputDataType` property. If set to `'Full precision'`, the output data type is computed based on the signal flow diagrams shown in the Fixed-Point Conversion on page 4-1024 section. If set to a custom numeric type, the output data type is cast to the specified numeric type.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `fi`  
 Complex Number Support: Yes

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

### Specific to `dsp.SOSFilter`

|                         |   |
|-------------------------|---|
| <code>freqz</code>      | Frequency response of discrete-time filter System object          |
| <code>fvtool</code>     | Visualize frequency response of DSP filters                       |
| <code>impz</code>       | Impulse response of discrete-time filter System object            |
| <code>info</code>       | Information about filter System object                            |
| <code>coeffs</code>     | Returns the filter System object coefficients in a structure      |
| <code>cost</code>       | Estimate cost of implementing filter System object                |
| <code>scale</code>      | Scale second-order sections                                       |
| <code>scaleopts</code>  | Create an options object for second-order section scaling         |
| <code>scalecheck</code> | Check scaling of biquadratic filter                               |
| <code>cumsec</code>     | Cumulative second-order section of the biquadratic filter         |
| <code>tf</code>         | Convert discrete-time filter System object to transfer function   |
| <code>reorder</code>    | Reorder second-order sections of biquadratic filter System object |

### Common to All System Objects

|                      |  |
|----------------------|--|
| <code>step</code>    | Run System object algorithm  |
| <code>release</code> | Release resources and allow changes to System object property values and input characteristics |
| <code>reset</code>   | Reset internal states of System object   |

## Examples

### Remove High-Frequency Noise Using Biquad SOS Filter

Lowpass filter a noisy sinusoidal signal using the `dsp.SOSFilter` System object. Visualize the original and filtered signals using a spectrum analyzer.

#### Input Signal

The input signal is a sinusoidal signal with two tones, one at 1 kHz and the other at 3 kHz. The sampling frequency is 8 kHz.

```
f1 = 1000;  
f2 = 3000;  
Fs = 8000;  
sine = dsp.SineWave('Frequency', [f1, f2], ...
```

```
'SampleRate',Fs,...  
'SamplesPerFrame',1024);
```

### Create Biquad SOS Filter

Design a 10th-order lowpass Butterworth IIR filter with a cutoff frequency of 2 kHz. The numerator and denominator coefficients are extracted from the designed SOS matrix.

```
Fcutoff = 2000;  
[z,p,k] = butter(10,Fcutoff/(Fs/2));  
[s, g] = zp2sos(z,p,k);  
num = s(:,1:3);  
den = s(:,4:6);
```

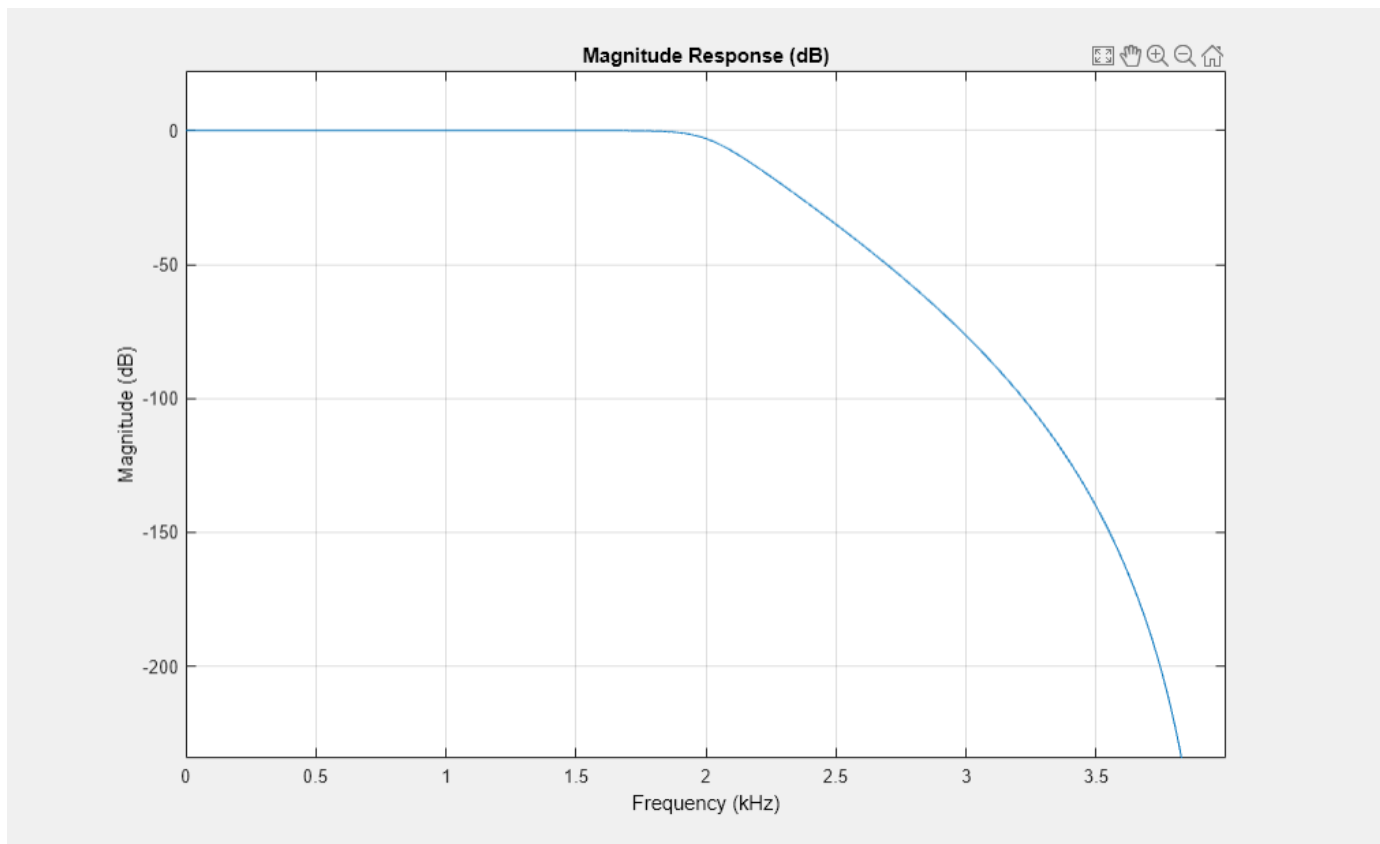
```
sosFilter = dsp.SOSFilter(num,den,...  
    'HasScaleValues',true,...  
    'ScaleValues',g)
```

```
sosFilter =  
dsp.SOSFilter with properties:  
  
    Structure: 'Direct form II transposed'  
CoefficientSource: 'Property'  
    Numerator: [5x3 double]  
    Denominator: [5x3 double]  
HasScaleValues: true  
    ScaleValues: [0.0029 1 1 1 1 1]
```

```
Show all properties
```

Visualize the frequency response of the designed SOS filter.

```
fvtool(sosFilter,'Fs',8000)
```



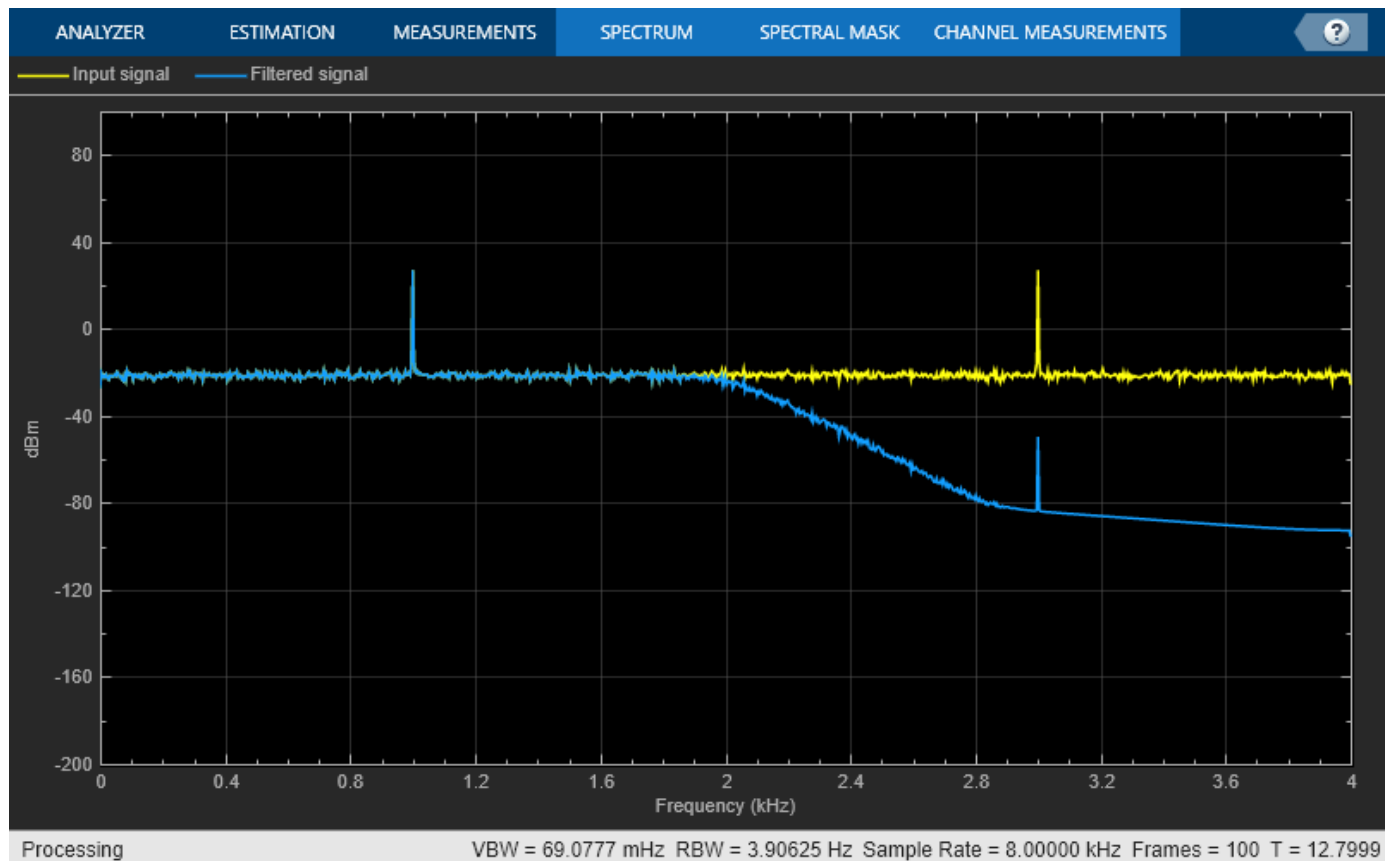
## Streaming

Add zero-mean white Gaussian noise with a standard deviation of 0.1 to the sum of sine waves. Filter the noisy sinusoidal signal with the designed SOS filter.

While running the simulation, the spectrum analyzer shows that the high-frequency tone above 2 kHz in the source signal is attenuated. The resulting signal maintains the peak at 1 kHz because it falls in the passband of the lowpass filter.

```
SA = spectrumAnalyzer(...
    'PlotAsTwoSidedSpectrum',false, ...
    'SampleRate',Fs, ...
    'ShowLegend',true,...
    'YLimits',[-200 100],...
    'ChannelNames',{'Input signal','Filtered signal'});

% Stream processing loop
for k = 1:100
    input = sum(sine(),2) + 0.1.*randn(sine.SamplesPerFrame,1);
    filteredOutput = sosFilter(input);
    SA(input,filteredOutput);
end
```



### Design a Time-Varying Lowpass IIR Filter

Design a lowpass biquadratic SOS filter with time-varying coefficients. Visualize the magnitude response of the filter using a dynamic filter visualizer.

```
dfv = dsp.DynamicFilterVisualizer('YLimits', [-120 10])
```

```
dfv =  
    dsp.DynamicFilterVisualizer with properties:
```

```
        FFTLength: 2048  
    NormalizedFrequency: 0  
        SampleRate: 44100  
    FrequencyRange: [0 22050]  
        XScale: 'Linear'  
    MagnitudeDisplay: 'Magnitude (dB)'  
    PlotAsMagnitudePhase: 0  
        PlotType: 'Line'  
    AxesScaling: 'Manual'
```

Show all properties

Create a `dsp.SOSFilter` object.

```
sosfilt = dsp.SOSFilter
```

```

sosfilt =
  dsp.SOSFilter with properties:

    Structure: 'Direct form II transposed'
    CoefficientSource: 'Property'
    Numerator: [0.0975 0.1950 0.0975]
    Denominator: [1 -0.9428 0.3333]
    HasScaleValues: false

  Show all properties

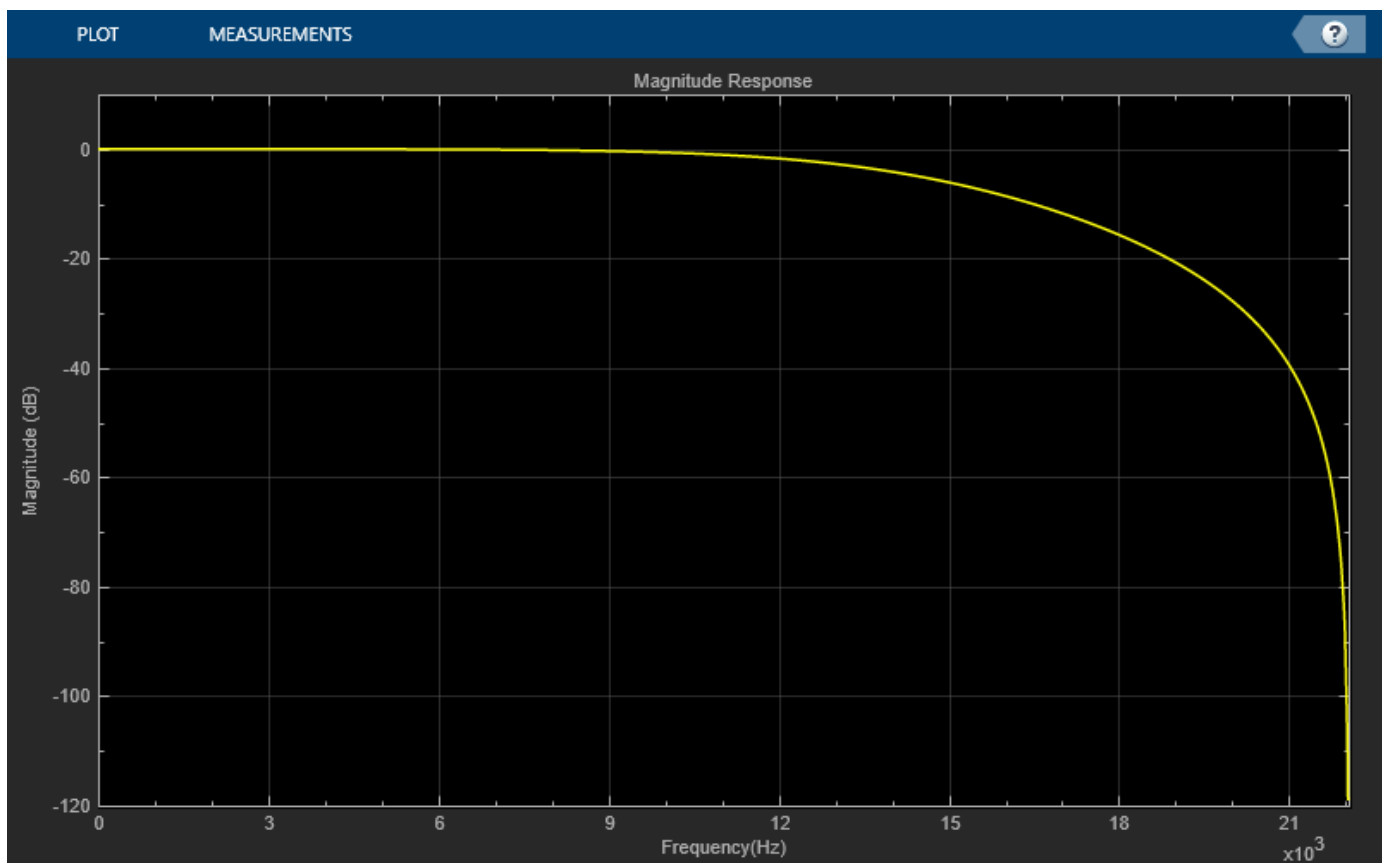
```

Use the `maxflat` function to design a lowpass maximally flat filter. Set the numerator and denominator order of the filter to 2 since the SOS filter is biquadratic. Vary the cutoff frequency in 0.001 increments and design the filter for each increment. Pass the computed coefficients to the SOS filter. Visualize the time-varying magnitude response of the SOS filter using the `dfv` object.

```

for Wn = 0.1:0.001:0.6
    [b,a] = maxflat(2,2,Wn);
    sosfilt.Numerator = b;
    sosfilt.Denominator = a;
    dfv(sosfilt)
end

```

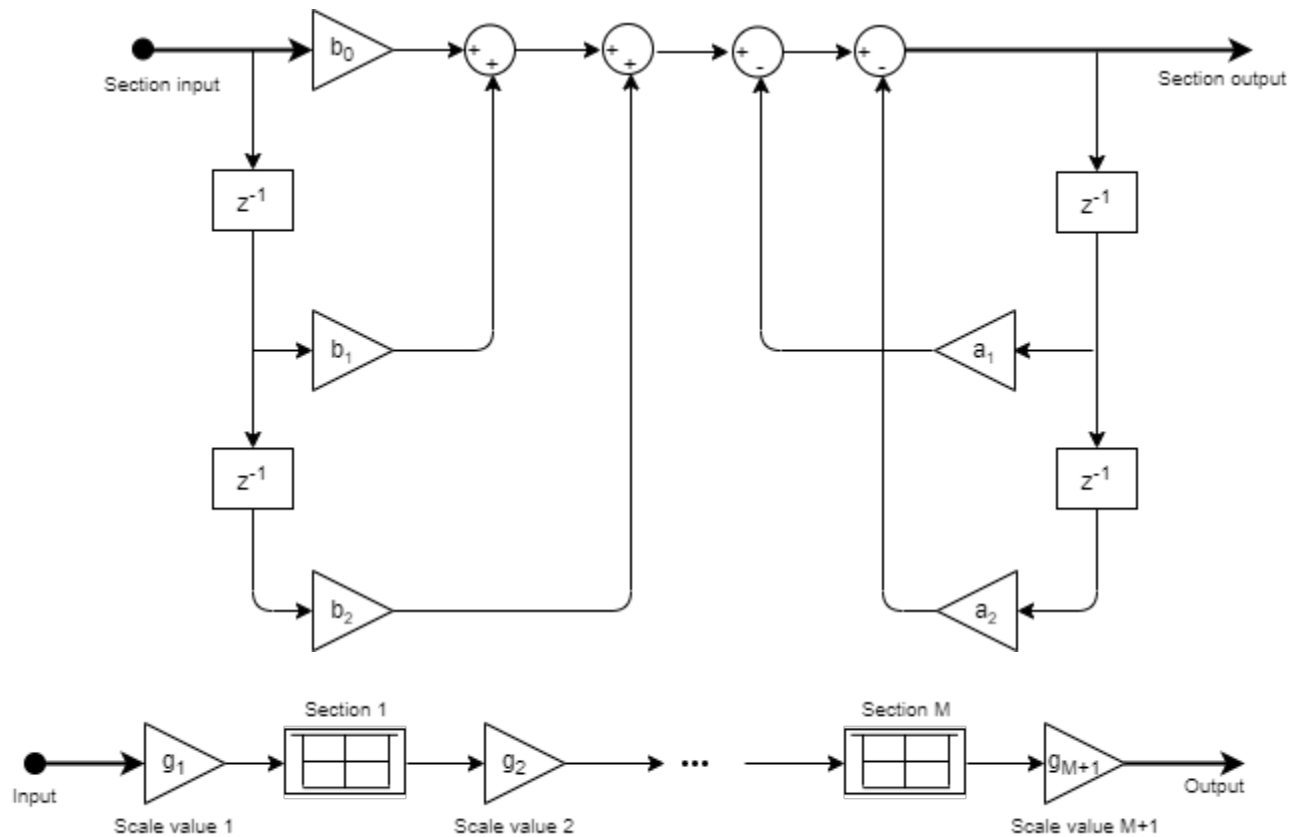


## More About

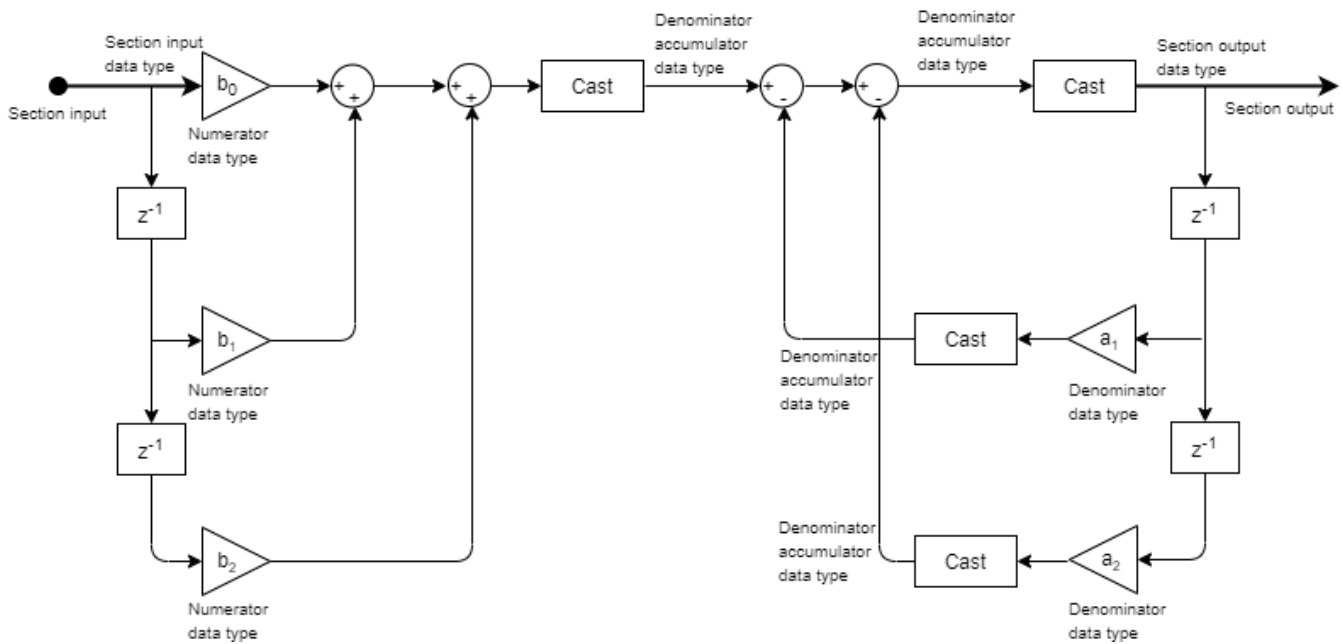
### Fixed Point

These diagrams show the data types used in the `dsp.SOSFilter` object when the input is fixed-point. For each filter structure the object supports, the data types shown in the diagrams can be set through the respective fixed-point properties of the object.

#### Direct Form I

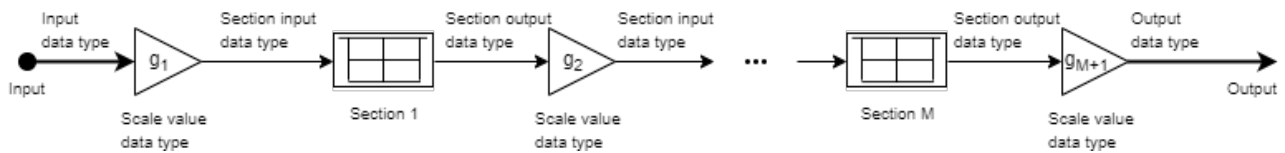


This diagram shows the data types for one section of the filter for fixed-point signals. The gain operations  $b_0$ ,  $b_1$ ,  $b_2$ ,  $a_1$ , and  $a_2$  operate in full precision.

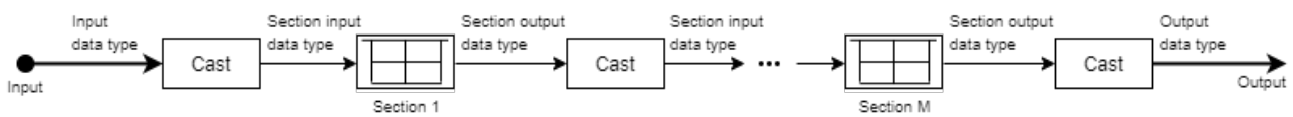


These diagrams show the fixed-point data types between filter sections.

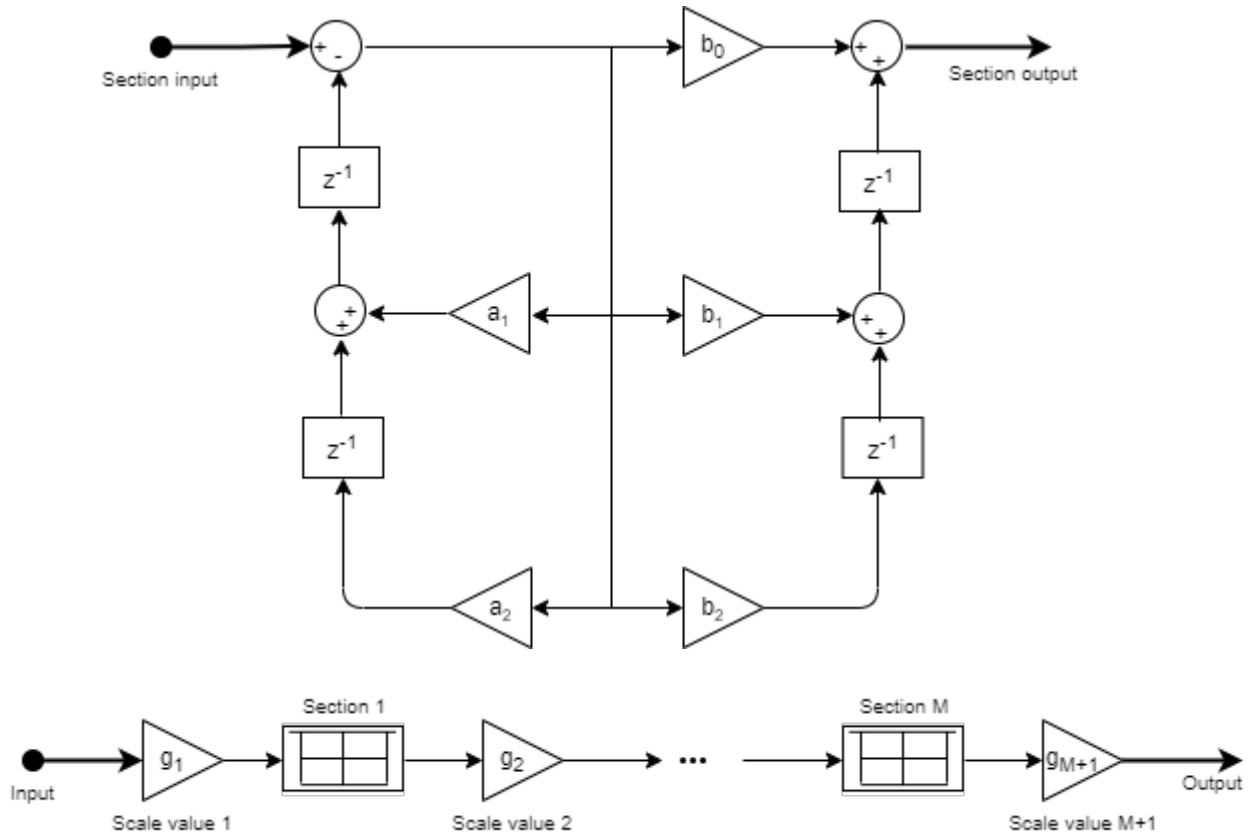
When the data is not optimized:



When you specify scale values to 1:

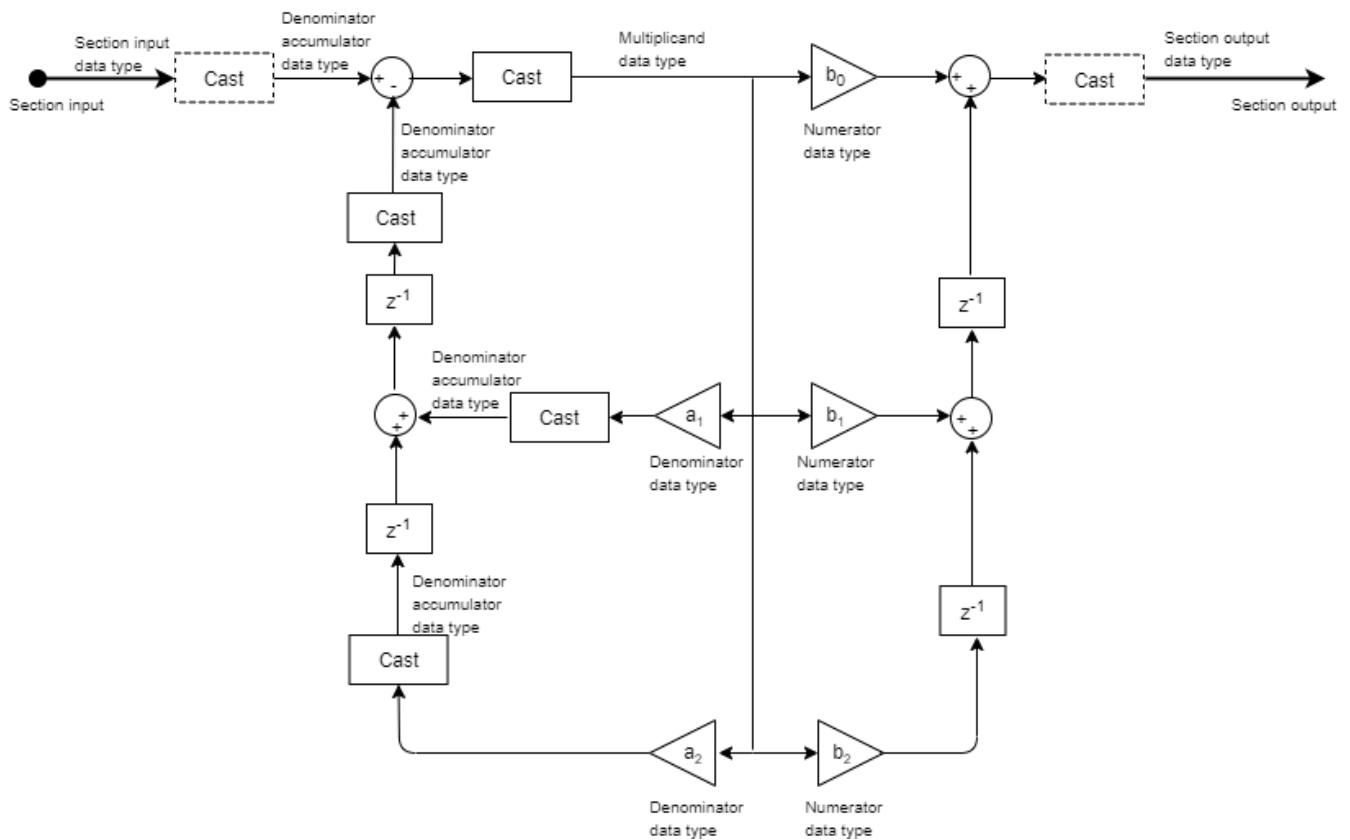


**Direct Form I Transposed**



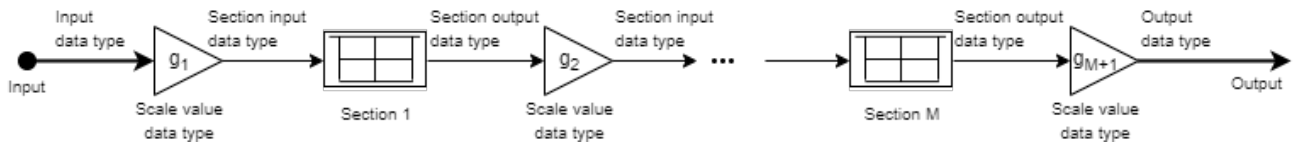
This diagram shows the data types for one section of the filter for fixed-point signals. The dashed casts are omitted when you set `HasScaleValues` to `false`. The gain operations  $b_0$ ,  $b_1$ ,  $b_2$ ,  $a_1$ , and  $a_2$  operate in full precision.



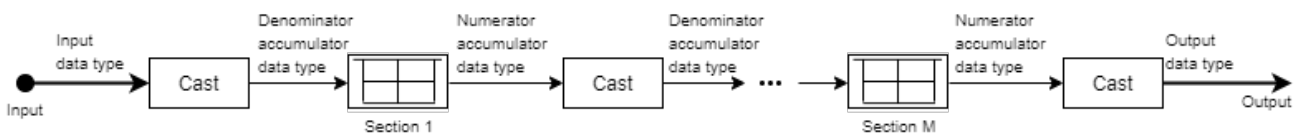


These diagrams show the fixed-point data types between filter sections.

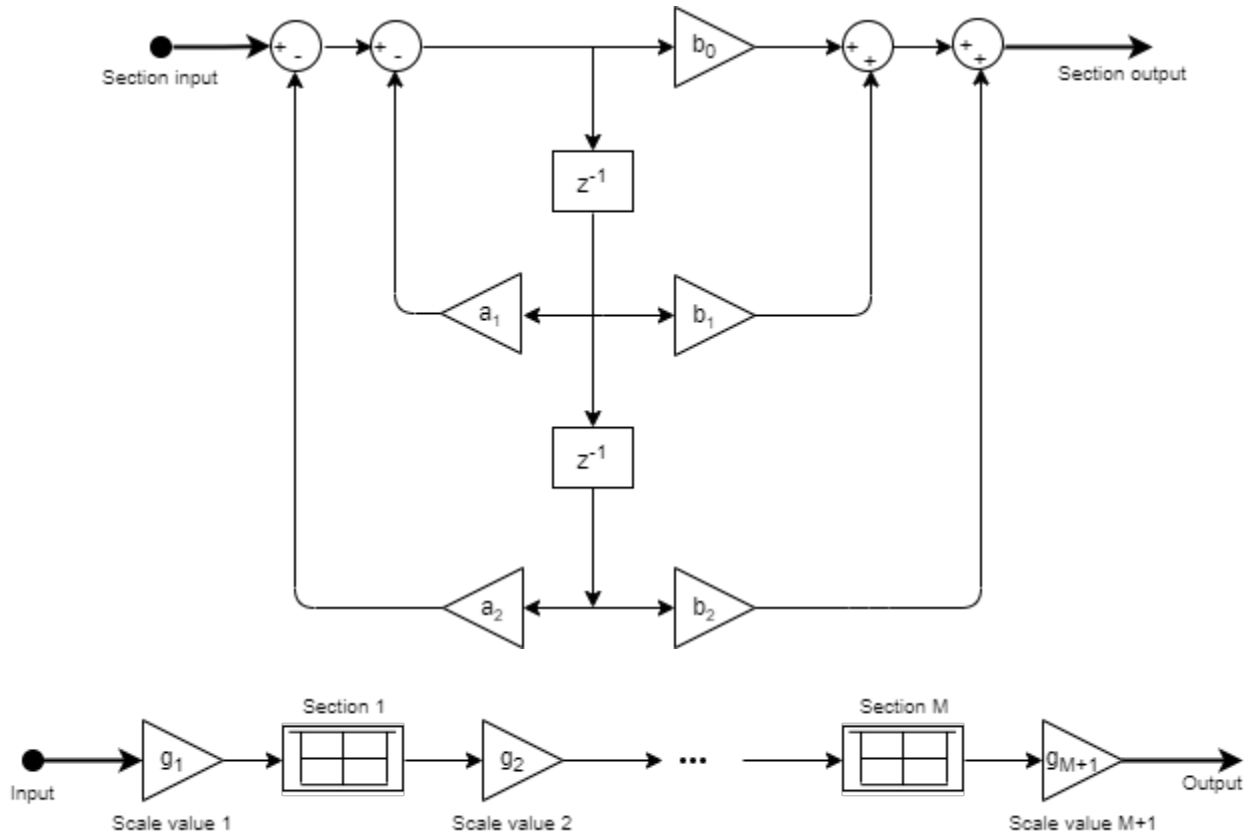
When the data is not optimized:



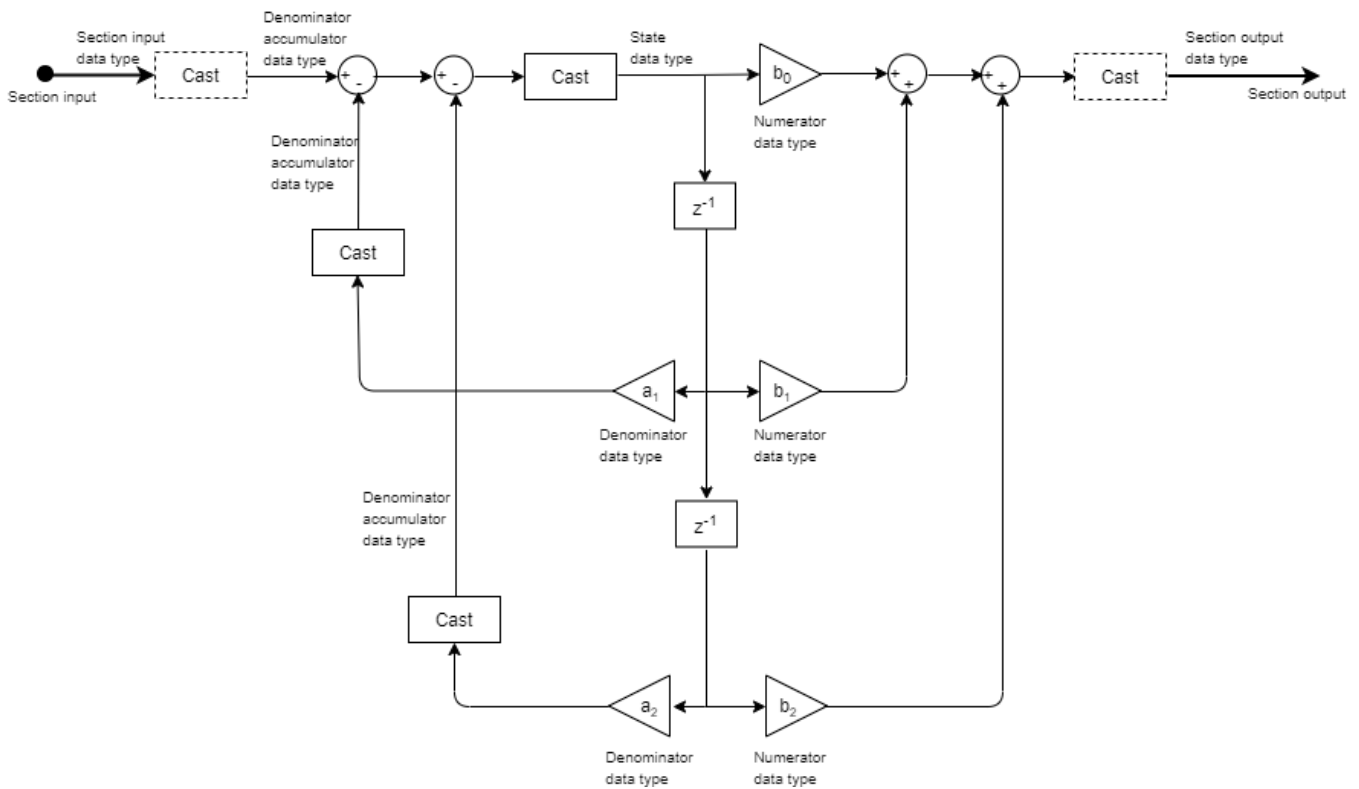
When you specify scale values to 1:



**Direct Form II**

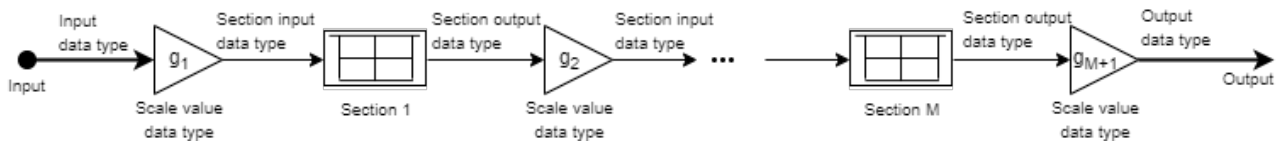


This diagram shows the data types for one section of the filter for fixed-point signals. The dashed casts are omitted when you set `HasScaleValues` to `false`. The gain operations  $b_0$ ,  $b_1$ ,  $b_2$ ,  $a_1$ , and  $a_2$  operate in full precision.

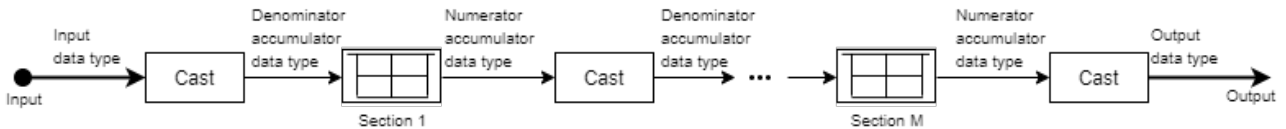


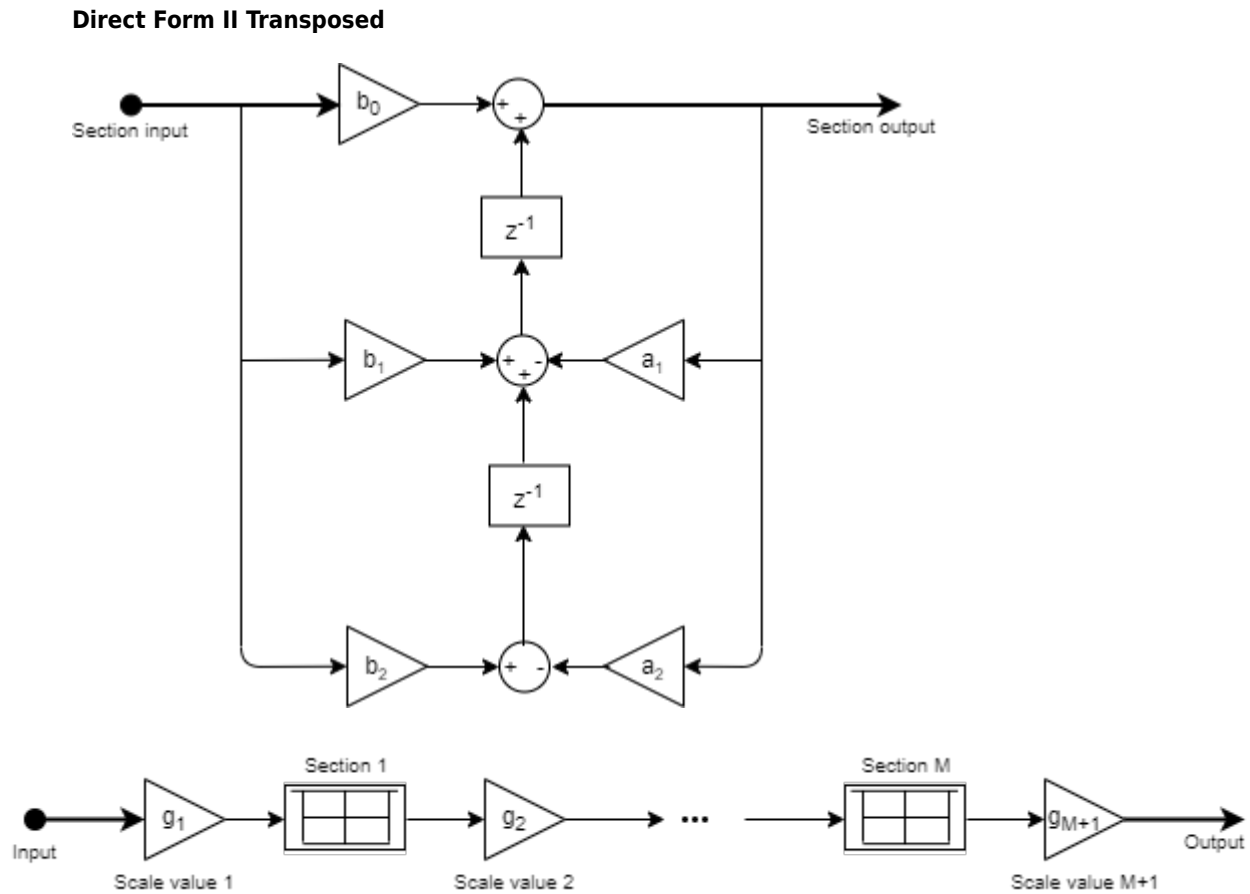
These diagrams show the fixed-point data types between filter sections.

When the data is not optimized:

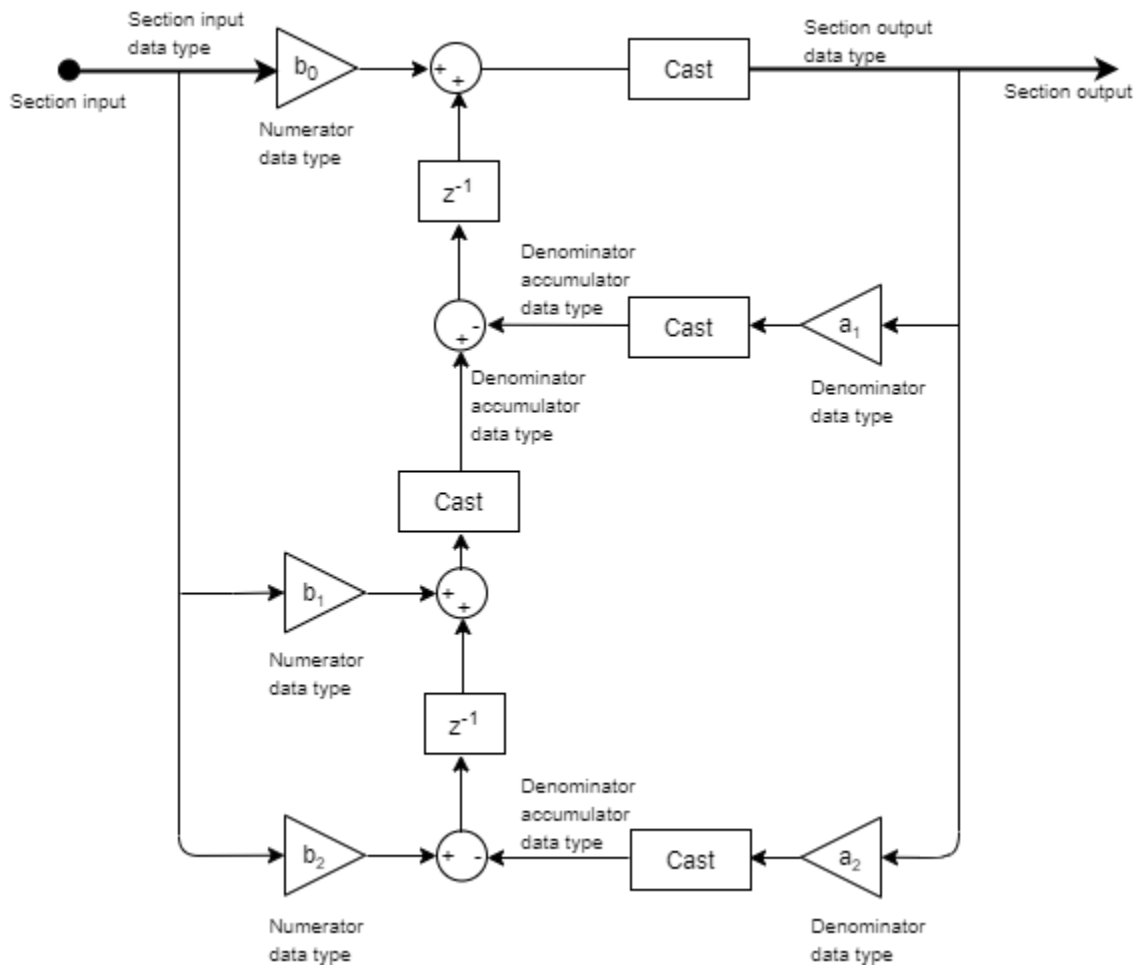


When you set scale values to 1:



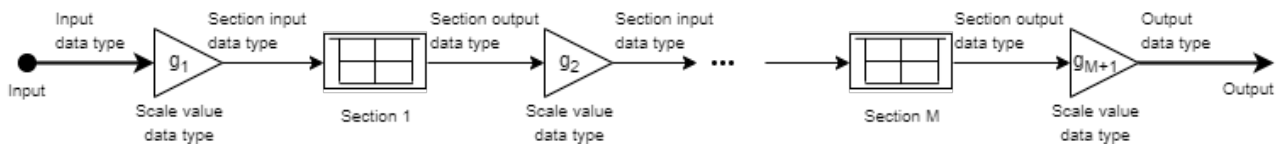


This diagram shows the data types for one section of the filter for fixed-point signals. The gain operations  $b_0$ ,  $b_1$ ,  $b_2$ ,  $a_1$ , and  $a_2$  operate in full precision. When you set `HasScaleValues` to `false`, the data type at the section output is automatically determined by the object algorithm and is not controlled by the value of the `SectionOutputDataType` property.

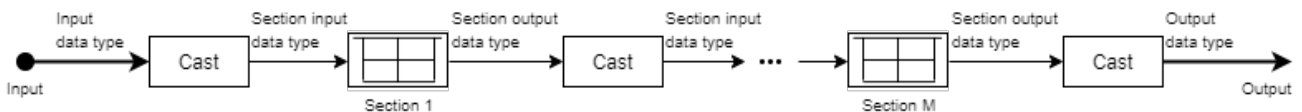


These diagrams show the fixed-point data types between filter sections.

When the data is not optimized:



When you specify scale values to 1:



## Version History

Introduced in R2020a

### **R2022b: Optimized C code generation for `dsp.SOSFilter` object on ARM Cortex-M processors**

In R2022b, you can generate optimized C code for the `dsp.SOSFilter` object on ARM Cortex-M processors under certain conditions. For more details, see “Code Generation” on page 4-1032.

### **Extended Capabilities**

#### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

The `dsp.SOSFilter` System object supports optimized C code generation on ARM Cortex-M processors under these conditions:

- Input is a single-channel real-valued signal.
- When you set `CoefficientSource` to 'Property' or 'Input port'.
- Data type of the input is `single`.
- When you set `Structure` to 'Direct form I' or 'Direct form II transposed'.
- When you set `HasScaleValues` to `false` after setting `CoefficientSource` to 'Input port'.

### **See Also**

#### **Functions**

`freqz` | `fvtool` | `impz` | `info` | `coeffs` | `cost` | `scale` | `scaleopts` | `scalecheck` | `cumsec` | `tf`

#### **Objects**

`dsp.BiquadFilter`

#### **Blocks**

Biquad Filter

# spectrumAnalyzer

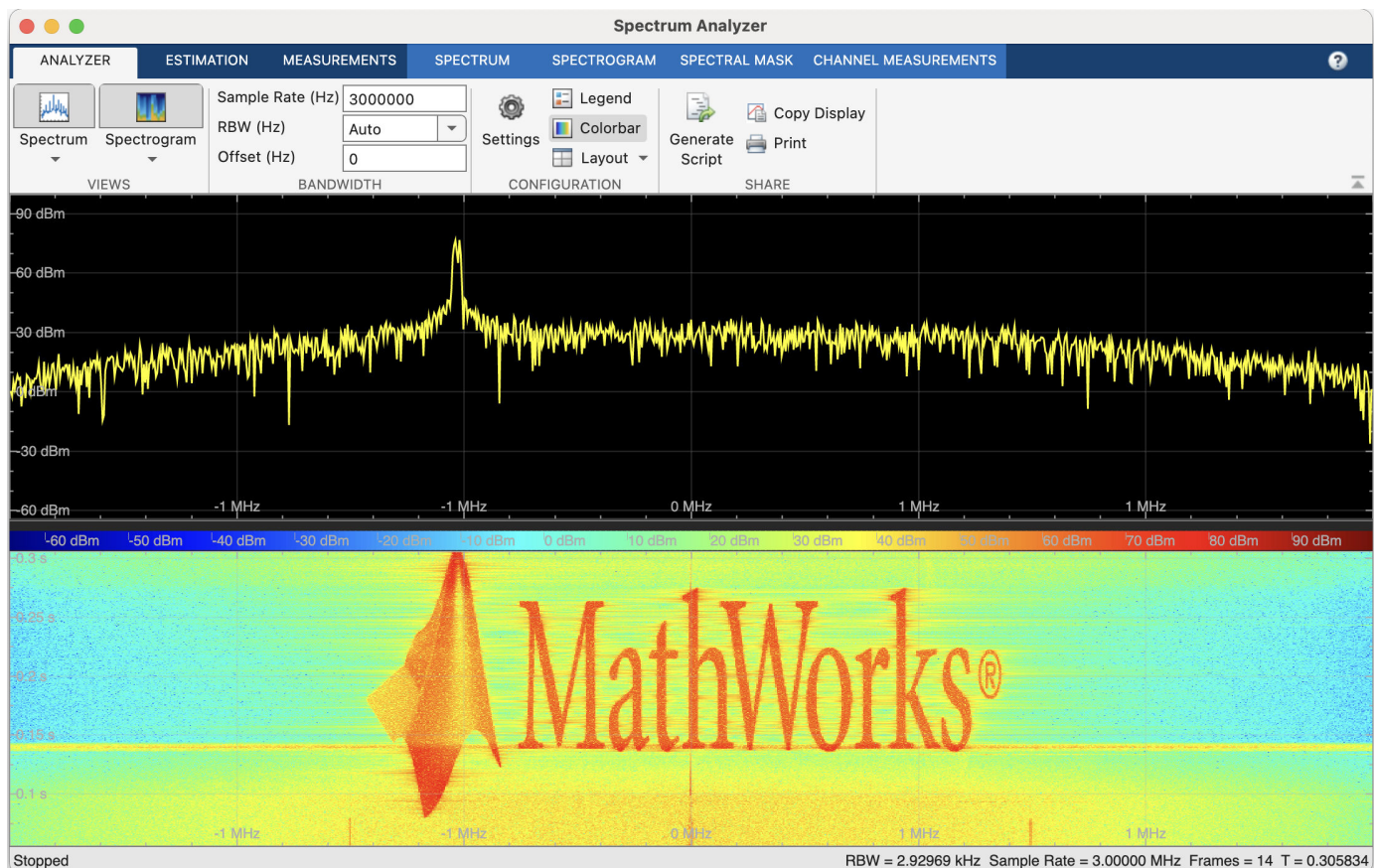
Display frequency spectrum of time-domain signals

## Description

The `spectrumAnalyzer` object displays frequency-domain signals and the frequency spectrum of time-domain signals. The scope shows the spectrum view and the spectrogram view. The object performs spectral estimation using the filter bank method and Welch's method of averaged modified periodograms. You can customize the spectrum analyzer display to show the data and the measurement information that you need. For more details, see “Algorithms” on page 4-1077. The spectrum analyzer can display the power spectrum of the signal in three units, Watts, dBm, and dBW. For more information on how to convert the power within these three units, see “Convert the Power Between Units” on page 4-1070.

To display the spectra of signals in the Spectrum Analyzer:

- 1 Create the `spectrumAnalyzer` object and set its properties.
- 2 Call the object with arguments, as if it were a function.



## Creation

### Syntax

```
scope = spectrumAnalyzer  
scope = spectrumAnalyzer(Name=Value)
```

### Description

`scope = spectrumAnalyzer` creates a `spectrumAnalyzer` object that displays the frequency spectrum of real or complex signals.

`scope = spectrumAnalyzer(Name=Value)` specifies nondefault properties for `scope` using one or more name-value arguments. For example, to display both spectrum and spectrogram, set `ViewType` to "spectrum-and-spectrogram".

## Properties

### Frequently Used

#### InputDomain — Domain of input signal

"time" (default) | "frequency"

The domain of the input signal you want to visualize, specified as "time" or "frequency". If you want to visualize time-domain signals, the Spectrum Analyzer transforms the signal to the frequency spectrum based on the algorithm you specify in the `Method` property.

#### Scope Window Use

In the **Estimation** tab on the Spectrum Analyzer toolstrip, set **Input Domain** to Time or Frequency.

Data Types: char | string

#### SpectrumType — Type of spectrum to display

"power" (default) | "power-density" | "rms"

The type of spectrum to display, specified as one of the following:

- "power" — Power spectrum.
- "power-density" — Power spectral density. The power spectral density is the squared magnitude of the spectrum normalized to a bandwidth of 1 hertz.
- "rms" — Root mean square. The root-mean-square shows the square root of the mean square. Use this option to view the frequency of voltage or current signals.

**Tunable:** Yes

#### Dependency

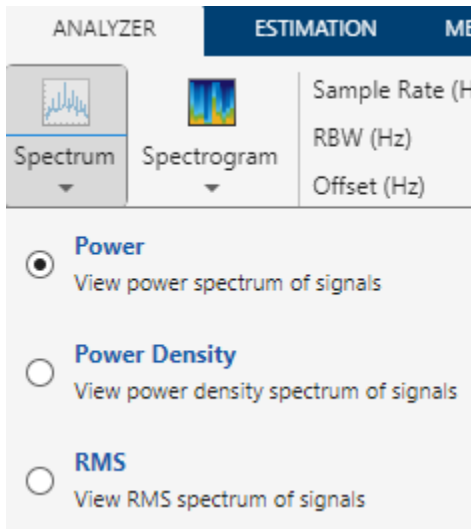
To enable this property, set `InputDomain` to "time".

#### Scope Window Use

In the **Analyzer** tab on the Spectrum Analyzer toolstrip, click the drop down arrow of **Spectrum** to select Power, Power Density, or RMS.



To enable these options, set the **Input Domain** on the **Estimation** tab to Time.



Data Types: char | string

#### ViewType — View to display

"spectrum" (default) | "spectrogram" | "spectrum-and-spectrogram"

View to display, specified as one of the following:

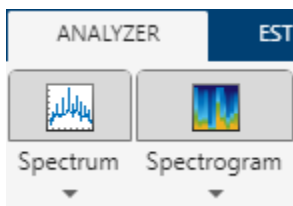
- "spectrum" — Display frequency spectrum of signals.
- "spectrogram" — Display spectrogram of signals. Spectrogram shows the frequency content over time. Each line of the spectrogram is one periodogram. Time scrolls from the top to the bottom of the display. The most recent spectrogram update is at the top of the display.
- "spectrum-and-spectrogram" — Display both spectrum and spectrogram.

To learn more about how the Spectrum Analyzer computes spectrum and spectrogram, see the "Algorithms" on page 4-1077 section.

**Tunable:** Yes

#### Scope Window Use

In the **Analyzer** tab on the Spectrum Analyzer toolstrip, select **Spectrum**, **Spectrogram**, or both.



Data Types: char | string

#### SampleRate — Sample rate of input

10000 (default) | positive scalar

The sample rate of the input in Hz, specified as a positive scalar.

**Tunable:** Yes

#### **Scope Window Use**

Click the **Analyzer** tab on the Spectrum Analyzer toolstrip. In the **Bandwidth** section, specify **Sample Rate (Hz)** to a finite scalar.

The Spectrum Analyzer shows the sample rate in the status bar at the bottom of the display.

Data Types: double

#### **Method — Spectrum estimation method**

"filter-bank" (default) | "welch"

Spectrum estimation method, specified as one of the following:

- "filter-bank" -- Use an analysis filter bank to estimate the power spectrum. Compared to Welch's method, this method has a lower noise floor, better frequency resolution, and lower spectral leakage and requires fewer samples per update.
- "welch" -- Use Welch's method of averaged modified periodograms.

For more details on these methods, see "Algorithms" on page 4-1077.

**Tunable:** Yes

#### **Dependency**

To enable this property, set InputDomain to "time".

#### **Scope Window Use**

In the **Estimation** tab of the Spectrum Analyzer toolstrip, set **Method** to Filter bank or Welch.

To enable this parameter, set **Input Domain** to Time in the **Estimation** tab.

Data Types: char | string

#### **PlotAsTwoSidedSpectrum — Option to plot a two-sided spectrum**

true (default) | false

Option to plot a two-sided spectrum, specified as one of the following:

- true — Compute and plot two-sided spectral estimates. When the input signal is complex valued, you must set this property to true.
- false — Compute and plot one-sided spectral estimates. If you set this property to false, then the input signal must be real valued.

When you set this property to false, the Spectrum Analyzer uses power-folding. The y-axis values are twice the amplitude that they would be if you were to set this property to true, except at 0 and the Nyquist frequency. A one-sided power spectral density (PSD) contains the total power of the signal in the frequency interval from DC to half the Nyquist rate. For more information, see `pwelch`.

**Tunable:** Yes

**Scope Window Use**

Click the **Spectrum** tab or the **Spectrogram** tab (if enabled) of the Spectrum Analyzer toolstrip. In the **Trace Options** section, select **Two-Sided Spectrum** to compute and plot two-sided spectral estimates.

Data Types: logical

**FrequencyScale — Scale to display frequency**

"linear" (default) | "log"

Scale to display frequency, specified as one of the following:

- "linear" — Use a linear scale to display frequencies on the x-axis.
- "log" — Use a logarithmic scale to display frequencies on the x-axis.

**Tunable:** Yes

**Dependency**

To set this property to "log", set the `PlotAsTwoSidedSpectrum` property to false.

**Scope Window Use**

Click the **Spectrum** tab or the **Spectrogram** tab (if enabled) of the Spectrum Analyzer toolstrip. In the **Scale** section, set the **Frequency Scale** to Linear or Log.

To set the **Frequency Scale** to Log, clear the **Two-Sided Spectrum** check box in the **Trace Options** section in the **Spectrum** or the **Spectrogram** tab (if enabled). If you select the **Two-Sided Spectrum** check box, then you must set the **Frequency Scale** to Linear.

Data Types: char | string

**PlotType — Plot type for normal traces**

"line" (default) | "stem"

Type of plot to use for displaying normal traces, specified as "line" or "stem". Normal traces are traces that display free-running spectral estimates.

**Tunable:** Yes

**Dependencies**

To enable this property, set:

- `ViewType` to "spectrum" or "spectrum-and-spectrogram".
- `PlotNormalTrace` to true.

**Scope Window Use**

Click the **Analyzer** tab on the Spectrum Analyzer toolstrip, navigate to the **Configuration** section and click **Settings**. In the Spectrum Analyzer Settings window that opens, under **Display and Labels**, set **Plot Type** to Line or Stem.

To enable the **Plot Type**, you must:

- Select **Spectrum** in the **Views** section of the **Analyzer** tab.

- Enable the **Normal Trace** check box in the **Trace Options** section of the **Spectrum** tab.

Data Types: char | string

### **AxesScaling** — Axes scaling mode

"auto" (default) | "manual" | "onceatstop" | "updates"

Axes scaling mode, specified as one of the following:

- "auto" — The scope scales the axes to fit the data, both during and after simulation.
- "manual" — The scope does not scale the axes automatically.
- "onceatstop" — The scope scales the axes when the simulation stops and you call the release function.
- "updates" — The scope scales the axes after a specific number of visual updates. It determines the number of updates using the AxesScalingNumUpdates property.

**Tunable:** Yes

Data Types: char | string

### **AxesScalingNumUpdates** — Number of updates before scaling

100 (default) | positive integer

Number of updates before scaling, specified as a positive integer.

**Tunable:** Yes

#### **Dependency**

To enable this property, set AxesScaling to "updates".

Data Types: double

#### **Advanced**

### **RBWSource** — Source of resolution bandwidth value

"auto" (default) | "property"

The source of the resolution bandwidth (RBW), specified as "auto" or "property".

- "auto" — The Spectrum Analyzer adjusts the spectral estimation resolution to ensure that there are 1024 RBW intervals over the defined frequency span.
- "property" — Specify the resolution bandwidth directly using the RBW property.

**Tunable:** Yes

#### **Scope Window Use**

Click the **Analyzer** tab on the Spectrum Analyzer toolstrip. In the **Bandwidth** section, set **RBW (Hz)** to either **Auto** or a positive scalar.

Data Types: char | string

### **RBW** — Resolution bandwidth

9.76 (default) | positive scalar

Resolution bandwidth (RBW) in Hz, specified as a positive scalar. Specify the value to ensure that there are at least two RBW intervals over the specified frequency span. The ratio of the overall span to RBW satisfies this condition:

$$\frac{\text{span}}{\text{RBW}} > 2$$

Specify the overall span based on how you set the `FrequencySpan` property.

RBW controls the spectral resolution of the displayed signal. The RBW value determines the spacing between frequencies that can be resolved. A smaller value gives a higher spectral resolution and lowers the noise floor. That is, the Spectrum Analyzer can resolve frequencies that are closer to each other. However, this comes at the cost of a longer sweep time.

**Tunable:** Yes

**Dependency**

To enable this property, set `RBWSource` to "property".

**Scope Window Use**

Click the **Analyzer** tab on the Spectrum Analyzer toolstrip. In the **Bandwidth** section, set **RBW (Hz)** to a positive scalar.

The Spectrum Analyzer shows the value of RBW in the status bar at the bottom of the display.

Data Types: double

**FilterSharpness — Sharpness of lowpass filter**

0.3 (default) | nonnegative scalar in the range [0,1]

Sharpness of the prototype lowpass filter, specified as a real nonnegative scalar in the range [0,1].

Increasing the filter sharpness decreases the spectral leakage and gives a more accurate power reading.

**Tunable:** Yes

**Dependencies**

To enable this property, set `Method` to "filter-bank".

**Scope Window Use**

Click the **Estimation** tab on the Spectrum Analyzer toolstrip. In the **Frequency Resolution** section, move the **Sharpness** slider.

To enable this parameter, set **Input Domain** to Time in the **Estimation** tab.

Data Types: double

**FrequencyVectorSource — Source of frequency vector**

"auto" (default) | "property"

Source of the frequency vector, specified as one of the following:

- "auto" — The Spectrum Analyzer computes the frequency vector based on the frame size of the input signal and the specified sample rate.

- "property" — Enter a custom vector in the FrequencyVector property.

**Tunable:** Yes

**Dependency**

To enable this property, set InputDomain to "frequency".

**Scope Window Use**

Click the **Estimation** tab on the Spectrum Analyzer toolstrip. In the **Domain** section, set **Frequency (Hz)** to either Auto or a monotonically increasing vector of length equal to the input signal frame size.

To enable the **Frequency (Hz)**, set **Input Domain** to Frequency.

Data Types: char | string

**FrequencyVector — Custom frequency vector**

[-5000 5000] (default) | monotonically increasing vector

Custom frequency vector, specified as a monotonically increasing vector. This vector determines the x-axis of the display. The vector must be monotonically increasing and must have the same length as the input signal frame size.

**Tunable:** Yes

**Dependency**

To enable this property, set:

- InputDomain to "frequency".
- FrequencyVectorSource to "property".

**Scope Window Use**

Click the **Estimation** tab on the Spectrum Analyzer toolstrip. In the **Domain** section, set **Frequency (Hz)** to either Auto or a monotonically increasing vector of length equal to the input signal frame size.

To enable the **Frequency (Hz)**, set **Input Domain** to Frequency.

Data Types: double

**FrequencySpan — Frequency span mode**

"full" (default) | "span-and-center-frequency" | "start-and-stop-frequencies"

Frequency span mode, specified as one of the following:

- "full" -- The Spectrum Analyzer computes and plots the spectrum over the entire "Nyquist Frequency Interval" on page 4-1082.
- "span-and-center-frequency" -- The Spectrum Analyzer computes and plots the spectrum over the interval specified by the Span and CenterFrequency properties.
- "start-and-stop-frequencies" -- The Spectrum Analyzer computes and plots the spectrum over the interval specified by the StartFrequency and StopFrequency properties.

**Tunable:** Yes

**Dependency**

To enable this property, set `InputDomain` to "time".

**Scope Window Use**

Click the **Estimation** tab on the Spectrum Analyzer toolstrip. In the **Frequency Options** section, set **Frequency Span** to Full, Span and Center Frequency, or Start and Stop Frequencies.

To enable the **Frequency Span**, set **Input Domain** to Time.

Data Types: char | string

**Span — Frequency span to compute spectrum**

10e3 (default) | real positive scalar

Frequency span over which the Spectrum Analyzer computes and plots the spectrum, specified as a positive scalar in Hz. The overall span, defined by this property and the `CenterFrequency` property, must fall within the "Nyquist Frequency Interval" on page 4-1082.

**Tunable:** Yes

**Dependency**

To enable this property, set:

- `InputDomain` to "time"
- `FrequencySpan` to "span-and-center-frequency"

**Scope Window Use**

Click the **Estimation** tab on the Spectrum Analyzer toolstrip. In the **Frequency Options** section, set **Frequency Span** to Span and Center Frequency and **Span (Hz)** to a positive scalar.

To enable the **Frequency Span**, set **Input Domain** to Time.

Data Types: double

**CenterFrequency — Center of frequency span**

0 (default) | real scalar

Center of frequency span over which the Spectrum Analyzer computes and plots the spectrum, specified as a real scalar in Hz. The overall frequency span, defined by `Span` and this property, must fall within the "Nyquist Frequency Interval" on page 4-1082.

**Tunable:** Yes

**Dependency**

To enable this property, set:

- `InputDomain` to "time"
- `FrequencySpan` to "span-and-center-frequency"

**Scope Window Use**

Click the **Estimation** tab on the Spectrum Analyzer toolstrip. In the **Frequency Options** section, set **Frequency Span** to Span and Center Frequency and **Center Frequency (Hz)** to a real scalar.

To enable the **Frequency Span**, set **Input Domain** to Time.

Data Types: double

### **StartFrequency — Starting frequency in frequency interval**

-5e3 (default) | real scalar

Starting frequency in the frequency interval over which the Spectrum Analyzer computes and plots the spectrum, specified as a real scalar in Hz. The overall span, which is defined by this property and **StopFrequency**, must fall within the “Nyquist Frequency Interval” on page 4-1082.

**Tunable:** Yes

#### **Dependency**

To enable this property, set:

- **InputDomain** to "time"
- **FrequencySpan** to "start-and-stop-frequencies"

#### **Scope Window Use**

Click the **Estimation** tab on the Spectrum Analyzer toolstrip. In the **Frequency Options** section, set **Frequency Span** to Start and Stop Frequencies and **Start Frequency (Hz)** to a real scalar.

To enable the **Frequency Span**, set **Input Domain** to Time.

Data Types: double

### **StopFrequency — Ending frequency in frequency interval**

5e3 (default) | real scalar

Ending frequency in the frequency interval over which the Spectrum Analyzer computes and plots the spectrum, specified as a real scalar in Hz. The overall span, which is defined by this property and the **StartFrequency** property, must fall within the “Nyquist Frequency Interval” on page 4-1082.

**Tunable:** Yes

#### **Dependency**

To enable this property, set:

- **InputDomain** to "time"
- **FrequencySpan** to "start-and-stop-frequencies"

#### **Scope Window Use**

Click the **Estimation** tab on the Spectrum Analyzer toolstrip. In the **Frequency Options** section, set **Frequency Span** to Start and Stop Frequencies and **Stop Frequency (Hz)** to a real scalar.

To enable the **Frequency Span**, set **Input Domain** to Time.

Data Types: double

### **OverlapPercent — Percentage of overlap**

0 (default) | scalar in the range [0 100)



Percentage of overlap between the previous and current buffered data segments, specified as a scalar in the range [0 100). The overlap creates a window segment that the object uses to compute a spectral estimate.

**Tunable:** Yes

**Dependency**

To enable this property, set:

- InputDomain to "time"
- Method to "welch"

**Scope Window Use**

Click the **Estimation** tab on the Spectrum Analyzer toolstrip. In the **Window Options** section, set the **Overlap (%)**.

To enable the **Overlap (%)**, set **Input Domain** to Time and **Method** to Welch in the **Estimation** tab on the Spectrum Analyzer toolstrip.

Data Types: double

**Window — Window function**

"hann" (default) | "blackman-harris" | "chebyshev" | "custom" | "flat-top" | "hamming" | "kaiser" | "rectangular"

Window function, specified as one of the following preset windows. For more information on a window option, click the link to the corresponding function.

| Window Option     | Corresponding Signal Processing Toolbox Function |
|-------------------|--|
| "hann"            | hann   |
| "blackman-harris" | blackmanharris                                   |
| "chebyshev"       | chebwin  |
| "flat-top"        | flattopwin                                       |
| "hamming"         | hamming  |
| "kaiser"          | kaiser   |
| "rectangular"     | rectwin  |

To use your own spectral estimation window, set this property to "custom" and specify a custom window function in the CustomWindow property.

**Tunable:** Yes

**Dependency**

To enable this property, set:

- InputDomain to "time"
- Method to "welch"

**Scope Window Use**

Click the **Estimation** tab on the Spectrum Analyzer toolstrip. In the **Window Options** section, set the **Window**.

To enable the **Window**, set **Input Domain** to Time and **Method** to Welch in the **Estimation** tab on the Spectrum Analyzer toolstrip.

Data Types: char | string

**CustomWindow — Name of custom window function**

"hann" (default) | character array | string scalar

Name of custom window function, specified as a character array or string scalar. The custom window function name must be on the MATLAB path. Use this property to customize a window function using additional properties available with the Signal Processing Toolbox.

**Tunable:** Yes

**Example:**

Define and use a custom window function.

```
function w = my_hann(L)
    w = hann(L, "periodic")
end

scope.Window = "custom";
scope.CustomWindow = "my_hann"
```

**Dependency**

To use this property, set Window to "custom".

**Scope Window Use**

Click the **Estimation** tab on the Spectrum Analyzer toolstrip. In the **Window Options** section, for the **Window**, enter a custom window function name.

Data Types: char | string

**SidelobeAttenuation — Sidelobe attenuation of window**

60 (default) | scalar greater than or equal to 45

Sidelobe attenuation of the window in decibels (dB), specified as a positive scalar greater than or equal to 45.

**Tunable:** Yes

**Dependency**

To enable this property, set Window to "chebyshev" or "kaiser".

**Scope Window Use**

Click the **Estimation** tab on the Spectrum Analyzer toolstrip. In the **Window Options** section, set the **Attenuation (dB)**.

To enable the **Attenuation (dB)**, set:

- **Input Domain** to Time
- **Method** to Welch
- **Window** to either Chebyshev or Kaiser in the **Estimation** tab on the Spectrum Analyzer toolstrip.

Data Types: double

### **AveragingMethod — Averaging method**

"vbw" (default) | "exponential"

Averaging method, specified as one of the following:

- "vbw" — Video bandwidth method. The object uses a lowpass filter to smooth the trace and decrease noise. Use the VBWSource and VBW properties to specify the VBW value.
- "exponential" — Weighted average of samples. The object computes the average over samples weighted by an exponentially decaying forgetting factor. Use the ForgettingFactor property to specify the weighted forgetting factor.

For more information on averaging methods, see "Averaging Method" on page 4-1085.

**Tunable:** Yes

#### **Dependency**

To enable this property, set InputDomain to "time".

#### **Scope Window Use**

Click the **Estimation** tab on the Spectrum Analyzer toolstrip. In the **Averaging** section, set **Averaging Method** to VBW or Exponential.

To enable the **Averaging Method**, set **Input Domain** to Time.

Data Types: char | string

### **ForgettingFactor — Forgetting factor of weighted average method**

0.9 (default) | scalar in the range [0,1]

Forgetting factor of the exponential weighted averaging method, specified as a scalar in the range [0,1].

**Tunable:** Yes

#### **Dependency**

To enable this property, set:

- InputDomain to "time".
- AveragingMethod to "exponential"

#### **Scope Window Use**

Click the **Estimation** tab on the Spectrum Analyzer toolstrip. In the **Averaging** section, adjust the slider for **Forgetting Factor**.

To enable the **Forgetting Factor**, set **Input Domain** to Time and **Averaging Method** to Exponential.

Data Types: double

**VBWSource — Source of video bandwidth**

"auto" (default) | "property"

Source of the video bandwidth (VBW), specified as either "auto" or "property".

- "auto" — The Spectrum Analyzer adjusts the VBW such that the equivalent forgetting factor is 0.9.
- "property" — The Spectrum Analyzer adjusts the VBW using the value specified in the VBW property.

For more details on the video bandwidth method, see "Averaging Method" on page 4-1085.

**Tunable:** Yes

**Dependency**

To enable this property, set `InputDomain` to "time" and `AveragingMethod` to "vbw".

**Scope Window Use**

Click the **Estimation** tab on the Spectrum Analyzer toolstrip. In the **Averaging** section, set **VBW (Hz)** to either Auto or a positive real scalar less than or equal to **Sample Rate (Hz)/2**.

To enable the **VBW (Hz)**, set **Input Domain** to Time and **Averaging Method** to VBW.

Data Types: char | string

**VBW — Video bandwidth**

10 (default) | positive scalar

Video bandwidth, specified as a positive scalar less than or equal to `SampleRate/2`. For more details on the video bandwidth method, see "Averaging Method" on page 4-1085.

The Spectrum Analyzer shows the value of VBW in the status bar at the bottom of the display.

**Tunable:** Yes

**Dependency**

To enable this property, set `VBWSource` to "property".

**Scope Window Use**

Click the **Estimation** tab on the Spectrum Analyzer toolstrip. In the **Averaging** section, set **VBW (Hz)** to either Auto or enter a positive real scalar that is less than or equal to **Sample Rate (Hz)/2**.

To enable the **VBW (Hz)**, set **Input Domain** to Time and **Averaging Method** to VBW.

Data Types: double

**InputUnits — Units of frequency input**

"dBm" (default) | "dBV" | "dBW" | "Vrms" | "Watts" | "none"

Units of the frequency-domain input, specified as "dBm", "dBV", "dBW", "Vrms", "Watts", or "none". The Spectrum Analyzer scales frequency data according to the specified display unit.

**Tunable:** Yes

**Dependency**

To enable this property, set `InputDomain` to "frequency".

**Scope Window Use**

Click the **Estimation** tab on the Spectrum Analyzer toolstrip. In the **Domain** section, set **Input Unit**.

To enable the **Input Unit**, set **Input Domain** to Frequency.

Data Types: char | string

**SpectrumUnits — Units of the spectrum**

"dBm" (default) | "dBFS" | "dBV" | "dBW" | "Vrms" | "Watts" | "dBm/Hz" | "dBW/Hz" | "dBFS/Hz" | "Watts/Hz" | "auto"

Units in which the Spectrum Analyzer displays power values, specified as one of the following:

- "dBm"
- "dBFS"
- "dBV"
- "dBW"
- "Vrms"
- "Watts"
- "dBm/Hz"
- "dBW/Hz"
- "dBFS/Hz"
- "Watts/Hz"
- "auto"

**Tunable:** Yes

**Dependency**

The spectrum units available depend on the value you specify in the `SpectrumType` property.

| InputDomain | SpectrumType    | Allowed SpectrumUnits                        |
|-------------|-----------------|--|
| "time"      | "power"         | "dBm", "dBW", "dBFS", "Watts"                |
|             | "power-density" | "dBm/Hz", "dBW/Hz", "dBFS/Hz", "Watts/Hz"    |
|             | "rms"           | "dBV", "Vrms"                                |
| "frequency" | —               | "auto", "dBm", "dBV", "dBW", "Vrms", "Watts" |

If you set the `InputDomain` property to "frequency" and the `SpectrumUnits` property to "auto", the Spectrum Analyzer assumes the spectrum units to be equal to input units specified in the `InputUnits` property. If you set `InputDomain` to "time" and `SpectrumUnits` to any option other than "auto", the Spectrum Analyzer converts the units specified in `InputUnits` to the units specified in `SpectrumUnits`.

### Scope Window Use

Click the **Spectrum** tab on the Spectrum Analyzer toolstrip. In the **Scale** section, set **Spectrum Unit**.

Data Types: char | string

### FullScaleSource — Source of full-scale value

"auto" (default) | "property"

Source of the dBFS scaling factor, specified as either "auto" or "property".

- "auto" -- The Spectrum Analyzer adjusts the scaling factor based on the input data.
- "property" -- Specify the full-scale scaling factor using the FullScale property.

**Tunable:** Yes

### Dependency

To enable this property, set:

- InputDomain to "time"
- SpectrumType to "power" or "power-density"
- SpectrumUnits to "dBFS" or "dBFS/Hz"

### Scope Window Use

Click the **Spectrum** tab on the Spectrum Analyzer toolstrip. In the **Scale** section, set the **Full Scale** to either Auto or a positive scalar.

To enable the **Full Scale**:

- In the **Analyzer** tab, set the spectrum type to Power or Power Density.
- In the **Estimation** tab, set **Input Domain** to Time.
- In the **Spectrum** tab, set **Spectrum Unit** to dBFS or dBFS/Hz (when spectrum type is set to Power Density).

Data Types: char | string

### FullScale — dBFS full scale

1 (default) | positive scalar

dBFS full scale, specified a positive scalar.

**Tunable:** Yes

### Dependency

To enable this property, set:

- 1 InputDomain to "time"
- 2 SpectrumType to "power" or "power-density"
- 3 SpectrumUnits to "dBFS" or "dBFS/Hz"
- 4 FullScaleSource to "property"

**Scope Window Use**

Click the **Spectrum** tab on the Spectrum Analyzer toolstrip. In the **Scale** section, set the **Full Scale** to either **Auto** or enter a positive scalar.

To enable the **Full Scale**:

- In the **Analyzer** tab, set the spectrum type to **Power** or **Power Density**.
- In the **Estimation** tab, set **Input Domain** to **Time**.
- In the **Spectrum** tab, set **Spectrum Unit** to **dBFS** or **dBFS/Hz** (when spectrum type is set to **Power Density**).

Data Types: double

**ReferenceLoad — Reference load to compute power levels**

1 (default) | positive scalar

Load that the scope uses as a reference to compute power levels, specified as a positive scalar in Ohms.

**Tunable:** Yes

**Dependency**

To enable this property, set:

- **SpectrumType** to "power" or "power-density".
- **SpectrumUnits** to any option other than "dBFS" or "dBFS/Hz".

**Scope Window Use**

Click the **Spectrum** tab on the Spectrum Analyzer toolstrip. In the **Scale** section, set **Reference Load ( $\Omega$ )**.

Data Types: double

**FrequencyOffset — Offset to apply to frequency axis**

0 (default) | scalar | vector

Offset to apply to the frequency axis (x-axis) in units of Hz, specified as one of the following:

- **Scalar** — Apply the same frequency offset to all channels.
- **Vector** — Apply a specific frequency offset for each channel. The vector length must be equal to the number of input channels.

The overall span must fall within the "Nyquist Frequency Interval" on page 4-1082. You can control the overall span in different ways based on how you set the **FrequencySpan** property.

**Tunable:** Yes

**Scope Window Use**

Click the **Analyzer** tab on the Spectrum Analyzer toolstrip. In the **Bandwidth** section, set **Offset (Hz)**.

Data Types: double

## Spectrogram

### **SpectrogramChannel** — Channel for which spectrogram is plotted

1 (default) | positive integer

Channel for which the spectrogram is plotted, specified as a positive integer in the range [1  $N$ ], where  $N$  is the number of input channels.

**Tunable:** Yes

#### **Dependency**

To enable this property, set `ViewType` to "spectrogram" or "spectrum-and-spectrogram".

#### **Scope Window Use**

Click the **Spectrogram** tab on the Spectrum Analyzer toolstrip. In the **Channel** section, select a **Channel**.

Data Types: double

### **TimeResolutionSource** — Source of the time resolution

"auto" (default) | "property"

Source of the time resolution of each spectrogram line, specified as either "auto" or "property".

When you set `RBWSource` and `TimeResolutionSource` to "auto", then RBW is set such that there are 1024 RBW intervals in one frequency span. The time resolution is set to 1/RBW.

When `RBWSource` is set to "auto" and `TimeResolutionSource` is set to "property", then time resolution becomes the main control and RBW is set to 1/`TimeResolution` Hz.

When `RBWSource` is set to "property" and `TimeResolutionSource` is set to "auto", then RBW becomes the main control and the time resolution is set 1/RBW s.

When both `RBWSource` and `TimeResolutionSource` are set to "property", then the specified time resolution value must be equal to or larger than the minimum attainable time resolution which is defined by 1/RBW. Several spectral estimates are combined into one spectrogram line to obtain the desired time resolution.

**Tunable:** Yes

#### **Dependency**

To enable this property, set `ViewType` to "spectrogram" or "spectrum-and-spectrogram".

#### **Scope Window Use**

Click the **Spectrogram** tab on the Spectrum Analyzer toolstrip. In the **Time Options** section, set the **Time Resolution (s)** to Auto or enter a positive scalar.

To enable the **Time Resolution (s)**, select **Spectrogram** in the **Analyzer** tab.

Data Types: char | string

### **TimeResolution** — Time resolution of each spectrogram line

0.001 (default) | positive scalar



Time resolution of each spectrogram line in seconds, specified as a positive scalar.

**Tunable:** Yes

**Dependency**

To enable this property, set:

- ViewType to "spectrogram" or "spectrum-and-spectrogram"
- TimeResolutionSource to "property".

**Scope Window Use**

Click the **Spectrogram** tab on the Spectrum Analyzer toolstrip. In the **Time Options** section, set the **Time Resolution (s)** to Auto or enter a positive scalar.

To enable the **Time Resolution (s)**, select **Spectrogram** in the **Analyzer** tab.

Data Types: double

**TimeSpanSource — Source of time span value**

"auto" (default) | "property"

Source for the time span of the spectrogram, specified as either one of these:

- "auto" -- The spectrogram displays 100 spectrogram lines at any given time.
- "property" -- The spectrogram uses the time duration you specify in seconds in the TimeSpan property.

The time span that you specify must be at least two times larger than the duration of the number of samples required for a spectral update.

**Tunable:** Yes

**Dependency**

To enable this property, set ViewType to "spectrogram" or "spectrum-and-spectrogram".

**Scope Window Use**

Click the **Spectrogram** tab on the Spectrum Analyzer toolstrip. In the **Time Options** section, set the **Time Span (s)** to Auto or enter a positive scalar.

Data Types: char | string

**TimeSpan — Time span of spectrogram**

0.1 (default) | positive scalar

Time span of the spectrogram display in seconds, specified as a positive scalar. You must set the time span to be at least twice as large as the duration of the number of samples required for a spectral update.

**Tunable:** Yes

**Dependency**

To enable this property, set:

- ViewType to "spectrogram" or "spectrum-and-spectrogram".
- TimeSpanSource to "property".

### Scope Window Use

Click the **Spectrogram** tab on the Spectrum Analyzer toolstrip. In the **Time Options** section, set the **Time Span (s)** to Auto or enter a positive scalar.

Data Types: double

### Measurements

#### MeasurementChannel — Channel for which to obtain measurements

1 (default) | positive integer

The channel for which you need to obtain measurements, specified as a positive integer in the range [1  $N$ ], where  $N$  is the number of input channels.

**Tunable:** Yes

### Scope Window Use

Click the **Measurements** tab on the Spectrum Analyzer toolstrip. In the **Channel** section, select a **Channel**.

Data Types: double

#### ChannelMeasurements — Channel measurements

ChannelMeasurementsConfiguration object

Channel measurements, specified as a ChannelMeasurementsConfiguration object. Enable channel measurements to compute and display the occupied bandwidth or adjacent channel power ratio. All ChannelMeasurementsConfiguration properties are tunable.

**Tunable:** Yes

### Dependency

To enable this property, set ViewType to either "spectrum" or "spectrum-and-spectrogram".

### Scope Window Use

Click the **Channel Measurements** tab on the Spectrum Analyzer toolstrip and modify the measurement settings.

The **Channel Measurements** tab appears when you select **Spectrum** in the **Analyzer** tab.

#### CursorMeasurements — Cursor measurements

CursorMeasurementsConfiguration object

Cursor measurements, specified as a CursorMeasurementsConfiguration object. Enable cursor measurements to display waveform cursors. All CursorMeasurementsConfiguration properties are tunable.

**Tunable:** Yes

**Scope Window Use**

Click the **Measurements** tab on the Spectrum Analyzer toolstrip and modify the cursor measurements in the **Cursors** section.

**DistortionMeasurements — Distortion measurements**

`DistortionMeasurementsConfiguration` object

Distortion measurements, specified as a `DistortionMeasurementsConfiguration` object. Enable distortion measurements to compute and display the harmonic distortion and intermodulation distortion. All `DistortionMeasurementsConfiguration` properties are tunable. For more details, see “Distortion Measurements” on page 4-1083 and “Harmonic Measurements” on page 4-1084.

**Tunable:** Yes

**Dependency**

To enable this property, set `ViewType` to either "spectrum" or "spectrum-and-spectrogram".

**Scope Window Use**

Click the **Measurements** tab on the Spectrum Analyzer toolstrip and modify the distortion measurements in the **Distortion** section.

The **Measurements** tab appears when you select **Spectrum** in the **Analyzer** tab.

**PeakFinder — Peak finder measurement**

`PeakFinderConfiguration` object

Peak finder measurement, specified as a `PeakFinderConfiguration` object. Enable peak finder to compute and display the largest calculated peak values. All `PeakFinderConfiguration` properties are tunable.

**Tunable:** Yes

**Dependency**

To enable this property, set `ViewType` to either "spectrum" or "spectrum-and-spectrogram".

**Scope Window Use**

Click the **Measurements** tab on the Spectrum Analyzer toolstrip and modify the peak finder measurements in the **Peaks** section.

The **Measurements** tab appears when you select **Spectrum** in the **Analyzer** tab.

**SpectralMask — Spectral mask configuration**

`SpectralMaskConfiguration` object

Spectral mask configuration, specified as a `SpectralMaskConfiguration` object. Use the spectral mask configuration to draw upper and lower or upper or lower mask lines in the power and power-density plots. All `SpectralMaskConfiguration` properties are tunable.

**Tunable:** Yes

**Dependency**

To enable this property, set:

- ViewType to either "spectrum" or "spectrum-and-spectrogram".
- SpectrumType to either "power" or "power-density".

**Scope Window Use**

Click the **Spectral Mask** tab on the Spectrum Analyzer toolstrip and modify the settings.

The **Spectral Mask** tab appears when you:

- Select **Spectrum** in the **Analyzer** tab.
- In the drop-down list under **Spectrum**, choose either Power or Power Density.

**Visualization****Name — Caption to display in spectrum Analyzer window**

"Spectrum Analyzer" (default) | character vector | string scalar

Caption to display in the scope window, specified as a character vector or string scalar.

**Tunable:** Yes

Data Types: char | string

**Position — Spectrum Analyzer window position in pixels**

[left bottom 800 500] (default) |[left bottom width height]

Spectrum Analyzer window position in pixels, specified as a four-element double vector of the form [left bottom width height]. You can place the scope window in a specific position on your screen by modifying the values of this property.

By default, the window appears at the center of your screen with a width of 800 pixels and height of 500 pixels. The exact center coordinates depend on your screen resolution.

**Tunable:** Yes

**MaximizeAxes — Maximize axes control**

"auto" (default) | "on" | "off"

Maximize axes control, specified as one of the following:

- "auto" -- The Spectrum Analyzer maximizes axes only if the display does not contain any labels or title annotations.
- "on" -- The Spectrum Analyzer maximizes axes in all displays.
- "off" -- The Spectrum Analyzer does not maximize axes in any display.

**Tunable:** Yes

Data Types: char | string

**PlotNormalTrace — Option to plot normal trace**

true (default) | false

To remove normal traces from the display, set this property to false. These traces display the free-running spectral estimates. The Spectrum Analyzer continues its spectral computations even when you set this property to false.

**Tunable:** Yes

**Dependency**

To enable this property, set `ViewType` to "spectrum" or "spectrum-and-spectrogram".

**Scope Window Use**

Click the **Spectrum** tab on the Spectrum Analyzer toolstrip and select the **Normal Trace** check box in the **Trace Options** section.

To enable the **Normal Trace** check box, select **Spectrum** in the **Analyzer** tab.

Data Types: `logical`

**PlotMaxHoldTrace — Option to plot max-hold trace**

`false` (default) | `true`

Option to plot max-hold trace, specified as `true` or `false`. To compute and plot the maximum-hold spectrum of each input channel, set this property to `true`. The Spectrum Analyzer computes the maximum-hold spectrum at each frequency bin by keeping the maximum value of all the power spectrum estimates. When you change the value of this property, the Spectrum Analyzer resets its maximum-hold computations.

**Tunable:** Yes

**Dependency**

To enable this property, set `ViewType` to "spectrum" or "spectrum-and-spectrogram".

**Scope Window Use**

Click the **Spectrum** tab on the Spectrum Analyzer toolstrip and select the **Max-Hold Trace** check box in the **Trace Options** section.

To enable the **Max-Hold Trace** check box, select **Spectrum** in the **Analyzer** tab.

Data Types: `logical`

**PlotMinHoldTrace — Option to plot min-hold trace**

`false` (default) | `true`

Option to plot min-hold trace, specified as `true` or `false`. To compute and plot the minimum-hold spectrum of each input channel, set this property to `true`. The Spectrum Analyzer computes the minimum-hold spectrum at each frequency bin by keeping the minimum value of all the power spectrum estimates. When you change the value of this property, the Spectrum Analyzer resets its minimum-hold computations.

**Tunable:** Yes

**Dependency**

To enable this property, set `ViewType` to "spectrum" or "spectrum-and-spectrogram".

**Scope Window Use**

Click the **Spectrum** tab on the Spectrum Analyzer toolstrip and select the **Min-Hold Trace** check box in the **Trace Options** section.

To enable the **Min-Hold Trace** check box, select **Spectrum** in the **Analyzer** tab.

Data Types: `logical`

### **Title – Display title**

' ' (default) | character vector | string scalar

Display title, specified as a character vector or a string scalar.

**Tunable:** Yes

### **Scope Window Use**

Click the **Analyzer** tab on the Spectrum Analyzer toolstrip. In the **Configuration** section, click **Settings**. In the Spectrum Analyzer Settings window that opens up, under **Display and labels**, enter **Title**.

Data Types: `char` | `string`

### **YLabel – y-axis label**

' ' (default) | character vector | string scalar

y-axis label, specified as a character vector or a string scalar. The Spectrum Analyzer displays the label to the left of the y-axis.

Regardless of this property, Spectrum Analyzer always displays power units as one of the `SpectrumUnits` values.

**Tunable:** Yes

### **Dependency**

To enable this property, set `ViewType` to "spectrum" or "spectrum-and-spectrogram".

### **Scope Window Use**

Click the **Analyzer** tab on the Spectrum Analyzer toolstrip. In the **Configuration** section, click **Settings**. In the Spectrum Analyzer Settings window that opens up, under **Display and labels**, enter **Y-Label**.

To enable the **Y-Label**, select **Spectrum** in the **Analyzer** tab.

Data Types: `char` | `string`

### **YLimits – y-axis limits**

[-80, 20] (default) | [ymin ymax]

y-axis limits, specified as a two-element numeric vector of the form [ymin ymax]. The units of the y-axis limits depend on the `SpectrumUnits` property.

Example: `scope.YLimits = [-10,20]`

**Tunable:** Yes

### **Dependencies**

To enable this property, set the `ViewType` property to "spectrum" or "spectrum-and-spectrogram".

**Scope Window Use**

Click the **Analyzer** tab on the Spectrum Analyzer toolstrip. In the **Configuration** section, click **Settings**. In the Spectrum Analyzer Settings window that opens up, under **Display and Labels**, enter **Y-Axis Limits**.

To enable the **Y-Axis Limits**, select **Spectrum** in the **Analyzer** tab.

**ColorLimits — Color limits of spectrogram**

`[-80, 20]` (default) | `[colorMin colorMax]`

Color limits of the spectrogram, specified as a two-element numeric vector of the form `[colorMin colorMax]`. The units of the color limits directly depend upon the `SpectrumUnits` property.

Example: `scope.ColorLimits = [-10,20]`

**Tunable:** Yes

**Dependencies**

To enable this property, set the `ViewType` property to "spectrogram" or "spectrum-and-spectrogram".

**Scope Window Use**

Click the **Analyzer** tab on the Spectrum Analyzer toolstrip. In the **Configuration** section, click **Settings**. In the Spectrum Analyzer Settings window that opens up, under **Display and Labels**, enter **Color Limits**.

To enable the **Color Limits**, select **Spectrogram** in the **Analyzer** tab.

**Colormap — Color look-up table**

"jet" (default) | "bone" | "cool" | "copper" | "gray" | "hot" | "parula" | three-column matrix

Color look-up table, specified as a valid colormap name or a three-column matrix with values in the range `[0,1]` defining RGB triplets.

**Tunable:** Yes

**Dependencies**

To enable this property, set the `ViewType` property to "spectrogram" or "spectrum-and-spectrogram".

**Scope Window Use**

Click the **Analyzer** tab on the Spectrum Analyzer toolstrip. In the **Configuration** section, click **Settings**. In the Spectrum Analyzer Settings window that opens up, under **Display and Labels**, enter **Color Map**.

To enable the **Color Map**, select **Spectrogram** in the **Analyzer** tab.

Data Types: `double` | `char` | `string`

**ShowGrid — Flag to show grid**

`true` (default) | `false`

Flag to show the grid, specified as `true` or `false`. Set this property to `true` to show grid lines in the plot.

**Tunable:** Yes

### Scope Window Use

Click the **Analyzer** tab on the Spectrum Analyzer toolstrip. In the **Configuration** section, click **Settings**. In the Spectrum Analyzer Settings window that appears, under **Display and Labels**, select **Show Grid**.

Data Types: `logical`

### ShowLegend — Show or hide legend

`false` (default) | `true`

Show or hide the legend, specified as `true` or `false`. To show a legend with the input names, set this property to `true`.

Use the legend to control which signals are visible. In the scope legend, click a signal name to hide the signal in the scope. To show the signal, click the signal name again. To show only one signal, right-click the signal name. To show all signals, press **Esc**.

**Tunable:** Yes

### Dependencies

To enable this property, set the `ViewType` property to `"spectrum"` or `"spectrum-and-spectrogram"`.

### Scope Window Use

Click the **Analyzer** tab on the Spectrum Analyzer toolstrip. To see the legend, click **Legend** in the **Configuration** section.

To enable the **Legend**, select **Spectrum** in the **Analyzer** tab.

Data Types: `logical`

### ShowColorbar — Show or hide color bar

`true` (default) | `false`

Show or hide color bar, specified as `true` or `false`.

**Tunable:** Yes

### Dependencies

To enable this property, set the `ViewType` property to `"spectrogram"` or `"spectrum-and-spectrogram"`.

### Scope Window Use

Click the **Analyzer** tab on the Spectrum Analyzer toolstrip. To see the color bar, click **Colorbar** in the **Configuration** section.

To enable the **Colorbar**, select **Spectrogram** in the **Analyzer** tab.

Data Types: `logical`



**ChannelNames — Channel names**

empty cell (default) | cell array of character vectors | array of strings

Channel names in the input data, specified as a cell array of character vectors or an array of strings. The names you specify in this property appear in the following locations:

- Legend
- **Spectrum Analyzer Settings > Color and styling** section
- **Measurements** and **Channel Measurements** tabs

If you do not specify channel names, the Spectrum Analyzer names the channels as Channel 1, Channel 2, and so on.

**Tunable:** Yes

**Dependency**

To see the channel names, set ShowLegend to true.

**Scope Window Use**

Click the **Analyzer** tab on the Spectrum Analyzer toolbar. To see the legend, click **Legend** in the **Configuration** section.

Data Types: char

**AxesLayout — Layout of axes**

"vertical" (default) | "horizontal"

Layout of the axes, specified as one of "vertical" or "horizontal". A vertical layout stacks the spectrum above the spectrogram. A horizontal layout puts the two views side-by-side.

**Tunable:** Yes

**Dependency**

To enable this property, set ViewType to "spectrum-and-spectrogram".

**Scope Window Use**

Click the **Analyzer** tab on the Spectrum Analyzer toolbar. Select **Spectrum** and **Spectrogram**. In the **Configuration** section, select and update **Layout**.

Data Types: char | string

**Usage****Syntax**

```
scope(signal)
scope(signal1,signal2,...,signalN)
```

## Description

`scope(signal)` displays the frequency spectrum of the time-domain signal in the Spectrum Analyzer. If `signal` is a frequency-domain signal, the signal is displayed directly in the Spectrum Analyzer.

`scope(signal1,signal2,...,signalN)` displays the frequency spectrum of multiple signals in the Spectrum Analyzer. The number of channels in each signal can be different but the frame size of each signal should be the same.

## Input Arguments

### **signal** — Input signal or signals to visualize

scalar | vector | matrix

Input signal or signals to visualize, specified as a scalar, vector, or a matrix. The number of channels in each signal can be different but the frame size of each signal should be the same.

This scope supports variable-size input signals. That is, the frame size (number of rows) of the input signals can change during simulation, but the number of channels (number of columns) cannot change.

When you set the `InputDomain` property to "time", the input signals can be real or complex. When you set the `InputDomain` property to "frequency", the input signals must be real.

Example: `scope(signal1,signal2)`

## Scope Window Use

To change the appearance of signals in the Spectrum Analyzer, click the **Analyzer** tab and then click **Settings**. In the Spectrum Analyzer Settings window, under **Color and styling**, select a signal and modify its style, width, color, and marker type.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

## Object Functions

### Specific to spectrumAnalyzer

|                                    |   |
|------------------------------------|---|
| <code>generateScript</code>        | Generate MATLAB script to create scope with current settings        |
| <code>getMeasurementsData</code>   | Get the current measurement data displayed on the spectrum analyzer |
| <code>getSpectralMaskStatus</code> | Get test results of current spectral mask                           |
| <code>getSpectrumData</code>       | Save spectrum data shown in spectrum analyzer                       |
| <code>isNewDataReady</code>        | Check spectrum analyzer for new data                                |

### Specific to Scopes

|                        |                               |
|------------------------|-------------------------------|
| <code>show</code>      | Display scope window          |
| <code>hide</code>      | Hide scope window             |
| <code>isVisible</code> | Determine visibility of scope |

### Common to All System Objects

|                   |                             |
|-------------------|-----------------------------|
| <code>step</code> | Run System object algorithm |
|-------------------|-----------------------------|

release Release resources and allow changes to System object property values and input characteristics

reset Reset internal states of System object

---

**Note** If you want to restart the simulation from the beginning, call `reset` to clear the scope window display. Do not call `reset` after calling `release`.

---

## Examples

### Spectrum Analyzer for One-Sided Power Spectrum

View a one-sided power spectrum made from the sum of fixed real sine waves with different amplitudes and frequencies.

```

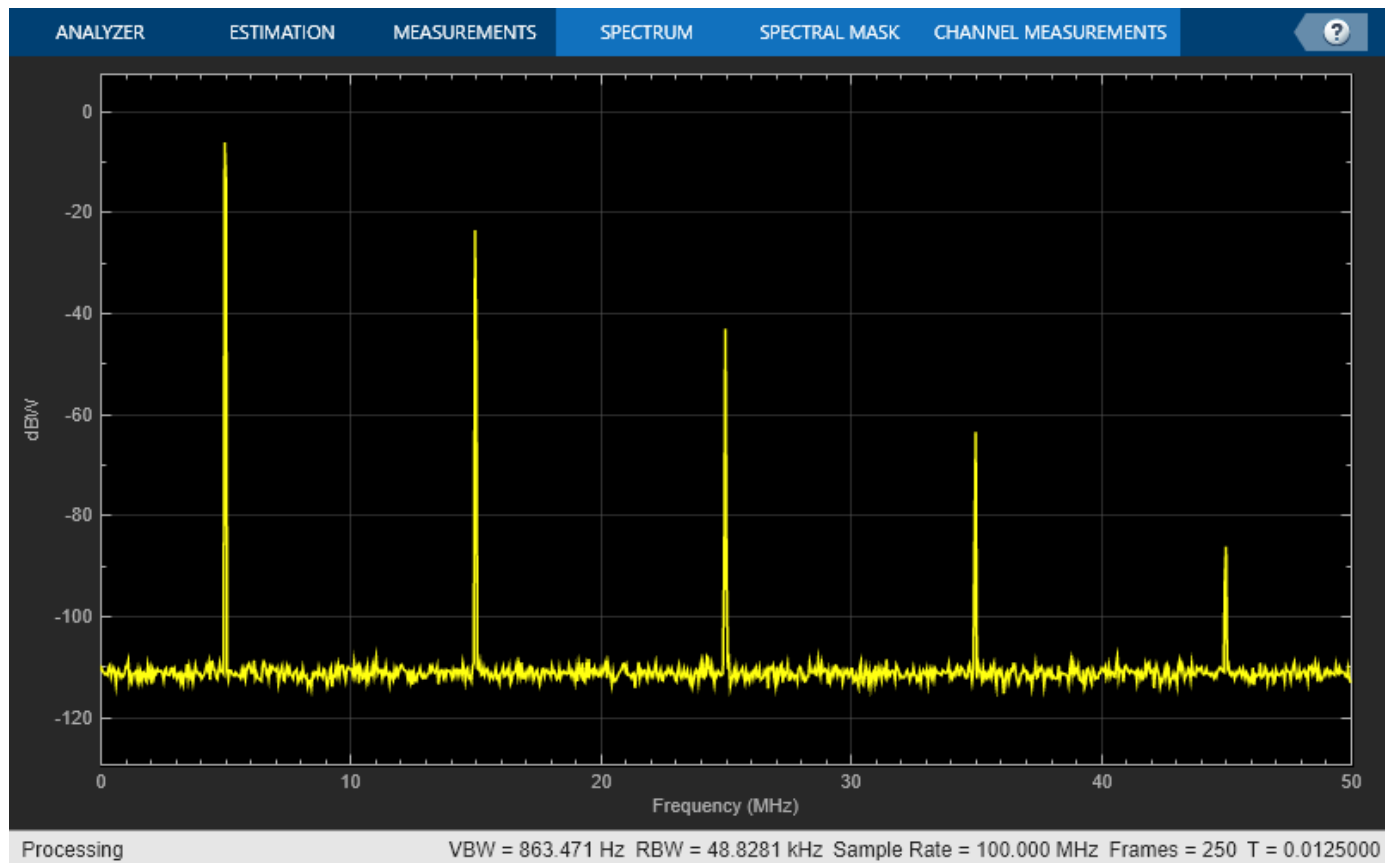
Fs = 100e6; % Sample rate
fSz = 5000; % Frame size

sin1 = dsp.SineWave(1e0,5e6,0,SamplesPerFrame=fSz,SampleRate=Fs);
sin2 = dsp.SineWave(1e-1,15e6,0,SamplesPerFrame=fSz,SampleRate=Fs);
sin3 = dsp.SineWave(1e-2,25e6,0,SamplesPerFrame=fSz,SampleRate=Fs);
sin4 = dsp.SineWave(1e-3,35e6,0,SamplesPerFrame=fSz,SampleRate=Fs);
sin5 = dsp.SineWave(1e-4,45e6,0,SamplesPerFrame=fSz,SampleRate=Fs);

scope = spectrumAnalyzer(SampleRate=Fs,AveragingMethod="exponential",...
    PlotAsTwoSidedSpectrum=false,...
    RBWSource="auto",SpectrumUnits="dBW");

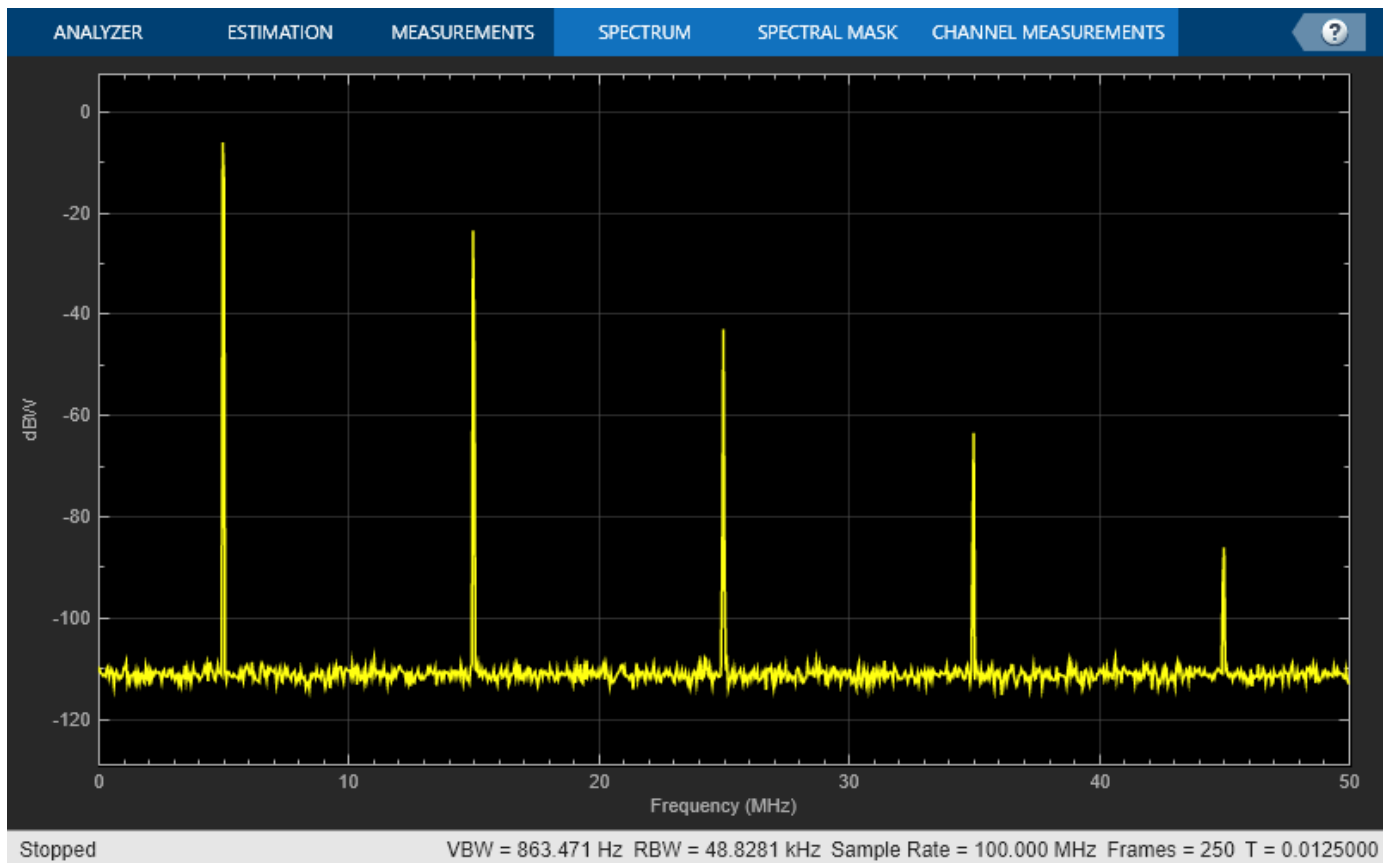
for idx = 1:250
    y1 = sin1();
    y2 = sin2();
    y3 = sin3();
    y4 = sin4();
    y5 = sin5();
    scope(y1+y2+y3+y4+y5+0.0001*randn(fSz,1));
end

```



Call the `release` function to let property values and input characteristics change. The scope automatically scales the axes.

```
release(scope)
```



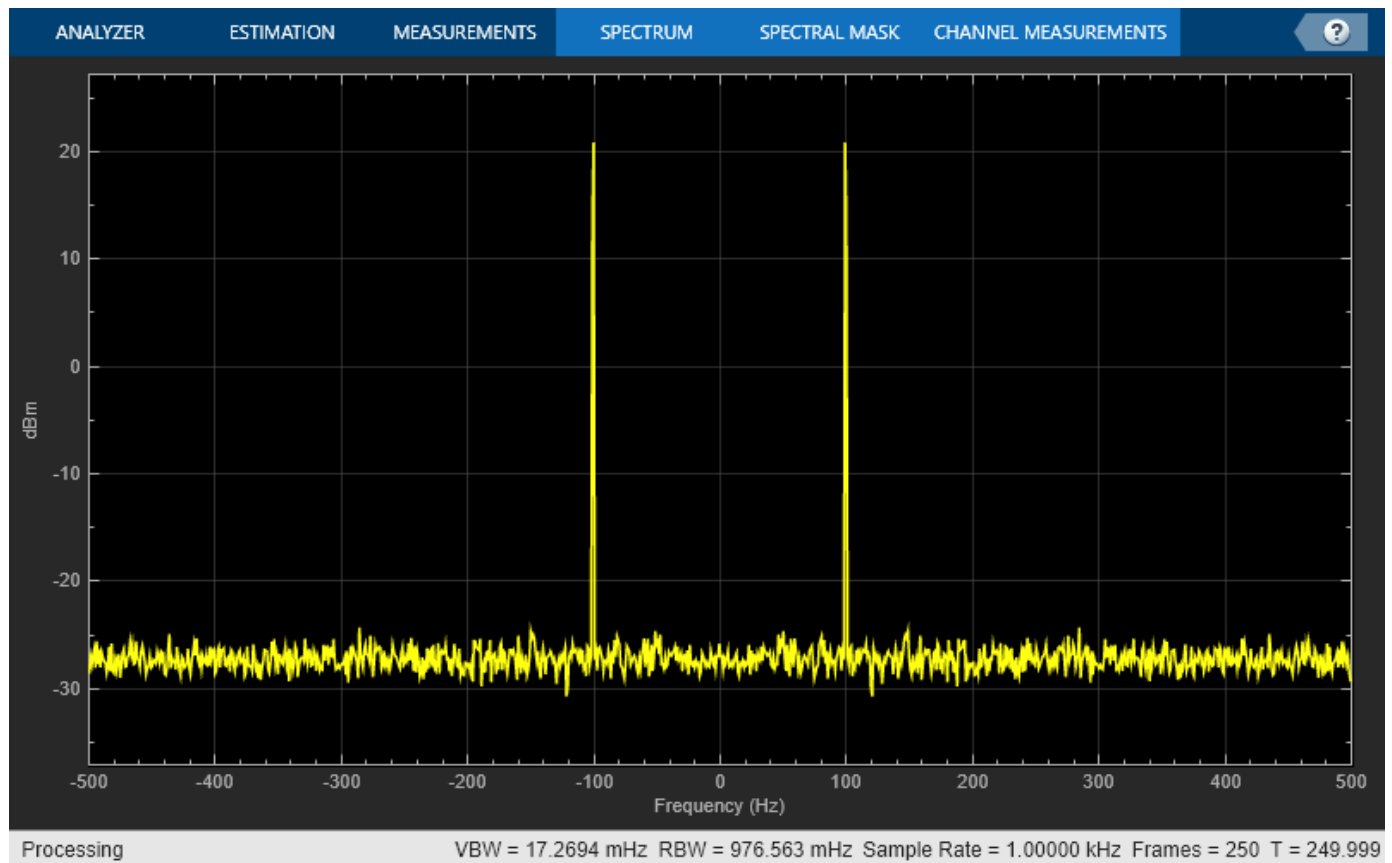
Run the `clear` function to close the Spectrum Analyzer window.

```
clear('scope');
```

### Spectrum Analyzer For Two-Sided Power Spectrum

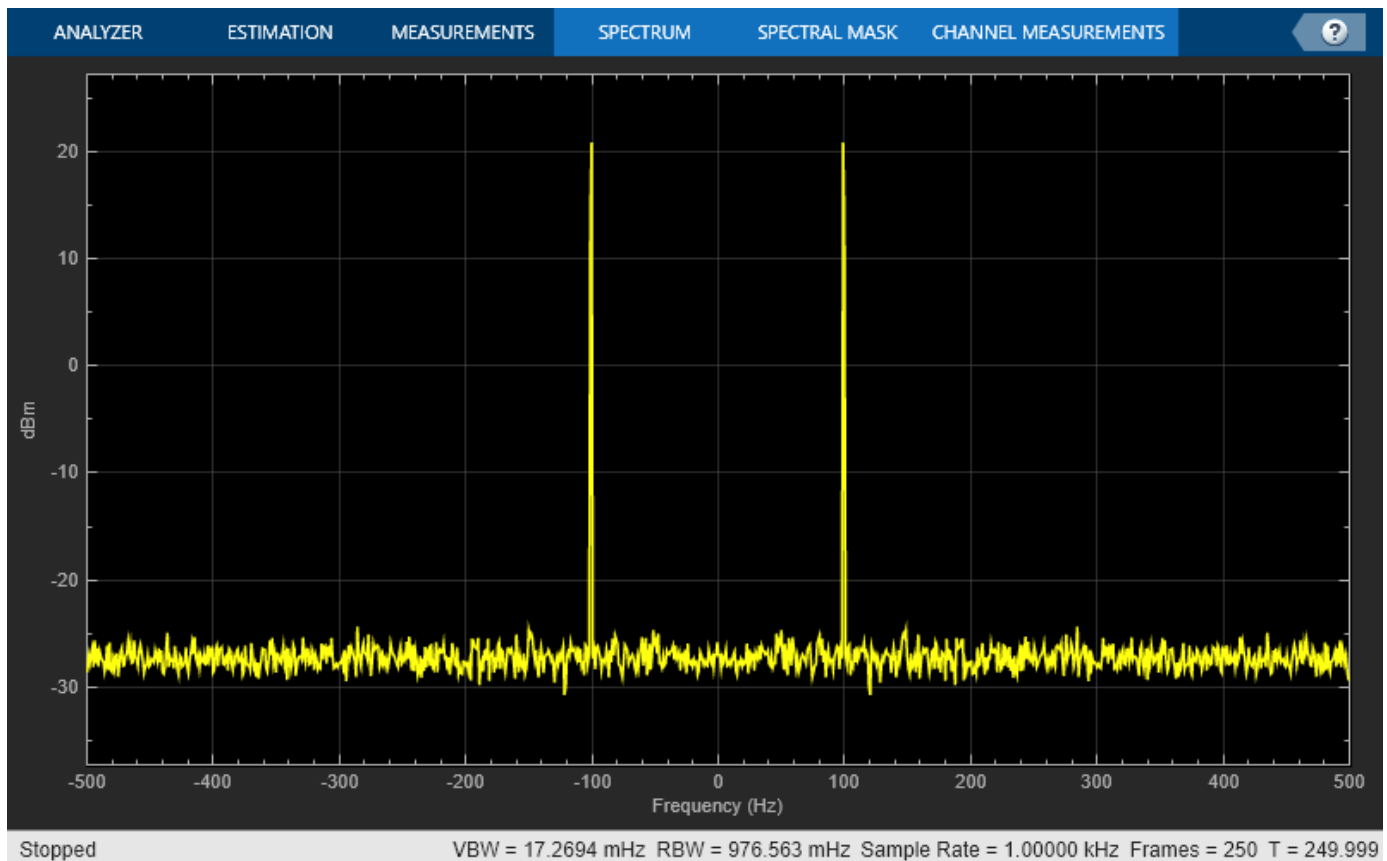
View a two-sided power spectrum of a noisy sine wave on the Spectrum Analyzer.

```
sin = dsp.SinWave(Frequency=100,SampleRate=1000,...
    SamplesPerFrame=1000);
scope = spectrumAnalyzer(SampleRate=sin.SampleRate);
for ii = 1:250
    x = sin() + 0.05*randn(1000,1);
    scope(x);
end
```



Call the `release` function to change property values and input characteristics. The scope automatically scales the axes and updates the display one more time if the internal buffer contains any more data.

```
release(scope);
```



Run the MATLAB `clear` function to close the Spectrum Analyzer window.

```
clear('scope');
```

### Spectrogram of Chirp Signal

Plot the spectrogram for a chirp signal with added random noise.

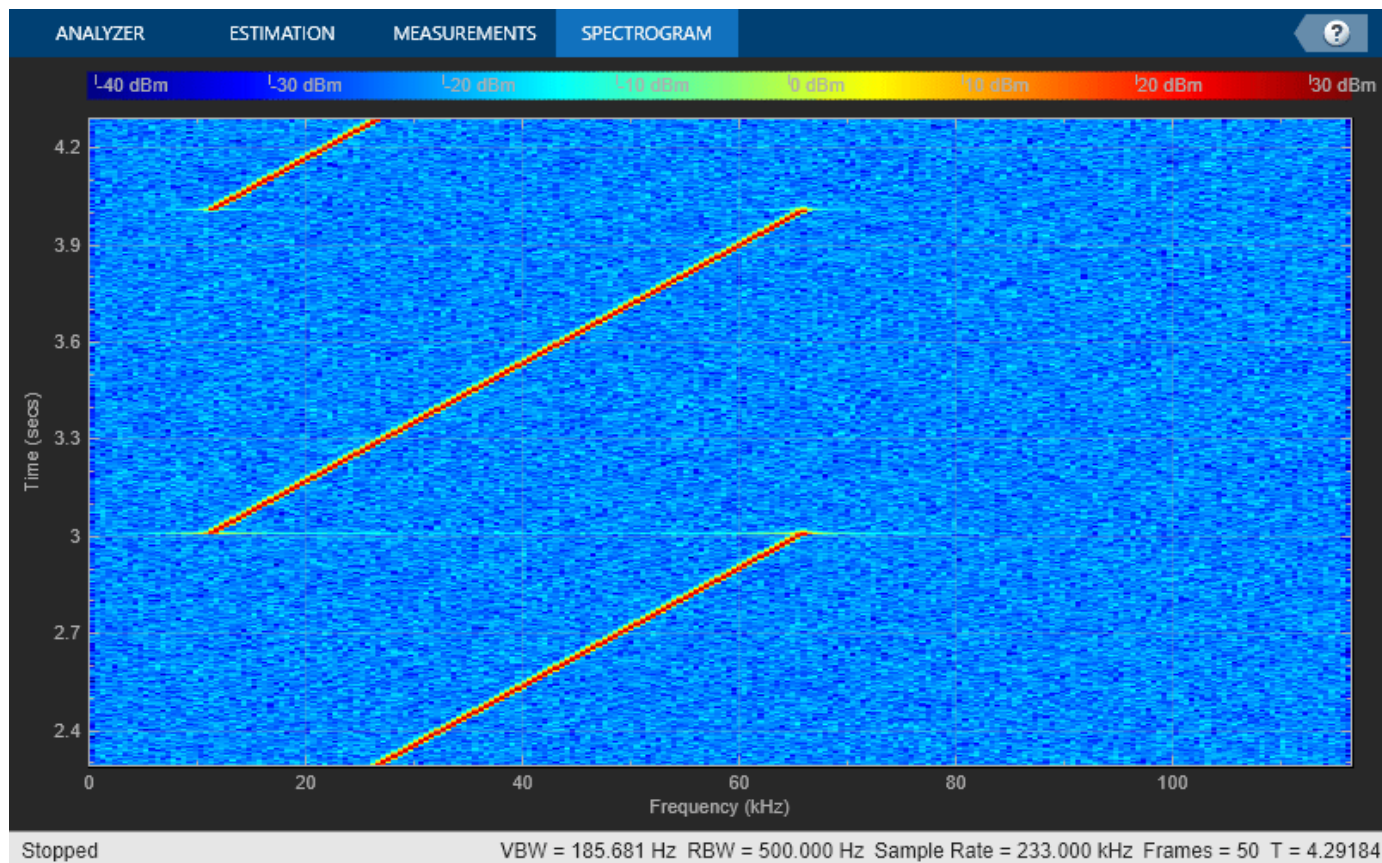
```
Fs = 233e3;
frameSize = 20e3;
chirp = dsp.Chirp(SampleRate=Fs, SamplesPerFrame=frameSize, ...
    InitialFrequency=11e3, TargetFrequency=11e3+55e3);

scope = spectrumAnalyzer(SampleRate=Fs, ...
    AveragingMethod="exponential", ...
    ForgettingFactor=0.3, ViewType="spectrogram", ...
    RBWSource="property", RBW=500, ...
    TimeSpanSource="property", TimeSpan=2);

scope.PlotAsTwoSidedSpectrum = false;

for idx = 1:50
    y = chirp()+0.05*randn(frameSize,1);
    scope(y);
end
```

```
end
release(scope)
```



### Display Frequency Input from Spectral Estimation

Use the Spectrum Analyzer to display frequency input from spectral estimates of sinusoids embedded in white Gaussian noise.

#### Initialization

Create two `dsp.SpectrumEstimator` objects. Set one object to use the Welch-based spectral estimation technique with a Hann window. Set the other object to use the filter bank estimation. Specify a noisy sine wave input signal with four sinusoids at 0.16, 0.2, 0.205, and 0.25 cycles per sample. View the spectral estimate using the `spectrumAnalyzer` object.

```
FrameSize = 420;
Fs = 1;
Frequency = [0.16 0.2 0.205 0.25];
singen = dsp.SineWave(SampleRate=Fs, SamplesPerFrame=FrameSize, ...
    Frequency=Frequency, Amplitude=[2e-5 1 0.05 0.5]);
NoiseVar = 1e-10;
numAvg = 8;

hannEstimator = dsp.SpectrumEstimator(PowerUnits="dBm", ...
```



```

Window="Hann",FrequencyRange="onesided",...
SpectralAverages=numAvgs,SampleRate=Fs);

filterBankEstimator = dsp.SpectrumEstimator(PowerUnits="dBm",...
Method="Filter bank",FrequencyRange="onesided",...
SpectralAverages=numAvgs,SampleRate=Fs);

spectrumPlotter = spectrumAnalyzer(InputDomain="frequency",...
SampleRate=Fs,SpectrumUnits="dBm",...
YLimits=[-120,40],PlotAsTwoSidedSpectrum=false,...
ChannelNames={'Hann window','Filter bank'},ShowLegend=true);

```

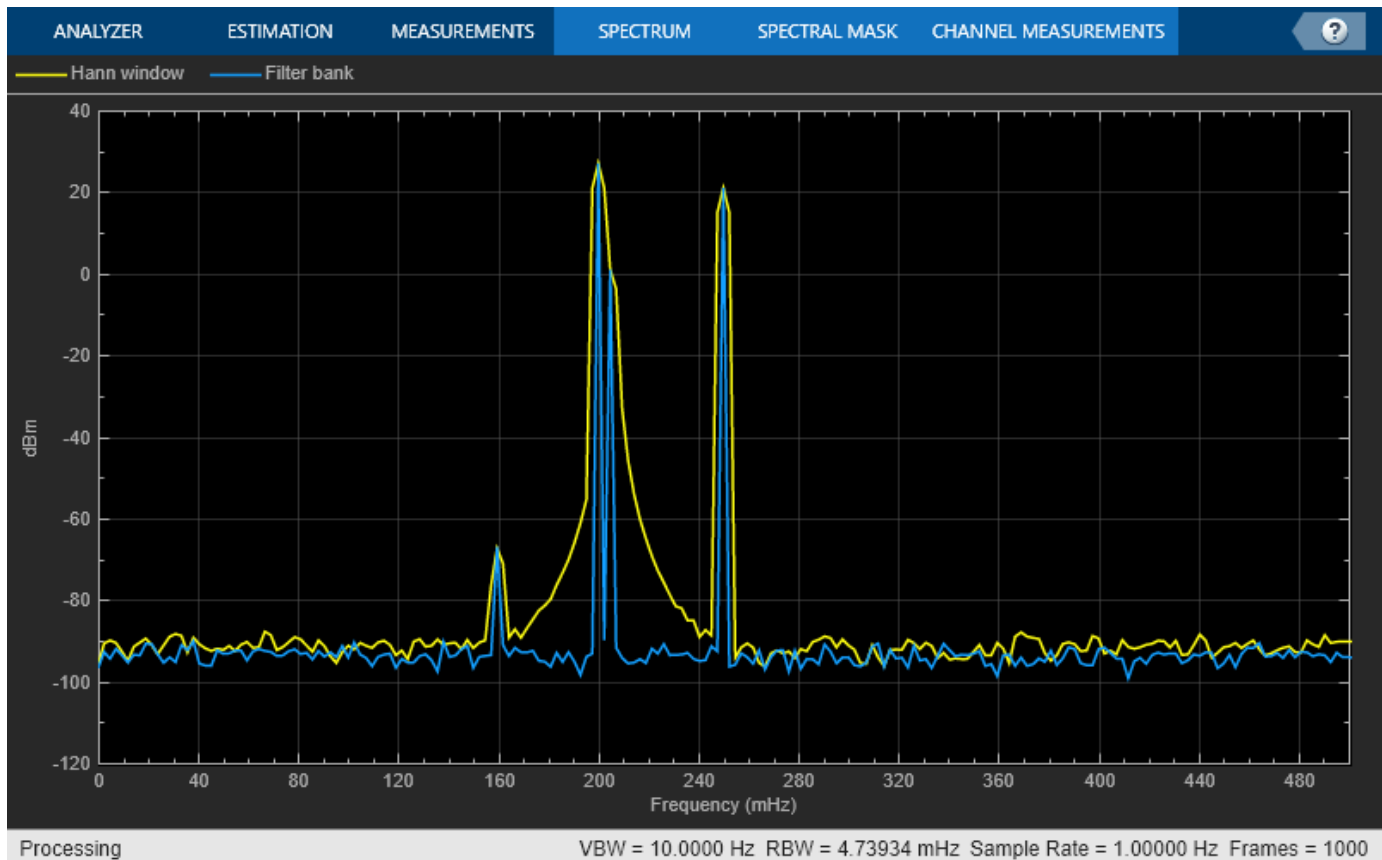
### Streaming

Stream the input. Compare the spectral estimates in the spectrum analyzer.

```

for i = 1:1000
    x = sum(sinegen(),2) + sqrt(NoiseVar)*randn(FrameSize,1);
    Pse_hann = hannEstimator(x);
    Pfb = filterBankEstimator(x);
    spectrumPlotter([Pse_hann,Pfb])
end

```



### Obtain Measurements Data Programmatically for spectrumAnalyzer object

Compute and display the power spectrum of a noisy sinusoidal input signal using the spectrumAnalyzer MATLAB® object. Measure the peaks, cursor placements, adjacent channel power ratio, and distortion values in the spectrum by enabling these properties:

- PeakFinder
- CursorMeasurements
- ChannelMeasurements
- DistortionMeasurements

#### Initialization

The input sine wave has two frequencies: 1000 Hz and 5000 Hz. Create two dsp.SinWave System objects to generate these two frequencies. Create a spectrumAnalyzer object to compute and display the power spectrum.

```
Fs = 44100;
Sineobject1 = dsp.SinWave(SamplesPerFrame=1024,PhaseOffset=10,...
    SampleRate=Fs,Frequency=1000);
Sineobject2 = dsp.SinWave(SamplesPerFrame=1024,...
    SampleRate=Fs,Frequency=5000);
SA = spectrumAnalyzer(SampleRate=Fs,SpectrumType="power",...
    PlotAsTwoSidedSpectrum=false,ChannelNames={'Power spectrum of the input'},...
    YLimits=[-120 40],ShowLegend=true);
```

#### Enable Measurements Data

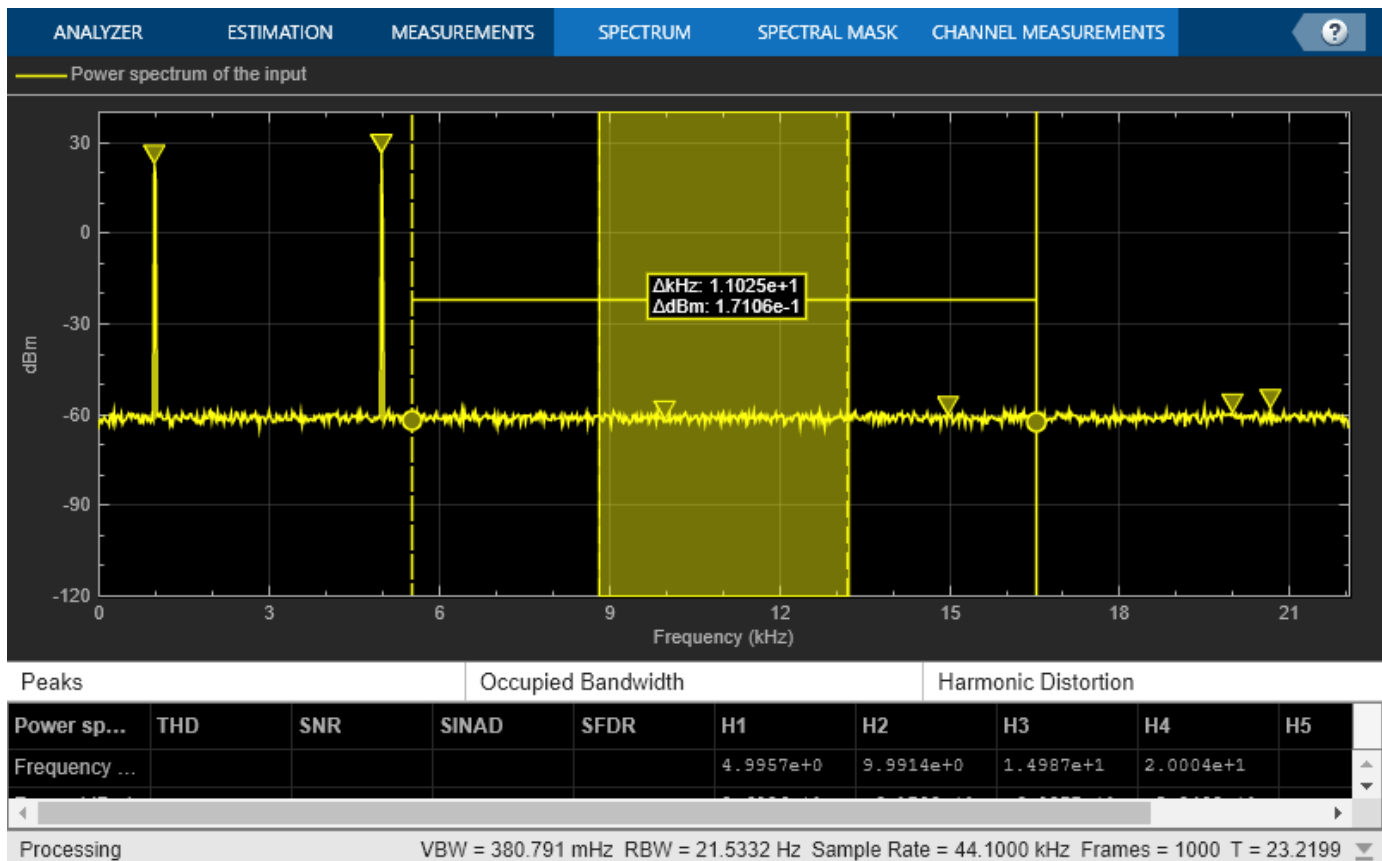
To obtain the measurements, set the Enabled property to true.

```
SA.CursorMeasurements.Enabled = true;
SA.ChannelMeasurements.Enabled = true;
SA.PeakFinder.Enabled = true;
SA.DistortionMeasurements.Enabled = true;
```

#### Use getMeasurementsData

Stream in the noisy sine wave input signal and estimate the power spectrum of the signal using the spectrumAnalyzer object. Measure the characteristics of the spectrum. Use the getMeasurementsData function to obtain these measurements programmatically. The isNewDataReady function returns true when there is new spectrum data. Store the measured data in the variable data.

```
data = [];
for Iter = 1:1000
    Sinewave1 = Sineobject1();
    Sinewave2 = Sineobject2();
    Input = Sinewave1 + Sinewave2;
    NoisyInput = Input + 0.001*randn(1024,1);
    SA(NoisyInput);
    if SA.isNewDataReady
        data = [data;getMeasurementsData(SA)];
    end
end
```



The panes at the bottom of the scope window display the measurements that you have enabled. The values in these panes match the values in the last time step of the data variable. You can access the individual fields of `data` to obtain the various measurements programmatically.

### Compare Peak Values

Use the `PeakFinder` property to obtain peak values. Verify that the peak values in the last time step of `data` match the values in the spectrum analyzer plot.

```
peakvalues = data.PeakFinder(end).Value
```

```
peakvalues = 3x1
```

```
26.3957
22.7830
-57.9977
```

```
frequencieskHz = data.PeakFinder(end).Frequency/1000
```

```
frequencieskHz = 3x1
```

```
4.9957
0.9905
```

20.6719

## More About

### Convert the Power Between Units

Conversion of power between units on the Spectrum Analyzer window.

The Spectrum Analyzer provides three units to specify the power spectral density: **Watts/Hz**, **dBm/Hz**, and **dBW/Hz**. Corresponding units of power are **Watts**, **dBm**, and **dBW**. For electrical engineering applications, you can also view the RMS of your signal in **Vrms** or **dBV**. The default spectrum type is **Power** in **dBm**.

### Convert the Power in Watts to dBW and dBm

Power in **dBW** is given by:

$$P_{\text{dBW}} = 10\log_{10}(\text{power in watt}/1 \text{ watt})$$

Power in **dBm** is given by:

$$P_{\text{dBm}} = 10\log_{10}(\text{power in watt}/1 \text{ milliwatt})$$

For a sine wave signal with an amplitude of 1 V, the power of a one-sided spectrum in **Watts** is given by:

$$P_{\text{Watts}} = A^2/2$$

$$P_{\text{Watts}} = 1/2$$

Corresponding power in **dBm** is given by this equation.

$$P_{\text{dBm}} = 10\log_{10}(\text{power in watt}/1 \text{ milliwatt})$$

$$P_{\text{dBm}} = 10\log_{10}(0.5/10^{-3})$$

$$P_{\text{dBm}} = 26.9897 \text{ dBm}$$

For a white noise signal, the spectrum is flat for all frequencies. Consider a white noise signal with a variance of  $1e-4$ . The power per unit bandwidth therefore ( $P_{\text{unitbandwidth}}$ ) is  $1e-4$ . For a one-sided spectrum in the range  $[0, Fs/2]$ , the total power of white noise in **Watts** is given by this equation.

$$P_{\text{whitenoise}} = P_{\text{unitbandwidth}} \cdot \text{number of frequency bins},$$

$$P_{\text{whitenoise}} = (10^{-4}) \cdot \left( \frac{Fs/2}{RBW} \right)$$

$F_s$  is the input sample rate. The number of frequency bins is the ratio of total bandwidth to RBW. For a one-sided spectrum, the total bandwidth is half the sample rate. Consider a sample rate of 44100 Hz and a RBW of 21.53 Hz. With these values, the total power of white noise in **Watts** is 0.1024 W.

$$P_{\text{whitenoise}} = 10\log_{10}(0.1024/10^{-3})$$

$$P_{\text{whitenoise}} = 20.103 \text{ dBm}$$

In **dBm**, the power of white noise can be calculated using  $10 \cdot \log_{10}(0.1024/10^{-3})$ , which equals 20.103 **dBm**.

### Convert Power in Watts to dBFS

If you set the spectral units to dBFS and set the full scale (FullScaleSource) to "auto", power in dBFS is computed as:

$$P_{\text{dBFS}} = 20 \cdot \log_{10}(\sqrt{P_{\text{watts}}}/\text{Full\_Scale})$$

where:

- $P_{\text{watts}}$  is the power in watts
- For double and float signals, *Full\_Scale* is the maximum value of the input signal.
- For fixed point or integer signals, *Full\_Scale* is the maximum value that can be represented.

If you specify a manual full scale (set FullScaleSource to "property"), power in dBFS is given by:

$$P_{\text{FS}} = 20 \cdot \log_{10}(\sqrt{P_{\text{watts}}}/\text{FS})$$

Where FS is the full-scaling factor specified in the FullScale property.

For a sine wave signal with an amplitude of 1 V, the power of a one-sided spectrum in Watts is given by:

$$P_{\text{Watts}} = A^2/2$$

$$P_{\text{Watts}} = 1/2$$

The corresponding power in dBFS is given by:

$$P_{\text{FS}} = 20 \cdot \log_{10}(\sqrt{1/2}/1)$$

$$P_{\text{FS}} = -3.0103 \text{ dBFS}$$

To confirm the power value in the spectrum analyzer, run these commands:

```
Fs = 1000; % Sampling frequency
sinef = dsp.SineWave(SampleRate=Fs, SamplesPerFrame=100);
scope = spectrumAnalyzer(SampleRate=Fs, ...
    SpectrumUnits="dBFS", PlotAsTwoSidedSpectrum=false)
```

```
for ii = 1:100000
xsine = sinef();
scope(xsine)
end
```

Enable **Peak Finder** in the **Measurements** tab of the spectrum analyzer toolstrip.

### Convert the Power in dBm to RMS in Vrms

Power in dBm is given by:

$$P_{\text{dBm}} = 10 \log_{10}(\text{power in watt}/1 \text{ milliwatt})$$

Voltage in RMS is given by:

$$V_{\text{rms}} = 10^{P_{\text{dBm}}/20} \sqrt{10^{-3}}$$

From the previous example,  $P_{\text{dBm}}$  equals 26.9897 dBm. The  $V_{\text{rms}}$  is calculated as

$$V_{\text{rms}} = 10^{26.9897/20} \sqrt{0.001}$$

which equals 0.7071.

To confirm this value using the peak finder:

- 1 In the **Analyzer** Tab of the Spectrum Analyzer toolstrip, select **Spectrum > RMS**.
- 2 Enable **Peak Finder** in the **Measurements** tab.

### Cursor Measurements

Measure signal values using vertical waveform cursors that track along the signal.

When you click the **Data Cursors** button in the **Measurements** tab of the Spectrum Analyzer, the spectrum display shows vertical cursors on each signal. Each cursor tracks a vertical line along the signal. The scope displays the difference between  $x$ - and  $y$ -values of the signal at the two cursors in the box between the cursors.

To enable cursor measurements, click the **Data Cursors** button in the **Measurements** tab. The cursors appear only when the Spectrum Analyzer has at least one signal in its display.

You can use the mouse to move the vertical cursors left and right.

In the **Measurements** tab, click the **Data Cursors** drop-down arrow to select one of these options:

- **Snap to Data** — To position the cursors on the signal data points.
- **Lock Cursor Spacing** — To lock the frequency difference between the two cursors.

For modifying the cursor measurements programmatically, see the `CursorMeasurementsConfiguration` object.

### Peak Finder Measurements

Compute and display peak values in the scope display.

When you click on the **Peak Finder** button in the **Measurements** tab of the Spectrum Analyzer, an arrow appears on the plot at each maxima and a **Peaks** panel appears at the bottom of the scope window. The Spectrum Analyzer computes peaks from the portion of the input signal that is currently on display in the scope, and the **Peaks** panel shows the peak values and the frequencies at which they occur.

The **Peaks** section in the **Measurements** tab allows you to specify the number of peaks you want the scope to display, the minimum height above which you want the scope to detect peaks, the minimum distance between peaks, and label the peaks.

The Spectrum Analyzer algorithm defines a peak as a local maximum with lower values present on either side of the peak. It does not consider end points as peaks. For more information on the algorithm, see the `findpeaks` function.

The peaks are valid for any units of the input signal. The letter after the value associated with each measurement indicates the abbreviation for the appropriate International System of Units (SI) prefix, such as *m* for *milli*-. For example, if the input signal is measured in volts, an *m* next to a measurement value indicates that this value is in units of millivolts.

For modifying the peak finder measurements programmatically, see the `PeakFinderConfiguration` object. For more information on these settings in the UI, see “Peaks” on page 2-0 .

## Distortion Measurements

Measure harmonic distortion and intermodulation distortion.

When you click the **Distortion** button in the **Distortion** section of the **Measurements** tab, a distortion panel opens at the bottom of the Spectrum Analyzer window. This panel shows the harmonic and distortion measurement values for the input signal currently on display in the scope. The **Distortion** section in the **Measurements** tab allows you to specify the distortion type, number of harmonics, and even label the harmonics.

---

**Note** For an accurate measurement, ensure that the fundamental signal (for harmonics) or primary tones (for intermodulation) is larger than any spurious or harmonic content. To do so, you may need to adjust the resolution bandwidth (RBW) of the Spectrum Analyzer. Make sure that the bandwidth is low enough to isolate the signal and harmonics from spurious noise content. In general, you should set the RBW value such that there is at least a 10 dB separation between the peaks of the sinusoids and the noise floor. You also might need to select a different spectral window to obtain a valid measurement.

---

You can set the **Distortion Type** parameter to one of these values:

- **Harmonic** -- Select **Harmonic** if your input is a single sinusoid.
- **Intermodulation** -- Select **Intermodulation** if your input is two equal-amplitude sinusoids. Intermodulation can help you determine distortion when the scope uses only a small portion of the available bandwidth.

See “Distortion Measurements” on page 2-1341 for information on how distortion measurements are calculated.

### Harmonic Distortion

When you set the **Distortion Type** to **Harmonic**, these fields appear in the **Harmonic Distortion** panel at the bottom of the Spectrum Analyzer window.

- **H1** — Fundamental frequency in Hz and its power in decibels of the measured power referenced to 1 milliwatt (dBm).
- **H2, H3, ...** — Harmonics frequencies in Hz and their power in decibels relative to the carrier (dBc). If the harmonics are at the same level or exceed the fundamental frequency, reduce the input power.
- **THD** — Total harmonic distortion. This value represents the ratio of the power in the harmonics  $D$  to the power in the fundamental frequency  $S$ . If the noise power is too high in relation to the harmonics, the THD value is not accurate. In this case, lower the resolution bandwidth or select a different spectral window.

$$THD = 10 \cdot \log_{10}(D/S)$$

- **SNR** — Signal-to-noise ratio (SNR). This value represents the ratio of the power in the fundamental frequency  $S$  to the power of all nonharmonic content  $N$ , including spurious signals, in decibels relative to the carrier (dBc).

$$SNR = 10 \cdot \log_{10}(S/N)$$

If you see — as the reported SNR, the total nonharmonic content of your signal is less than 30% of the total signal.

- **SINAD** — Signal-to-noise-and-distortion ratio. This value represents the ratio of the power in the fundamental frequency  $S$  to all other content (including noise  $N$  and harmonic distortion  $D$ ) in decibels relative to the carrier (dBc).

$$SINAD = 10 \cdot \log_{10}\left(\frac{S}{N + D}\right)$$

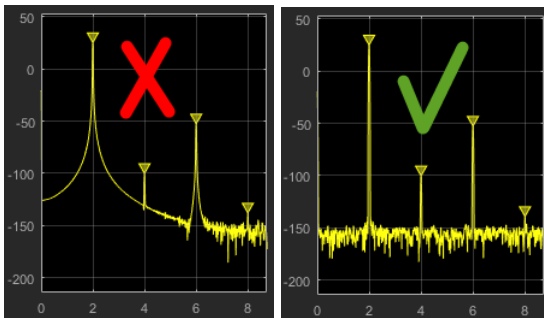
- **SFDR** — Spurious-free dynamic range (SFDR). This value represents the ratio of the power in the fundamental frequency  $S$  to power of the largest spurious signal  $R$  regardless of where it falls in the frequency spectrum. The worst spurious signal might or might not be a harmonic of the original signal. SFDR represents the smallest value of a signal that can be distinguished from a large interfering signal. SFDR includes harmonics.

$$SNR = 10 \cdot \log_{10}(S/R)$$

The harmonic distortion measurement automatically locates the largest sinusoidal component (fundamental signal frequency). It then computes the harmonic frequencies and power in each harmonic in your signal and ignores any DC component. The measurement does not include any harmonics that are outside the Spectrum Analyzer frequency span. Adjust your frequency span so that it includes all the desired harmonics.

**Note** To view the best harmonics, make sure that your fundamental frequency is set high enough to resolve the harmonics. However, this frequency should not be so high that aliasing occurs. For the best display of harmonic distortion, your plot should not show skirts, which indicate frequency leakage. The noise floor should be visible.

For a better display, try a Kaiser window with a large sidelobe attenuation (e.g. between 100–300 db).



### Intermodulation Distortion

When you set the **Distortion Type** to **Intermodulation**, the following fields appear in the **Intermodulation Distortion** panel at the bottom of the Spectrum Analyzer window.

- **F1** — Lower fundamental first-order frequency.
- **F2** — Upper fundamental first-order frequency.
- **2F1 - F2** — Lower intermodulation product from third-order harmonics.



- **2F2 - F1** — Upper intermodulation product from third-order harmonics.
- **TOI** — Third-order intercept point. If the noise power is too high in relation to the harmonics, the TOI value will not be accurate. In this case, you should lower the resolution bandwidth or select a different spectral window. If the TOI has the same amplitude as the input two-tone signal, reduce the power of that input signal.

The intermodulation distortion measurement automatically locates the fundamental and the first-order frequencies ( $F1$  and  $F2$ ). It then computes the frequencies of the third-order intermodulation products ( $2F1 - F2$  and  $2F2 - F1$ ).

For modifying the distortion measurements programmatically, see the `DistortionMeasurementsConfiguration` object. For more information on these settings in the UI, see “Distortion” on page 2-0 .

### Channel Measurements

Measure the occupied bandwidth or adjacent channel power ratio (ACPR).

When you click the **Channel Measurements** button in the **Channel Measurements** tab, a channel measurements panel opens at the bottom of the Spectrum Analyzer window. This panel displays the occupied bandwidth or the adjacent channel power ratio measurements. In the **Channel Measurements** tab, you can specify the occupied bandwidth or the ACPR settings, frequency span, center frequency, and start and stop frequencies.

You can select the channel measurements **Type** to:

- **Occupied BW** -- Occupied bandwidth
- **ACPR** -- Ratio of the power of the main channel to the power of the adjacent channel

For more details on how the Spectrum Analyzer calculates the occupied bandwidth, see “Occupied BW” on page 2-1341.

### Occupied Bandwidth

When you set the **Type** of channel measurement to compute and display to **Occupied BW**, these fields appear in the measurements panel at the bottom of the scope window.

- **Channel Power** — Total power in the channel
- **Occupied BW** — Bandwidth containing the specified **Occupied BW (%)** of the total power of the spectrum.
- **Frequency Error** — Difference between the center of the occupied band and the center frequency (**Center Frequency (Hz)**) of the channel

### ACPR

When you set the **Type** of channel measurement to compute and display to ACPR, these fields appear in the measurements panel at the bottom of the scope window.

- **Lower (Rel Power (dBc))** — Ratio of the power of the lower sideband to the power of the main channel
- **Upper (Rel Power (dBc))** — Ratio of the power of the upper sideband to the power of the main channel

To modify the channel measurements programmatically, see the `ChannelMeasurementsConfiguration` object. For more information on these settings in the UI, see “Channel Measurements” on page 2-0 .

### Spectral Mask

Visualize spectrum limits and compare spectrum values to specification values.

Add upper and lower masks to the Spectrum Analyzer to visualize spectrum limits and compare spectrum values to specification values. To enable the **Spectral Mask** tab, select **Spectrum** in the **Analyzer** tab. When you click the **Upper Mask** and **Lower Mask** buttons in the **Spectral Mask** tab, a **Spectral Mask** panel opens at the bottom of the Spectrum Analyzer window. This panel provides information on pass-fail statistics of masks, names of masks currently failing or passing, and names of channels causing the failure.

You can modify the mask settings in the **Spectral Mask** tab. For more information on these settings in the UI, see Spectral Mask on page 2-1322. For modifying the channel measurements programmatically, see the `SpectralMaskConfiguration` object.


### Check Spectral Masks

You can check the status of the spectral mask using the `getSpectralMaskStatus` function. This function gives details on the number of times a mask succeeded or failed, names of channels causing mask failure, and so on.

You can even use the `MaskTestFailed` event to perform an action every time the mask fails. To trigger a function when the mask fails, create a listener to the `MaskTestFailed` event and define a callback function to trigger it. For more details about using events, see “Events”.

### Customize Visualization

Set configuration and style settings in the Spectrum Analyzer.

To control the settings of the display and labels, color and styling, click on **Settings** () in the **Analyzer** tab of the Spectrum Analyzer toolbar.


In the dialog box that opens, you can customize the font size, plot type, y-axis properties of the spectrum plot, and color map properties of the spectrogram plot. You can change the color of the spectrum plot, background, axes, and labels and also change the line properties.




When you view the spectrum or the spectrogram, you see only the relevant options. For more details about these options, see Configuration > Spectrum Settings on page 2-0 .

### Display Controls

Zoom and pan axes using display controls.

To scale the plot axes, use the mouse to pan around the axes and the scroll button on your mouse to zoom in and out of the plot. Additionally, you can use the buttons that appear when you hover over the plot window.

-  — Maximize the axes, hide all labels and inset the axes values.

-  — Zoom in on the plot.
-  — Pan the plot.
-  — Autoscale the axes to fit the shown data.

## Tips

- To close the scope window and clear its associated data, use the MATLAB `clear` function.
- To hide or show the scope window, use the `hide` and `show` functions.
- Use the MATLAB `mcc` function to compile code containing a Spectrum Analyzer.

## Algorithms

### Spectrum Estimation — Filter Bank

When you choose the `Filter Bank` method, the Spectrum Analyzer uses an analysis filter bank to estimate the power spectrum.

The filter bank splits the broadband input signal  $x(n)$ , of sample rate  $fs$ , into multiple narrow band signals  $y_0(m)$ ,  $y_1(m)$ , ...,  $y_{M-1}(m)$ , of sample rate  $fs/M$ .

The variable  $M$  represents the number of frequency bands in the filter bank. In the Spectrum Analyzer,  $M$  is equal to the number of data points needed to achieve the specified RBW value or 1024, whichever is larger. For more information on the analysis filter bank and its implementation, see the “More About” on page 4-198 and the “Algorithm” on page 4-200 sections in the `dsp.Channelizer` object.

After the Spectrum Analyzer splits the broadband input signal into multiple narrow bands, it computes the power in each narrow frequency band using the following equation. Each  $Z_i$  value is the power estimate over that narrow frequency band.

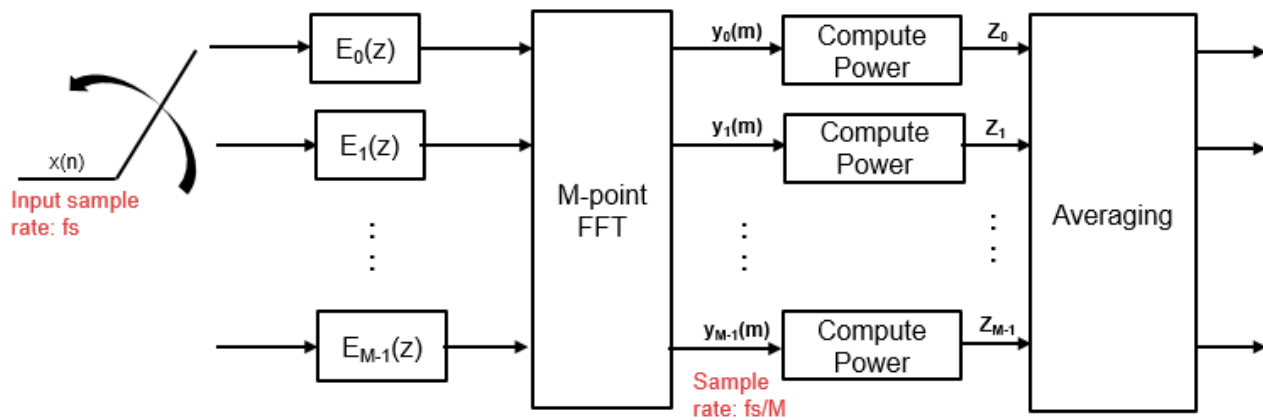
$$Z_i = \frac{1}{L} \sum_{m=0}^{L-1} |y_i[m]|^2$$

$L$  is length of the narrowband signal  $y_i(m)$  and  $i = 1, 2, \dots, M-1$ .

The power values in all the narrow frequency bands (denoted by  $Z_i$ ) form the  $Z$  vector.

$$Z = [Z_0, Z_1, Z_2, \dots, Z_{M-1}]$$

The Spectrum Analyzer averages the current  $Z$  vector with the previous  $Z$  vectors using one of the two moving average methods: video bandwidth or exponential weighting. The output of the averaging operation forms the spectral estimate vector. For details on the two averaging methods, see “Averaging Method” on page 4-1085.



The Spectrum Analyzer uses the value you specify in the **RBW (Hz)** parameter to determine the input frame length. To view the **RBW (Hz)** parameter in the scope, click the **Analyzer** tab on the Spectrum Analyzer toolstrip and navigate to the **Bandwidth** section.

Spectrum Analyzer requires a minimum number of samples to compute a spectral estimate. This value is directly related to the resolution bandwidth property **RBW (Hz)**.

When you set **RBW (Hz)** to:

- **Auto** -- The Spectrum Analyzer requires 1024 samples to update the display. The Spectrum Analyzer determines the appropriate resolution bandwidth to ensure that there are 1024 RBW intervals over the specified frequency span. When you set **RBW (Hz)** to Auto, the Spectrum Analyzer calculates RBW using this equation.

$$RBW_{auto} = \frac{span}{1024}$$

- **scalar value** -- The Spectrum Analyzer calculates the number of samples  $N_{samples}$  using this equation.

$$N_{samples} = \frac{F_s}{RBW}$$

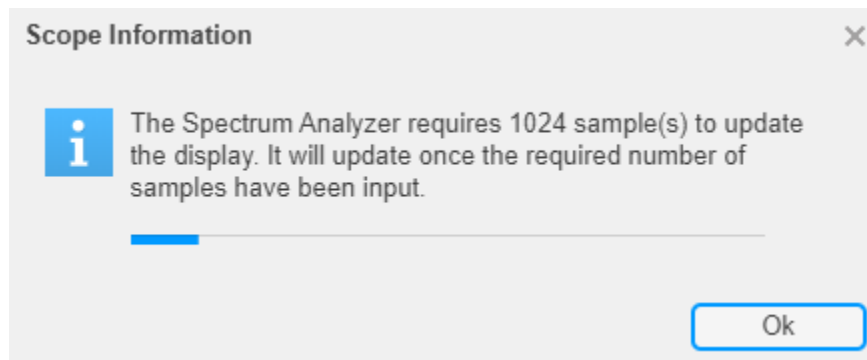
$F_s$  is the sample rate of the input signal as specified in the **Sample Rate (Hz)** property. To view the **Sample Rate (Hz)** in the scope, click the **Analyzer** tab on the Spectrum Analyzer toolstrip and navigate to the **Bandwidth** section.

When you specify a resolution bandwidth using the **RBW (Hz)** parameter, you must specify a value such that there are at least two RBW intervals over the specified frequency span. The ratio of the overall span to RBW must be greater than two.

$$\frac{span}{RBW} > 2$$

$span$  is the frequency span over which the Spectrum Analyzer computes and plots the spectrum. To view the **Span (Hz)** in the scope, click the **Estimation** tab on the Spectrum Analyzer toolstrip and navigate to the **Frequency Options** section. To enable this property, set **Frequency Span** to Span and Center Frequency.

When the number of input samples is not sufficient to achieve the specified resolution bandwidth, the Spectrum Analyzer displays a message similar to this one.



The Spectrum Analyzer removes this message and displays a spectral estimate once you provide enough input samples.

### Spectrum Estimation – Welch's Method

When you select the Welch method, the power spectrum estimate is the averaged modified periodograms.

The algorithm in the Spectrum Analyzer consists of these steps:

- 1 The block buffers the input into  $N$ -point data segments. Each data segment is split into  $P$  overlapping data segments, each of length  $M$ , overlapping by  $D$  points. The data segments can be represented as:

$$x_i(n) = x(n + iD), \quad n = 0, 1, \dots, M - 1$$

$$i = 0, 1, \dots, P - 1$$

- If  $D = M/2$ , the overlap is 50%.
- If  $D = 0$ , the overlap is 0%.

- 2 Apply a window to each of the  $P$  overlapping data segments in the time domain.

The Spectrum Analyzer uses **RBW (Hz)** in the **Analyzer** tab to determine the data window length  $N_{window}$  internally. Then, it partitions the input signal into a number of windowed data segments.

Most window functions afford more influence to the data at the center of the set than to the data at the edges, which represents a loss of information. To mitigate that loss, the individual data sets are commonly overlapped in time. For each windowed segment, compute the periodogram by computing the discrete Fourier transform. Then compute the squared magnitude of the result and divide the result by  $M$ .

$$P_{xx}^i(f) = \frac{1}{MU} \left| \sum_{n=0}^{M-1} x_i(n)w(n)e^{-j2\pi fn} \right|^2, \quad i = 0, 1, \dots, P - 1$$

where  $U$  is the normalization factor for the power in the window function and is given by

$$U = \frac{1}{M} \sum_{n=0}^{M-1} w^2(n)$$

You can specify the window using the **Window** parameter in the **Estimation** tab of the Spectrum Analyzer toolstrip.

- 3 The Spectrum Analyzer calculates and plots the power spectrum, power spectrum density, and RMS using the modified Periodogram estimator. For more information about the Periodogram method, see **periodogram**.

To determine the power spectrum estimate for Welch's method, the Spectrum Analyzer averages the result of the periodograms for the last  $P$  data segments. The averaging reduces the variance, compared to the original  $N$ -point data segment. For more details on the averaging, see "Averaging Method" on page 4-1085.

$$\text{PSD}(f) = \frac{1}{P} \sum_{i=0}^{P-1} P_{xx}^i(f)$$

- 4 The Spectrum Analyzer computes the power spectral density using:

$$\text{PSD}(f) = \frac{1}{P * F_s} \sum_{i=0}^{P-1} P_{xx}^i(f)$$

- 5 The power spectrum is the product of the power spectral density and the resolution bandwidth, as given by this equation.

$$P_{\text{spectrum}}(f) = \text{PSD}(f) \times \text{RBW} = \text{PSD}(f) \times \frac{F_s \times \text{NENBW}}{N_{\text{window}}}$$

- 6 The Spectrum Analyzer plots the power as a spectrogram in the **Spectrogram** mode. Each line of the spectrogram is one periodogram. The time resolution of each line is  $1/\text{RBW}$ , which is the minimum attainable resolution. Achieving the resolution you want might require combining several periodograms. You then use interpolation to calculate noninteger values of  $1/\text{RBW}$ . In the spectrogram display, time scrolls from top to bottom, so the most recent data appears at the top of the display. The offset shows the time value at which the center of the most current spectrogram line occurred.

The Spectrum Analyzer requires a minimum number of samples to compute a spectral estimate. This value is directly related to the resolution bandwidth (RBW).

$$N_{\text{samples}} = \frac{\left(1 - \frac{O_p}{100}\right) \times \text{NENBW} \times F_s}{\text{RBW}}$$

where  $O_p$  on page 4-1080 is the overlap percentage, NENBW on page 4-1081 is the normalized effective noise bandwidth,  $F_s$  on page 4-1081 is the input sample rate, and RBW on page 4-1081 is the resolution bandwidth.

The Spectrum Analyzer shows the number of samples per update in the Spectrum Analyzer status bar.

You can enable **Samples/Update** in the status bar only when you set **Input Domain** to Time and **Method** to Welch in the **Estimation** tab on the Spectrum Analyzer toolstrip.

### Overlap Percentage ( $O_p$ )

The overlap percentage  $O_p$  is the value you specify in the **Overlap %** property. To view the **Overlap %** in the scope, click the **Estimation** tab on the Spectrum Analyzer toolstrip and navigate to the **Window Options** section.

When you increase the overlap percentage, the Spectrum Analyzer needs fewer new input samples to compute a new spectral update.

| $O_p$ | $N_{samples}$ |
|-------|---------------|
| 0%    | 100           |
| 50%   | 50            |
| 80%   | 20            |

### Normalized Effective Noise Bandwidth (NENBW)

The normalized effective noise bandwidth  $NENBW$  is a window parameter that measures the noise performance of the window.  $NENBW$  is determined using the window length and the window coefficients, and is given by the following equation:

$$NENBW = N_{window} \times \frac{\sum_{n=1}^{N_{window}} w^2(n)}{\left[ \sum_{n=1}^{N_{window}} w(n) \right]^2}$$

$w(n)$  denotes the vector of window coefficients (calculated internally).  $N_{window}$  is the window length the Spectrum Analyzer needs to compute one spectral update, and is directly related to the resolution bandwidth and normalized effective noise bandwidth.

$$N_{window} = \frac{NENBW \times F_s}{RBW}$$

The rectangular window has the smallest  $NENBW$ , with a value of 1. All other windows have a larger  $NENBW$  value. For example, the Hann window has an  $NENBW$  value of approximately 1.5.

The Spectrum Analyzer shows the value of  $NENBW$  in the Spectrum Analyzer status bar.

You can enable  $NENBW$  only when you set **Input Domain** to Time and **Method** to Welch in the **Estimation** tab on the Spectrum Analyzer toolstrip.

### Input Sample Rate ( $F_s$ )

$F_s$  is the sample rate of the input signal. To view the **Sample Rate (Hz)** in the scope, click the **Analyzer** tab on the Spectrum Analyzer toolstrip and navigate to the **Bandwidth** section. You can enable this property in the status bar at the bottom of the Spectrum Analyzer window. Right-click the status bar and select **Sample Rate**.

### Resolution Bandwidth (RBW)

Resolution bandwidth controls the spectral resolution of the displayed signal. The RBW value determines the spacing between frequencies that the scope can resolve. A smaller value gives a higher spectral resolution and lowers the noise floor, that is, the Spectrum Analyzer can resolve frequencies that are closer to each other. However, this comes at the cost of a longer sweep time.

You can set the resolution bandwidth through the **RBW (Hz)** property. To view **RBW (Hz)** in the scope, click the **Analyzer** tab on the Spectrum Analyzer toolstrip and navigate to the **Bandwidth** section.

When you set **RBW (Hz)** to:

- **Auto** -- The Spectrum Analyzer requires 1024 samples to update the display. The Spectrum Analyzer determines the appropriate resolution bandwidth to ensure that there are 1024 RBW intervals over the specified frequency span. When you set **RBW (Hz)** to **Auto**, the Spectrum Analyzer calculates using this equation.

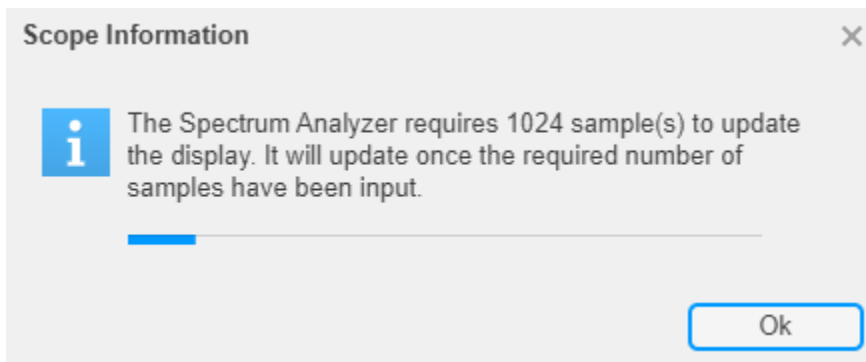
$$RBW_{auto} = \frac{span}{1024}$$

- **scalar value** -- Specify a value such that there are at least two RBW intervals over the specified frequency span. The ratio of the overall span to RBW must be greater than two:

$$\frac{span}{RBW} > 2$$

*span* is the frequency span over which the Spectrum Analyzer computes and plots the spectrum. Spectrum Analyzer shows the span through the **Span (Hz)** property. To view the **Span (Hz)** in the scope, click the **Estimation** tab on the Spectrum Analyzer toolstrip, navigate to the **Frequency Options** section, and set **Frequency Span** to **Span and Center Frequency**.

When the number of input samples is not sufficient to achieve the specified resolution bandwidth, the Spectrum Analyzer displays a message similar to this one.



The Spectrum Analyzer removes this message and displays a spectral estimate once you provide enough input samples.

You can enable this property in the status bar at the bottom of the Spectrum Analyzer window. Right-click the status bar and select **RBW**.

### Nyquist Frequency Interval

When you plot the two-sided spectrum by selecting **Two-Sided Spectrum** in the **Spectrum** or **Spectrogram** tab, the Nyquist frequency interval is

$$\left[ -\frac{SampleRate}{2}, \frac{SampleRate}{2} \right] + FrequencyOffset \text{ Hz.}$$

When you clear the **Two-Sided Spectrum**, the Nyquist frequency interval is

$$\left[ 0, \frac{SampleRate}{2} \right] + FrequencyOffset \text{ Hz.}$$



## Frequency Vector

When you set **Frequency (Hz)** to Auto, the software calculates the frequency vector for the frequency-domain input.

When you plot the two-sided spectrum by selecting **Two-Sided Spectrum** in the **Spectrum** or **Spectrogram** tab, the frequency vector is:

$$\left[ -\frac{\text{SampleRate}}{2}, \frac{\text{SampleRate}}{2} \right]$$

When you clear the **Two-Sided Spectrum**, the frequency vector is:

$$\left[ 0, \frac{\text{SampleRate}}{2} \right]$$

## Occupied BW

The Spectrum Analyzer calculates *Occupied BW* using these steps.

- 1 Calculate the total power in the measured frequency range.
- 2 Determine the lower frequency value. Starting at the lowest frequency in the range and moving upward, sum the power distributed in each frequency until the result is

$$\frac{100 - \text{OccupiedBW}\%}{2}$$

of the total power.

- 3 Determine the upper frequency value. Starting at the highest frequency in the range and moving downward, sum the power distributed in each frequency until the result reaches

$$\frac{100 - \text{OccupiedBW}\%}{2}$$

of the total power.

- 4 The bandwidth between the lower and upper power frequency values is the occupied bandwidth.
- 5 The frequency halfway between the lower and upper frequency values is the center frequency.

## Distortion Measurements

The Spectrum Analyzer calculates Distortion Measurements using these steps.

- 1 Estimate spectral content by finding peaks in the spectrum. When the algorithm detects a peak, it records the width of the peak and clears all monotonically decreasing values by treating all these values as if they belong to the peak. Using this method, the algorithm removes all spectral content centered at DC (0 Hz) from the spectrum and records the amount of bandwidth cleared ( $W_0$ ).
- 2 Determine the fundamental power ( $P_1$ ) from the remaining maximum value of the displayed spectrum. Create a local estimate ( $Fe_1$ ) of the fundamental frequency by computing the central moment of the power near the peak. Record the bandwidth of the fundamental power content ( $W_1$ ). Then remove the power from the fundamental as in step 1.
- 3 Determine the power and width of the higher-order harmonics ( $P_2, W_2, P_3, W_3$ , etc.) in succession by examining the frequencies closest to the appropriate multiple of the local estimate ( $Fe_1$ ).

Remove any spectral content that decreases monotonically about the harmonic frequency from the spectrum before proceeding to the next harmonic.

- 4 After removing the DC, fundamental, and harmonic content from the spectrum, examine the power of the remaining spectrum for its sum ( $P_{remaining}$ ), peak value ( $P_{maxspur}$ ), and median value ( $P_{estnoise}$ ).
- 5 Compute the sum of all the removed bandwidth as  $W_{sum} = W_0 + W_1 + W_2 + \dots + W_n$ .

Compute the sum of powers of the second and higher-order harmonics as  $P_{harmonic} = P_2 + P_3 + P_4 + \dots + P_n$ .

- 6 Estimate the sum of the noise power as:

$$P_{noise} = (P_{remaining} \cdot dF + P_{est.noise} \cdot W_{sum})/RBW$$

Where  $dF$  is the absolute difference between frequency bins, and  $RBW$  is the resolution bandwidth of the window.

- 7 Then compute the metrics for SNR, THD, SINAD, and SFDR from the estimates.

$$THD = 10 \cdot \log_{10} \left( \frac{P_{harmonic}}{P_1} \right)$$

$$SINAD = 10 \cdot \log_{10} \left( \frac{P_1}{P_{harmonic} + P_{noise}} \right)$$

$$SNR = 10 \cdot \log_{10} \left( \frac{P_1}{P_{noise}} \right)$$

$$SFDR = 10 \cdot \log_{10} \left( \frac{P_1}{\max(P_{maxspur}, \max(P_2, P_3, \dots, P_n))} \right)$$

### Harmonic Measurements

- 1 The harmonic distortion measurements use the spectrum trace shown in the display as the input to the measurements. The default Hann window setting of the Spectrum Analyzer might exhibit leakage that can completely mask the noise floor of the measured signal.

The harmonic measurements attempt to correct for leakage by ignoring all frequency content that decreases monotonically away from the maximum of harmonic peaks. If the window leakage covers more than 70% of the frequency bandwidth in your spectrum, you may see a blank reading (-) reported for **SNR** and **SINAD**. If your application can tolerate the increased equivalent noise bandwidth (ENBW), consider using a Kaiser window with a high attenuation (up to 330 dB) to minimize spectral leakage.

- 2 Ignore the DC component.
- 3 After windowing, the width of each harmonic component masks the noise power in the neighborhood of the fundamental frequency and harmonics. To estimate the noise power in each region, the Spectrum Analyzer computes the median noise level in the nonharmonic areas of the spectrum. It then extrapolates that value into each region.
- 4  $N^{\text{th}}$  order intermodulation products occur at  $A \cdot F1 + B \cdot F2$ ,

where  $F1$  and  $F2$  are the sinusoid input frequencies and  $|A| + |B| = N$ .  $A$  and  $B$  are integer values.

- 5 For intermodulation measurements, compute the third-order intercept (TOI) point as follows.

- $TOI_{lower} = P_{F1} + (P_{F2} - P_{(2F1-F2)})/2$
- $TOI_{upper} = P_{F2} + (P_{F1} - P_{(2F2-F1)})/2$
- $TOI = (TOI_{lower} + TOI_{upper})/2$

Where  $P$  is power in decibels of the measured power referenced to 1 milliwatt (dBm).

### Averaging Method

The Spectrum Analyzer can calculate the moving average using two methods:

- Video bandwidth — The Spectrum Analyzer uses a time-domain lowpass filter to smooth the noise in the signal. The video bandwidth (VBW) filter smooths the trace and decreases noise, and the Spectrum Analyzer applies the filter to the data before displaying it.

Video bandwidth is the bandwidth of the lowpass filter that Spectrum Analyzer uses to average or smooth the noise in the signal before displaying it in the scope. The Spectrum Analyzer computes the video bandwidth using this equation:

$$VBW = \frac{(1 - \lambda)RBW}{2\pi\lambda NENBW}$$

where,

- $\lambda$  is the forgetting factor.
- RBW on page 4-1081 is the resolution bandwidth.
- NENBW on page 4-1081 is the normalized effective noise bandwidth.

Video bandwidth does not affect the level of the noise (noise floor), but only increases the signal-to-noise ratio and smooths the trace of the noise. When you decrease the value of VBW, the signal-to-noise ratio improves.

The cutoff frequency of the video bandwidth filter is given by:

$$\omega_c = \frac{2\pi VBW}{F_s/NFFT}$$

where  $F_s$  is the input sample rate and NFFT is the number of FFT points.

The Spectrum Analyzer shows the values of sample rate, VBW, and NFFT in the status bar at the bottom of the display. To enable, right-click the status bar and select **Sample Rate**, **VBW**, and **NFFT**.

- Exponential — The moving average algorithm uses the exponential weighting method to update the weights and compute the moving average recursively for each  $Z$  vector that comes in by using the following recursive equations:

$$w_N = \lambda w_{N-1} + 1$$

$$\bar{z}_N = \left(1 - \frac{1}{w_N}\right)\bar{z}_{N-1} + \left(\frac{1}{w_N}\right)z_N$$

- $\lambda$  — Forgetting factor
- $w_N$  — Weighting factor applied to the current  $Z$  vector
- $z_N$  — Current  $Z$  vector

- $\bar{z}_{N-1}$  — Moving average until the previous  $Z$  vector
- $\left(1 - \frac{1}{w_N}\right)\bar{z}_{N-1}$  — Effect of the previous  $Z$  vectors on the average
- $\bar{z}_N$  — Moving average including the current  $Z$  vector

## Version History

### Introduced in R2022a

#### R2022b: Channel names support array of strings

Starting in R2022b, you can specify the `ChannelNames` property of the `spectrumAnalyzer` object as an array of strings.

```
sa = spectrumAnalyzer(ChannelNames=["Input", "Output"]);
```

#### R2022b: New FilterSharpness property

When the object uses the filter bank spectral estimation method, you can now increase the sharpness of the filter bank in the `spectrumAnalyzer` object. Increasing the filter sharpness decreases the spectral leakage and gives a more accurate power reading.

#### R2022b: New fields on Spectrum Analyzer status bar

Starting in R2022b, the Spectrum Analyzer displays these fields on the status bar:

- **Updates** -- This field shows when you select **Spectrum updates processed** in the Customize Status Bar window.
- **Channel number** -- This field shows when you select **Spectrogram channel** in the Customize Status Bar window.

The **Spectrogram channel** field is enabled only when the display shows spectrogram.

To enable these fields, right-click the status bar and select the properties in the Customize Status Bar window.

For more details, see "Configure Spectrum Analyzer".

#### R2022b: Support for waveform cursors in spectrogram mode

Starting in R2022b, the Spectrum Analyzer supports data cursors even when the scope shows only the spectrogram display.

In the command line, you can edit the `CursorMeasurementsConfiguration` object when you set the `ViewType` to "spectrogram".

#### R2022b: Enhancements to Data Cursor Measurements

Starting in R2022b, the `CursorMeasurementsConfiguration` object has the new `LockSpacing` property. Using this property, you can lock the spacing between waveform cursors in the scope window.

In the **Measurements** tab of the scope UI window, these data cursor settings are new:

- **Lock cursor spacing** -- This setting corresponds to the `LockSpacing` property in the `CursorMeasurementsConfiguration` object.
- **X location** -- These fields are enabled and correspond to the `XLocation` property in the `CursorMeasurementsConfiguration` object.

### **R2023a: SampleRate and PlotAsTwoSidedSpectrum properties are tunable**

Starting in R2023a, the `SampleRate` and `PlotAsTwoSidedSpectrum` properties are tunable, so you can change the values of these properties even when the object is locked.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The `spectrumAnalyzer` object supports MEX code generation by treating the calls to the object as extrinsic.
- The object does not support code generation for standalone applications.

## **See Also**

### **Objects**

`dsp.SpectrumEstimator` | `dsp.CrossSpectrumEstimator` | `timescope` | `dsp.ArrayPlot` | `dsp.LogicAnalyzer` | `dsp.DynamicFilterVisualizer` | `powermeter`

### **Blocks**

Spectrum Analyzer

### **Topics**

“Configure Spectrum Analyzer”

“Spectral Analysis”

“Estimate the Power Spectrum in MATLAB”

## **dsp.SpectrumAnalyzer**

**Package:** dsp

(To be removed) Display frequency spectrum of time-domain signals

---

### **Note**

- The `dsp.SpectrumAnalyzer` object will be removed in a future release. Use the `spectrumAnalyzer` MATLAB® object instead.
- The `CCDFMeasurements` property of the `dsp.SpectrumAnalyzer` object will be removed in a future release. Use the `powermeter` object instead to compute and visualize CCDF measurements.

For more information on how to replace your existing code, see [“Compatibility Considerations”](#).

---

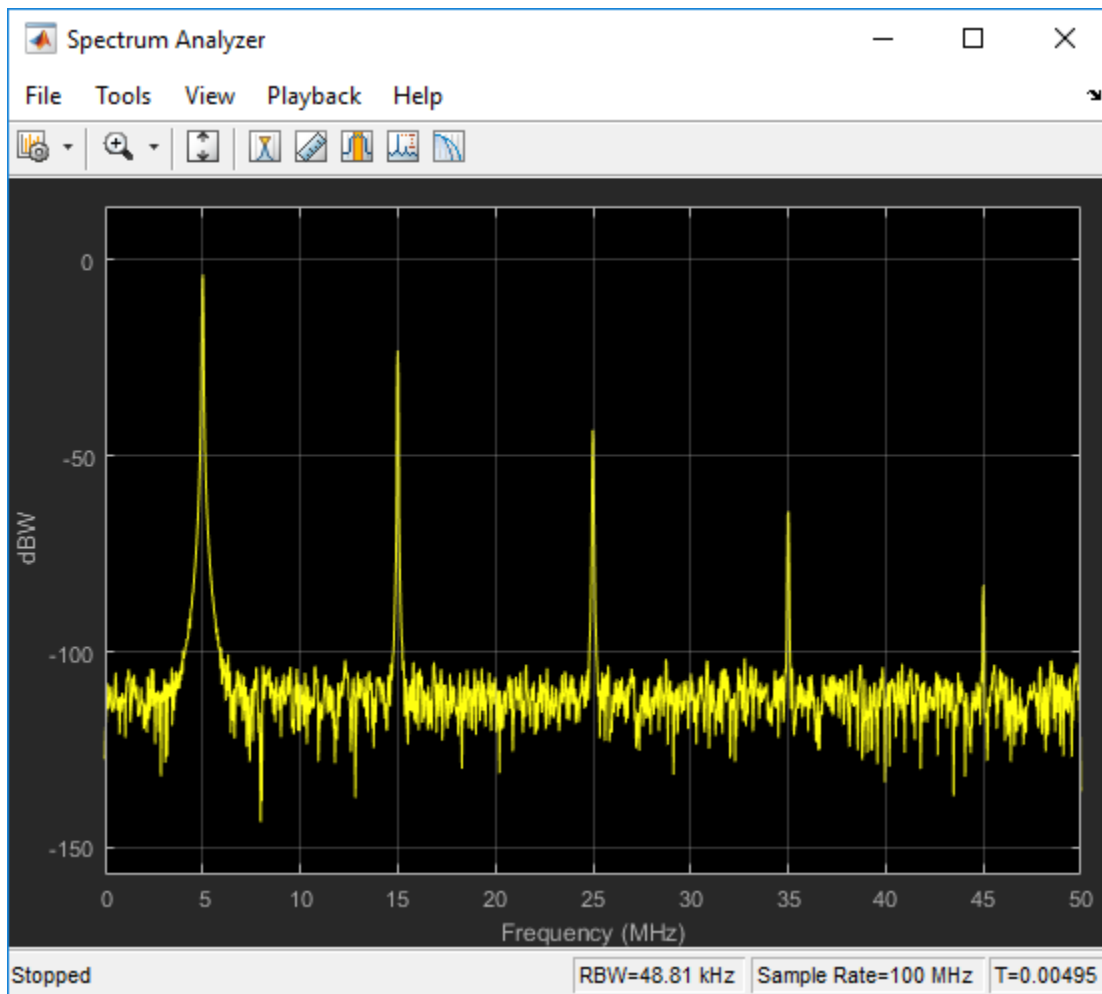
### **Description**

The Spectrum Analyzer System object displays the frequency spectrum of time-domain signals. This scope supports variable-size input, which allows the input frame size to change. Frame size is the first dimension of the input vector. The number of input channels must remain constant.

To display the spectra of signals in the Spectrum Analyzer:

- 1** Create the `dsp.SpectrumAnalyzer` object and set its properties.
- 2** Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)



## Creation

### Syntax

```
scope = dsp.SpectrumAnalyzer
scope = dsp.SpectrumAnalyzer(ports)
scope = dsp.SpectrumAnalyzer(Name,Value)
```

### Description

`scope = dsp.SpectrumAnalyzer` creates a Spectrum Analyzer System object. This object displays the frequency spectrum of real- and complex-valued floating- and fixed-point signals.

`scope = dsp.SpectrumAnalyzer(ports)` creates a Spectrum Analyzer object and sets the `NumInputPorts` property to the value of `ports`.

`scope = dsp.SpectrumAnalyzer(Name,Value)` sets properties using one or more name-value pairs. Enclose each property name in single quotes.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see *System Design in MATLAB Using System Objects*.

### Frequently Used

#### **NumInputPorts** — Number of input ports

1 (default) | integer between [1, 96]

Number of input ports, specified as a positive integer. Each signal coming through a separate input becomes a separate channel in the scope. You must invoke the scope with the same number of inputs as the value of this property.

#### **InputDomain** — Domain of input signal

"Time" (default) | "Frequency"

The domain of the input signal you want to visualize, specified as "Time" or "Frequency". If you visualize time-domain signals, the Spectrum Analyzer transforms the signal to the frequency spectrum based on the algorithm specified in the `Method` property.

#### **Scope Window Use**

In the **Estimation** tab on the Spectrum Analyzer toolstrip, set **Input Domain** to Time or Frequency.

Data Types: char | string

#### **SpectrumType** — Spectrum type

"Power" (default) | "Power density" | "RMS"

Spectrum type, specified as one of these:

"Power" — Power spectrum

"Power density" — Power spectral density. The power spectral density is the magnitude squared of the spectrum normalized to a bandwidth of 1 Hz.

"RMS" — Root mean square. The root mean square shows the square root of the mean square. Use this option to view the frequency of voltage or current signals.

**Tunable:** Yes

#### **Dependency**

To enable this property, set `InputDomain` to "Time".

#### **Scope Window Use**

In the **Analyzer** tab on the Spectrum Analyzer toolstrip, select **Spectrum**. Click **Spectrum** to select Power, Power Density, or RMS.



To enable these options, set the **Input Domain** on the **Estimation** tab to Time.

Data Types: char | string

### ViewType – Viewer type

"Spectrum" (default) | "Spectrogram" | "Spectrum and spectrogram"

Specify the spectrum type as one of "Spectrum", "Spectrogram", or "Spectrum and spectrogram".

- "Spectrum" — Shows the power spectrum.
- "Spectrogram" — Shows frequency content over time. Each line of the spectrogram is one periodogram. Time scrolls from the bottom to the top of the display. The most recent spectrogram update is at the bottom of the display.
- "Spectrum and Spectrogram" — Shows a dual view of a spectrum and spectrogram.

**Tunable:** Yes

### Scope Window Use

In the **Analyzer** tab on the Spectrum Analyzer toolstrip, select **Spectrum**, **Spectrogram**, or both.

Data Types: char | string

### SampleRate – Sample rate of input

10000 (default) | finite scalar

Specify the sample rate, in hertz, of the input signals as a finite numeric scalar.

### Scope Window Use

Open the **Spectrum Settings**. In the **Main options** section, set **Sample rate (Hz)**.

### Method – Spectrum estimation method

"Filter Bank" (default) | "Welch"

Spectrum estimation method, specified as one of the following:

- "Filter bank" -- Use an analysis filter bank to estimate the power spectrum. Compared to Welch's method, this method has a lower noise floor, better frequency resolution, and lower spectral leakage and requires fewer samples per update.
- "Welch" -- Use Welch's method of averaged modified periodograms.

For more details on these methods, see "Algorithms" on page 2-1335.

**Tunable:** Yes

### Dependency

To enable this property, set InputDomain to "Time".

### Scope Window Use

In the **Estimation** tab of the Spectrum Analyzer toolstrip, set **Method** to Filter bank or Welch.

To enable this parameter, set **Input Domain** to Time in the **Estimation** tab.

Data Types: char | string

**PlotAsTwoSidedSpectrum — Option to plot a two-sided spectrum**`true (default) | false`

Option to plot a two-sided spectrum, specified as one of the following:

- `true` — Compute and plot two-sided spectral estimates. When the input signal is complex valued, you must set this property to `true`.
- `false` — Compute and plot one-sided spectral estimates. If you set this property to `false`, then the input signal must be real valued.

When you set this property to `false`, the Spectrum Analyzer uses power-folding. The y-axis values are twice the amplitude that they would be if you were to set this property to `true`, except at 0 and the Nyquist frequency. A one-sided power spectral density (PSD) contains the total power of the signal in the frequency interval from DC to half the Nyquist rate. For more information, see `pwelch`.

**Tunable:** Yes

**Scope Window Use**

Click the **Spectrum** tab or the **Spectrogram** tab (if enabled) of the Spectrum Analyzer toolstrip. In the **Trace Options** section, select **Two-Sided Spectrum** to compute and plot two-sided spectral estimates.

Data Types: `logical`

**FrequencyScale — Frequency scale**`"Linear" (default) | "Log"`

Scale to display frequency, specified as one of the following:

- `"Linear"` — Use a linear scale to display frequencies on the x-axis. To use the `"Linear"` setting, you must also set the `PlotAsTwoSidedSpectrum` property to `true`.
- `"Log"` — Use a logarithmic scale to display frequencies on the x-axis. To use the `"Log"` setting, you must also set the `PlotAsTwoSidedSpectrum` property to `false`.

**Tunable:** Yes

**Scope Window Use**

Click the **Spectrum** tab or the **Spectrogram** tab (if enabled) of the Spectrum Analyzer toolstrip. In the **Scale** section, set the **Frequency Scale** to `Linear` or `Log`.

To set the **Frequency Scale** to `Log`, clear the **Two-Sided Spectrum** check box in the **Trace Options** section in the **Spectrum** or the **Spectrogram** tab (if enabled). If you select the **Two-Sided Spectrum** check box, then you must set the **Frequency Scale** to `Linear`.

Data Types: `char` | `string`

**Advanced****FrequencySpan — Frequency span mode**`"Full" (default) | "Span and center frequency" | "Start and stop frequencies"`

Frequency span mode, specified as one of the following:

- "Full" -- The Spectrum Analyzer computes and plots the spectrum over the entire "Nyquist Frequency Interval" on page 4-1124.
- "Span and center frequency" -- The Spectrum Analyzer computes and plots the spectrum over the interval specified by the Span and CenterFrequency properties.
- "Start and stop frequencies" -- The Spectrum Analyzer computes and plots the spectrum over the interval specified by the StartFrequency and StopFrequency properties.

**Tunable:** Yes

**Dependency**

To enable this property, set InputDomain to "Time".

**Scope Window Use**

Click the **Estimation** tab on the Spectrum Analyzer toolstrip. In the **Frequency Options** section, set **Frequency Span** to Full, Span and Center Frequency, or Start and Stop Frequencies.

To enable the **Frequency Span**, set **Input Domain** to Time.

Data Types: char | string

**Span — Frequency span to compute spectrum**

10e3 (default) | real positive scalar

Specify the frequency span, in hertz, over which the Spectrum Analyzer computes and plots the spectrum. The overall span, defined by this property and the CenterFrequency property, must fall within the "Nyquist Frequency Interval" on page 4-1124.

**Tunable:** Yes

**Dependency**

To enable this property, set FrequencySpan to "Span and center frequency".

**Scope Window Use**

Open the **Spectrum Settings**. In the **Main options** section, clear the **Full frequency span** check box and set Span.

**StartFrequency — Start frequency to compute spectrum**

-5e3 (default) | real scalar

Start of the frequency interval over which spectrum is computed, specified in hertz as a real scalar. The overall span, which is defined by this property and StopFrequency, must fall within the "Nyquist Frequency Interval" on page 4-1124.

**Tunable:** Yes

**Dependency**

To enable this property, set FrequencySpan to "Start and stop frequencies".

**Scope Window Use**

Open the **Spectrum Settings**. In the **Main options** section, clear the **Full frequency span** and change Span to FStart. Set **FStart (Hz)**.

### StopFrequency — Stop frequency to compute spectrum

5e3 (default) | real scalar

End of the frequency interval over which spectrum is computed, specified in hertz as a real scalar. The overall span, which is defined by this property and the StartFrequency property, must fall within the “Nyquist Frequency Interval” on page 4-1124.

**Tunable:** Yes

#### Dependency

To enable this property, set FrequencySpan to "Start and stop frequencies".

#### Scope Window Use

Open the **Spectrum Settings**. In the **Main options** section, clear the **Full frequency span** and change Span to FStart. Set **FStop (Hz)**.

### CenterFrequency — Center of frequency span

0 (default) | real scalar

Specify in hertz the center frequency of the span over which the Spectrum Analyzer computes and plots the spectrum. The overall frequency span, defined by the Span and this property, must fall within the “Nyquist Frequency Interval” on page 4-1124.

**Tunable:** Yes

#### Dependency

To enable this property, set FrequencySpan to "Span and center frequency".

#### Scope Window Use

Open the **Spectrum Settings**. In the **Main**, clear **Full frequency span** and set **CF (Hz)**.

### FrequencyResolutionMethod — Frequency resolution method

"RBW" (default) | "WindowLength" | "NumFrequencyBands"

Specify the frequency resolution method of the Spectrum Analyzer.

- "RBW" - The RBWSource and RBW properties control the frequency resolution (in Hz) of the analyzer. The FFT length is the window length that results from achieving the specified RBW value or 1024, whichever is larger.
- "WindowLength" - Applies only when the Method property is set to "Welch". The WindowLength property controls the frequency resolution. You can control the number of FFT points only when the FrequencyResolutionMethod property is "WindowLength".
- "NumFrequencyBands" - Applies only when the Method property is set to "Filter Bank". The FFTLengthSource and FFTLength properties control the frequency resolution.

**Tunable:** Yes

#### Dependency

To enable this property, set InputDomain to "Time".

**Scope Window Use**

Open the **Spectrum Settings**. In the **Main options** section, set the frequency resolution method by selecting the **RBW (Hz)** dropdown.

Data Types: char | string

**RBWSource — Source of resolution bandwidth value**

"Auto" (default) | "Property"

Specify the source of the resolution bandwidth (RBW) as either "Auto" or "Property".

- "Auto" — The Spectrum Analyzer adjusts the spectral estimation resolution to ensure that there are 1024 RBW intervals over the defined frequency span.
- "Property" — Specify the resolution bandwidth directly using the RBW property.

**Tunable:** Yes

**Dependency**

To enable this property, set either:

- InputDomain to "Time" and FrequencyResolutionMethod to "RBW".
- InputDomain to "Frequency".

**Scope Window Use**

Open the **Spectrum Settings**. In the **Main options** section, set **RBW (Hz)**.

Data Types: char | string

**RBW — Resolution bandwidth**

9.76 (default) | real positive scalar

RBW controls the spectral resolution of Spectrum Analyzer. Specify the resolution bandwidth in hertz as a real positive scalar. You must specify a value to ensure that there are at least two RBW intervals over the specified frequency span. Thus, the ratio of the overall span to RBW must be greater than two:

$$\frac{\text{span}}{\text{RBW}} > 2$$

You can specify the overall span in different ways based on how you set the FrequencySpan property.

**Dependency**

To enable, set:

- RBWSource to "Property"

**Scope Window Use**

Open the **Spectrum Settings**. In the **Main options** section, set **RBW (Hz)**.

**WindowLength — Window length**

1024 (default) | integer greater than 2

Control the frequency resolution by specifying the window length, in samples used to compute the spectral estimates. The window length must be an integer scalar greater than 2.

**Tunable:** Yes

### Dependencies

To enable this property, set:

- FrequencyResolutionMethod to "WindowLength", which controls the frequency resolution based on your window length setting
- Method to "Welch"

### Scope Window Use

Open the **Spectrum Settings**. Change the **RBW (Hz)** dropdown to Window length.

### FFTLengthSource — Source of the FFT length

"Auto" (default) | "Property"

- "Auto" - sets the FFT length to the window length specified in the WindowLength property or 1024, whichever is larger.
- "Property" - number of FFT points using the FFTLength property. FFTLength must be greater than WindowLength.

**Tunable:** Yes

### Dependency

To enable this property, set FrequencyResolutionMethod to "WindowLength".

### Scope Window Use

Open the **Spectrum Settings**. In the **Main options** section, next to the **RBW (Hz)** option, enter a number or select Auto.

Data Types: char | string

### FFTLength — Length of FFT

1024 (default) | positive integer

Specify the length of the FFT that the Spectrum Analyzer uses to compute spectral estimates.

If FrequencyResolutionMethod is "RBW", the FFT length is set as the window length required to achieve the specified resolution bandwidth value or 1024, whichever is larger.

**Tunable:** Yes

### Dependencies

To use this property, the following must be true:

- FrequencyResolutionMethod is set to "WindowLength" or "NumFrequencyBands"
- FFTLength is greater than or equal to the WindowLength.
- FFTLengthSource is set to "Property".

**Scope Window Use**

Open the **Spectrum Settings**. In the **Main options** section, next to the **RBW (Hz)** option, enter a number or select Auto.

**NumTapsPerBand — Number of filter taps per frequency band**

12 (default) | positive even scalar

Specify the number of filter taps or coefficients for each frequency band. This number must be a positive even integer. This value corresponds to the number of filter coefficients per polyphase branch. The total number of filter coefficients is equal to NumTapsPerBand + FFTLength.

**Dependency**

To enable this property, set Method to "Filter Bank"

**Scope Window Use**

Open the **Spectrum Settings**. In the **Main options** section, set **Taps per band**.

**FrequencyVectorSource — Source of frequency vector**

"Auto" (default) | "Property"

- "Auto" — The frequency vector is calculated from the length of the input. See "Frequency Vector" on page 4-1124.
- "Property" — Enter a custom vector as the frequency vector.

**Dependency**

To enable this property, set InputDomain to "Frequency".

**Scope Window Use**

Open the **Spectrum Settings**. In the **Frequency input options** section, set **Frequency (Hz)**.

Data Types: char | string

**FrequencyVector — Custom frequency vector**

[-5000 5000] (default) | monotonically increasing vector

Custom frequency vector, specified as a monotonically increasing vector. This vector determines the x-axis of the display. The vector must be monotonically increasing and must have the same length as the input signal frame size.

**Tunable:** Yes

**Dependency**

To enable this property, set:

- InputDomain to "Frequency".
- FrequencyVectorSource to "Property".

to .

**Scope Window Use**

Click the **Estimation** tab on the Spectrum Analyzer toolstrip. In the **Domain** section, set **Frequency (Hz)** to a monotonically increasing vector of length equal to the input signal frame size.

To enable the **Frequency (Hz)**, set **Input Domain** to Frequency.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**OverlapPercent – Overlap percentage**

0 (default) | real, scalar value

The percentage overlap between the previous and current buffered data segments, specified as a real, scalar value. The overlap creates a window segment that is used to compute a spectral estimate. The value must be greater than or equal to zero and less than 100.

**Tunable:** Yes

**Scope Window Use**

Open the **Spectrum Settings**. In the **Window options** section, set **Overlap (%)**.

**Window – Window function**

"Hann" (default) | "Rectangular" | "Chebyshev" | "Flat Top" | "Hamming" | "Kaiser" | "Blackman-Harris" | "Custom"

Specify a window function for the spectral estimator. The following table shows preset windows. For more information, follow the link to the corresponding function reference in the Signal Processing Toolbox documentation.

| Window Option     | Corresponding Signal Processing Toolbox Function |
|-------------------|--|
| "Rectangular"     | rectwin  |
| "Chebyshev"       | chebwin  |
| "Flat Top"        | flattopwin                                       |
| "Hamming"         | hamming  |
| "Hann"            | hann   |
| "Kaiser"          | kaiser   |
| "Blackman-Harris" | blackmanharris                                   |

To set your own spectral estimation window, set this property to "Custom" and specify a custom window function in the CustomWindow property.

**Tunable:** Yes

**Dependency**

To enable this property, set:

- InputDomain to "Time".
- Method to "Welch"



**Scope Window Use**

Click the **Estimation** tab on the Spectrum Analyzer toolstrip. In the **Window Options** section, set the **Window**.

To enable the **Window**, set **Input Domain** to Time and **Method** to Welch in the **Estimation** tab on the Spectrum Analyzer toolstrip.

Data Types: char | string

**CustomWindow — Name of custom window function**

"hann" (default) | character vector | string scalar

Name of the custom window function, specified as a character vector or string scalar. The name of the custom window function must be on the MATLAB path. Use this property if you want to customize the window using additional properties available with the Signal Processing Toolbox version of the window function.

**Tunable:** Yes

**Example**

Define and use a custom window function.

```
function w = my_hann(L)
    w = hann(L, 'periodic')
end

scope.Window = 'Custom';
scope.CustomWindow = 'my_hann'
```

**Dependency**

To use this property, set Window to "Custom".

**Scope Window Use**

Click the **Estimation** tab on the Spectrum Analyzer toolstrip. In the **Window Options** section, for the **Window**, enter the name of the custom window function.

Data Types: char | string

**SidelobeAttenuation — Sidelobe attenuation of window**

60 (default) | real positive scalar

The window sidelobe attenuation, in decibels (dB). The value must be greater than or equal to 45.

**Tunable:** Yes

**Dependency**

To enable this property, set Window to "Chebyshev" or "Kaiser".

**Scope Window Use**

Open the **Spectrum Settings**. In the **Window options** section, set **Attenuation (dB)**.

**InputUnits — Units of frequency input**

"dBm" (default) | "dBV" | "dBW" | "Vrms" | "Watts"

Select the units of the frequency-domain input. This property allows the Spectrum Analyzer to scale frequency data if you choose a different display unit with the SpectrumUnits property.

#### Dependency

This option is only available when InputDomain is set to Frequency.

#### Scope Window Use

Open the **Spectrum Settings**. In the **Frequency input options** section, set **Input units**.

Data Types: char | string

#### SpectrumUnits — Units of the spectrum

"dBm" (default) | "dBFS" | "dBV" | "dBW" | "Vrms" | "Watts" | "dBm/Hz" | "dBW/Hz" | "dBFS/Hz" | "Watts/Hz" | "Auto"

Specify the units in which the Spectrum Analyzer displays power values.

**Tunable:** Yes

#### Dependency

The spectrum units available depend on the value you specify in the SpectrumType property.

| InputDomain | SpectrumType    | Allowed SpectrumUnits                        |
|-------------|-----------------|--|
| "Time"      | "Power"         | "dBm", "dBW", "dBFS", "Watts"                |
|             | "Power density" | "dBm/Hz", "dBW/Hz", "dBFS/Hz", "Watts/Hz"    |
|             | "RMS"           | "dBV", "Vrms"                                |
| "Frequency" | —               | "Auto", "dBm", "dBV", "dBW", "Vrms", "Watts" |

If you set the InputDomain property to "Frequency" and the SpectrumUnits property to "Auto", the Spectrum Analyzer assumes the spectrum units to be equal to input units specified in the InputUnits property. If you set InputDomain to "Time" and SpectrumUnits to any option other than "Auto", then the Spectrum Analyzer converts the units specified in InputUnits to the units specified in SpectrumUnits.

#### Scope Window Use

Click the **Spectrum** tab on the Spectrum Analyzer toolbar. In the **Scale** section, set **Spectrum Unit**.

Data Types: char | string

#### FullScaleSource — Source of full scale

"Auto" (default) | "Property"

Specify the source of the dBFS scaling factor as either "Auto" or "Property".

- "Auto" -- The Spectrum Analyzer adjusts the scaling factor based on the input data.
- "Property" -- Specify the full-scale scaling factor using the FullScale property.

**Tunable:** Yes

**Dependency**

To enable this property, set:

- InputDomain to "Time"
- SpectrumType to "Power" or "Power density"
- SpectrumUnits to "dBFS" or "dBFS/Hz" (when spectrum type is set to "Power density")

**Scope Window Use**

Click the **Spectrum** tab on the Spectrum Analyzer toolstrip. In the **Scale** section, set the **Full Scale** to either Auto or a positive scalar.

To enable the **Full Scale**:

- In the **Analyzer** tab, set the spectrum type to Power or Power Density.
- In the **Estimation** tab, set **Input Domain** to Time.
- In the **Spectrum** tab, set **Spectrum Unit** to dBFS or dBFS/Hz (when spectrum type is set to Power Density).

Data Types: char | string

**FullScale — Full scale**

1 (default) | positive scalar

Specify a real positive scalar for the dBFS full scale.

**Tunable:** Yes

**Dependency**

To enable this option set:

- SpectrumUnits to "dBFS"
- FullScaleSource to "Property"

**Scope Window Use**

Open the **Spectrum Settings**. In the **Trace options** section, set **Full scale** to Auto or enter a number.

**AveragingMethod — Smoothing method**

"VBW" (default) | "Exponential"

Averaging method, specified as one of the following:

- "VBW" — Video bandwidth method. The object uses a lowpass filter to smooth the trace and decrease noise. Use the VBWSource and VBW properties to specify the VBW value.
- "Exponential" — Weighted average of samples. The object computes the average over samples weighted by an exponentially decaying forgetting factor. Use the ForgettingFactor property to specify the weighted forgetting factor.

For more information, see "Averaging Method" on page 2-1343.

**Tunable:** Yes

### Dependency

To enable this property, set `InputDomain` to "Time".

### Scope Window Use

Click the **Estimation** tab on the Spectrum Analyzer toolstrip. In the **Averaging** section, set **Averaging Method** to VBW or Exponential.

To enable the **Averaging Method**, set **Input Domain** to Time.

Data Types: char | string

### SpectralAverages — Number of spectral averages

1 (default) | positive integer

The Spectrum Analyzer computes the current power spectrum estimate by computing a running average of the last  $N$  power spectrum estimates. This property defines  $N$ .

**Tunable:** Yes

### Dependency

To enable this property, set `ViewType` to "Spectrum".

### Dependency

This property applies only when the `AveragingMethod` is "Running".

### Scope Window Use

Open the **Spectrum Settings**. In the **Trace options** section, set **Averages**.

### ForgettingFactor — Weighting forgetting factor

0.9 (default) | scalar in the range (0,1]

Specify the exponential weighting as a scalar value greater than 0 and less than or equal to 1.

### Dependency

This property applies only when the `AveragingMethod` is "Exponential".

### Scope Window Use

Open the **Spectrum Settings**. In the **Trace options** section, set **Forgetting factor**.

### ReferenceLoad — Reference load

1 (default) | real positive scalar

The load the scope uses as a reference to compute power levels.

**Tunable:** Yes

### Scope Window Use

Open the **Spectrum Settings**. In the **Trace options** section, set **Ref. load (Ohms)**.

### FrequencyOffset — Frequency offset

0 (default) | scalar | vector

- **Scalar** — Apply the same frequency offset to all channels, specified in hertz as a character vector.
- **Vector** — apply a specific frequency offset for each channel, specify a vector of frequencies. The vector length must be equal to number of input channels.

The frequency-axis values are offset by the values specified in this property. The overall span must fall within the “Nyquist Frequency Interval” on page 4-1124. You can control the overall span in different ways based on how you set the `FrequencySpan` property.

**Tunable:** Yes

#### Scope Window Use

Open the **Spectrum Settings**. In the **Trace options** section, set **Offset (Hz)**.

### Spectrogram

#### **SpectrogramChannel** — Channel for which spectrogram is plotted

1 (default) | positive scalar integer

Specify the channel for which the spectrogram is plotted, as a real, positive scalar integer in the range [1  $N$ ], where  $N$  is the number of input channels.

**Tunable:** Yes

#### Dependency

To enable this property, set `ViewType` to "Spectrogram" or "Spectrum and spectrogram".

#### Scope Window Use

Open the **Spectrum Settings**. In the **Spectrogram options** section, select a **Channel**.

#### **TimeResolutionSource** — Source of the time resolution value

"Auto" (default) | "Property"

Specify the source for the time resolution of each spectrogram line as either "Auto" or "Property". The `TimeResolution` property shows the time resolution for the different frequency resolution methods and time resolution properties.

**Tunable:** Yes

#### Dependency

To enable this property, set `ViewType` to "Spectrogram" or "Spectrum and spectrogram".

#### Scope Window Use

Click the **Spectrogram** tab on the Spectrum Analyzer toolstrip. In the **Time Options** section, set the **Time Resolution (s)** to Auto or enter a positive scalar.

To enable the **Time Resolution (s)**, select **Spectrogram** in the **Analyzer** tab.

Data Types: char | string

#### **TimeResolution** — Time resolution

0.001 (default) | positive scalar

Specify the time resolution of each spectrogram line as a positive scalar, expressed in seconds.

The Spectrum Analyzer determines the time resolution value based on the frequency resolution method, RBW, and time resolution properties.

| Method               | RBW              | Time Resolution  | Resulting Time Resolution in Seconds  |
|----------------------|------------------|------------------|---|
| Welch or Filter Bank | Auto             | Auto             | 1/RBW   |
| Welch or Filter Bank | Auto             | Manually entered | 1/Time Resolution Hz  |
| Welch or Filter Bank | Manually entered | Auto             | 1/RBW seconds   |
| Welch or Filter Bank | Manually entered | Manually entered | Equal to or greater than the minimum attainable time resolution 1/RBW. The Spectrum Analyzer combines several spectral estimates into one spectrogram line to obtain the desired time resolution. It uses interpolation to obtain time resolution values that are not integer multiples of 1/RBW. |

**Tunable:** Yes

**Dependency**

To enable this property, set:

- ViewType to "Spectrogram" or "Spectrum and spectrogram"
- TimeResolutionSource to "Property".

**Scope Window Use**

Open the **Spectrum Settings**. In the **Spectrogram options** section, in the **Time res (s)** box, enter a number.

**TimeSpanSource — Source of time span value**

"Auto" (default) | "Property"

Source for the time span of the spectrogram, specified as either "Auto" or "Property". If you set this property to "Auto", the spectrogram displays 100 spectrogram lines at any given time. If you set this property to "Property", the spectrogram uses the time duration you specify in seconds in the TimeSpan property.

**Tunable:** Yes

**Dependency**

To enable this property, set ViewType to "Spectrogram" or "Spectrum and spectrogram".

**Scope Window Use**

Click the **Spectrogram** tab on the Spectrum Analyzer toolbar. In the **Time Options** section, set the **Time Span (s)** to Auto or enter a positive scalar.

Data Types: char | string

**TimeSpan — Time span**

0.1 (default) | positive scalar

Specify the time span of the spectrogram display in seconds. You must set the time span to be at least twice as large as the duration of the number of samples required for a spectral update.

**Tunable:** Yes**Dependency**

To enable this property, set:

- ViewType to "Spectrogram" or "Spectrum and spectrogram".
- TimeSpanSource to "Property".

**Scope Window Use**

Open the **Spectrum Settings**. In the **Spectrogram options** section, in the **Time span (s)** box, enter a number.

**Measurements****MeasurementChannel — Channel for which measurements are obtained**

1 (default) | positive integer

Channel for which the measurements are obtained, specified as a real, positive integer greater than 0 and less than or equal to 100. The maximum number you can specify is the number of channels (columns) in the input signal.

**Tunable:** Yes**Scope Window Use**

Click on **Tools > Measurements** and open the **Trace Selection** settings.

Data Types: double

**SpectralMask — Spectral mask lines**

SpectralMaskSpecification object

Specify whether to display upper and lower spectral mask lines on a spectrum plot. This property uses properties from a SpectralMaskSpecification object to enable and configure the spectral masks.

**Tunable:** Yes**Scope Window Use**

Open the **Spectral Mask** pane and modify the **Settings** options.

**PeakFinder — Peak finder measurement**

PeakFinderConfiguration object

Peak finder measurement, specified as a PeakFinderConfiguration object. Enable peak finder to compute and display the largest calculated peak values. All PeakFinderConfiguration properties are tunable.

**Tunable:** Yes

### Dependency

To enable this property, set `ViewType` to "Spectrum" or "Spectrum and spectrogram".

### Scope Window Use

Click the **Measurements** tab on the Spectrum Analyzer toolstrip and modify the peak finder measurements in the **Peaks** section.

The **Measurements** tab appears when you select **Spectrum** in the **Analyzer** tab.

### CursorMeasurements — Cursor measurements

`CursorMeasurementsConfiguration` object

Cursor measurements, specified as a `CursorMeasurementsConfiguration` object. Enable cursor measurements to display waveform cursors. All `CursorMeasurementsConfiguration` properties are tunable.

**Tunable:** Yes

### Dependency

To enable this property, set `ViewType` to "Spectrum" or "Spectrum and spectrogram".

### Scope Window Use

Click the **Measurements** tab on the Spectrum Analyzer toolstrip and modify the cursor measurements in the **Cursors** section.

The **Measurements** tab appears when you select **Spectrum** in the **Analyzer** tab.

### ChannelMeasurements — Channel measurements

`ChannelMeasurementsSpecification` object

Enable channel measurements to compute and display the occupied bandwidth or adjacent channel power ratio. The `ChannelMeasurements` property uses the `ChannelMeasurementsSpecification` properties.

The `ChannelMeasurementsSpecification` properties are:

- `Algorithm` -- Type of measurement data to display, specified as either "Occupied BW" or "ACPR".  
Default: "Occupied BW"
- `FrequencySpan` -- Frequency span mode, specified as either "Span and center frequency" or "Start and stop frequencies"  
Default: "Span and center frequency"
- `Span` -- Frequency span over which the channel measurements are computed, specified as a real, positive scalar in Hz. This property applies when you set the `FrequencySpan` property to "Span and center frequency".  
Default: 2000 Hz
- `CenterFrequency` -- Center frequency of the span over which the channel measurements are computed, specified as a real scalar in Hz. This property applies when you set the `FrequencySpan` property to "Span and center frequency".



Default: 0 Hz

- **StartFrequency** -- Start frequency over which the channel measurements are computed, specified as a real scalar in Hz. This property applies when you set the **FrequencySpan** property to "Start and stop frequencies".

Default: -1000 Hz

- **StopFrequency** -- Stop frequency over which the channel measurements are computed, specified as a real scalar in Hz. This property applies when you set the **FrequencySpan** property to "Start and stop frequencies".

Default: 1000 Hz

- **PercentOccupiedBW** -- Percent of power over which to compute the occupied bandwidth, specified as a positive real scalar. This property applies when you set the **Algorithm** property to "Occupied BW".

Default: 99

- **NumOffsets** -- Number of adjacent channel pairs, specified as a real, positive integer. This property applies when you set the **Algorithm** property to "ACPR".

Default: 2

- **AdjacentBW** -- Adjacent channel bandwidth, specified as a real, positive scalar. This property applies when you set the **Algorithm** property to "ACPR".

Default: 1000

- **FilterShape** -- Filter shape for both main and adjacent channels, specified as "None", "Gaussian", or "RRC". This property applies when you set the **Algorithm** property to "ACPR".

Default: "None"

- **FilterCoeff** -- Channel filter coefficient, specified as a real scalar between 0 and 1. This property applies when you set the **Algorithm** property to "ACPR" and the **FilterShape** property to either "Gaussian" or "RRC".

Default: 0.5

- **ACPROffsets** -- Frequency of the adjacent channel relative to the center frequency of the main channel, specified as a real vector of length equal to the number of offset pairs specified in **NumOffsets**. This property applies when you set the **Algorithm** property to "ACPR".


Default: [2000 3500]

- **Enable** -- Set this property to **true** to enable channel measurements. Valid values are **true** or **false**.

Default: false

All **ChannelMeasurementsSpecification** properties are tunable.

### Scope Window Use

Open the **Channel Measurements** pane () and modify the **Measurement** and **Channel Settings** options.

### DistortionMeasurements — Distortion measurements

*DistortionMeasurementsConfiguration* object

Distortion measurements, specified as a `DistortionMeasurementsConfiguration` object. Enable distortion measurements to compute and display the harmonic distortion and intermodulation distortion. All `DistortionMeasurementsConfiguration` properties are tunable.

**Tunable:** Yes

### Dependency

To enable this property, set `ViewType` to "Spectrum" or "Spectrum and spectrogram".

### Scope Window Use

Click the **Measurements** tab on the Spectrum Analyzer toolstrip and modify the distortion measurements in the **Distortion** section.

The **Measurements** tab appears when you select **Spectrum** in the **Analyzer** tab.

### Visualization

#### Name — Caption to display in spectrum Analyzer window

"Spectrum Analyzer" (default) | character vector | string scalar

Caption to display in the scope window, specified as a character vector or string scalar.

**Tunable:** Yes

Data Types: char | string

#### Position — Window position

screen center (default) | [left bottom width height]

Spectrum Analyzer window position in pixels, specified by the size and location of the scope window as a four-element double vector of the form [left bottom width height]. You can place the scope window in a specific position on your screen by modifying the values to this property.

By default, the window appears in the center of your screen with a width of 800 pixels and height of 450 pixels. The exact center coordinates depend on your screen resolution.

**Tunable:** Yes

#### PlotType — Plot type to display normal traces

"Line" (default) | "Stem"

Plot type to display normal traces, specified as "Line" or "Stem". Normal traces are traces that display free-running spectral estimates.

**Tunable:** Yes

### Dependencies

To enable this property, set:

- `ViewType` to "Spectrum" or "Spectrum and spectrogram".
- `PlotNormalTrace` to true.

**Scope Window Use**

Click the **Analyzer** tab on the Spectrum Analyzer toolstrip, navigate to the **Configuration** section and click **Settings**. In the Spectrum Analyzer Settings window, under **Display and Labels**, set **Plot Type** to Line or Stem.

To enable the **Plot Type**, you must:

- Select **Spectrum** in the **Views** section of the **Analyzer** tab.
- Enable the **Normal Trace** check box in the **Trace Options** section of the **Spectrum** tab.

Data Types: char | string

**PlotNormalTrace — Normal trace flag**

true (default) | false

To remove normal traces from the display, set this property to `false`. These traces display the free-running spectral estimates. The Spectrum Analyzer continues its spectral computations even when you set this property to `false`.

**Tunable:** Yes

**Dependency**

To enable this property, set `ViewType` to "Spectrum" or "Spectrum and spectrogram".

**Scope Window Use**

Click the **Spectrum** tab on the Spectrum Analyzer toolstrip and select the **Normal Trace** check box in the **Trace Options** section.

To enable the **Normal Trace** check box, select **Spectrum** in the **Analyzer** tab.

Data Types: logical

**PlotMaxHoldTrace — Max-hold trace flag**

false (default) | true

To compute and plot the maximum-hold spectrum of each input channel, set this property to `true`. The maximum-hold spectrum at each frequency bin is computed by keeping the maximum value of all the power spectrum estimates. When you toggle this property, the Spectrum Analyzer resets its maximum-hold computations.

**Tunable:** Yes

**Dependency**

To enable this property, set `ViewType` to "Spectrum" or "Spectrum and spectrogram".

**Scope Window Use**

Click the **Spectrum** tab on the Spectrum Analyzer toolstrip and select the **Max-Hold Trace** check box in the **Trace Options** section.

To enable the **Max-Hold Trace** check box, select **Spectrum** in the **Analyzer** tab.

Data Types: logical

**PlotMinHoldTrace — Min-hold trace flag**`false (default) | true`

To compute and plot the minimum-hold spectrum of each input channel, set this property to `true`. The minimum-hold spectrum at each frequency bin is computed by keeping the minimum value of all the power spectrum estimates. When you toggle this property, the Spectrum Analyzer resets its minimum-hold computations.

**Tunable:** Yes

**Dependency**

To enable this property, set `ViewType` to "Spectrum" or "Spectrum and spectrogram".

**Scope Window Use**

Click the **Spectrum** tab on the Spectrum Analyzer toolstrip and select the **Min-Hold Trace** check box in the **Trace Options** section.

To enable the **Min-Hold Trace** check box, select **Spectrum** in the **Analyzer** tab.

Data Types: `logical`

**Title — Display title**`'' (default) | character vector | string scalar`

Display title, specified as a character vector or a string scalar.

**Tunable:** Yes

**Scope Window Use**

Click the **Analyzer** tab on the Spectrum Analyzer toolstrip. In the **Configuration** section, click **Settings**. In the Spectrum Analyzer Settings window that opens up, under **Display and labels**, enter **Title**.

Data Types: `char | string`

**YLabel — Y-axis label**`'' (default) | character vector | string scalar`

y-axis label, specified as a character vector or a string scalar. The Spectrum Analyzer displays the label to the left of the y-axis.

Regardless of the value of this property, Spectrum Analyzer always displays power units as one of the `SpectrumUnits` values.

**Tunable:** Yes

**Dependency**

To enable this property, set `ViewType` to "Spectrum" or "Spectrum and spectrogram".

**Scope Window Use**

Click the **Analyzer** tab on the Spectrum Analyzer toolstrip. In the **Configuration** section, click **Settings**. In the Spectrum Analyzer Settings window that opens up, under **Display and labels**, enter **Y-Label**.

To enable the **Y-Label**, select **Spectrum** in the **Analyzer** tab.

Data Types: char | string

### ShowLegend — Flag to show legend

false (default) | true

Flag to show the legend, specified as true or false. To show a legend with the input names, set this property to true.

Use the legend to control which signals are visible. In the scope legend, click a signal name to hide the signal in the scope. To show the signal, click the signal name again. To show only one signal, right-click the signal name. To show all signals, press **Esc**.

**Tunable:** Yes

### Dependencies

To enable this property, set the ViewType property to "Spectrum" or "Spectrum and spectrogram".

### Scope Window Use

Click the **Analyzer** tab on the Spectrum Analyzer toolstrip. To see the legend, click **Legend** in the **Configuration** section.

To enable the **Legend**, select **Spectrum** in the **Analyzer** tab.

Data Types: logical

### ChannelNames — Channel names

empty cell (default) | cell array of character vectors | array of strings

Channel names in the input data, specified as a cell array of character vectors or an array of strings. The names you specify in this property appear in the following locations:

- Legend
- **Spectrum Analyzer Settings > Color and styling** section
- **Measurements** and **Channel Measurements** tabs

If you do not specify channel names, the Spectrum Analyzer names the channels as Channel 1, Channel 2, and so on.

**Tunable:** Yes

### Dependency

To see the channel names, set ShowLegend to true.

### Scope Window Use

Click the **Analyzer** tab on the Spectrum Analyzer toolstrip. To see the legend, click **Legend** in the **Configuration** section.

Data Types: char

### ShowGrid — Flag to show grid

true (default) | false

Flag to show the grid, specified as `true` or `false`. Set this property to `true` to show grid lines in the plot.

**Tunable:** Yes

#### **Scope Window Use**

Click the **Analyzer** tab on the Spectrum Analyzer toolstrip. In the **Configuration** section, click **Settings**. In the Spectrum Analyzer Settings window that appears, under **Display and Labels**, select **Show Grid**.

Data Types: `logical`

#### **YLimits — Y-axis limits**

`[-80, 20]` (default) | `[ymin ymax]`

y-axis limits, specified as a two-element numeric vector of the form `[ymin ymax]`. The units of the y-axis limits depend on the `SpectrumUnits` property.

Example: `scope.YLimits = [-10,20]`

**Tunable:** Yes

#### **Dependencies**

- To enable this property, set the `ViewType` property to "Spectrum" or "Spectrum and spectrogram".
- The units directly depend upon the `SpectrumUnits` property.

#### **Scope Window Use**

Click the **Analyzer** tab on the Spectrum Analyzer toolstrip. In the **Configuration** section, click **Settings**. In the Spectrum Analyzer Settings window that opens up, under **Display and Labels**, enter **Y-Axis Limits**.

To enable the **Y-Axis Limits**, select **Spectrum** in the **Analyzer** tab.

#### **ColorLimits — Scale spectrogram color limits**

`[-80, 20]` (default) | `[colorMin colorMax]`

Color limits of the spectrogram, specified as a two-element numeric vector of the form `[colorMin colorMax]`. The units of the color limits directly depend upon the `SpectrumUnits` property.

Example: `scope.ColorLimits = [-10,20]`

**Tunable:** Yes

#### **Dependencies**

- To enable this property, set the `ViewType` property to "Spectrogram" or "Spectrum and spectrogram".
- The units directly depend upon the `SpectrumUnits` property.

#### **Scope Window Use**

Click the **Analyzer** tab on the Spectrum Analyzer toolstrip. In the **Configuration** section, click **Settings**. In the Spectrum Analyzer Settings window that opens up, under **Display and Labels**, enter **Color Limits**.

To enable the **Color Limits**, select **Spectrogram** in the **Analyzer** tab.

### AxesScaling — Axes scaling mode

"Auto" (default) | "Manual" | "OnceAtStop" | "Updates"

Axes scaling mode, specified as one of these:

- "Auto" — The scope scales the axes to fit the data, both during and after simulation.
- "Manual" — The scope does not scale the axes automatically.
- "OnceAtStop" — The scope scales the axes when the simulation stops.
- "Updates" — The scope scales the axes after a specific number of visual updates. It determines the number of updates using the `AxesScalingNumUpdates` property.

**Tunable:** Yes

Data Types: char | string

### AxesLayout — Orientation of the spectrum and spectrogram

"Vertical" (default) | "Horizontal"

Layout of the axes, specified as one of "Vertical" or "Horizontal". A vertical layout stacks the spectrum above the spectrogram. A horizontal layout puts the two views side-by-side.

**Tunable:** Yes

#### Dependency

To enable this property, set `ViewType` to "Spectrum and spectrogram".

#### Scope Window Use

Click the **Analyzer** tab on the Spectrum Analyzer toolstrip. Select **Spectrum** and **Spectrogram**. In the **Configuration** section, select and update **Layout**.

Data Types: char | string

## Usage

### Syntax

```
scope(signal)
scope(signal1,signal2,...,signalN)
```

### Description

`scope(signal)` updates the spectrum of the signal in the spectrum analyzer.

`scope(signal1,signal2,...,signalN)` displays multiple signals in the spectrum analyzer. The signals must have the same frame length, but can vary in number of channels. You must set the `NumInputPorts` property to enable multiple input signals.

### Input Arguments

#### signal — Input signal or signals to visualize

scalar | vector | matrix

Specify one or more input signals to visualize in the `dsp.SpectrumAnalyzer`. Signals can have a different number of channels, but must have the same frame length.

Example: `scope(signal1, signal2)`

Data Types: `single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fi`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

### Specific to `dsp.SpectrumAnalyzer`

|                                    |   |
|------------------------------------|---|
| <code>generateScript</code>        | Generate MATLAB script to create scope with current settings        |
| <code>getMeasurementsData</code>   | Get the current measurement data displayed on the spectrum analyzer |
| <code>getSpectralMaskStatus</code> | Get test results of current spectral mask                           |
| <code>getSpectrumData</code>       | Save spectrum data shown in spectrum analyzer                       |
| <code>isNewDataReady</code>        | Check spectrum analyzer for new data                                |

### Specific to Scopes

|                        |                               |
|------------------------|-------------------------------|
| <code>show</code>      | Display scope window          |
| <code>hide</code>      | Hide scope window             |
| <code>isVisible</code> | Determine visibility of scope |

### Common to All System Objects

|                      |  |
|----------------------|--|
| <code>step</code>    | Run System object algorithm  |
| <code>release</code> | Release resources and allow changes to System object property values and input characteristics |
| <code>reset</code>   | Reset internal states of System object   |

If you want to restart the simulation from the beginning, call `reset` to clear the scope window displays. Do not call `reset` after calling `release`.

## Examples

### Spectrum Analyzer for One-Sided Power Spectrum

View a one-sided power spectrum made from the sum of fixed real sine waves with different amplitudes and frequencies.

```
Fs = 100e6; % Sampling frequency  
fSz = 5000; % Frame size
```

```
sin1 = dsp.SineWave(1e0, 5e6,0,'SamplesPerFrame',fSz,'SampleRate',Fs);  
sin2 = dsp.SineWave(1e-1,15e6,0,'SamplesPerFrame',fSz,'SampleRate',Fs);  
sin3 = dsp.SineWave(1e-2,25e6,0,'SamplesPerFrame',fSz,'SampleRate',Fs);  
sin4 = dsp.SineWave(1e-3,35e6,0,'SamplesPerFrame',fSz,'SampleRate',Fs);  
sin5 = dsp.SineWave(1e-4,45e6,0,'SamplesPerFrame',fSz,'SampleRate',Fs);
```



```

scope = dsp.SpectrumAnalyzer;
scope.SampleRate = Fs;
scope.SpectralAverages = 1;
scope.PlotAsTwoSidedSpectrum = false;
scope.RBWSource = 'Auto';
scope.PowerUnits = 'dBW';
for idx = 1:1e2
    y1 = sin1();
    y2 = sin2();
    y3 = sin3();
    y4 = sin4();
    y5 = sin5();
    scope(y1+y2+y3+y4+y5+0.0001*randn(fSz,1));
end

```

Run the `release` method to let property values and input characteristics change. The scope automatically scales the axes.

```
release(scope)
```

Run the `clear` function to close the Spectrum Analyzer window.

```
clear('scope');
```

### Spectrum Analyzer For Two-Sided Power Spectrum

View a two-sided power spectrum of a sine wave with noise on the Spectrum Analyzer.

```

sin = dsp.SineWave('Frequency',100,'SampleRate',1000);
sin.SamplesPerFrame = 1000;
scope = dsp.SpectrumAnalyzer('SampleRate',sin.SampleRate);
for ii = 1:250
    x = sin() + 0.05*randn(1000,1);
    scope(x);
end

```

Run the `release` method to change property values and input characteristics. The scope automatically scales the axes. It updates the display one more time if any data is in the internal buffer.

```
release(scope);
```

Run the MATLAB `clear` function to close the Spectrum Analyzer window.

```
clear('scope');
```

### Spectrogram of Chirp Signal

This example shows the spectrogram for a chirp signal with added random noise.

```

Fs = 233e3;
frameSize = 20e3;
chirp = dsp.Chirp('SampleRate',Fs,...

```

```
'SamplesPerFrame',frameSize,...
'InitialFrequency',11e3,...
'TargetFrequency',11e3+55e3);

scope = dsp.SpectrumAnalyzer('SampleRate',Fs);
scope.ViewType = 'Spectrogram';
scope.RBWSource = 'Property';
scope.RBW = 500;
scope.TimeSpanSource = 'Property';
scope.TimeSpan = 2;
scope.PlotAsTwoSidedSpectrum = false;

for idx = 1:50
    y = chirp()+ 0.05*randn(frameSize,1);
    scope(y);
end

release(scope)
```

### Display Frequency Input from Spectral Estimation

Use the Spectrum Analyzer to display frequency input from spectral estimates of sinusoids embedded in white Gaussian noise.

#### Initialization

Initialize two `dsp.SpectrumEstimator` objects to display. Set one object to use the Welch-based spectral estimation technique with a Hann window, set the other object use a filter bank estimation. Specify a noisy sine wave input signal with four sinusoids at 0.16, 0.2, 0.205, and 0.25 cycles/sample. View the spectral estimate using a third object, a spectrum analyzer, set to process frequency input.

```
FrameSize = 420;
Fs = 1;
Frequency = [0.16 0.2 0.205 0.25];
sinegen = dsp.SineWave('SampleRate',Fs,'SamplesPerFrame',FrameSize,...
    'Frequency',Frequency,'Amplitude',[2e-5 1 0.05 0.5]);
NoiseVar = 1e-10;
numAvg = 8;

hannEstimator = dsp.SpectrumEstimator('PowerUnits','dBm',...
    'Window','Hann','FrequencyRange','onesided',...
    'SpectralAverages',numAvg,'SampleRate',Fs);

filterBankEstimator = dsp.SpectrumEstimator('PowerUnits','dBm',...
    'Method','Filter bank','FrequencyRange','onesided',...
    'SpectralAverages',numAvg,'SampleRate',Fs);

spectrumPlotter = dsp.SpectrumAnalyzer('InputDomain','Frequency',...
    'SampleRate',Fs,...
    'SpectrumUnits','dBm','YLimits',[-120,40],...
    'PlotAsTwoSidedSpectrum',false,...
    'ChannelNames',{'Hann window','Filter bank'},'ShowLegend',true);
```

#### Streaming

Stream the input. Compare the spectral estimates in the spectrum analyzer.

```

for i = 1:1000
    x = sum(sinegen(),2) + sqrt(NoiseVar)*randn(FrameSize,1);
    Pse_hann = hannEstimator(x);
    Pfb = filterBankEstimator(x);
    spectrumPlotter([Pse_hann,Pfb])
end

```

### Obtain Measurement Data Programmatically for dsp.SpectrumAnalyzer System object

Compute and display the power spectrum of a noisy sinusoidal input signal using the `dsp.SpectrumAnalyzer` System object. Measure the peaks, cursor placements, adjacent channel power ratio, and distortion in the spectrum by enabling the following properties:

- PeakFinder
- CursorMeasurements
- ChannelMeasurements
- DistortionMeasurements

#### Initialization

The input sine wave has two frequencies: 1000 Hz and 5000 Hz. Create two `dsp.SineWave` System objects to generate these two frequencies. Create a `dsp.SpectrumAnalyzer` System object to compute and display the power spectrum.

```

Fs = 44100;
Sineobject1 = dsp.SineWave('SamplesPerFrame',1024,...
    'PhaseOffset',10,...
    'SampleRate',Fs,'Frequency',1000);
Sineobject2 = dsp.SineWave('SamplesPerFrame',1024,...
    'SampleRate',Fs,'Frequency',5000);
SA = dsp.SpectrumAnalyzer('SampleRate',Fs,'Method','Filter bank',...
    'SpectrumType','Power','PlotAsTwoSidedSpectrum',false,...
    'ChannelNames',{'Power spectrum of the input'},...
    'YLimits',[-120 40],'ShowLegend',true);

```

#### Enable Measurements Data

To obtain the measurements, set the `Enable` property of the measurements to `true`.

```

SA.CursorMeasurements.Enable = true;
SA.ChannelMeasurements.Enable = true;
SA.PeakFinder.Enable = true;
SA.DistortionMeasurements.Enable = true;

```

#### Use getMeasurementsData

Stream in the noisy sine wave input signal and estimate the power spectrum of the signal using the spectrum analyzer. Measure the characteristics of the spectrum. Use the `getMeasurementsData` function to obtain these measurements programmatically. The `isNewDataReady` function indicates when there is new spectrum data. The measured data is stored in the variable `data`.

```

data = [];
for Iter = 1:1000
    Sinewave1 = Sineobject1();

```

```
Sinewave2 = Sineobject2();  
Input = Sinewave1 + Sinewave2;  
NoisyInput = Input + 0.001*randn(1024,1);  
SA(NoisyInput);  
    if SA.isNewDataReady  
        data = [data;getMeasurementsData(SA)];  
    end  
end
```

The right side of the spectrum analyzer shows the enabled measurement panes. The values shown in these panes match with the values shown in the last time step of the `data` variable. You can access the individual fields of `data` to obtain the various measurements programmatically.

### Compare Peak Values

Peak values are obtained by the `PeakFinder` property. Verify that the peak values obtained in the last time step of `data` match the values shown on the spectrum analyzer plot.

```
peakvalues = data.PeakFinder(end).Value
```

```
peakvalues = 3×1
```

```
    26.9851  
    24.1735  
   -51.1973
```

```
frequencieskHz = data.PeakFinder(end).Frequency/1000
```

```
frequencieskHz = 3×1
```

```
    4.9957  
    0.9905  
    0.2369
```

## Tips

- To close the scope window and clear its associated data, use the MATLAB `clear` function.
- To hide or show the scope window, use the `hide` and `show` functions.
- Use the MATLAB `mcc` function to compile code containing a Spectrum Analyzer.

You cannot open Spectrum Analyzer configuration dialog boxes if you have more than one compiled component in your application.

## Algorithms

### Spectrum Estimation — Filter Bank

When you choose the `Filter Bank` method, the Spectrum Analyzer uses an analysis filter bank to estimate the power spectrum.

The filter bank splits the broadband input signal  $x(n)$ , of sample rate  $f_s$ , into multiple narrow band signals  $y_0(m)$ ,  $y_1(m)$ , ...,  $y_{M-1}(m)$ , of sample rate  $f_s/M$ .

The variable  $M$  represents the number of frequency bands in the filter bank. In the Spectrum Analyzer,  $M$  is equal to the number of data points needed to achieve the specified RBW value or 1024, whichever is larger. For more information on the analysis filter bank and its implementation, see the “More About” on page 4-198 and the “Algorithm” on page 4-200 sections in the `dsp.Channelizer` object.

After the Spectrum Analyzer splits the broadband input signal into multiple narrow bands, it computes the power in each narrow frequency band using the following equation. Each  $Z_i$  value is the power estimate over that narrow frequency band.

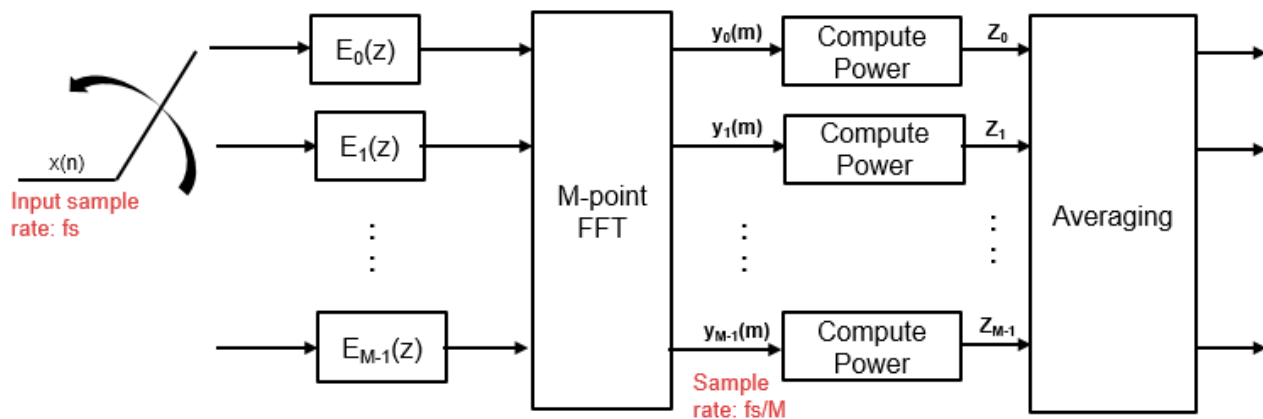
$$Z_i = \frac{1}{L} \sum_{m=0}^{L-1} |y_i[m]|^2$$

$L$  is length of the narrowband signal  $y_i(m)$  and  $i = 1, 2, \dots, M-1$ .

The power values in all the narrow frequency bands (denoted by  $Z_i$ ) form the  $Z$  vector.

$$Z = [Z_0, Z_1, Z_2, \dots, Z_{M-1}]$$

The Spectrum Analyzer averages the current  $Z$  vector with the previous  $Z$  vectors using one of the two moving average methods: video bandwidth or exponential weighting. The output of the averaging operation forms the spectral estimate vector. For details on the two averaging methods, see “Averaging Method” on page 4-1126.



The Spectrum Analyzer uses the value you specify in the **RBW (Hz)** parameter to determine the input frame length. To view the **RBW (Hz)** parameter in the scope, click the **Analyzer** tab on the Spectrum Analyzer toolstrip and navigate to the **Bandwidth** section.

Spectrum Analyzer requires a minimum number of samples to compute a spectral estimate. This value is directly related to the resolution bandwidth property **RBW (Hz)**.

When you set **RBW (Hz)** to:

- Auto -- The Spectrum Analyzer requires 1024 samples to update the display. The Spectrum Analyzer determines the appropriate resolution bandwidth to ensure that there are 1024 RBW

intervals over the specified frequency span. When you set **RBW (Hz)** to Auto, the Spectrum Analyzer calculates RBW using this equation.

$$RBW_{auto} = \frac{span}{1024}$$

- scalar value -- The Spectrum Analyzer calculates the number of samples  $N_{samples}$  using this equation.

$$N_{samples} = \frac{F_s}{RBW}$$

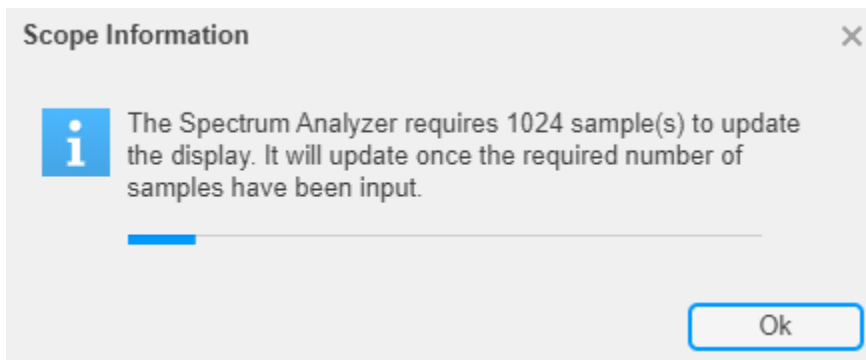
$F_s$  is the sample rate of the input signal as specified in the **Sample Rate (Hz)** property. To view the **Sample Rate (Hz)** in the scope, click the **Analyzer** tab on the Spectrum Analyzer toolstrip and navigate to the **Bandwidth** section.

When you specify a resolution bandwidth using the **RBW (Hz)** parameter, you must specify a value such that there are at least two RBW intervals over the specified frequency span. The ratio of the overall span to RBW must be greater than two.

$$\frac{span}{RBW} > 2$$

$span$  is the frequency span over which the Spectrum Analyzer computes and plots the spectrum. To view the **Span (Hz)** in the scope, click the **Estimation** tab on the Spectrum Analyzer toolstrip and navigate to the **Frequency Options** section. To enable this property, set **Frequency Span** to Span and Center Frequency.

When the number of input samples is not sufficient to achieve the specified resolution bandwidth, the Spectrum Analyzer displays a message similar to this one.



The Spectrum Analyzer removes this message and displays a spectral estimate once you provide enough input samples.

### Spectrum Estimation — Welch's Method

When you select the Welch method, the power spectrum estimate is the averaged modified periodograms.

The algorithm in the Spectrum Analyzer consists of these steps:

- 1 The block buffers the input into  $N$ -point data segments. Each data segment is split into  $P$  overlapping data segments, each of length  $M$ , overlapping by  $D$  points. The data segments can be represented as:

$$x_i(n) = x(n + iD), \quad n = 0, 1, \dots, M - 1$$

$$i = 0, 1, \dots, P - 1$$

- If  $D = M/2$ , the overlap is 50%.
  - If  $D = 0$ , the overlap is 0%.
- 2 Apply a window to each of the  $P$  overlapping data segments in the time domain.

The Spectrum Analyzer uses **RBW (Hz)** in the **Analyzer** tab to determine the data window length  $N_{window}$  internally. Then, it partitions the input signal into a number of windowed data segments.

Most window functions afford more influence to the data at the center of the set than to the data at the edges, which represents a loss of information. To mitigate that loss, the individual data sets are commonly overlapped in time. For each windowed segment, compute the periodogram by computing the discrete Fourier transform. Then compute the squared magnitude of the result and divide the result by  $M$ .

$$P_{xx}^i(f) = \frac{1}{MU} \left| \sum_{n=0}^{M-1} x_i(n)w(n)e^{-j2\pi fn} \right|^2, \quad i = 0, 1, \dots, P - 1$$

where  $U$  is the normalization factor for the power in the window function and is given by

$$U = \frac{1}{M} \sum_{n=0}^{M-1} w^2(n)$$

You can specify the window using the **Window** parameter in the **Estimation** tab of the Spectrum Analyzer toolstrip.

- 3 The Spectrum Analyzer calculates and plots the power spectrum, power spectrum density, and RMS using the modified Periodogram estimator. For more information about the Periodogram method, see [periodogram](#).

To determine the power spectrum estimate for Welch's method, the Spectrum Analyzer averages the result of the periodograms for the last  $P$  data segments. The averaging reduces the variance, compared to the original  $N$ -point data segment. For more details on the averaging, see "Averaging Method" on page 4-1126.

$$\text{PSD}(f) = \frac{1}{P} \sum_{i=0}^{P-1} P_{xx}^i(f)$$

- 4 The Spectrum Analyzer computes the power spectral density using:

$$\text{PSD}(f) = \frac{1}{P * F_s} \sum_{i=0}^{P-1} P_{xx}^i(f)$$

- 5 The power spectrum is the product of the power spectral density and the resolution bandwidth, as given by this equation.

$$P_{\text{spectrum}}(f) = \text{PSD}(f) \times \text{RBW} = \text{PSD}(f) \times \frac{F_s \times NENBW}{N_{\text{window}}}$$

- 6 The Spectrum Analyzer plots the power as a spectrogram in the **Spectrogram** mode. Each line of the spectrogram is one periodogram. The time resolution of each line is  $1/RBW$ , which is the minimum attainable resolution. Achieving the resolution you want might require combining several periodograms. You then use interpolation to calculate noninteger values of  $1/RBW$ . In the spectrogram display, time scrolls from top to bottom, so the most recent data appears at the top of the display. The offset shows the time value at which the center of the most current spectrogram line occurred.

The Spectrum Analyzer requires a minimum number of samples to compute a spectral estimate. This value is directly related to the resolution bandwidth (RBW).

$$N_{samples} = \frac{\left(1 - \frac{O_p}{100}\right) \times NENBW \times F_s}{RBW}$$

where  $O_p$  on page 4-1122 is the overlap percentage,  $NENBW$  on page 4-1122 is the normalized effective noise bandwidth,  $F_s$  on page 4-1123 is the input sample rate, and  $RBW$  on page 4-1123 is the resolution bandwidth.

The Spectrum Analyzer shows the number of samples per update in the Spectrum Analyzer status bar.

You can enable **Samples/Update** in the status bar only when you set **Input Domain** to Time and **Method** to Welch in the **Estimation** tab on the Spectrum Analyzer toolstrip.

### Overlap Percentage ( $O_p$ )

The overlap percentage  $O_p$  is the value you specify in the **Overlap %** property. To view the **Overlap %** in the scope, click the **Estimation** tab on the Spectrum Analyzer toolstrip and navigate to the **Window Options** section.

When you increase the overlap percentage, the Spectrum Analyzer needs fewer new input samples to compute a new spectral update.

| $O_p$ | $N_{samples}$ |
|-------|---------------|
| 0%    | 100           |
| 50%   | 50            |
| 80%   | 20            |

### Normalized Effective Noise Bandwidth (NENBW)

The normalized effective noise bandwidth  $NENBW$  is a window parameter that measures the noise performance of the window.  $NENBW$  is determined using the window length and the window coefficients, and is given by the following equation:

$$NENBW = N_{window} \times \frac{\sum_{n=1}^{N_{window}} w^2(n)}{\left[\sum_{n=1}^{N_{window}} w(n)\right]^2}$$



$w(n)$  denotes the vector of window coefficients (calculated internally).  $N_{window}$  is the window length the Spectrum Analyzer needs to compute one spectral update, and is directly related to the resolution bandwidth and normalized effective noise bandwidth.

$$N_{window} = \frac{NENBW \times F_s}{RBW}$$

The rectangular window has the smallest NENBW, with a value of 1. All other windows have a larger NENBW value. For example, the Hann window has an NENBW value of approximately 1.5.

The Spectrum Analyzer shows the value of *NENBW* in the Spectrum Analyzer status bar.

You can enable NENBW only when you set **Input Domain** to Time and **Method** to Welch in the **Estimation** tab on the Spectrum Analyzer toolstrip.

### Input Sample Rate ( $F_s$ )

$F_s$  is the sample rate of the input signal. To view the **Sample Rate (Hz)** in the scope, click the **Analyzer** tab on the Spectrum Analyzer toolstrip and navigate to the **Bandwidth** section. You can enable this property in the status bar at the bottom of the Spectrum Analyzer window. Right-click the status bar and select **Sample Rate**.

### Resolution Bandwidth (RBW)

Resolution bandwidth controls the spectral resolution of the displayed signal. The RBW value determines the spacing between frequencies that the scope can resolve. A smaller value gives a higher spectral resolution and lowers the noise floor; that is, the Spectrum Analyzer can resolve frequencies that are closer to each other. However, this comes at the cost of a longer sweep time.

You can set the resolution bandwidth through the **RBW (Hz)** property. To view **RBW (Hz)** in the scope, click the **Analyzer** tab on the Spectrum Analyzer toolstrip and navigate to the **Bandwidth** section.

When you set **RBW (Hz)** to:

- **Auto** -- The Spectrum Analyzer requires 1024 samples to update the display. The Spectrum Analyzer determines the appropriate resolution bandwidth to ensure that there are 1024 RBW intervals over the specified frequency span. When you set **RBW (Hz)** to **Auto**, the Spectrum Analyzer calculates using this equation.

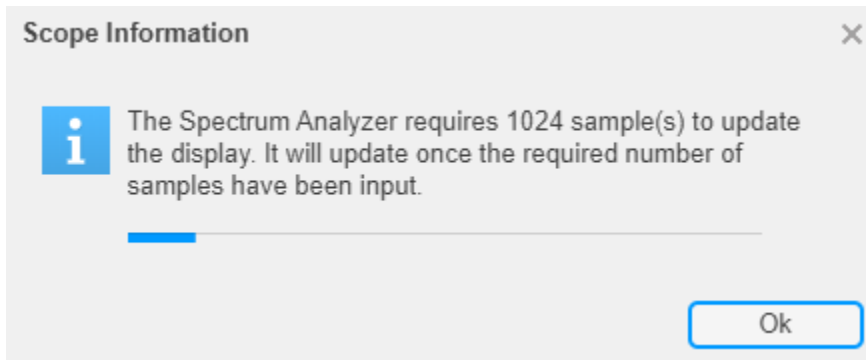
$$RBW_{auto} = \frac{span}{1024}$$

- **scalar value** -- Specify a value such that there are at least two RBW intervals over the specified frequency span. The ratio of the overall span to RBW must be greater than two:

$$\frac{span}{RBW} > 2$$

*span* is the frequency span over which the Spectrum Analyzer computes and plots the spectrum. Spectrum Analyzer shows the span through the **Span (Hz)** property. To view the **Span (Hz)** in the scope, click the **Estimation** tab on the Spectrum Analyzer toolstrip, navigate to the **Frequency Options** section, and set **Frequency Span** to **Span and Center Frequency**.

When the number of input samples is not sufficient to achieve the specified resolution bandwidth, the Spectrum Analyzer displays a message similar to this one.



The Spectrum Analyzer removes this message and displays a spectral estimate once you provide enough input samples.

You can enable this property in the status bar at the bottom of the Spectrum Analyzer window. Right-click the status bar and select RBW.

### Nyquist Frequency Interval

When you plot the two-sided spectrum by selecting **Two-Sided Spectrum** in the **Spectrum** or **Spectrogram** tab, the Nyquist frequency interval is

$$\left[ -\frac{SampleRate}{2}, \frac{SampleRate}{2} \right] + FrequencyOffset \text{ Hz.}$$

When you clear the **Two-Sided Spectrum**, the Nyquist frequency interval is

$$\left[ 0, \frac{SampleRate}{2} \right] + FrequencyOffset \text{ Hz.}$$

### Frequency Vector

When you set **Frequency (Hz)** to Auto, the software calculates the frequency vector for the frequency-domain input.

When you plot the two-sided spectrum by selecting **Two-Sided Spectrum** in the **Spectrum** or **Spectrogram** tab, the frequency vector is:

$$\left[ -\frac{SampleRate}{2}, \frac{SampleRate}{2} \right]$$

When you clear the **Two-Sided Spectrum**, the frequency vector is:

$$\left[ 0, \frac{SampleRate}{2} \right]$$

### Occupied BW

The Spectrum Analyzer calculates *Occupied BW* using these steps.

- 1 Calculate the total power in the measured frequency range.
- 2 Determine the lower frequency value. Starting at the lowest frequency in the range and moving upward, sum the power distributed in each frequency until the result is

$$\frac{100 - OccupiedBW\%}{2}$$

of the total power.

- 3 Determine the upper frequency value. Starting at the highest frequency in the range and moving downward, sum the power distributed in each frequency until the result reaches

$$\frac{100 - \text{OccupiedBW}\%}{2}$$

of the total power.

- 4 The bandwidth between the lower and upper power frequency values is the occupied bandwidth.
- 5 The frequency halfway between the lower and upper frequency values is the center frequency.

### Distortion Measurements

The Spectrum Analyzer calculates Distortion Measurements using these steps.

- 1 Estimate spectral content by finding peaks in the spectrum. When the algorithm detects a peak, it records the width of the peak and clears all monotonically decreasing values by treating all these values as if they belong to the peak. Using this method, the algorithm removes all spectral content centered at DC (0 Hz) from the spectrum and records the amount of bandwidth cleared ( $W_0$ ).
- 2 Determine the fundamental power ( $P_1$ ) from the remaining maximum value of the displayed spectrum. Create a local estimate ( $Fe_1$ ) of the fundamental frequency by computing the central moment of the power near the peak. Record the bandwidth of the fundamental power content ( $W_1$ ). Then remove the power from the fundamental as in step 1.
- 3 Determine the power and width of the higher-order harmonics ( $P_2, W_2, P_3, W_3$ , etc.) in succession by examining the frequencies closest to the appropriate multiple of the local estimate ( $Fe_1$ ). Remove any spectral content that decreases monotonically about the harmonic frequency from the spectrum before proceeding to the next harmonic.
- 4 After removing the DC, fundamental, and harmonic content from the spectrum, examine the power of the remaining spectrum for its sum ( $P_{remaining}$ ), peak value ( $P_{maxspur}$ ), and median value ( $P_{estnoise}$ ).
- 5 Compute the sum of all the removed bandwidth as  $W_{sum} = W_0 + W_1 + W_2 + \dots + W_n$ .

Compute the sum of powers of the second and higher-order harmonics as  $P_{harmonic} = P_2 + P_3 + P_4 + \dots + P_n$ .

- 6 Estimate the sum of the noise power as:

$$P_{noise} = (P_{remaining} \cdot dF + P_{est.noise} \cdot W_{sum})/RBW$$

Where  $dF$  is the absolute difference between frequency bins, and  $RBW$  is the resolution bandwidth of the window.

- 7 Then compute the metrics for SNR, THD, SINAD, and SFDR from the estimates.

$$THD = 10 \cdot \log_{10} \left( \frac{P_{\text{harmonic}}}{P_1} \right)$$

$$SINAD = 10 \cdot \log_{10} \left( \frac{P_1}{P_{\text{harmonic}} + P_{\text{noise}}} \right)$$

$$SNR = 10 \cdot \log_{10} \left( \frac{P_1}{P_{\text{noise}}} \right)$$

$$SFDR = 10 \cdot \log_{10} \left( \frac{P_1}{\max(P_{\text{maxspur}}, \max(P_2, P_3, \dots, P_n))} \right)$$

### Harmonic Measurements

- 1 The harmonic distortion measurements use the spectrum trace shown in the display as the input to the measurements. The default Hann window setting of the Spectrum Analyzer might exhibit leakage that can completely mask the noise floor of the measured signal.

The harmonic measurements attempt to correct for leakage by ignoring all frequency content that decreases monotonically away from the maximum of harmonic peaks. If the window leakage covers more than 70% of the frequency bandwidth in your spectrum, you may see a blank reading (-) reported for **SNR** and **SINAD**. If your application can tolerate the increased equivalent noise bandwidth (ENBW), consider using a Kaiser window with a high attenuation (up to 330 dB) to minimize spectral leakage.

- 2 Ignore the DC component.
- 3 After windowing, the width of each harmonic component masks the noise power in the neighborhood of the fundamental frequency and harmonics. To estimate the noise power in each region, the Spectrum Analyzer computes the median noise level in the nonharmonic areas of the spectrum. It then extrapolates that value into each region.
- 4  $N^{\text{th}}$  order intermodulation products occur at  $A \cdot F1 + B \cdot F2$ ,

where  $F1$  and  $F2$  are the sinusoid input frequencies and  $|A| + |B| = N$ .  $A$  and  $B$  are integer values.

- 5 For intermodulation measurements, compute the third-order intercept (TOI) point as follows.

- $TOI_{\text{lower}} = P_{F1} + (P_{F2} - P_{(2F1-F2)})/2$
- $TOI_{\text{upper}} = P_{F2} + (P_{F1} - P_{(2F2-F1)})/2$
- $TOI = + (TOI_{\text{lower}} + TOI_{\text{upper}})/2$

Where  $P$  is power in decibels of the measured power referenced to 1 milliwatt (dBm).

### Averaging Method

The Spectrum Analyzer can calculate the moving average using two methods:

- Video bandwidth — The Spectrum Analyzer uses a time-domain lowpass filter to smooth the noise in the signal. The video bandwidth (VBW) filter smoothes the trace and decreases noise, and the Spectrum Analyzer applies the filter to the data before displaying it.

Video bandwidth is the bandwidth of the lowpass filter that Spectrum Analyzer uses to average or smooth the noise in the signal before displaying it in the scope. The Spectrum Analyzer computes the video bandwidth using this equation:

$$VBW = \frac{(1 - \lambda)RBW}{2\pi\lambda NENBW}$$

where,

- $\lambda$  is the forgetting factor.
- RBW on page 4-1123 is the resolution bandwidth.
- NENBW on page 4-1122 is the normalized effective noise bandwidth.

Video bandwidth does not affect the level of the noise (noise floor), but only increases the signal-to-noise ratio and smoothes the trace of the noise. When you decrease the value of VBW, the signal-to-noise ratio improves.

The cutoff frequency of the video bandwidth filter is given by:

$$\omega_c = \frac{2\pi VBW}{F_s/NFFT}$$

where  $F_s$  is the input sample rate and NFFT is the number of FFT points.

The Spectrum Analyzer shows the values of sample rate, VBW, and NFFT in the status bar at the bottom of the display. To enable, right-click the status bar and select **Sample Rate, VBW, and NFFT**.

- **Exponential** — The moving average algorithm uses the exponential weighting method to update the weights and compute the moving average recursively for each  $Z$  vector that comes in by using the following recursive equations:

$$w_N = \lambda w_{N-1} + 1$$

$$\bar{z}_N = \left(1 - \frac{1}{w_N}\right)\bar{z}_{N-1} + \left(\frac{1}{w_N}\right)z_N$$

- $\lambda$  — Forgetting factor
- $w_N$  — Weighting factor applied to the current  $Z$  vector
- $z_N$  — Current  $Z$  vector
- $\bar{z}_{N-1}$  — Moving average until the previous  $Z$  vector
- $\left(1 - \frac{1}{w_N}\right)\bar{z}_{N-1}$  — Effect of the previous  $Z$  vectors on the average
- $\bar{z}_N$  — Moving average including the current  $Z$  vector

## Version History

**Introduced in R2012b**

**R2022a: dsp.SpectrumAnalyzer System object will be removed**

*Not recommended starting in R2022a*

The dsp.SpectrumAnalyzer System object will be removed in a future release. Use the spectrumAnalyzer object instead.

**Update Code**

The `spectrumAnalyzer` object has the same properties as the `dsp.SpectrumAnalyzer` object. However, the default value of the `Method` property has changed to `'filter-bank'`, and the default value of the `AveragingMethod` property has changed to `'vbw'`, which is video bandwidth.

No updates to your code are required except for:

- Replacing instances of `dsp.SpectrumAnalyzer` with `spectrumAnalyzer`.
- Updating the values of `Method` and `AveragingMethod` properties, if required.

This table shows how the System object is typically used and explains how to update existing code to use the `spectrumAnalyzer` object.

| Discouraged Usage   | Recommended Replacement  |
|---|--|
| <p><b>Using the default version of <code>dsp.SpectrumAnalyzer</code> object</b></p> <pre>sa = dsp.SpectrumAnalyzer  dsp.SpectrumAnalyzer with properties:      NumInputPorts: 1     InputDomain: 'Time'     SpectrumType: 'Power'     ViewType: 'Spectrum'     SampleRate: 10000     Method: 'Welch'     PlotAsTwoSidedSpectrum: 1     FrequencyScale: 'Linear'  Advanced     FrequencySpan: 'Full'     FrequencyResolutionMethod: 'RBW'     RBWSource: 'Auto'     OverlapPercent: 0     Window: 'Hann'     SpectrumUnits: 'dBm'     AveragingMethod: 'Running'     SpectralAverages: 1     ReferenceLoad: 1     FrequencyOffset: 0</pre> | <p><b>Replacing with the <code>spectrumAnalyzer</code> object</b></p> <p>The <code>spectrumAnalyzer</code> object uses "filter-bank" as the default spectral estimation method. The filter bank method has a lower noise floor, better frequency resolution, and also requires fewer samples per update compared to the Welch method.</p> <p>To retain the same default behavior as the <code>dsp.SpectrumAnalyzer</code> System object, change the <code>Method</code> property to "welch" and the <code>AveragingMethod</code> property to "exponential". If needed, adjust the value of the forgetting factor.</p> <pre>sa = spectrumAnalyzer(Method="welch",AveragingMethod= spectrumAnalyzer with properties:      InputDomain: 'time'     SpectrumType: 'power'     ViewType: 'spectrum'     SampleRate: 10000     Method: 'welch'     PlotAsTwoSidedSpectrum: 1     FrequencyScale: 'linear'     PlotType: 'line'     AxesScaling: 'auto'  Advanced     RBWSource: 'auto'     FrequencySpan: 'full'     OverlapPercent: 0     Window: 'hann'     AveragingMethod: 'exponential'     ForgettingFactor: 0.9000     SpectrumUnits: 'dBm'     ReferenceLoad: 1     FrequencyOffset: 0</pre> |

| Discouraged Usage   | Recommended Replacement   |
|---|---|
| <p><b>Display spectrum</b></p> <p>Display spectrum data on the Spectrum Analyzer using the <code>dsp.SpectrumAnalyzer</code> object.</p> <pre>swv = dsp.SineWave(Frequency=100,SampleRate=1000); swv.SamplesPerFrame = 1000; san = dsp.SpectrumAnalyzer(SampleRate=swv.SampleRate); data = []; for ii = 1:250     x = swv() + 0.05*randn(1000,1);     san(x);     if san.isNewDataReady         data = [data;getSpectrumData(san)];     end end release(san);</pre> | <p><b>Display spectrum</b></p> <p>Display spectrum data on the Spectrum Analyzer using the <code>spectrumAnalyzer</code> object.</p> <pre>swv = dsp.SineWave(Frequency=100,SampleRate=1000); swv.SamplesPerFrame = 1000; san = spectrumAnalyzer(SampleRate=swv.SampleRate, ...     Method="welch",AveragingMethod="exponential"); data = []; for ii = 1:250     x = swv() + 0.05*randn(1000,1);     san(x);     if san.isNewDataReady         data = [data;getSpectrumData(san)];     end end release(san);</pre> |

**R2022a: CCDF measurements will be removed from the `dsp.SpectrumAnalyzer` object**  
*Warns starting in R2022a*

The `CCDFMeasurements` property will be removed from the `dsp.SpectrumAnalyzer` object. If you try to edit these measurements from the command line or from the user interface (UI), the object throws a warning message.

Use the `powermeter` object instead to plot and visualize the CCDF measurements.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Supports MEX code generation by treating the calls to the object as extrinsic. Does not support code generation for standalone applications.
- See “System Objects in MATLAB Code Generation” (MATLAB Coder).

## See Also

### Objects

`timescope` | `dsp.ArrayPlot` | `dsp.LogicAnalyzer` | `dsp.DynamicFilterVisualizer` | `powermeter`

### Blocks

Spectrum Analyzer

### Topics

“Spectral Analysis”  
 “Estimate the Power Spectrum in MATLAB”  
 “View the Spectrogram Using Spectrum Analyzer”

# **dsp.SpectrumEstimator**

**Package:** dsp

Estimate power spectrum or power density spectrum

## **Description**

The `dsp.SpectrumEstimator` System object computes the power spectrum or the power density spectrum of a signal using the Welch algorithm or the filter bank approach.

When you choose the Welch method, the object computes the averaged modified periodograms to compute the spectral estimate. When you choose the filter bank approach, an analysis filter bank splits the broadband input signal into multiple narrow subbands. The object computes the power in each narrow frequency band, and the computed value is the spectral estimate over the respective frequency band. For signals with relatively small FFT lengths, the filter bank approach produces a spectral estimate with a higher resolution, a more accurate noise floor, and peaks more precise than the Welch method, with low or no spectral leakage. These advantages come at the expense of increased computation and slower tracking.

The spectrum can be expressed in watts or in decibels. This object can also estimate the max-hold and min-hold spectra of the signal.

To estimate the power density spectrum:

- 1 Create the `dsp.SpectrumEstimator` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## **Creation**

### **Syntax**

```
SE = dsp.SpectrumEstimator  
SE = dsp.SpectrumEstimator(Name,Value)
```

### **Description**

`SE = dsp.SpectrumEstimator` returns a System object, `SE`, that computes the frequency power spectrum or the power density spectrum of real or complex signals. This System object uses the Welch's averaged modified periodogram method or the filter bank-based spectral estimation method.

`SE = dsp.SpectrumEstimator(Name,Value)` returns a `dsp.SpectrumEstimator` System object with each specified property name set to the specified value. Unspecified properties have default values.



## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see *System Design in MATLAB Using System Objects*.

### SpectrumType — Spectrum type

'Power' (default) | 'Power density'

Spectrum type, specified as either 'Power' or 'Power density'. When the spectrum type is 'Power', the power density spectrum is scaled by the equivalent noise bandwidth of the window (in Hz).

**Tunable:** Yes

### FFTLengthSource — Source of FFT length value

'Auto' (default) | 'Property'

Source of the FFT length value, specified as either 'Auto' or 'Property'. If you set this property to 'Auto', the spectrum estimator sets the FFT length to the input frame size. If you set this property to 'Property', then you specify the number of FFT points using the `FFTLength` property.

### FFTLength — FFT length

128 (default) | positive integer

Specify the length of the FFT that the spectrum estimator uses to compute spectral estimates as a positive integer.

### Dependencies

This property applies when you set the `FFTLengthSource` property to 'Property'.

Data Types: `double`

### Method — Welch or filter bank

'Welch' (default) | 'Filter bank'

Specify the spectral estimation method:

- 'Welch' — The object uses Welch's averaged modified periodograms method.
- 'Filter bank' — An analysis filter bank splits the broadband input signal into multiple narrow subbands. The object computes the power in each narrow frequency band, and the computed value is the spectral estimate over the respective frequency band.

### Window — Window function

'Hann' (default) | 'Rectangular' | 'Chebyshev' | 'Flat Top' | 'Hamming' | 'Kaiser'

Specify a window function for the spectral estimator as one of 'Rectangular', 'Chebyshev', 'Flat Top', 'Hamming', 'Hann', or 'Kaiser'.

**Dependencies**

This property applies when you set `Method` to `'Welch'`.

**NumTapsPerBand — Number of filter taps per frequency band**

12 (default) | positive integer

Specify the number of filter coefficients, or taps, for each frequency band. This value corresponds to the number of filter coefficients per polyphase branch. The total number of filter coefficients is given by `NumTapsPerBand × FFTLength`.

**Dependencies**

This property applies when you set `Method` to `'Filter bank'`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**FrequencyRange — Frequency range of the spectrum estimate**

`'twosided'` (default) | `'onesided'` | `'centered'`

Specify the frequency range of the spectrum estimator as one of `'twosided'`, `'onesided'`, or `'centered'`.

If you set the `FrequencyRange` to `'onesided'`, then the spectrum estimator computes the one-sided spectrum of a real input signal. When the FFT length, `NFFT`, is even, the spectrum estimate has length  $(NFFT/2) + 1$  and is computed over the frequency range  $[0, \text{SampleRate}/2]$ , where `SampleRate` is the sample rate of the input signal. When `NFFT` is odd, the spectrum estimate has length  $(NFFT + 1)/2$  and is computed over the frequency range  $[0, \text{SampleRate}/2)$ .

If you set the `FrequencyRange` to `'twosided'`, then the spectrum estimator computes the two-sided spectrum of a complex or real input signal. The length of the spectrum estimate is equal to the FFT length. The spectrum estimate is computed over the frequency range  $[0, \text{SampleRate})$ , where `SampleRate` is the sample rate of the input signal.

If you set the `FrequencyRange` to `'centered'`, then the spectrum estimator computes the centered two-sided spectrum of a complex or real input signal. The length of the spectrum estimate is equal to the FFT length. The spectrum estimate is computed over the frequency range  $(-\text{SampleRate}/2, \text{SampleRate}/2]$  when the FFT length is even and  $(-\text{SampleRate}/2, \text{SampleRate}/2)$  when the FFT length is odd.

**PowerUnits — Power units**

`'Watts'` (default) | `'dBW'` | `'dBm'`

Specify the units used to measure power as one of `'Watts'`, `'dBW'`, or `'dBm'`.

**AveragingMethod — Averaging method**

`'Running'` (default) | `'Exponential'`

Specify the averaging method as `'Running'` or `'Exponential'`. In the running averaging method, the object computes an equally weighted average of a specified number of spectrum estimates defined by the `SpectralAverages` property. In the exponential method, the object computes the average over samples weighted by an exponentially decaying forgetting factor.

**SpectralAverages — Number of spectral averages**

8 (default) | positive integer

Number of spectral averages, specified as a positive integer. The spectrum estimator computes the current power spectrum or power density spectrum estimate by averaging the last  $N$  estimates.  $N$  is the number of spectral averages defined in the `SpectralAverages` property.

**Dependencies**

This property applies when you set `AveragingMethod` to 'Running'.

Data Types: double

**ForgettingFactor — Forgetting factor**

0.9 (default) | scalar in the range (0,1]

Specify the exponential weighting forgetting factor as a scalar value greater than zero and smaller than or equal to one.

**Tunable:** Yes

**Dependencies**

This property applies when you set `AveragingMethod` to 'Exponential'.

Data Types: single | double

**ReferenceLoad — Reference load**

1 (default) | positive scalar

Specify the load that the spectrum estimator uses as a reference to compute power values as a real, positive scalar in ohms.

Data Types: single | double

**SidelobeAttenuation — Side lobe attenuation of window**

60 dB (default) | positive scalar

Specify the side lobe attenuation of the window as a real, positive scalar, in decibels (dB).

**Dependencies**

This property applies when you set `Method` to 'Welch' and `Window` to 'Chebyshev' or 'Kaiser'.

Data Types: double

**OutputMaxHoldSpectrum — Output max-hold spectrum**

false (default) | true

Set this property to `true` so that the spectrum estimator computes and outputs the max-hold spectrum of each input channel. The max-hold spectrum is computed by keeping, at each frequency bin, the maximum value of all the power spectrum estimates.

**OutputMinHoldSpectrum — Output min-hold spectrum**

false (default) | true

Set this property to `true` so that the spectrum estimator computes and outputs the min-hold spectrum of each input channel. The min-hold spectrum is computed by keeping, at each frequency bin, the minimum value of all the power spectrum estimates.

**SampleRate — Sample rate of input**

1 (default) | scalar

Sample rate of the input in Hz, specified as a finite numeric scalar. The sample rate is the rate at which the signal is sampled in time.

Data Types: `single` | `double`

## Usage

## Syntax

```
pxx = SE(x)
[pxx, pmax] = SE(x)
[pxx, pmin] = SE(x)
[pxx, pmax, pmin] = SE(x)
```

## Description

`pxx = SE(x)` computes the power spectrum or power-density spectrum, `pxx`, of the input signal, `x`. The System object treats the columns of `x` as independent channels.

`[pxx, pmax] = SE(x)` also computes the max-hold frequency spectrum, `pmax`, of `x`. To determine the max-hold spectrum, the method keeps the maximum of all the power spectrum estimates computed at each frequency bin. Set `OutputMaxHoldSpectrum` to `true` to obtain the max-hold spectrum.

`[pxx, pmin] = SE(x)` also computes the min-hold frequency spectrum, `pmin`, of `x`. To determine the min-hold spectrum, the method keeps the minimum of all the power spectrum estimates computed at each frequency bin. Set `OutputMinHoldSpectrum` to `true` to obtain the min-hold spectrum.

`[pxx, pmax, pmin] = SE(x)` computes the power spectrum or power-density spectrum, the max-hold spectrum, and the min-hold spectrum of `x`. Set `OutputMaxHoldSpectrum` and `OutputMinHoldSpectrum` to `true` to obtain the max-hold and min-hold spectra.

## Input Arguments

### **x** — Input signal

vector | matrix

Input signal, specified as a vector or matrix. The row length of `x` is the frame size or channel length. Each column of `x` is treated as a separate channel. The column length of `x` is the number of channels.

Data Types: `single` | `double`

## Output Arguments

### **pxx** — Power or power-density spectrum estimate

vector | matrix

Power or power-density spectrum estimate, returned as a vector or matrix of the same data type and complexity as the input signal, `x`.

When `FFTLengthSource` is set to:

- `'Auto'` -- The size of `pxx` is same as the size of the input signal, `x`.

- 'Property' -- The size of pxx is the same as the specified FFT length.

By default, the unit of pxx is 'Watts'. You can also specify the spectrum to be in 'dBm' or 'dBW' through the PowerUnits property.

Data Types: single | double

#### **pmax — Max-hold spectrum estimate**

vector | matrix

Max-hold spectrum estimate, returned as a vector or matrix of the same size, data type, and complexity as the output signal, pxx.

Data Types: single | double

#### **pmin — Min-hold spectrum estimate**

vector | matrix

Min-hold spectrum estimate, returned as a vector or matrix of the same size, data type, and complexity as the output signal, pxx.

Data Types: single | double

## **Object Functions**

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named obj, use this syntax:

```
release(obj)
```

### **Specific to dsp.SpectrumEstimator**

getFrequencyVector    Vector of frequencies at which estimation is done

getRBW                Resolution bandwidth of spectrum

### **Common to All System Objects**

step                Run System object algorithm

release            Release resources and allow changes to System object property values and input characteristics

reset              Reset internal states of System object

## **Examples**

### **Power Spectrum of Multichannel Sinusoidal Signal**

Compute the power spectrum of a multichannel sinusoidal signal using the dsp.SpectrumEstimator System object™. You can get the vector of frequencies at which the spectrum is estimated using the getFrequencyVector function. To compute the resolution bandwidth of the estimate (RBW), use the getRBW function.

Generate a three-channel sinusoid sampled at 1 kHz. Specify sinusoidal frequencies of 100, 200, and 300 Hz. The second and third channels have their phases offset from the first by  $\pi/2$  and  $\pi/4$ , respectively.

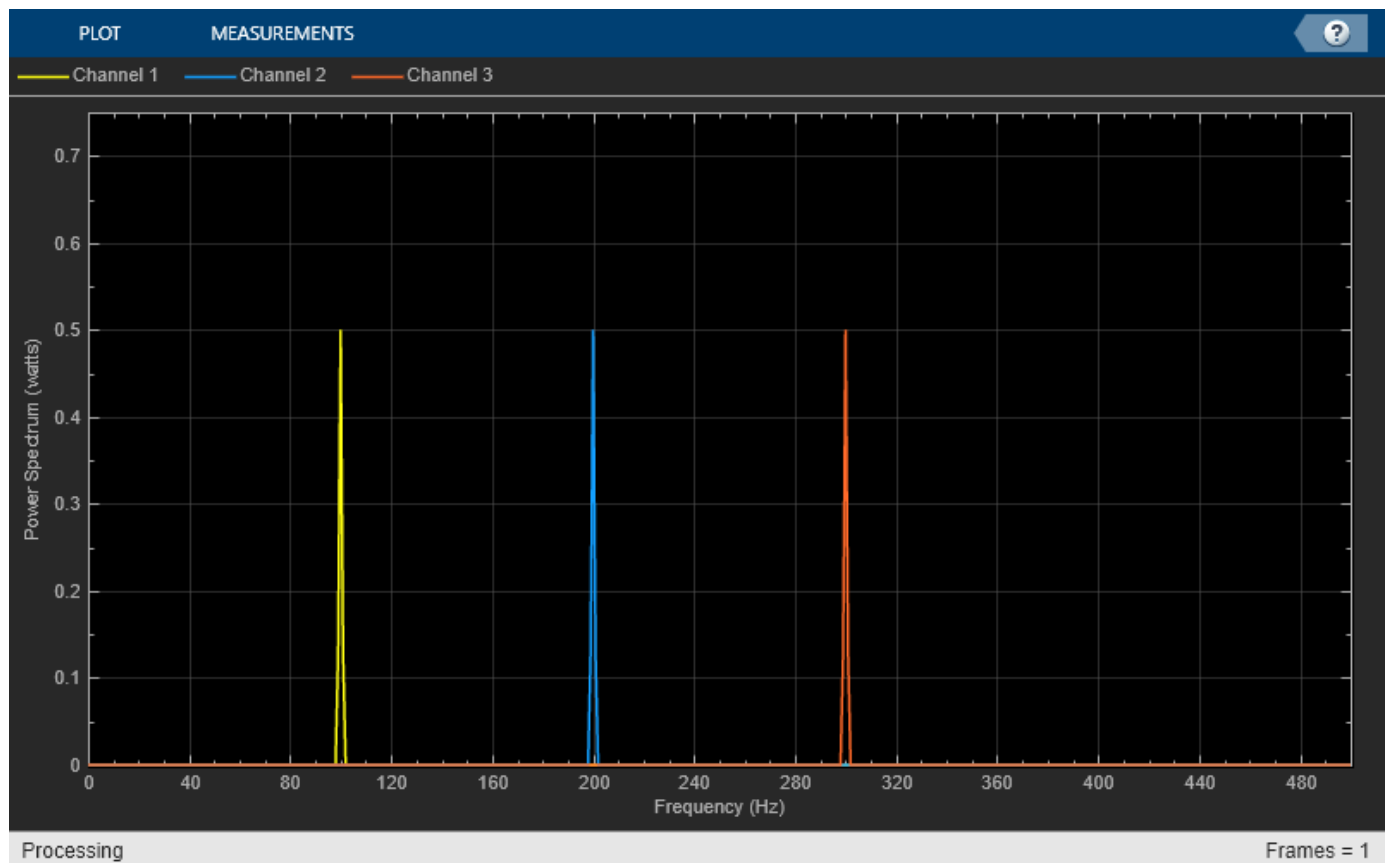
```
sineSignal = dsp.SineWave('SamplesPerFrame',1000,'SampleRate',1000, ...
    'Frequency',[100 200 300],'PhaseOffset',[0 pi/2 pi/4]);
```

Estimate and plot the one-sided spectrum of the signal. Use the `dsp.SpectrumEstimator` object for the computation and the `dsp.ArrayPlot` for the plotting.

```
estimator = dsp.SpectrumEstimator('FrequencyRange','onesided');
plotter = dsp.ArrayPlot('PlotType','Line','YLimits',[0 0.75], ...
    'YLabel','Power Spectrum (watts)','XLabel','Frequency (Hz)');
```

Step through to obtain the data streams and display the spectra of the three channels.

```
y = sineSignal();
pxx = estimator(y);
plotter(pxx)
```



Get the vector of frequencies at which the spectrum is estimated in Hz, using the `getFrequencyVector` function.

```
f = getFrequencyVector(estimator);
```

Compute the resolution bandwidth (RBW) of the estimate using the `getRBW` function.

```
rbw = getRBW(estimator)
```

```
rbw =
```

```
0.0015
```

The resolution bandwidth of the signal power spectrum is 0.0015 Hz. This frequency is the smallest frequency that can be resolved on the spectrum.

## Spectral Estimation Using Filter Bank

Compare spectral estimates of sinusoids embedded in white Gaussian noise using a Hann window-based Welch method and filter bank method.

### Initialization

Initialize two `dsp.SpectrumEstimator` objects. Specify one estimator to use the Welch-based spectral estimation technique with a Hann window. Specify the other estimator to use an analysis filter bank to perform the spectral estimation. Specify a noisy sine wave input signal with 4 sinusoids at 0.16, 0.2, 0.205, and 0.25 cycles/sample. View the spectral estimate using an array plot.

```
FrameSize = 420;
Fs = 1;
sinegen = dsp.SineWave('SampleRate',Fs,...
    'SamplesPerFrame',FrameSize,...
    'Frequency',[0.16 0.2 0.205 0.25],...
    'Amplitude',[2e-5 1 0.05 0.5]);
NoiseVar = 1e-10;
numAvgs = 8;

hannEstimator = dsp.SpectrumEstimator('PowerUnits','dBm',...
    'Window','Hann','FrequencyRange','onesided',...
    'SpectralAverages',numAvgs,'SampleRate',Fs);

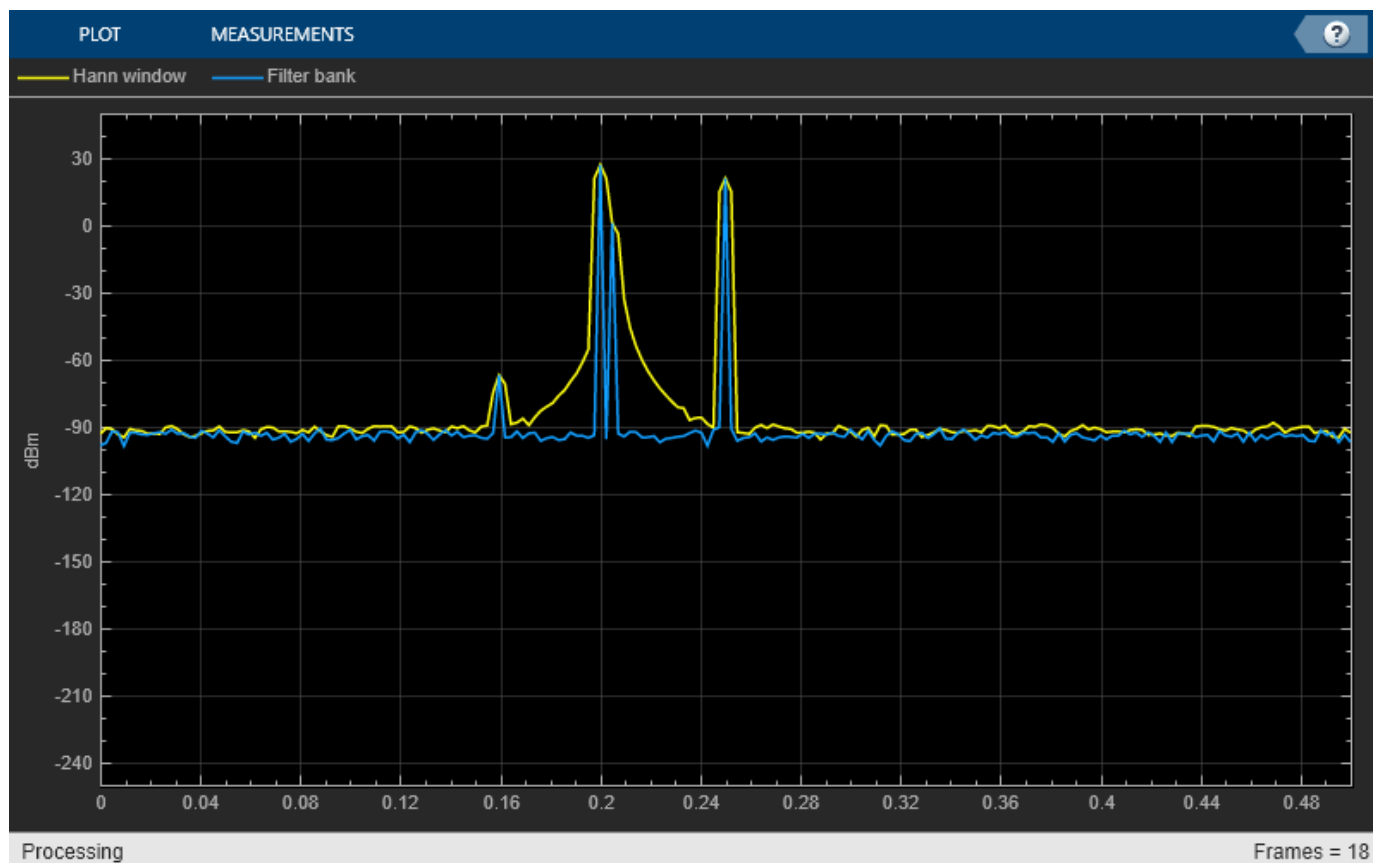
filterBankEstimator = dsp.SpectrumEstimator('PowerUnits','dBm',...
    'Method','Filter bank','FrequencyRange','onesided',...
    'SpectralAverages',numAvgs,'SampleRate',Fs);

spectrumPlotter = dsp.ArrayPlot(...
    'PlotType','Line','SampleIncrement',Fs/FrameSize,...
    'YLimits',[-250,50],'YLabel','dBm',...
    'ShowLegend',true,'ChannelNames',{'Hann window','Filter bank'});
```

### Streaming

Stream the input. Compare the spectral estimates computed using the Hann window and the analysis filter bank

```
for i = 1:1000
    x = sum(sinegen(),2) + sqrt(NoiseVar)*randn(FrameSize,1);
    Pse_hann = hannEstimator(x);
    Pfb = filterBankEstimator(x);
    spectrumPlotter([Pse_hann,Pfb])
end
```



The Hann window misses the peak at 0.205 cycles/sample. In addition, the window has a significant spectral leakage that makes the peak at 0.16 cycles/sample hard to distinguish, and the noise floor is not correct.

The filter bank estimate has a very good resolution with no spectral leakage.

### Power and Max-Hold Spectra of Noisy Sine Wave

Generate a sine wave.

```
sineWave = dsp.SineWave('Frequency',100,...
    'SampleRate',1000, ...
    'SamplesPerFrame',1000);
```

Use the spectrum estimator to compute the power spectrum and the max-hold spectrum of the sine wave. Use the Array Plot to display the spectra.

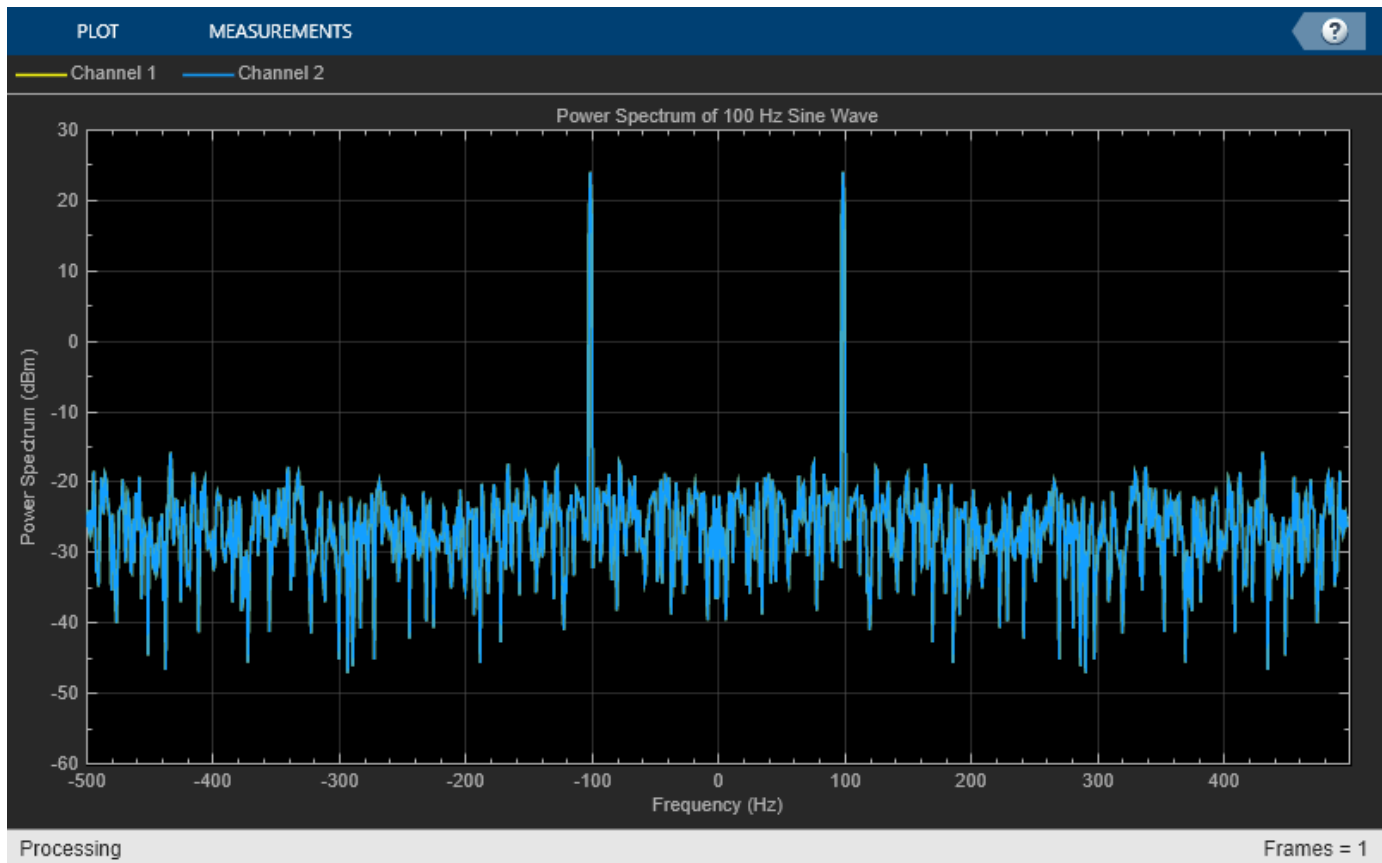
```
SE = dsp.SpectrumEstimator(...
    'SampleRate',sineWave.SampleRate,...
    'SpectrumType','Power','PowerUnits','dBm', ...
    'FrequencyRange','centered',...
    'OutputMaxHoldSpectrum',true);
plotter = dsp.ArrayPlot('PlotType','Line',...
    'Xoffset',-500, ...
    'YLimits',[-60 30],...
    'Yoffset',-500);
```



```
'Title','Power Spectrum of 100 Hz Sine Wave', ...
'YLabel','Power Spectrum (dBm)',...
'XLabel','Frequency (Hz)');
```

Add random noise to the sine wave. Stream in the data, and plot the power spectrum of the signal.

```
for ii = 1:10
    x = sineWave() + 0.05*randn(1000,1);
    [Pxx,Pmax] = SE(x);
    plotter([Pxx Pmax])
end
```



## Algorithms

### Welch's Method of Averaged Modified Periodograms

When you choose the Welch method, the power spectrum estimate is averaged modified periodograms.

Given the signal input,  $x$ :

- 1 Multiply  $x$  by the window and scale the result by the window power.
- 2 Compute the FFT of the signal,  $Y$ , and take the square magnitude using  $Z = Y \cdot \text{conj}(Y)$ .

- 3 Compute the current power spectrum estimate by taking the moving average of the last  $N$  number of  $Z$ 's, and scaling the answer by the sample rate. For details on the moving average methods, see "Averaging Method" on page 4-1141.

### Filter Bank

The filter-bank-based spectrum estimator uses an analysis filter bank to estimate the power spectrum. The filter bank splits a broadband input signal,  $x(n)$ , of sample rate  $f_s$  into multiple narrow band signals,  $y_0(m)$ ,  $y_1(m)$ , ...,  $y_{M-1}(m)$ , of sample rate  $f_s/M$ .

The variable  $M$  represents the number of frequency bands in the filter bank. When you specify FFT length,  $M$  equals the FFT length. When you do not specify FFT length,  $M$  is equal to the number of rows in the input signal. The number of taps per frequency band sets the number of filter coefficients for each frequency band of the filter bank. The total number of filter coefficients is equal to number of taps per band times the number of frequency bands  $M$ . For more information on the analysis filter bank and how it is implemented, see the "More About" on page 4-198 and the "Algorithm" on page 4-200 sections in `dsp.Channelizer`.

After the broadband input signal is split into multiple narrow bands, the spectrum estimator computes the power in each narrow band using the following equation. Each  $Z_i$  value becomes the estimate of the power over that narrow frequency band.

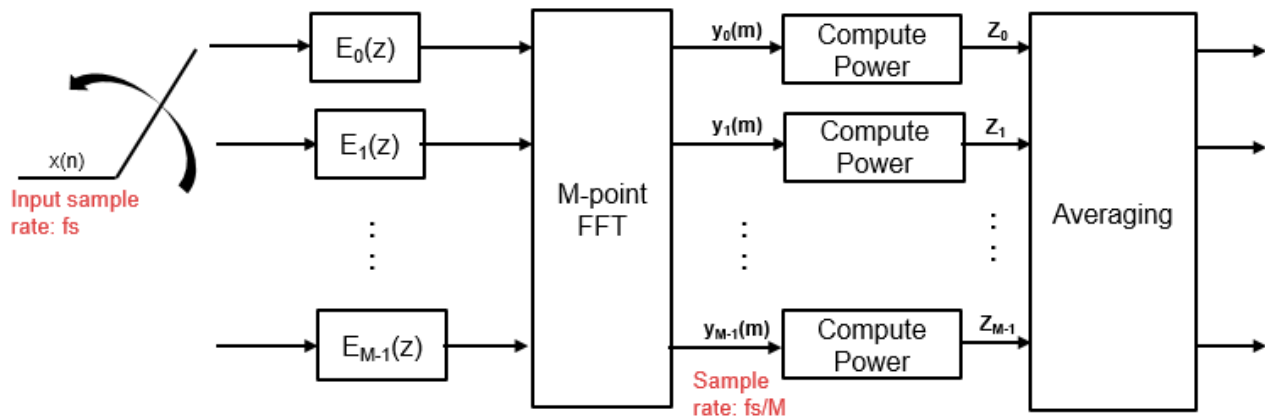
$$Z_i = \frac{1}{L} \sum_{m=0}^{L-1} |y_i[m]|^2$$

$L$  is length of the narrow band signal  $y_i(m)$ , where  $i = 1, 2, \dots, M-1$ .

The power values in all the narrow bands (denoted by  $Z_i$ ) form the  $Z$  vector.

$$Z = [Z_0, Z_1, Z_2, \dots, Z_{M-1}]$$

The filter bank estimator algorithm averages the current  $Z$  vector with the previous  $Z$  vectors using one of the two moving average methods: running or exponential weighting. The output of the averaging operation forms the spectral estimate vector. For details on the two averaging methods, see "Averaging Method" on page 4-1141.



## Averaging Method

The moving average is calculated using one of the two methods:

- **Running** — For each frame of input, average the last  $N$ -scaled  $Z$  vectors, which are computed by the algorithm. The variable  $N$  is the value you specify for the number of spectral averages. If the algorithm does not have enough  $Z$  vectors, the algorithm uses zeros to fill the empty elements.

|                                 |                                    |       |       |     |     |     |     |     |   |
|---------------------------------|------------------------------------|-------|-------|-----|-----|-----|-----|-----|---|
| Input frame size:<br>[1000 1]   | Number of spectral averages (N): 6 |       |       |     |     |     |     |     |   |
|                                 | ← 1000 elements →                  |       |       |     |     |     |     |     |   |
| Z vector →                      | $Z_1$                              | 3     | 4     | 5   | ... | ... | ... | ... | 4 |
|                                 | $Z_2$                              | 5     | 6     | 7   | ... | ... | ... | ... | 8 |
|                                 | $Z_3$                              | 8     | 4     | 2   | ... | ... | ... | ... | 9 |
|                                 | $Z_4$                              | 3     | 2     | 7   | ... | ... | ... | ... | 3 |
|                                 | $Z_5$                              | 0     | 0     | 0   | ... | ... | ... | ... | 0 |
|                                 | $Z_6$                              | 0     | 0     | 0   | ... | ... | ... | ... | 0 |
| Output: $(Z_1+Z_2+\dots+Z_6)/6$ |                                    | 3.167 | 2.667 | 3.5 | ... | ... | ... | ... | 4 |

- **Exponential** — The moving average algorithm using the exponential weighting method updates the weights and computes the moving average recursively for each  $Z$  vector that comes in by using the following recursive equations:

$$w_N = \lambda w_{N-1} + 1$$

$$\bar{z}_N = \left(1 - \frac{1}{w_N}\right)\bar{z}_{N-1} + \left(\frac{1}{w_N}\right)z_N$$

- $\lambda$  — Forgetting factor
- $w_N$  — Weighting factor applied to the current  $Z$  vector
- $z_N$  — Current  $Z$  vector
- $\bar{z}_{N-1}$  — Moving average until the previous  $Z$  vector
- $\left(1 - \frac{1}{w_N}\right)\bar{z}_{N-1}$  — Effect of the previous  $Z$  vectors on the average
- $\bar{z}_N$  — Moving average including the current  $Z$  vector

## Version History

Introduced in R2013b

## References

- [1] Hayes, Monson H. *Statistical Digital Signal Processing and Modeling*. Hoboken, NJ: John Wiley & Sons, 1996
- [2] Kay, Steven M. *Modern Spectral Estimation: Theory and Application*. Englewood Cliffs, NJ: Prentice Hall, 1999
- [3] Stoica, Petre and Randolph L. Moses. *Spectral Analysis of Signals*. Upper Saddle River, NJ: Prentice Hall, 2005
- [4] Welch, P. D. "The use of fast Fourier transforms for the estimation of power spectra: A method based on time averaging over short modified periodograms," *IEEE Transactions on Audio and Electroacoustics*, Vol. 15, 1967, pp. 70-73.
- [5] Harris, F.J. *Multirate Signal Processing for Communication Systems*. Prentice Hall. 2004, pp. 208-209.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- "System Objects in MATLAB Code Generation" (MATLAB Coder)

## See Also

### Functions

`getFrequencyVector` | `getRBW`

### Objects

`spectrumAnalyzer` | `dsp.TransferFunctionEstimator` | `dsp.CrossSpectrumEstimator` | `powermeter`

### Blocks

Spectrum Estimator

### Topics

"Estimate the Power Spectrum in MATLAB"

"High Resolution Spectral Analysis in MATLAB"

# dsp.StateLevels

**Package:** dsp

(Removed) State-level estimation for bilevel rectangular waveform

---

**Note** `dsp.StateLevels` has been removed. Use `statelevels` instead. For more information, see “Compatibility Considerations”.

---

## Description

The `dsp.StateLevels` object estimates the state levels of a bilevel rectangular waveform.

To estimate the state levels of a bilevel waveform:

- 1 Create the `dsp.StateLevels` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
sl = dsp.StateLevels  
sl = dsp.StateLevels(Name,Value)
```

### Description

`sl = dsp.StateLevels` creates a state-level estimation System object, `sl`, that estimates state levels in a bilevel rectangular waveform using the histogram method with 100 bins.

`sl = dsp.StateLevels(Name,Value)` returns a `StateLevels` System object, `sl`, with each specified property set to the specified value.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

### **HistogramBounds — Minimum and maximum levels of histogram**

[0 5] (default) | two-element row vector

Minimum and maximum levels of the histogram. Specify the range of the histogram as a two-element real-valued row vector. Signal values outside the range defined by this property are ignored.

**Dependencies**

This property applies when you set the `Method` property to `'Histogram mode'` or `'Histogram mean'`, and either `RunningStateLevels` is `true`, or the `HistogramBoundsSource` property is set to `'Property'`.

Data Types: `double`

**HistogramBoundsSource — Source of histogram bounds**

`'Auto'` (default) | `'Property'`

Source of histogram bounds. Specify how to determine the histogram bounds as one of `'Auto'` or `'Property'`. When you set this property to `'Auto'`, the histogram bounds are determined by the minimum and maximum input values. When you set this property to `'Property'`, the histogram bounds are determined by the value of the `HistogramBounds` property.

**Dependencies**

This property applies when you set the `Method` property to `'Histogram mode'` or `'Histogram mean'`, and the `RunningStateLevels` property is `false`.

**HistogramNumBins — Number of bins in histogram**

`100` (default) | positive integer

Number of bins in the histogram. Specify the number of bins in the histogram.

**Dependencies**

This property applies when you set the `Method` property to `'Histogram mode'` or `'Histogram mean'`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**HistogramOutputPort — Enable histogram output**

`false` (default) | `true`

Enable histogram output. Set this property to `true` to output the histogram used in the computation of the state levels.

**Dependencies**

This property applies when you set the `Method` property to `'Histogram mode'` or `'Histogram mean'`.

**Method — Algorithm used to compute state levels**

`'Histogram mode'` (default) | `'Histogram mean'` | `'Peak to peak'`

Algorithm used to compute state levels. Specify the method used to compute state levels as one of `'Histogram mean'`, `'Histogram mode'`, or `'Peak to peak'`.

**RunningStateLevels — Calculation over successive calls**

`false` (default) | `true`

Calculation over successive calls to the algorithm. Set this property to `true` to enable computation of the state levels over successive calls to the algorithm. Otherwise, the object computes the state levels

of only the current input. When you set the `RunningStateLevels` property to `false` and you are using a histogram to compute your state levels, you must set the `HistogramBoundsSource` property to `'Property'`.

## Usage

### Syntax

```
levels = sl(x)
[levels,histogram] = sl(x)
```

### Description

`levels = sl(x)` returns a two-element row vector, `levels`, containing the estimated state levels for the input, `x`.

`[levels,histogram] = sl(x)` returns a double-precision column vector, `histogram`, containing the histogram of the sample values in `x`. You can obtain this output only when you set the `Method` property to either `'Histogram mean'` or `'Histogram mode'`, and you set the `HistogramOutputPort` property to `true`.

### Input Arguments

#### **x** — Input

column vector

Input data, specified as a real-valued column vector.

Data Types: `double`

### Output Arguments

#### **levels** — State levels

two-element row vector

State levels, returned as a two-element row vector.

Data Types: `double`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Specific to `dsp.StateLevels`

`plot` (To be removed) Plot signal, state levels, and histogram

## Common to All System Objects

`step` Run System object algorithm

release Release resources and allow changes to System object property values and input characteristics  
reset Reset internal states of System object

## Examples

### State Levels of 2.3 V Underdamped Noisy Clock

Compute and plot the state levels of a 2.3 V underdamped noisy clock. Load the clock data in the variable, `x`, and the sampling instants in the variable `t`.

```
load('clockex.mat','x','t');
```

Estimate the state levels.

```
sl = dsp.StateLevels;  
levels = sl(x);
```

Plot the clock data along with the estimated state levels and histograms.

```
plot(sl)
```

## More About

### State

A state is a particular level, which can be associated with an upper and lower state boundary. States are ordered from the most negative to the most positive. In a bilevel waveform, the most negative state is the low state. The most positive state is the high state.

### State-Level Tolerances

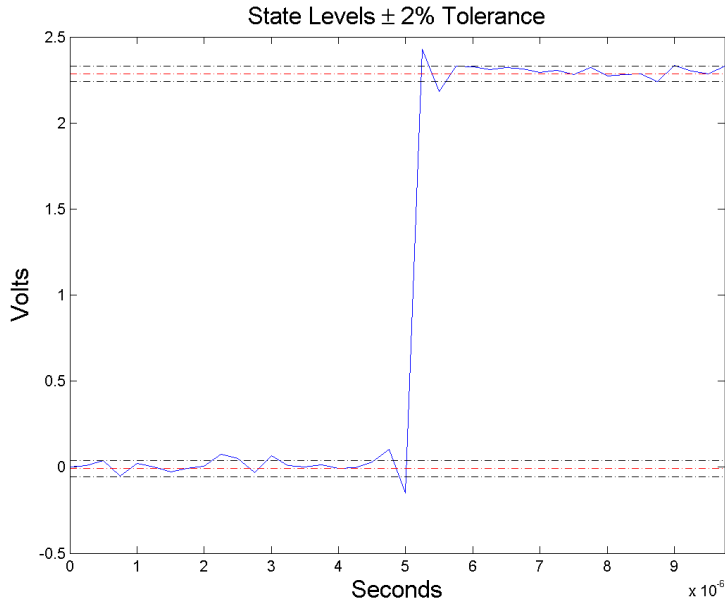
Each state level can have associated lower- and upper-state boundaries. These state boundaries are defined as the state level plus or minus a scalar multiple of the difference between the high state and low state. To provide a useful tolerance region, the scalar is typically a small number such as 2/100 or 3/100. In general, the  $\alpha\%$  tolerance region for the low state is defined as

$$S_1 \pm \frac{\alpha}{100}(S_2 - S_1)$$

where  $S_1$  is the low-state level and  $S_2$  is the high-state level. Replace the first term in the equation with  $S_2$  to obtain the  $\alpha\%$  tolerance region for the high state.

This figure shows the lower and upper 2% state boundaries (tolerance regions) for a positive-polarity bilevel waveform. The estimated state levels are indicated by a dashed red line.





## Algorithms

The `dsp.StateLevels` System object uses the histogram method to estimate the states of a bilevel waveform. The histogram method is described in [1]. To summarize the method:

- 1 Determine the maximum and minimum amplitudes and amplitude range of the data.
- 2 For the specified number of histogram bins, determine the bin width as the ratio of the amplitude range to the number of bins.
- 3 Sort the data values into the histogram bins.
- 4 Identify the lowest-indexed histogram bin,  $i_{low}$ , and highest-indexed histogram bin,  $i_{high}$ , with nonzero counts.
- 5 Divide the histogram into two subhistograms. The lower-histogram bins are  $i_{low} \leq i \leq 1/2(i_{high} - i_{low})$ .

The upper-histogram bins are  $i_{low} + 1/2(i_{high} - i_{low}) \leq i \leq i_{high}$ .

- 6 Compute the state levels by determining the mode or mean of the lower and upper histograms.

## Version History

**Introduced in R2012a**

**R2021b: `dsp.StateLevels` System object will be removed**

*Warns starting in R2021b*

`dsp.StateLevels` System object will be removed in a future release. Use the `statelevels` function instead.

**R2023a: `dsp.StateLevels` System object has been removed**

*Errors starting in R2023a*

The `dsp.StateLevels` System object has been removed. Use the `statelevels` function instead.

### Update Code

This table shows how to update existing code to use the `statelevels` function.

| Discouraged Usage  | Recommended Replacement  |
|--|--|
| <pre>load('clockex.mat', 'x', 't'); slevel = dsp.StateLevels; levelsObj = slevel(x)  levelsObj =  0.0027    2.3068</pre> <p><b>Plot state levels and the corresponding histogram</b></p> <pre>figure; plot(slevel)</pre> | <pre>levelsfn = statelevels(x)  levelsfn =  0.0027    2.3068</pre> <p><b>Plot state levels and the corresponding histogram</b></p> <pre>figure; statelevels(x)</pre> |

### References

- [1] *IEEE Standard on Transitions, Pulses, and Related Waveforms*, IEEE Standard 181, 2003, pp. 15-17.

### Extended Capabilities

#### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

### See Also

#### Functions

`statelevels`

# dsp.SubbandAnalysisFilter

**Package:** dsp

Decompose signal into high-frequency and low-frequency subbands

## Description

The `dsp.SubbandAnalysisFilter` object decomposes a signal into high-frequency and low-frequency subbands, each with half the bandwidth of the input.

To decompose a signal into high-frequency and low-frequency subbands:

- 1 Create the `dsp.SubbandAnalysisFilter` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
subAna = dsp.SubbandAnalysisFilter
subAna = dsp.SubbandAnalysisFilter(lpc,hpc)
subAna = dsp.SubbandAnalysisFilter(Name,Value)
```

### Description

`subAna = dsp.SubbandAnalysisFilter` returns a two-channel subband analysis filter, `subAna`, that decomposes the input signal into a high-frequency subband and a low-frequency subband, each with half the bandwidth of the input.

`subAna = dsp.SubbandAnalysisFilter(lpc,hpc)` returns a two-channel subband analysis filter, `subAna`, with the “LowpassCoefficients” on page 4-0 property set to `lpc` and the `HighpassCoefficients` property set to `lpc`.

`subAna = dsp.SubbandAnalysisFilter(Name,Value)` returns a two-channel subband analysis filter, `subAna`, with each specified property set to the specified value.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

**LowpassCoefficients — Lowpass FIR filter coefficients**`[0.0352 -0.0854 -0.1350 0.4599 0.8069 0.3327]` (default) | row vector

Specify a vector of lowpass FIR filter coefficients, in descending powers of  $z$ . For the lowpass filter, use a half-band filter that passes the frequency band stopped by the filter specified in the `HighpassCoefficients` property. The default values of this property specify a filter based on a third-order Daubechies wavelet.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**HighpassCoefficients — Highpass FIR filter coefficient**`[-0.3327 0.8069 -0.4599 -0.1350 0.0854 0.0352]` (default) | row vector

Specify a vector of highpass FIR filter coefficients, in descending powers of  $z$ . For the highpass filter, use a half-band filter that passes the frequency band stopped by the filter specified in the “LowpassCoefficients” on page 4-0 property. The default values of this property specify a filter based on a third-order Daubechies wavelet.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**Fixed-Point Properties****FullPrecisionOverride — Full precision override for fixed-point arithmetic**`true` (default) | `false`

Specify whether to use full precision rules. If you set `FullPrecisionOverride` to `true`, which is the default, the object computes all internal arithmetic and output data types using full precision rules. These rules provide the most accurate fixed-point numerics. It also turns off the display of other fixed-point properties because they do not apply individually. These rules guarantee that no quantization occurs within the object. Bits are added, as needed, to ensure that no roundoff or overflow occurs. If you set `FullPrecisionOverride` to `false`, fixed-point data types are controlled through individual fixed-point property settings. For more information, see “Full Precision for Fixed-Point System Objects”.

Data Types: `logical`

**RoundingMethod — Rounding method for fixed-point operations**`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

Specify the rounding method.

**Dependencies**

This property applies only if the object is not in full precision mode.

**OverflowAction — Action to take when integer input is out of range**`Wrap` (default) | `Saturate`

Specify the overflow action as `Wrap` or `Saturate`.

**Dependencies**

This property applies only if the object is not in full precision mode.

**CoefficientsDataType — Data type of the coefficients**`Same word length as input` (default) | `Custom`

Specify the FIR filter coefficients fixed-point data type as Same word length as input or Custom.

#### **CustomCoefficientsDataType — Coefficients word and fraction lengths**

`numericType([],16,15)` (default) | `numericType`

Specify the FIR filter coefficients fixed-point type as a `numericType` object with a Signedness of Auto.

#### **Dependencies**

This property applies only when you set the `CoefficientsDataType` property to Custom.

#### **ProductDataType — Data type of product**

`Full precision` (default) | `Same as input` | `Custom`

Specify the product data type as one of `Full precision` | `Same as input` | `Custom`.

#### **CustomProductDataType — Product word and fraction lengths**

`numericType([],32,30)` (default) | `numericType`

Specify the product fixed-point type as a scaled `numericType` object with a Signedness of Auto.

#### **Dependencies**

This property applies only when you set the `ProductDataType` property to Custom.

#### **AccumulatorDataType — Data type of accumulator**

`Full precision` (default) | `Same as input` | `Same as product` | `Custom`

Specify the accumulator data type as `Full precision`, `Same as input`, `Same as product`, or `Custom`.

#### **CustomAccumulatorDataType — Accumulator word and fraction lengths**

`numericType([],32,30)` (default) | `numericType`

Specify the accumulator fixed-point type as a scaled `numericType` object with a Signedness of Auto.

#### **Dependencies**

This property applies only when you set the `AccumulatorDataType` property to Custom.

#### **OutputDataType — Data type of output**

`Same as accumulator` (default) | `Same as product` | `Same as input` | `Custom`

Specify the output data type as `Same as accumulator`, `Same as product`, `Same as input`, or `Custom`.

#### **CustomOutputDataType — Output word and fraction lengths**

`numericType([],16,14)` (default) | `numericType`

Specify the output fixed-point type as a scaled `numericType` object with a Signedness of Auto.

#### **Dependencies**

This property applies only when you set the “`OutputDataType`” on page 4-0 property to Custom.

## Usage

### Syntax

```
[hi,lo] = subAna(x)
```

### Description

`[hi,lo] = subAna(x)` decomposes the input signal, `x`, into a high-frequency subband, `hi`, and a low-frequency subband, `lo`.

### Input Arguments

#### **x** — Data input

column vector | matrix

Data input, specified as a column vector or a matrix size  $P$ -by- $Q$ . The number of input rows  $P$  can be arbitrary and does not have to be even.

This object supports variable-size input signals, that is, the frame length (number of rows) of the signal can change even when the object is locked. However, the number of channels (columns) must remain constant.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fi`

### Output Arguments

#### **hi** — High-frequency subband

column vector | matrix

High-frequency subband, returned as a column vector or a matrix.

When the input is of size  $P$ -by- $Q$ , and  $P$  is not even, the high-frequency subband output has an upper bound size of  $\text{ceil}(P/2)$ -by- $Q$ . If  $P$  is even, then the output is of size  $(P/2)$ -by- $Q$ . The number of channels (columns) does not change.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fi`

#### **lo** — Low-frequency subband

column vector | matrix

Low-frequency subband, returned as a column vector or a matrix.

When the input is of size  $P$ -by- $Q$ , and  $P$  is not even, the low-frequency subband output has an upper bound size of  $\text{ceil}(P/2)$ -by- $Q$ . If  $P$  is even, then the output is of size  $(P/2)$ -by- $Q$ . The number of channels (columns) does not change.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fi`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Common to All System Objects

step     Run System object algorithm  
 release   Release resources and allow changes to System object property values and input characteristics  
 reset     Reset internal states of System object

## Examples

### Decompose and Reconstruct a Signal

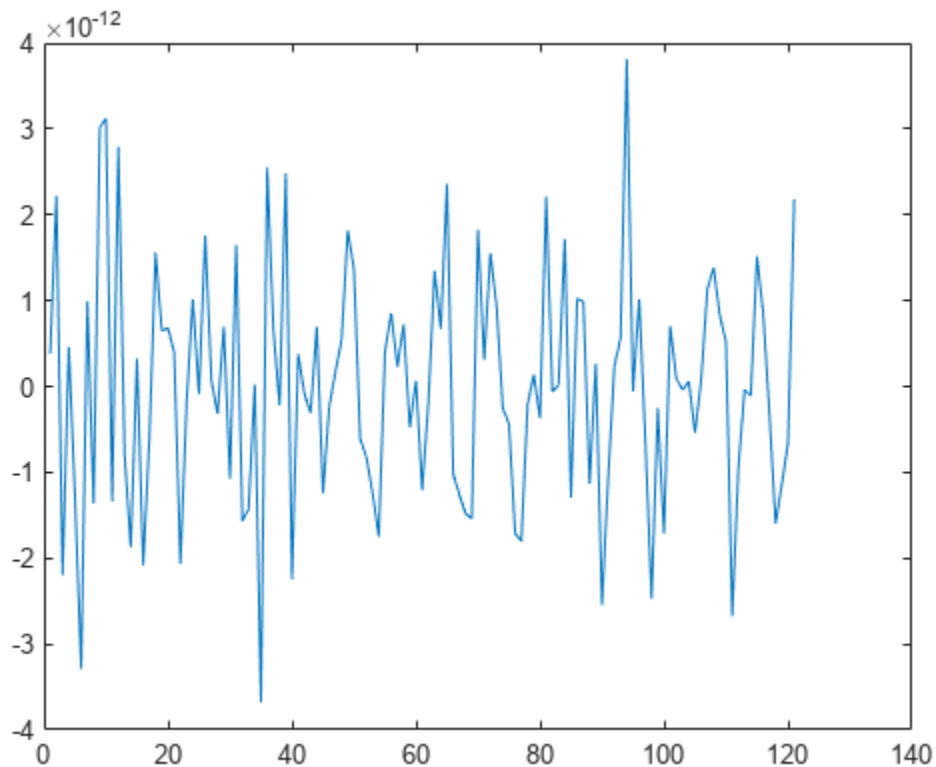
Decompose a signal into low frequency and high frequency subbands using the subband analysis filter. Reconstruct the signal using the subband synthesis filter.

```
load dspwlets; % load the filter coefficients lod, hid, lor and hir
subAna = dsp.SubbandAnalysisFilter(lod, hid);
subSynth = dsp.SubbandSynthesisFilter(lor, hir);
u = randn(128,1);
```

```
[hi, lo] = subAna(u); % Two channel analysis
y = subSynth(hi, lo); % Two channel synthesis
```

Plot difference between original and reconstructed signals with filter latency compensated.

```
plot(u(1:end-7)-y(8:end));
```



## Algorithms

This object implements the algorithm, inputs, and outputs described on the Two-Channel Analysis Subband Filter block reference page. The object properties correspond to the block parameters, except:

- The `SubbandAnalysisFilter` object does not have a property that corresponds to the **Input processing** parameter of the Two-Channel Analysis Subband Filter block. The object assumes the input is frame based and always maintains the input frame rate.
- The **Rate options** block parameter is not supported by the `dsp.SubbandAnalysisFilter` object.

## Version History

**Introduced in R2012a**

### **R2023a: Support for arbitrary input frame length**

This object supports an input signal with an arbitrary frame length, so the input frame length does not have to be even.

### **R2023a: Support for variable-size input signal**

The input signal to this object can be a variable-size signal, that is, the frame length (number of rows) of the signal can change even when the object is locked. However, the number of channels (columns) must remain constant.

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

In the code you generate from this object, the low-frequency and high-frequency subband outputs are fixed-size signals if the input signal is even and is of fixed size. If the input is not even or if the input is a variable-size signal, then the frequency subband outputs are of variable-size.

## See Also

### **Objects**

`dsp.SubbandSynthesisFilter` | `dsp.DyadicAnalysisFilterBank`



# dsp.SubbandSynthesisFilter

**Package:** dsp

Reconstruct signal from high-frequency and low-frequency subbands

## Description

The `dsp.SubbandSynthesisFilter` System object reconstructs a signal from high-frequency and low-frequency subbands.

To reconstruct a signal from high-frequency and low-frequency subbands:

- 1 Create the `dsp.SubbandSynthesisFilter` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
subSyn = dsp.SubbandSynthesisFilter
subSyn = dsp.SubbandSynthesisFilter(lpc,hpc)
subSyn = dsp.SubbandSynthesisFilter(Name,Value)
```

### Description

`subSyn = dsp.SubbandSynthesisFilter` returns a two-channel subband synthesis filter, `subSyn`, that reconstructs a signal from its high-frequency subband and low-frequency subband. Each subband contains half the bandwidth of the original signal.

`subSyn = dsp.SubbandSynthesisFilter(lpc,hpc)` returns a two-channel subband synthesis filter, `subSyn`. The object has the “LowpassCoefficients” on page 4-0 property set to `lpc` and the `HighpassCoefficients` property set to `hpc`.

`subSyn = dsp.SubbandSynthesisFilter(Name,Value)` returns a two-channel subband synthesis filter, `subSyn`, with each specified property set to the specified value.

### Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

**LowpassCoefficients — Lowpass FIR filter coefficients**`[0.3327 0.8069 0.4599 -0.1350 -0.0854 0.0352]` (default) | row vector

Specify a vector of lowpass FIR filter coefficients, in descending powers of  $z$ . For the lowpass filter, use a half-band filter that passes the frequency band stopped by the filter specified in the `HighpassCoefficients` property. The default values of this property specify a filter based on a third-order Daubechies wavelet.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**HighpassCoefficients — Highpass FIR filter coefficients**`[0.0352 0.0854 -0.1350 -0.4599 0.8069 -0.3327]` (default) | row vector

Specify a vector of highpass FIR filter coefficients, in descending powers of  $z$ . For the highpass filter, use a half-band filter that passes the frequency band stopped by the filter specified in the `LowpassCoefficients` property. The default values of this property specify a filter based on a third-order Daubechies wavelet.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**Fixed-Point Properties****FullPrecisionOverride — Full precision override for fixed-point arithmetic**`true` (default) | `false`

Specify whether to use full precision rules. If you set `FullPrecisionOverride` to `true`, which is the default, the object computes all internal arithmetic and output data types using full precision rules. These rules provide the most accurate fixed-point numerics. It also turns off the display of other fixed-point properties because they do not apply individually. These rules guarantee that no quantization occurs within the object. Bits are added, as needed, to ensure that no roundoff or overflow occurs. If you set `FullPrecisionOverride` to `false`, fixed-point data types are controlled through individual fixed-point property settings. For more information, see “Full Precision for Fixed-Point System Objects”.

**RoundingMethod — Rounding method for fixed-point operations**`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

Specify the rounding method.

**Dependencies**

This property applies only if the object is not in full precision mode.

**OverflowAction — Action to take when integer input is out of range**`Wrap` (default) | `Saturate`

Specify the overflow action as `Wrap` or `Saturate`.

**Dependencies**

This property applies only if the object is not in full precision mode.

**CoefficientsDataType — Data type of the coefficients**`Same word length as input` (default) | `Custom`

Specify the FIR filter coefficients fixed-point data type as `Same word length as input` or `Custom`.

**CustomCoefficientsDataType — Coefficient word and fraction lengths**

numericType([],16,15) (default) | numericType

Specify the FIR filter coefficients fixed-point type as a numericType object with a Signedness of Auto.

**Dependencies**

This property applies only when you set the CoefficientsDataType property to Custom.

**ProductDataType — Data type of product**

Full precision (default) | Same as input | Custom

Specify the product data type as Full precision, Same as input, or Custom.

**CustomProductDataType — Product word and fraction lengths**

numericType([],32,30) (default) | numericType

Specify the product fixed-point type as a scaled numericType object with a Signedness of Auto.

**Dependencies**

This property applies only when you set the ProductDataType property to Custom.

**AccumulatorDataType — Data type of accumulator**

Full precision (default) | Same as input | Same as product | Custom

Specify the accumulator data type as Full precision, Same as input, Same as product, or Custom.

**CustomAccumulatorDataType — Accumulator word and fraction lengths**

numericType([],32,30) (default) | numericType

Specify the accumulator fixed-point type as a scaled numericType object with a Signedness of Auto.

**Dependencies**

This property applies only when you set the AccumulatorDataType property to Custom.

**OutputDataType — Data type of output**

Same as accumulator (default) | Same as product | Same as input | Custom

Specify the output data type as Same as accumulator, Same as product, Same as input, or Custom.

**CustomOutputDataType — Output word and fraction lengths**

numericType([],16,14) (default) | numericType

Specify the output fixed-point type as a scaled numericType object with a Signedness of Auto.

**Dependencies**

This property applies only when you set the OutputDataType property to Custom.

## Usage

### Syntax

```
y = subSyn(hi,lo)
```

### Description

`y = subSyn(hi, lo)` reconstructs a signal from a high-frequency subband, `hi`, and a low-frequency subband, `lo`.

### Input Arguments

#### **hi** — High-frequency subband

vector | matrix

High-frequency subband, specified as a column vector or a matrix. Both inputs must have the same size and data type.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fi`

#### **lo** — Low-frequency subband

vector | matrix

Low-frequency subband, specified as a column vector or a matrix. Both inputs must have the same size and data type.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fi`

### Output Arguments

#### **y** — Synthesized output

vector | matrix

Synthesized output, reconstructed as a vector or a matrix. The number of rows in the output is the sum of the number of rows of the input signals.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fi`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Common to All System Objects

|                      |  |
|----------------------|--|
| <code>step</code>    | Run System object algorithm  |
| <code>release</code> | Release resources and allow changes to System object property values and input characteristics |
| <code>reset</code>   | Reset internal states of System object   |

## Examples

## Decompose and Reconstruct a Signal

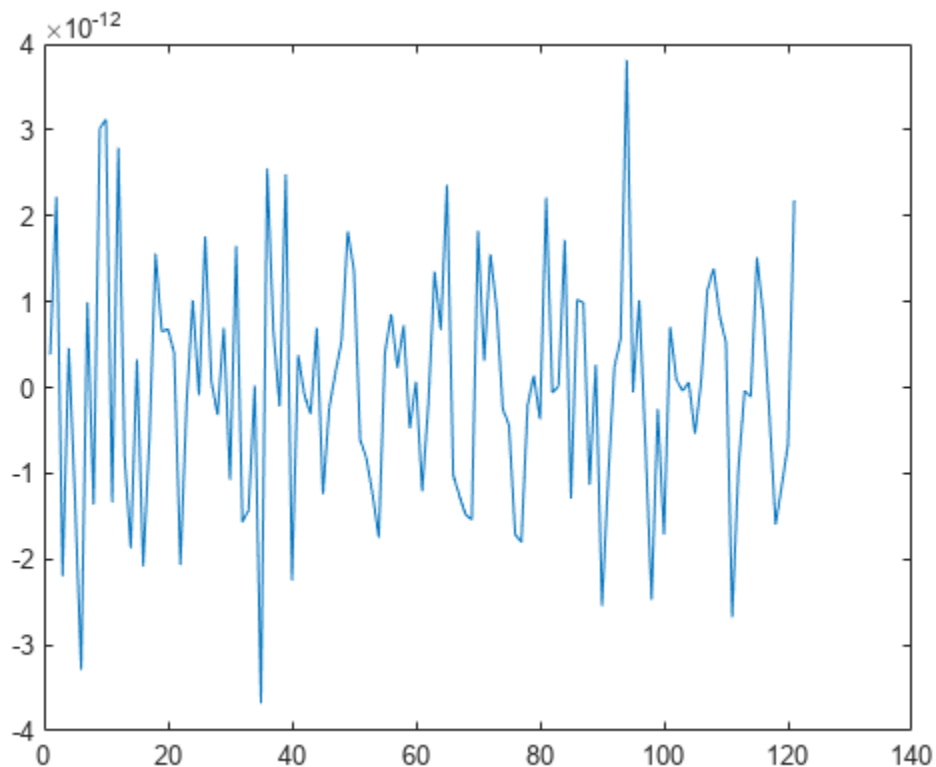
Decompose a signal into low frequency and high frequency subbands using the subband analysis filter. Reconstruct the signal using the subband synthesis filter.

```
load dspwlets; % load the filter coefficients lod, hid, lor and hir
subAna = dsp.SubbandAnalysisFilter(lod, hid);
subSynth = dsp.SubbandSynthesisFilter(lor, hir);
u = randn(128,1);
```

```
[hi, lo] = subAna(u); % Two channel analysis
y = subSynth(hi, lo); % Two channel synthesis
```

Plot difference between original and reconstructed signals with filter latency compensated.

```
plot(u(1:end-7)-y(8:end));
```



## Algorithms

This object implements the algorithm, inputs, and outputs described on the Two-Channel Synthesis Subband Filter block reference page. The object properties correspond to the block parameters, except:

- The `SubbandSynthesisFilter` object does not have a property that corresponds to the **Input processing** parameter of the Two-Channel Synthesis Subband Filter block. The object only performs sample-based processing and always maintains the input frame rate.

- The **Rate options** block parameter is not supported by the `SubbandSynthesisFilter` object.

## Version History

Introduced in R2012a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

## See Also

### Objects

`dsp.SubbandAnalysisFilter` | `dsp.DyadicSynthesisFilterBank`

# dsp.TimeScope

**Package:** dsp

(To be removed) Time domain signal display and measurement

## Description

---

**Note** `dsp.TimeScope` is not recommended. Use `timescope` instead. For more information, see “Compatibility Considerations” on page 4-1171

---

The `dsp.TimeScope` System object displays time-domain signals. You can use the scope to measure signal values, find peaks, display bilevel measurements and statistics.

To see time-domain signals in the scope:

- 1 Create the `dsp.TimeScope` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

Oscilloscope features:

- Triggers — Set triggers to sync repeating signals and pause the display when events occur.
- Cursor Measurements — Measure signal values using vertical and horizontal cursors.
- Signal Statistics — Display the maximum, minimum, peak-to-peak difference, mean, median, and RMS values of a selected signal.
- Peak Finder — Find maxima, showing the x-axis values at which they occur.
- Bilevel Measurements — Measure transitions, overshoots, undershoots, and cycles.

For information on measurements and triggers, see “Configure Time Scope Block”.

Scope display features:

- Multiple signals — Plot multiple signals on the same y-axis (display) using multiple input ports.
- Multiple y-axes (displays) — Display multiple y-axes. All the y-axes have a common time range on the x-axis.
- Modify parameters — Modify scope parameter values before and during a simulation.
- Axis autoscaling — Autoscaling during or at the end of a simulation. Margins are drawn at the top and bottom of the axes.

## Creation

### Syntax

```
scope = dsp.TimeScope
```

```
scope = dsp.TimeScope(numInputs, sampleRate)
scope = dsp.TimeScope( ___, Name, Value)
```

### Description

`scope = dsp.TimeScope` returns a Time Scope System object, `scope`. This object displays real- and complex-valued floating and fixed-point signals in the time domain.

`scope = dsp.TimeScope(numInputs, sampleRate)` creates a Time Scope and sets the `NumInputPorts` property to `numInputs` and the `SampleRate` property to `sampleRate`.

`scope = dsp.TimeScope( ___, Name, Value)` sets properties specified as `Name, Value` pairs.

### Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

### Frequently Used

#### **NumInputPorts** — Number of input ports

1 (default) | integer between [1, 96]

Number of input ports, specified as a positive integer. Each signal coming through a separate input becomes a separate channel in the scope. You must invoke the scope with the same number of inputs as the value of this property.

#### **SampleRate** — Sample rate of inputs

1 (default) | scalar | vector

Specify the sample rate, in hertz, of the input signals.

You can specify a scalar or a numeric vector with length equal to the value of `NumInputPorts`. The inverse of the sample rate determines the spacing between points on the time axis in the displayed signal. When you set `SampleRate` to a scalar value and `NumInputPorts` is greater than 1, the object uses the same sample rate for all inputs.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

#### **TimeSpan** — Time span

10 (default) | positive scalar

Specify the time span, in seconds, as a positive, numeric scalar value. The time-axis limits are calculated as follows.

- Minimum time-axis limit = `min(TimeDisplayOffset)`
- Maximum time-axis limit = `max(TimeDisplayOffset) + TimeSpan`

**Tunable:** Yes



**Dependencies**

To use this property, set `FrameBasedProcessing` to `false`, or set `FrameBasedProcessing` to `true` and `TimeSpanSource` to `'Property'`.

**Scope Window Use**

Open the **Configuration Properties**. On the **Time** tab, set **Time Span**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**TimeSpanOverrunAction — Wrap or scroll when TimeSpan is overrun**

`'Wrap'` (default) | `'Scroll'`

Specify how the scope displays new data beyond the visible time span.

- **Wrap** — In this mode, the scope displays new data until the data reaches the maximum time-axis limit. When the data reaches the maximum time-axis limit of the scope window, the scope clears the display. The scope then updates the time offset value and begins displaying subsequent data points starting from the minimum time-axis limit.
- **Scroll** — In this mode, the scope scrolls old data to the left to make room for new data on the right side of the scope display. This mode is graphically intensive and can affect run-time performance. However, it is beneficial for debugging and monitoring time-varying signals.

**Tunable:** Yes

**Scope Window Use**

Open the **Configuration Properties**. On the **Time** tab, set **Time Span overrun action**.

Data Types: `char` | `string`

**TimeSpanSource — Source of time span**

`'Property'` (default) | `'Auto'`

Specify the source of the time span for frame-based input signals as:

- `'Property'` - The object derives the x-axis limits from the `TimeDisplayOffset` and `TimeSpan` properties.
- `'Auto'` - The time-axis limits are derived from the `TimeDisplayOffset` and `SampleRate` properties and the `FrameSize` (number of rows in each input signal). The limits are calculated as:
  - Minimum time-axis limit =  $\min(\text{TimeDisplayOffset})$
  - Maximum time-axis limit =  $\max(\text{TimeDisplayOffset}) + \max(1/\text{SampleRate} * \text{FrameSize})$

**Tunable:** Yes

**Dependencies**

To use this property, set `FrameBasedProcessing` to `true`.

**Scope Window Use**

Open the **Configuration Properties**. On the **Time** tab, set **Time Span**.

Data Types: `char` | `string`

**AxesScaling — Axes scaling mode**`"OnceAtStop" (default) | "Auto" | "Manual" | "Updates"`

Specify when the scope scales the axes. Valid values are:

- `"Auto"` — The scope scales the axes as needed to fit the data, both during and after simulation.
- `"Manual"` — The scope does not scale the axes automatically.
- `"OnceAtStop"` — The scope scales the axes when the simulation stops.
- `"Updates"` — The scope scales the axes once and only once after 10 updates.

**Scope Window Use**

Select **Tools > Axes Scaling**.

Data Types: `char` | `string`

**Advanced****Name — Window name**`'Time Scope' (default) | character vector | string scalar`

Specify the name of the scope as a character vector or string scalar. This name appears as the title of the scope's figure window. To specify a title of a scope plot, use the `Title` property.

**Tunable:** Yes

Data Types: `char` | `string`

**Position — Window position**`screen center (default) | [left bottom width height]`

Scope window position in pixels, specified by the size and location of the scope window as a 4-element vector of the form `[left bottom width height]`. You can place the scope window in a specific position on your screen by modifying the values to this property.

By default, the window appears in the center of your screen with a width of 410 pixels and height of 300 pixels. The exact position value depends on your screen resolution.

**Tunable:** Yes

**ReduceUpdates — Reduce updates to improve performance**`true (default) | false`

- `true` — The scope logs data for later use and updates the window periodically.
- `false` — The scope updates every time the scope is called.

The simulation speed is faster when this property is set to `true`.

**Tunable:** Yes

**Scope Window Use**

Select **Simulation > Reduce Updates to Improve Performance**.

**LayoutDimensions — Display layout grid dimensions**`[1,1] (default) | [numberOfRows, numberOfColumns]`

Specify the layout grid dimensions as a 2-element vector: `[numberOfRows, numberOfColumns]`. You can use up to 16 rows and 16 columns.

Example: `scope.LayoutDimensions = [2,4]`

**Tunable:** Yes

#### Scope Window Use

Select **View > Layout**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

#### PlotType — Control the type of plot

`'Line'` (default) | `'Stairs'`

Specify the type of plot to use.

- `Line` — Line graph, similar to the `line` or `plot` function.
- `Stairs` — Stair-step graph, similar to the `stairs` function. Stair-step graphs are useful for drawing time history graphs of digitally sampled data.

**Tunable:** Yes

#### Scope Window Use

Open the **Style** properties. Set **Plot type**.

Data Types: `char` | `string`

#### BufferLength — Length of buffer used for each input signal

`5000` (default) | `scalar`

Specify the size of the buffer that the scope holds in its memory cache. Memory is limited by available memory on your system. If your signal has  $M$  rows of data and  $N$  data points in each row,  $M \times N$  is the number of data points per time step. Multiply this result by the number of time steps for your model to obtain the required buffer length. For example, if you have 10 rows of data with each row having 100 data points and your run will be 10 time steps, you should enter 10,000 ( $10 \times 100 \times 10$ ) as the buffer length.

#### Scope Window Use

Open the **Data History Properties**. Set **Buffer length**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

#### FrameBasedProcessing — Process input in frames

`true` (default) | `false`

- `true` — Enable frame-based processing.
- `false` — Enable sample-based processing.

#### Scope Window Use

Open the **Configuration Properties**. On the **Main** tab, set **Input processing**.

#### TimeUnits — Units of time axis

`'Metric'` (default) | `'Seconds'` | `'None'`

Specify the units used to describe the time axis. You can select one of the following options:

- **Metric** — In this mode, the scope converts the times on the time axis to the most appropriate measurement units. These units include milliseconds, microseconds, nanoseconds, minutes, days, etc. The scope chooses the appropriate measurement units based on the minimum time-axis limit and the maximum time-axis limit of the scope window.
- **Seconds** — In this mode, the scope always displays the units on the time axis as seconds.
- **None** — In this mode, the scope does not display any units on the time axis. The scope only shows the word **Time** on the time axis.

**Tunable:** Yes

#### **Scope Window Use**

Open the **Configuration Properties**. On the **Time** tab, set **Time units**.

Data Types: char | string

#### **TimeDisplayOffset — Offset x-axis limits**

0 (default) | scalar | vector

Specify, in seconds, how far to move the data on x-axis. The signal value does not change, only the displayed x-axis.

If you specify this property as a scalar, then that value is the time display offset for all channels.

If you specify a vector, each vector element is the time offset for the corresponding channel. For vectors with length less than the number of input channels, the time display offsets for the remaining channels are set to 0. If a vector has a length greater than the number of input channels, the extra vector elements are ignored.

**Tunable:** Yes

#### **Scope Window Use**

Open the **Configuration Properties**. On the **Time** tab, set **Time display offset**.

#### **TimeAxisLabels — Time-axis labels**

'All' (default) | 'Bottom' | 'None'

Specify how time-axis labels should appear in the scope displays as:

- **'All'** — Time-axis labels appear in all displays.
- **'Bottom'** — Time-axis labels appear in the bottom display of each column.
- **'None'** — No labels appear in any display.

**Tunable:** Yes

#### **Scope Window Use**

Open the **Configuration Properties**. On the **Time** tab, set **Time-axis labels**.

Data Types: char | string

#### **MaximizeAxes — Maximize axes control**

"Auto" (default) | "On" | "Off"

Specify whether to display the scope in maximized-axes mode. In this mode, the axes are expanded to fit into the entire display. To conserve space, labels do not appear in each display. Instead, tick-mark values appear on top of the plotted data. You can select one of the following options:

- "Auto" — The axes appear maximized in all displays only if the `Title` and `YLabel` properties are empty for every display. If you enter any value in any display for either of these properties, the axes are not maximized.
- "On" — The axes appear maximized in all displays. Any values entered into the `Title` and `YLabel` properties are hidden.
- "Off" — None of the axes appear maximized.

**Tunable:** Yes

#### Scope Window Use

Open the **Configuration Properties**. On the **Main** tab, set **Maximize axes**.

Data Types: `char` | `string`

#### Display

##### ActiveDisplay — Active display for setting properties

1 (default) | integer

Specify the active display, by integer display number, to get and set relevant properties. The number of a display corresponds to its column-wise placement index. Set this property to control which display has its axes colors, line properties, marker properties, and visibility changed.

**Tunable:** Yes

#### Scope Window Use

Open the **Configuration Properties**. On the **Display** tab, set **Active Display**.

##### Title — Display title

' ' (default) | character vector | string scalar

Display title, specified as a character vector or a string scalar.

**Tunable:** Yes

#### Scope Window Use

Click the **Analyzer** tab on the Spectrum Analyzer toolstrip. In the **Configuration** section, click **Settings**. In the Spectrum Analyzer Settings window that opens up, under **Display and labels**, enter **Title**.

Data Types: `char` | `string`

##### ShowLegend — Flag to show legend

false (default) | true

Flag to show the legend, specified as `true` or `false`. To show a legend with the input names, set this property to `true`.

Use the legend to control which signals are visible. In the scope legend, click a signal name to hide the signal in the scope. To show the signal, click the signal name again. To show only one signal, right-click the signal name. To show all signals, press **Esc**.

**Tunable:** Yes

### Dependencies

To enable this property, set the `ViewType` property to "Spectrum" or "Spectrum and spectrogram".

### Scope Window Use

Click the **Analyzer** tab on the Spectrum Analyzer toolstrip. To see the legend, click **Legend** in the **Configuration** section.

To enable the **Legend**, select **Spectrum** in the **Analyzer** tab.

Data Types: `logical`

### ShowTimeAxisLabel — Show legend on display

`true` (default) | `false`

When you set this property to `true`, the scope displays the time-axis label. When you set this property to `false`, the scope does not display the time-axis label, but still displays tick marks and other time-axis items. This property applies only if the `TimeAxisLabels` property is `All` or `Bottom`.

**Tunable:** Yes

### Dependency

To control which display's axis is labeled, use the `ActiveDisplay` property.

### Scope Window Use

Open the **Configuration Properties**. On the **Time** tab, set **Show time-axis label**.

### ChannelNames — Channel names

empty cell (default) | cell array of character vectors | array of strings

Channel names in the input data, specified as a cell array of character vectors or an array of strings. The names you specify in this property appear in the following locations:

- Legend
- **Spectrum Analyzer Settings > Color and styling** section
- **Measurements** and **Channel Measurements** tabs

If you do not specify channel names, the Spectrum Analyzer names the channels as `Channel 1`, `Channel 2`, and so on.

**Tunable:** Yes

### Dependency

To see the channel names, set `ShowLegend` to `true`.

**Scope Window Use**

Click the **Analyzer** tab on the Spectrum Analyzer toolstrip. To see the legend, click **Legend** in the **Configuration** section.

Data Types: char

**ShowGrid — Grid visibility**

false (default) | true

Set this property to `true` to show gridlines on the plot.

**Tunable:** Yes

**Scope Window Use**

Open the **Configuration Properties**. On the **Display** tab, set **Show grid**.

**PlotAsMagnitudePhase — Plot signal as magnitude and phase**

false (default) | true

When you set this property to `true`, the scope plots the magnitude and phase of the input signal on two separate axes within the same active display. When you set this property to `false`, the scope plots the real and imaginary parts of the input signal on two separate axes within the same active display.

This property is useful for complex-valued input signals. Turning on this property affects the phase for real-valued input signals. When the amplitude of the input signal is nonnegative, the phase is 0 degrees. When the amplitude of the input signal is negative, the phase is 180 degrees.

**Tunable:** Yes

**Scope Window Use**

Open the **Configuration Properties**. On the **Display** tab, select **Plot signals as magnitude and phase**.

**YLimits — y-axis limits**

[-10,10] (default) | [ymin, ymax]

Specify the y-axis limits as a two-element numeric vector, [ymin, ymax].

If `PlotAsMagnitudePhase` is `false`, the default is [-10, 10]. If `PlotAsMagnitudePhase` is `true`, the default is [0, 10].

**Tunable:** Yes

**Dependencies**

When `PlotAsMagnitudePhase` is `true`, this property specifies the y-axis limits of only the magnitude plot. The y-axis limits of the phase plot are always [-180, 180].

**Scope Window Use**

Open the **Configuration Properties**. On the **Display** tab, set **Y-limits (Minimum)** and **Y-limits (Maximum)**.

**YLabel — y-axis label**

"Amplitude" (default) | character vector | string scalar

Specify the text for the scope to display to the left of the y-axis.

**Tunable:** Yes

**Dependencies**

This property applies only when `PlotAsMagnitudePhase` is `false`. When `PlotAsMagnitudePhase` is `true`, the two y-axis labels are read-only values. The y-axis labels are set to "Magnitude" and "Phase" for the magnitude plot and the phase plot, respectively.

**Scope Window Use**

Open the **Configuration Properties**. On the **Display** tab, set **Y-Label**.

Data Types: char | string

**Usage****Syntax**

```
scope(signal)
scope(signal, signal2, ..., signalN)
```

**Description**

`scope(signal)` displays the signal, `signal`, in the time scope display.

`scope(signal, signal2, ..., signalN)` displays the signals `signal`, `signal2`, ..., `signalN` in the time scope display when you set the `NumInputPorts` property to `N`. In this case, `signal1`, `signal2`, ..., `signalN` can have different data types and dimensions.

**Input Arguments****signal — Input signal or signals to visualize**

scalar | vector | matrix

Specify one or more input signals to visualize in the `dsp.TimeScope`. Signals can have different data types and dimensions.

Example: `scope(signal1, signal2)`

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fi

**Object Functions**

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```



## Specific to Scopes

show      Display scope window  
hide      Hide scope window  
isVisible Determine visibility of scope

## Common to All System Objects

step      Run System object algorithm  
release   Release resources and allow changes to System object property values and input characteristics  
reset     Reset internal states of System object

If you want to restart the simulation from the beginning, call `reset` to clear the scope window displays. Do not call `reset` after calling `release`.

## Examples

### Display Simple Sine Wave Input Signal

Create `dsp.SineWave` and `dsp.TimeScope` objects. Run the scope to display the signal

```
sine = dsp.SineWave('Frequency',100,'SampleRate',1000);
sine.SamplesPerFrame = 10;
scope = dsp.TimeScope('SampleRate',sine.SampleRate,'TimeSpan',0.1);
for ii = 1:10
    x = sine();
    scope(x);
end
```

Run the `release` method to allow changes to property values and input characteristics. The scope automatically scales the axes.

```
release(scope)
```

## Tips

- To close the scope window and clear its associated data, use the MATLAB `clear` function.
- To hide or show the scope window, use the `hide` and `show` functions.
- Use the MATLAB `mcc` function to compile code containing a scope.

You cannot open scope configuration dialogs if you have more than one compiled component in your application.

## Version History

### Introduced in R2011a

### R2021a: `dsp.TimeScope` will be removed

*Warns starting in R2021a*

`dsp.TimeScope` will be removed in a future release. Use `timescope` instead.

### Extended Capabilities

#### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Supports MEX code generation by treating the calls to the object as extrinsic. Does not support code generation for standalone applications.
- See “System Objects in MATLAB Code Generation” (MATLAB Coder).

### See Also

#### Objects

`dsp.DynamicFilterVisualizer`

#### Objects

`timescope` | `dsp.ArrayPlot` | `spectrumAnalyzer`

#### Blocks

Time Scope

#### Topics

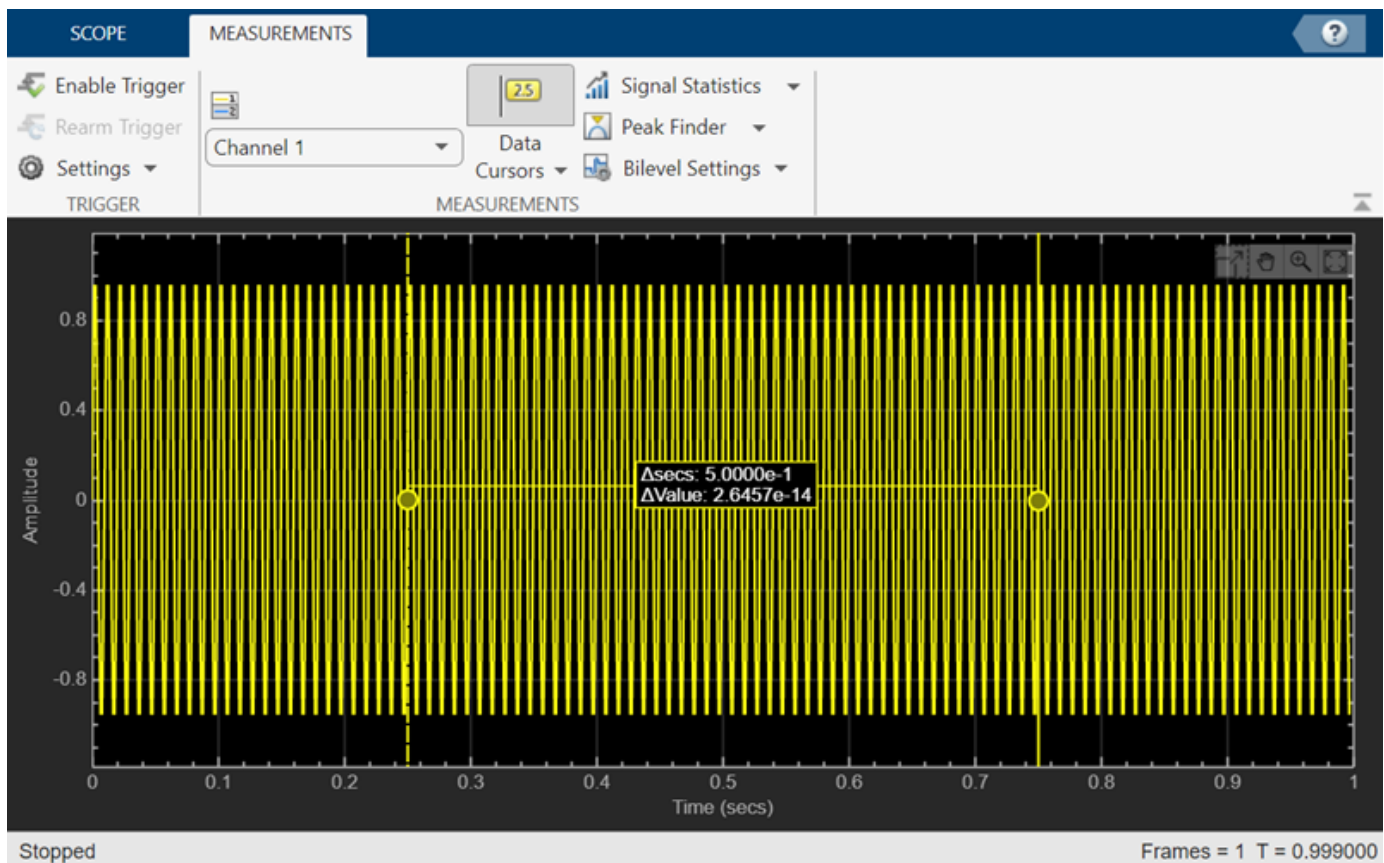
“Display Time-Domain Data”

# timescope

Display time-domain signals

## Description

The `timescope` object displays signals in the time domain.



Scope features:

- “Data Cursors” — Measure signal values using vertical and horizontal cursors.
- “Signal Statistics” — Display the maximum, minimum, peak-to-peak difference, mean, median, and RMS values of a selected signal.
- “Peak Finder” — Find maxima, showing the x-axis values at which they occur.
- “Bilevel Measurements” — Measure transitions, overshoots, undershoots, and cycles.
- “Triggers” — Set triggers to sync repeating signals and pause the display when events occur.

Use “Object Functions” on page 4-1182 to show, hide, and determine visibility of the scope window.

You can enable these measurements either programmatically or on the scope UI. For more details, see “Measurements” on page 4-1178.

## Creation

### Syntax

```
scope = timescope  
scope = timescope(Name=Value)
```

### Description

`scope = timescope` returns a `timescope` object, `scope`. This object displays real- and complex-valued floating and fixed-point signals in the time domain.

`scope = timescope(Name=Value)` returns a `timescope` sets properties using `Name=Value` arguments. You can specify the name-value arguments in any order.

### Properties

Most properties can be changed from the `timescope` UI.

#### Frequently Used

##### SampleRate — Sample rate of inputs

1 (default) | finite numeric scalar | vector

Sampling rate of the input signal, in hertz, specified as a finite numeric scalar or vector of scalars.

The inverse of the sample rate determines the x-axis (time axis) spacing between points in the displayed signal. When the value of `NumInputPorts` is greater than 1 and the sample rate is scalar, the object uses the same sample rate for all inputs. To specify different sample rates for each input, use a vector.

You can only set this property when creating the object or after calling `release`.

##### Scope Window Use

On the **Scope** tab, click **Settings**. In the Time Scope Settings dialog box, under **Data and Axes**, specify the sample rate in the **Sample Rate** box.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

##### TimeSpanSource — Source of time span

"auto" (default) | "property"

Source of the time span for frame-based input signals, specified as one of the following:

- "property" - The object derives the x-axis limits from the `TimeDisplayOffset` and `TimeSpan` properties.
- "auto" - The x-axis limits are derived from the `TimeDisplayOffset` property, `SampleRate` property, and the number of rows in each input signal (`FrameSize` in the equations below). The limits are calculated as:
  - Minimum time-axis limit = `TimeDisplayOffset`
  - Maximum time-axis limit = `TimeDisplayOffset + max(1/SampleRate.*FrameSize)`

**Dependency**

When you set the `TimeSpan` property, `TimeSpanSource` is automatically set to "property".

**Scope Window Use**

On the **Scope** tab, click **Settings**. In the Time Scope Settings dialog box, under **Data and Axes**, select Auto or specify the time span in the **Time Span** box.

Data Types: `char` | `string`

**TimeSpan — Time span**

10 (default) | positive scalar

Time span, in seconds, specified as a positive, numeric scalar value. The time-axis limits are calculated as:

- Minimum time-axis limit = `TimeDisplayOffset`
- Maximum time-axis limit = `TimeDisplayOffset` + `TimeSpan`

**Dependencies**

To enable this property, set `TimeSpanSource` to "property".

**Scope Window Use**

On the **Scope** tab, click **Settings**. In the Time Scope Settings dialog box, under **Data and Axes**, specify the time span in the **Time Span** box.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**TimeSpanOverrunAction — Data overrun behavior**

"scroll" (default) | "wrap"

Specify how the scope displays new data beyond the visible time span as either:

- "scroll" — In this mode, the scope scrolls old data to the left to make room for new data on the right of the scope display. This mode is beneficial for debugging and monitoring time-varying signals.
- "wrap" — In this mode, the scope adds data to the left of the plot after overrunning the right of the plot.

**Scope Window Use**

On the **Scope** tab, click **Settings**. In the Time Scope Settings dialog box, under **Data and Axes**, select the overrun behavior in **Overrun Action**.

Data Types: `char` | `string`

**PlotType — Type of plot**

"line" (default) | "stairs"

Type of plot, specified as either:

- "line" — Line graph, similar to the `line` or `plot` function.
- "stairs" — Stair-step graph, similar to the `stairs` function. Stair-step graphs are useful for drawing time history graphs of digitally sampled data.

### Scope Window Use

On the **Scope** tab, click **Settings**. In the Time Scope Settings dialog box, under **Data and Axes**, select the plot type in **Plot Type**.

Data Types: char | string

### AxesScaling — Axes scaling mode

"onceatstop" (default) | "auto" | "manual" | "updates"

When this property is set to:

- "onceatstop" -- The limits are updated once at the end of the simulation (when release is called).
- "auto" -- The scope attempts to always keep the data in the display while minimizing the number of updates to the axes limits.
- "manual" -- The scope takes no action unless specified by the user.
- "updates" -- The scope scales the axes once after a set number of visual updates. The number of updates is determined by the value of the AxesScalingNumUpdates property.

You can set this property only when creating the object.

Data Types: char | string

### AxesScalingNumUpdates — Number of updates before scaling

100 (default) | real positive integer

Specify the number of updates before scaling as a real, positive scalar integer.

### Dependency

To enable this property, set AxesScaling to "updates".

Data Types: double

### Advanced

#### LayoutDimensions — Display layout grid dimensions


[1,1] (default) | [numberOfRows, numberOfColumns]

Specify the layout grid dimensions as a two-element vector: [numberOfRows, numberOfColumns]. The grid can have a maximum of 4 rows and 4 columns.

If you create a grid of multiple axes, to modify the settings of individual axes, use the ActiveDisplay.

Example: scope.LayoutDimensions = [2,4]

### Scope Window Use

On the **Scope** tab, click **Display Grid** () and select a specific number of rows and columns in the grid.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### TimeUnits — Units of x-axis

"seconds" (default) | "none" | "metric"

Specify the units used to describe the x-axis (time axis). You can select one of the following options:

- "seconds" —The scope always displays the units on the x-axis as seconds. The scope shows the word `Time(s)` on the x-axis.
- "none" — The scope does not display any units on the x-axis. The scope only shows the word `Time` on the x-axis.
- "metric" — The scope displays the units on the x-axis as `Time (s)` changing the units to day, weeks, months, or years as you plot more data points.

#### Scope Window Use

On the **Scope** tab, click **Settings**. In the Time Scope Settings dialog box, under **Data and Axes**, specify the x-axis units in **Time Units**.

Data Types: `char` | `string`

#### TimeDisplayOffset — Offset x-axis limits

0 (default) | `scalar` | `vector`

Specify, in seconds, how far to move the data on the x-axis. The signal value does not change, only the limits displayed on the x-axis change.

If you specify this property as a scalar, then that value is the time display offset for all channels. If you specify this property as a vector, each input channel can be a different time display offset

#### Scope Window Use

On the **Scope** tab, click **Settings**. In the Time Scope Settings dialog box, under **Data and Axes**, specify the x-axis offset in **Time Offset**.

#### TimeAxisLabels — Time-axis labels

"all" (default) | "bottom" | "none"

Time-axis labels, specified as:

- "all" — Time-axis labels appear in all displays.
- "bottom" — Time-axis labels appear in the bottom display of each column.
- "none" — No labels appear in any display.

#### Scope Window Use

On the **Scope** tab, click **Settings**. In the Time Scope Settings dialog box, under **Data and Axes**, select which x-axis label to display in **Time Labels**.

Data Types: `char` | `string`


#### MaximizeAxes — Maximize axes control

"auto" (default) | "on" | "off"

Specify whether to display the scope in the maximized-axes mode. In this mode, the axes are expanded to fit into the entire display. To conserve space, labels do not appear in each display. Instead, the tick-marks and their values appear on top of the plotted data. You can select one of the following options:

- "auto" — The axes appear maximized in all displays only if the `Title` and `YLabel` properties are empty for every display. If you enter any value in any display for either of these properties, the axes are not maximized.
- "on" — The axes appear maximized in all displays. Any values entered into the `Title` and `YLabel` properties are hidden.
- "off" — None of the axes appear maximized.

### Scope Window Use

On the scope window, click  to maximize axes, hide all labels and inset the axes values.

Data Types: `char` | `string`

### BufferLength — Buffer length

50000 (default) | positive integer

Specify the length of the buffer used for each input signal as a positive integer.

You can set this property only when creating the object.

### Scope Window Use

On the **Scope** tab, click **Settings**. In the Time Scope Settings dialog box, under **Data and Axes**, specify the buffer length in the **Buffer Length** box.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

## Measurements

### MeasurementChannel — Channel for which to obtain measurements

1 (default) | positive integer

Channel for which to obtain measurements, specified as a positive integer in the range [1 *N*], where *N* is the number of input channels.

### Scope Window Use

On the **Measurements** tab, select a **Channel**.

Data Types: `double`

### BilevelMeasurements — Bilevel measurements

`BilevelMeasurementsConfiguration` object

Bilevel measurements to measure transitions, aberrations, and cycles of bilevel signals, specified as a `BilevelMeasurementsConfiguration` object.

All `BilevelMeasurementsConfiguration` properties are tunable.

### Scope Window Use

On the **Measurements** tab, in the **Measurements** section, click **Bilevel Settings** to modify the bilevel measurements.

### CursorMeasurements — Cursor measurements

`CursorMeasurementsConfiguration` object



Cursor measurements to display screen or waveform cursors, specified as a `CursorMeasurementsConfiguration` object.

All `CursorMeasurementsConfiguration` properties are tunable.

#### Scope Window Use

On the **Measurements** tab, in the **Measurements** section, select **Data Cursors** to enable the cursors on the display. Click **Data Cursors** to modify the cursor settings.

#### PeakFinder — Peak finder measurements

`PeakFinderConfiguration` object

Peak finder measurements to compute and display the largest calculated peak values, specified as a `PeakFinderConfiguration` object.

All `PeakFinderConfiguration` properties are tunable.

#### Scope Window Use

On the **Measurements** tab, in the **Measurements** section, select **Peak Finder** to enable the peak finder. Click **Peak Finder** to modify the peak finder settings.

#### SignalStatistics — Signal statistics measurements

`SignalStatisticsConfiguration` object

Signal statistics measurements to compute and display signal statistics, specified as a `SignalStatisticsConfiguration` object.

All `SignalStatisticsConfiguration` properties are tunable.

#### Scope Window Use

On the **Measurements** tab, in the **Measurements** section, select **Signal Statistics**. Click **Signal Statistics** to choose the statistics to compute.

#### Trigger — Trigger measurements

`TriggerConfiguration` object

Trigger measurements, specified as a `TriggerConfiguration` object. Define a trigger event to identify the simulation time of specified input signal characteristics. You can use trigger events to stabilize periodic signals such as a sine wave or capture non-periodic signals such as a pulse that occurs intermittently.

All `TriggerConfiguration` properties are tunable.

#### Scope Window Use

On the **Measurements** tab, in the **Trigger** section, select **Enable Trigger** and click **Settings** to modify the trigger settings.

### Visualization

#### Name — Window name

"Time Scope" (default) | character vector | string scalar

Specify the name of the scope as a character vector or string scalar. This name appears as the title of the scope's figure window. To specify a title of a scope plot, use the `Title` property.

Data Types: char | string

**Position — Window position**

screen center (default) | [left bottom width height]

Scope window position in pixels, specified by the size and location of the scope window as a four-element vector of the form [left bottom width height]. You can place the scope window in a specific position on your screen by modifying the values of this property.

By default, the window appears in the center of your screen with a width of 800 pixels and height of 500 pixels. The exact values of the position depend on your screen resolution.

**ChannelNames — Channel names**

{ ' ' } (default) | cell array of character vectors | array of strings

Specify the input channel names as a cell array of character vectors or an array of strings. The channel names appear in the legend, and on the **Measurements** tab under **Select Channel**. If you do not specify names, the channels are labeled as Channel 1, Channel 2, etc.

**Dependency**

To enable this property, set ShowLegend to true.

Data Types: char

**ActiveDisplay — Active display for setting properties**

1 (default) | integer

Active display used to set properties, specified by the integer display number. The number of a display corresponds to the display's row-wise placement index. Setting this property controls which display is used for the following properties: YLimits, YLabel, ShowLegend, ShowGrid, Title, and PlotAsMagnitudePhase.

**Scope Window Use**

On the **Scope** tab, click **Settings**. In the Time Scope Settings dialog box, under **Display and Labels**, specify the active display using **Active Display**.

**Title — Display title**

' ' (default) | character vector | string scalar

Specify the display title as a character vector or a string scalar.

**Dependency**

When you set this property, ActiveDisplay controls the display that is updated.

**Scope Window Use**

On the **Scope** tab, click **Settings**. In the Time Scope Settings dialog box, under **Display and Labels**, specify the display title in the **Title** box.

Data Types: char | string

**YLabel — y-axis label**

"Amplitude" (default) | character vector | string scalar

Specify the text for the scope to display to the left of the y-axis.

**Dependencies**

This property applies only when `PlotAsMagnitudePhase` is `false`. When `PlotAsMagnitudePhase` is `true`, the two y-axis labels are read-only values "Magnitude" and "Phase", for the magnitude plot and the phase plot, respectively.

When you set this property, `ActiveDisplay` controls the display that is updated.

**Scope Window Use**

On the **Scope** tab, click **Settings**. In the Time Scope Settings dialog box, under **Display and Labels**, specify a label for the y-axis in the **Y-Label** box.

Data Types: `char` | `string`

**YLimits — y-axis limits**

`[-10,10]` (default) | `[ymin, ymax]`

Specify the y-axis limits as a two-element numeric vector, `[ymin, ymax]`.

- If `PlotAsMagnitudePhase` is `false`, the default is `[-10,10]`.
- If `PlotAsMagnitudePhase` is `true`, the default is `[0,10]`. This property specifies the y-axis limits of only the magnitude plot. The y-axis limits of the phase plot are always `[-180,180]`

**Dependency**

When you set this property, `ActiveDisplay` controls the display that is updated.

**Scope Window Use**

On the **Scope** tab, click **Settings**. In the Time Scope Settings dialog box, under **Display and Labels**, specify the limits for the y-axis in the **Y-Axis Limits** box.

**ShowLegend — Show legend**

`false` (default) | `true`

To show a legend with the input names, set this property to `true`.

From the legend, you can control which signals are visible. In the scope legend, click a signal name to hide the signal in the scope. To show the signal, click the signal name again.

**Scope Window Use**

On the **Scope** tab, click **Settings**. In the Time Scope Settings dialog box, under **Display and Labels**, select **Show Legend**.

Data Types: `logical`

**ShowGrid — Grid visibility**

`true` (default) | `false`

Set this property to `true` to show grid lines on the plot.

**Scope Window Use**

On the **Scope** tab, click **Settings**. In the Time Scope Settings dialog box, under **Display and Labels**, select **Show Grid**.

**PlotAsMagnitudePhase — Plot signal as magnitude and phase**`false` (default) | `true`

Plot signal as magnitude and phased, specified as either:

- `true` - The scope plots the magnitude and phase of the input signal on two separate axes within the same active display.
- `false` - The scope plots the real and imaginary parts of the input signal on two separate axes within the same active display.

This property is useful for complex-valued input signals. Turning on this property affects the phase for real-valued input signals. When the amplitude of the input signal is nonnegative, the phase is 0 degrees. When the amplitude of the input signal is negative, the phase is 180 degrees.

**Scope Window Use**

On the **Scope** tab, click **Settings**. In the Time Scope Settings dialog box, under **Display and Labels**, select **Magnitude Phase Plot**.

**Object Functions**

To use an object function, specify the object as the first input argument.

|                             |  |
|-----------------------------|--|
| <code>hide</code>           | Hide scope window  |
| <code>show</code>           | Display scope window   |
| <code>isVisible</code>      | Determine visibility of scope  |
| <code>generateScript</code> | Generate MATLAB script to create scope with current settings                                   |
| <code>step</code>           | Run System object algorithm  |
| <code>release</code>        | Release resources and allow changes to System object property values and input characteristics |
| <code>reset</code>          | Reset internal states of System object   |

If you want to restart the simulation from the beginning, call `reset` to clear the scope window displays. Do not call `reset` after calling `release`.

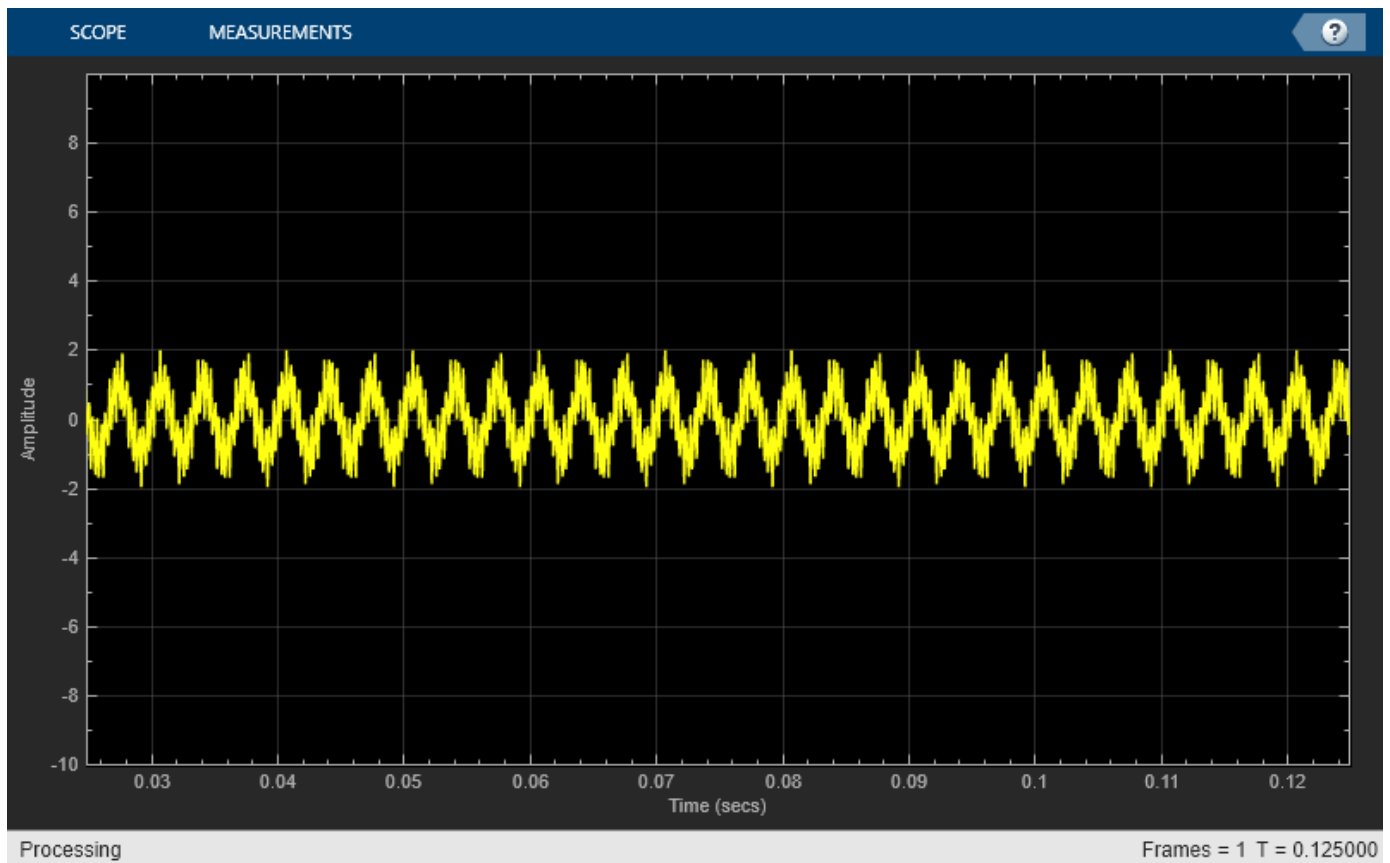
**Examples****View Sine Wave on Time Scope**

Create a sinusoidal signal with two tones, one at 0.3 kHz and the other at 3 kHz.

```
t = (0:1000)'/8e3;  
xin = sin(2*pi*0.3e3*t)+sin(2*pi*3e3*t);
```

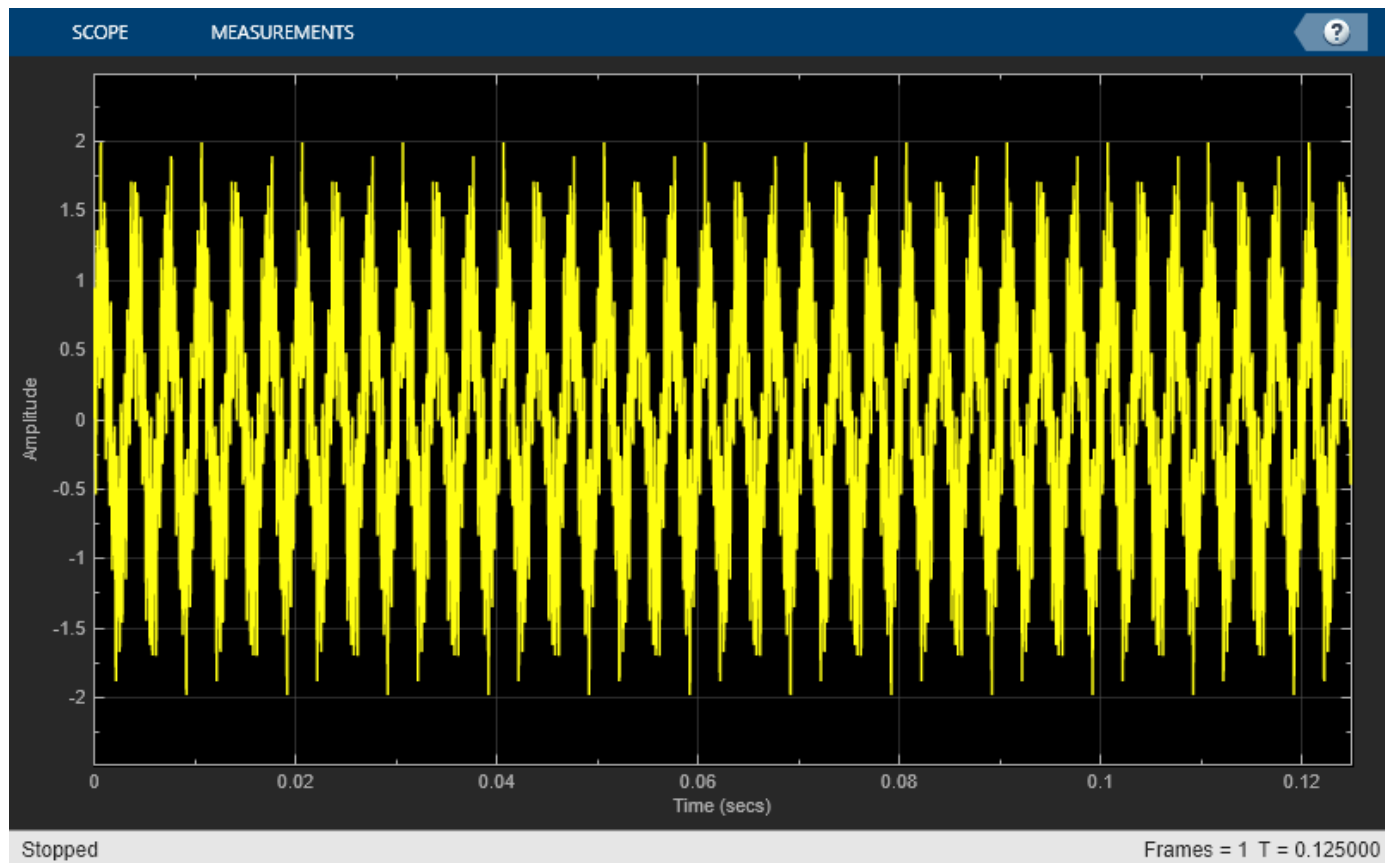
Create a `timescope` object and view the sinusoidal signal by calling the time scope object `scope`.

```
scope = timescope(SampleRate=8e3,...  
    TimeSpanSource="property",...  
    TimeSpan=0.1);  
scope(xin)
```



Run `release` to allow changes to property values and input characteristics. The scope automatically scales the axes.

```
release(scope);
```

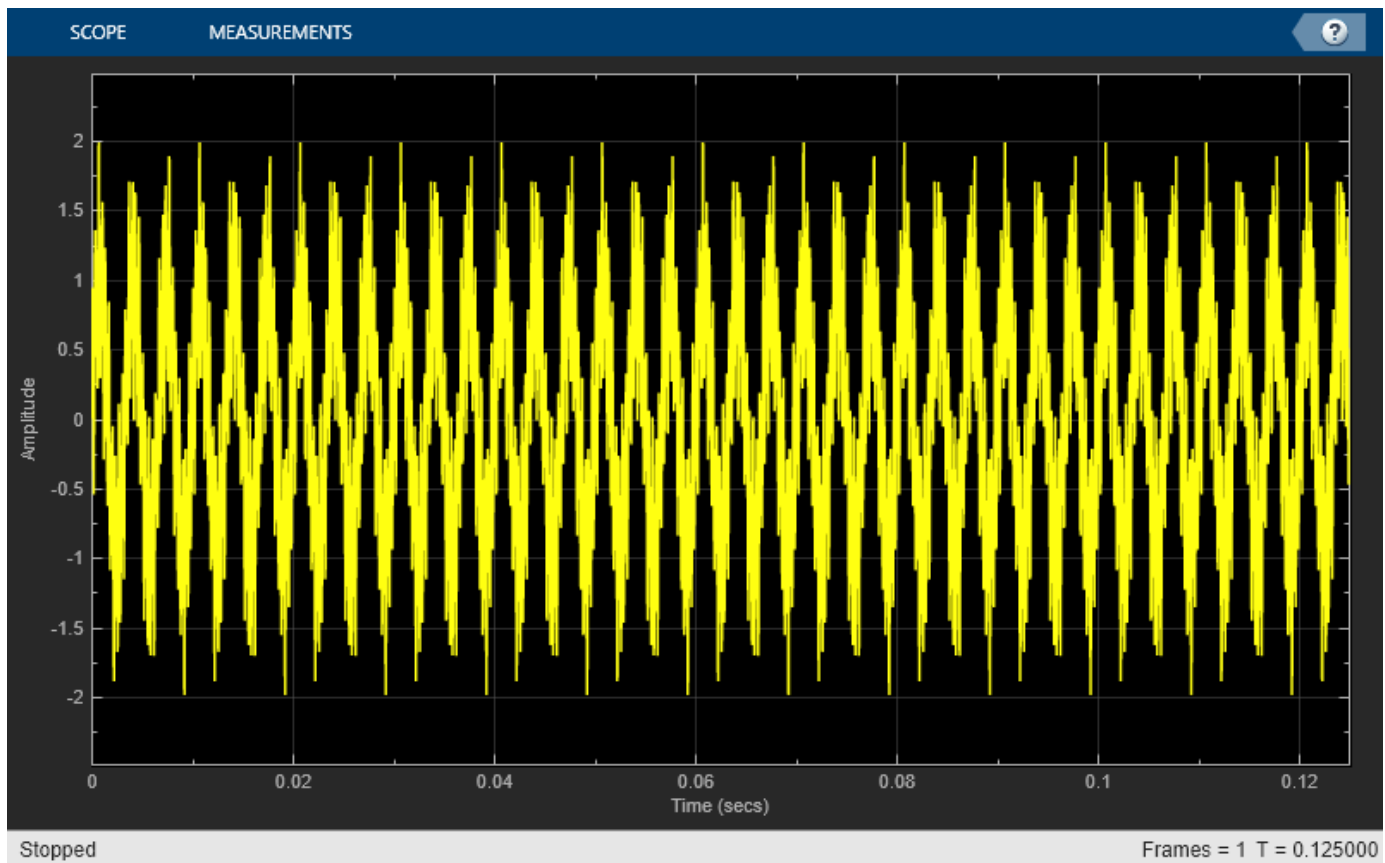


Hide the scope window.

```
if(isVisible(scope))  
    hide(scope)  
end
```

Show the scope window.

```
if(~isVisible(scope))  
    show(scope)  
end
```



## Use Bilevel Measurements Panel with Clock Input Signal

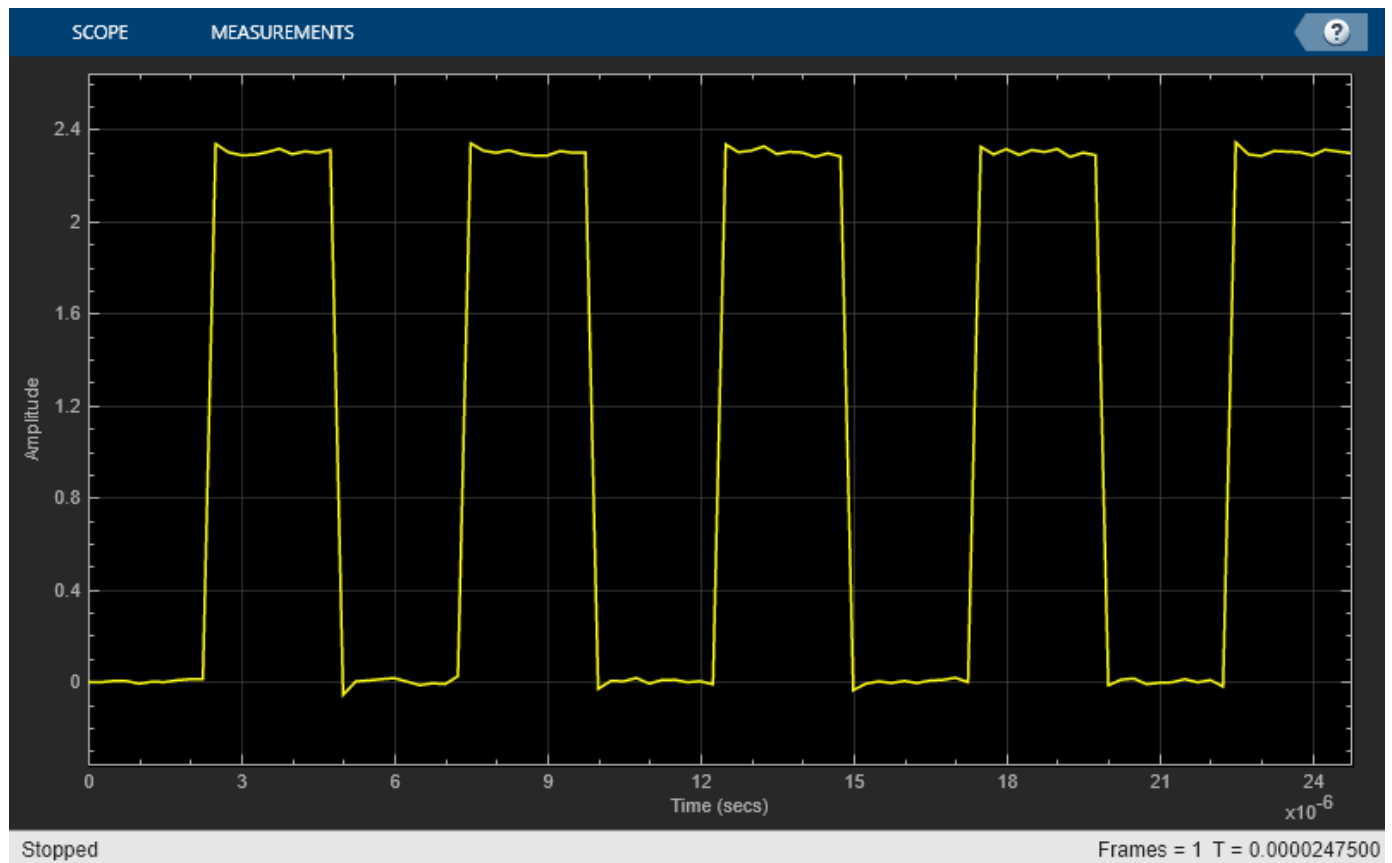
### Create and Display Clock Input Signal

Load the clock data, `x` and `t`. Find the sample time `ts`.

```
load clockx  
ts = t(2)-t(1);
```

Create a `timescope` object and call the object to display the signal. To autoscale the axes and enable changes to property values and input characteristics, call `release`.

```
scope = timescope(SampleRate=1/ts,TimeSpanSource="auto");  
scope(x);  
release(scope);
```

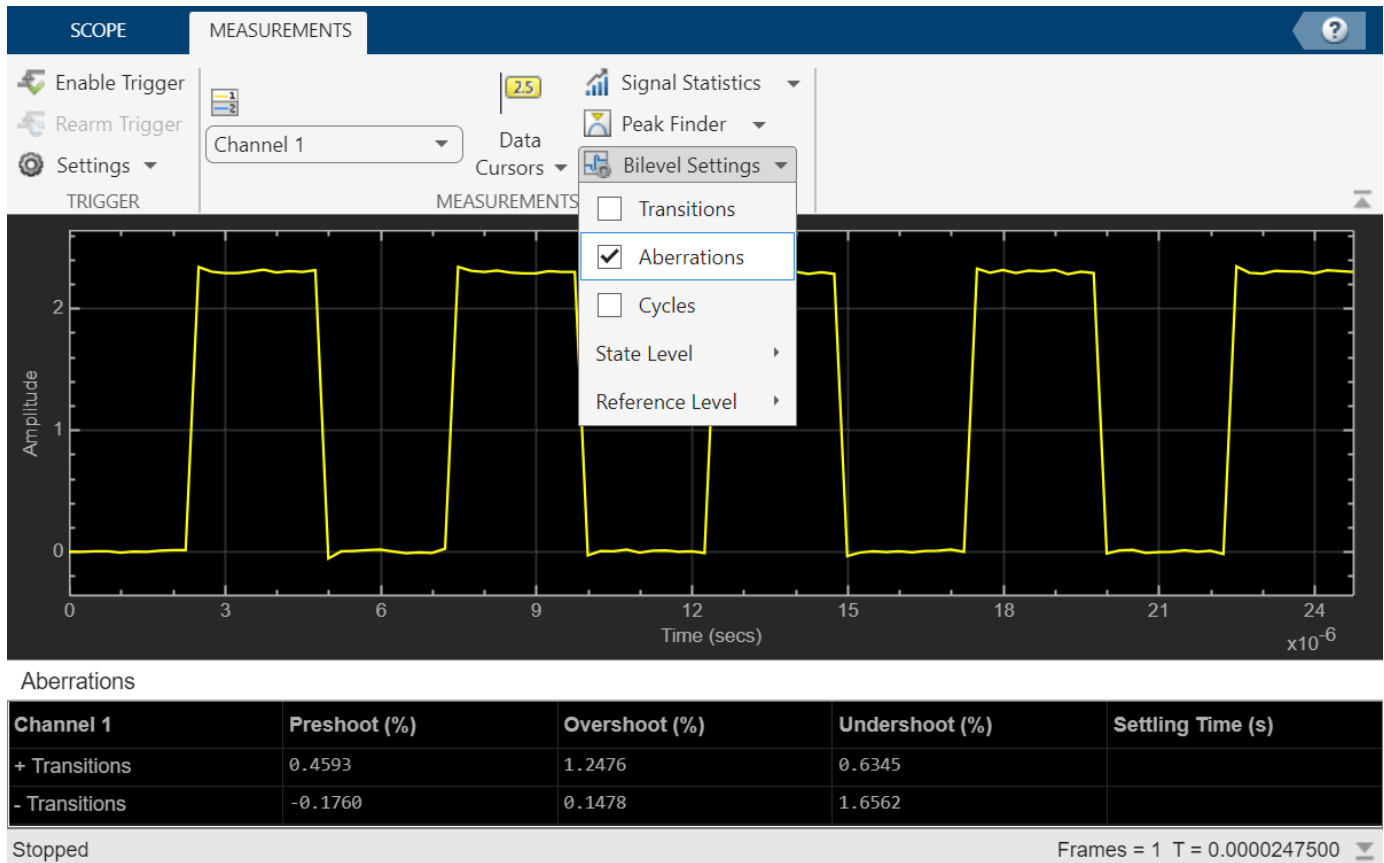


### Use Bilevel Measurements Panel to Find Settling Time

On the **Measurements** tab, under **Bilevel Settings**, select **Aberrations**.

Initially, the Time Scope does not display the **Settling Time(s)** measurement. This absence occurs because the default value of the **Settle Seek** parameter is longer than the entire simulation duration.

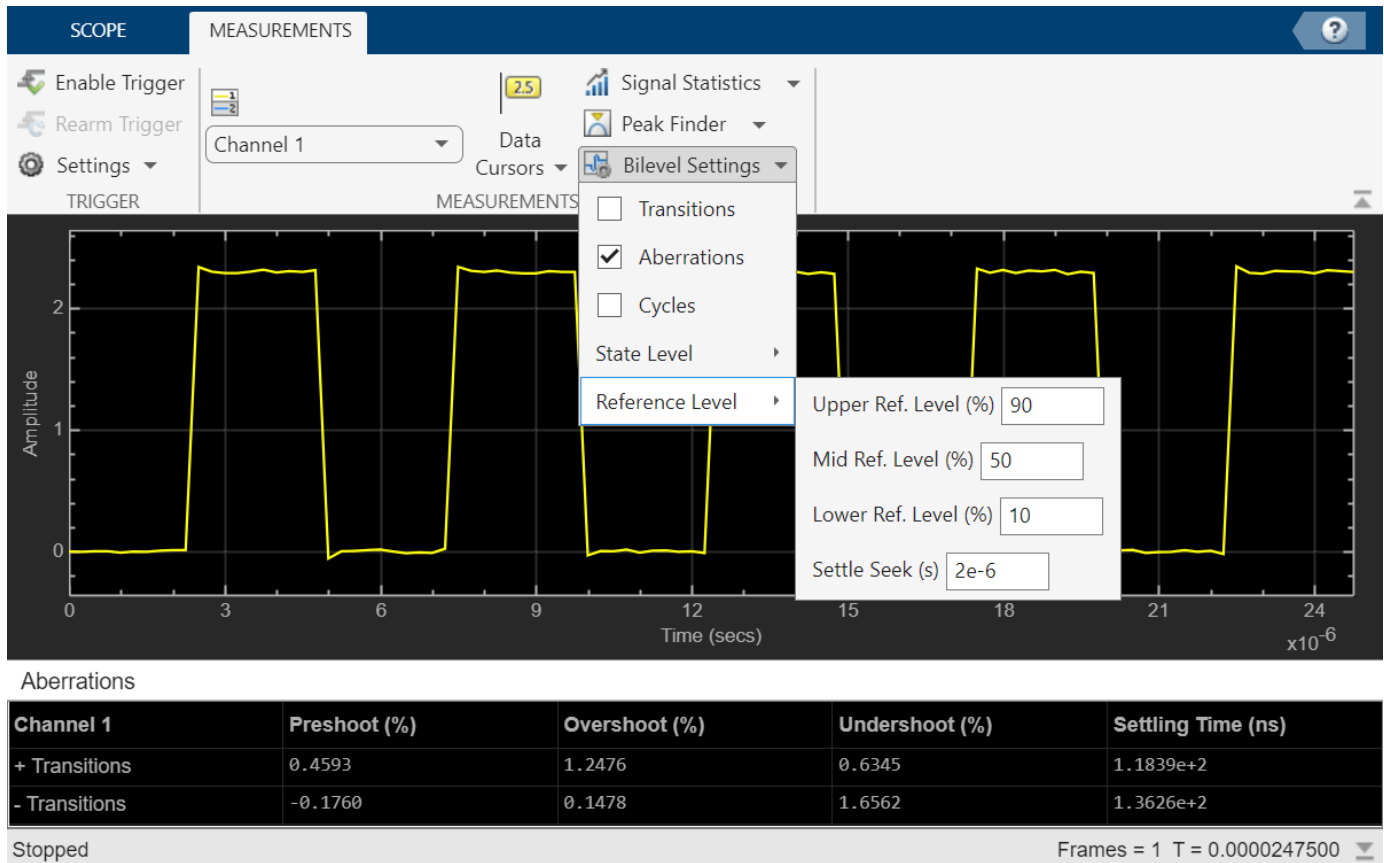




To change the value of the **Settle Seek (s)** parameter, click **Bilevel Settings**, and under **Reference Level**, set the settle seek value to  $2e-6$  and press **Enter**.

Time Scope now displays a rising edge **Settling Time** value of 118.392 ns.

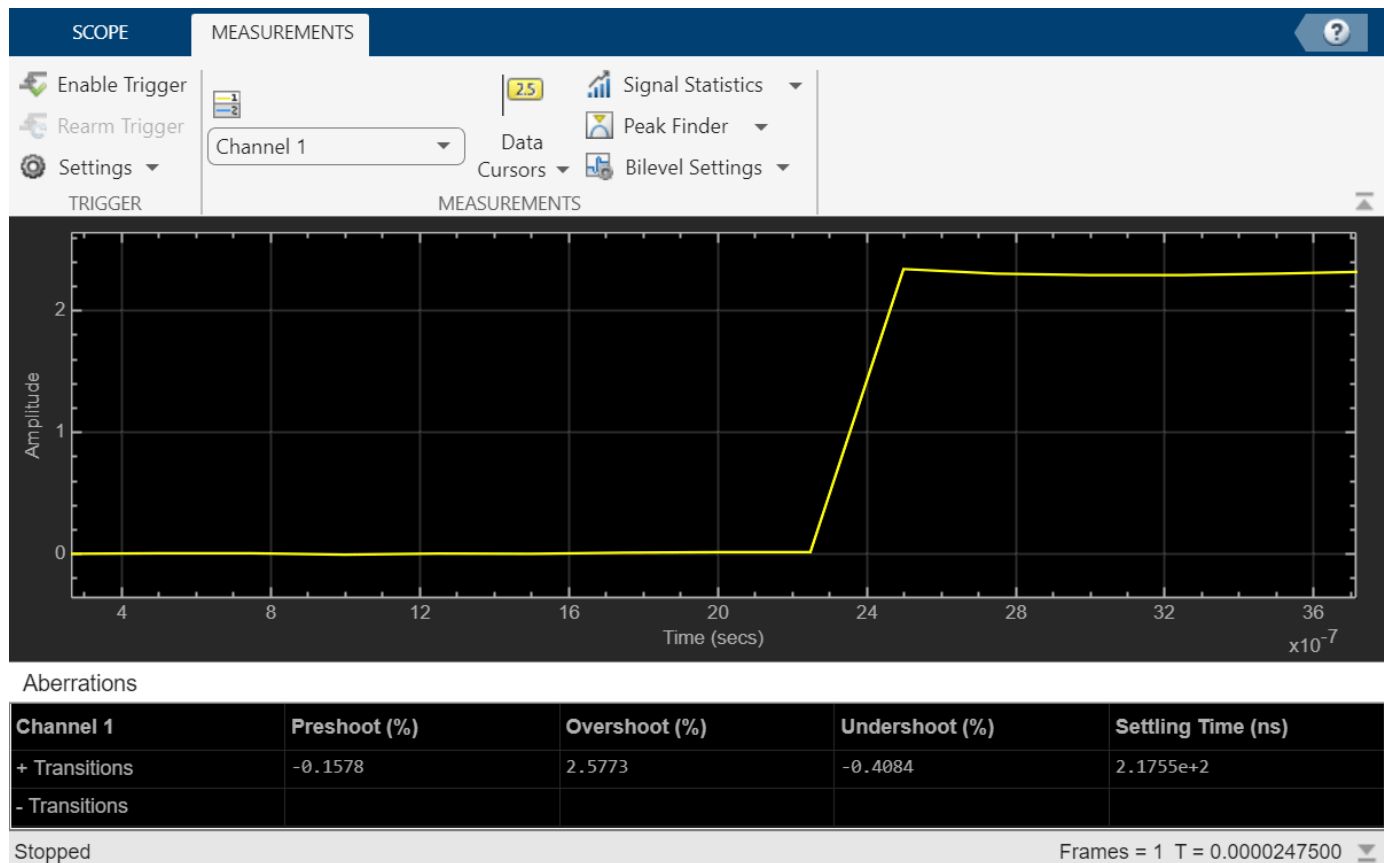
This settling time value is actually the statistical average of the settling times for all five rising edges. To show the settling time for only one rising edge, you can zoom in on that transition.



Hover over the upper right corner of the scope axes and click the zoom button.

Click and drag to zoom in on one of the transitions. Set **Settle Seek (s)** to  $2e-7$  and press **Enter**.

The Time Scope updates the rising edge **Settling Time** value to reflect the new time window.



## Configure Bilevel Measurements Programmatically in Time Scope MATLAB Object

Create a sine wave and view it in the Time Scope. Programmatically compute the bilevel measurements related to signal transitions, aberrations, and cycles.

### Initialization

Create the input sine wave using the `sin` function. Create a `timescope` MATLAB® object to display the signal. Set the `TimeSpan` property to 1 second.

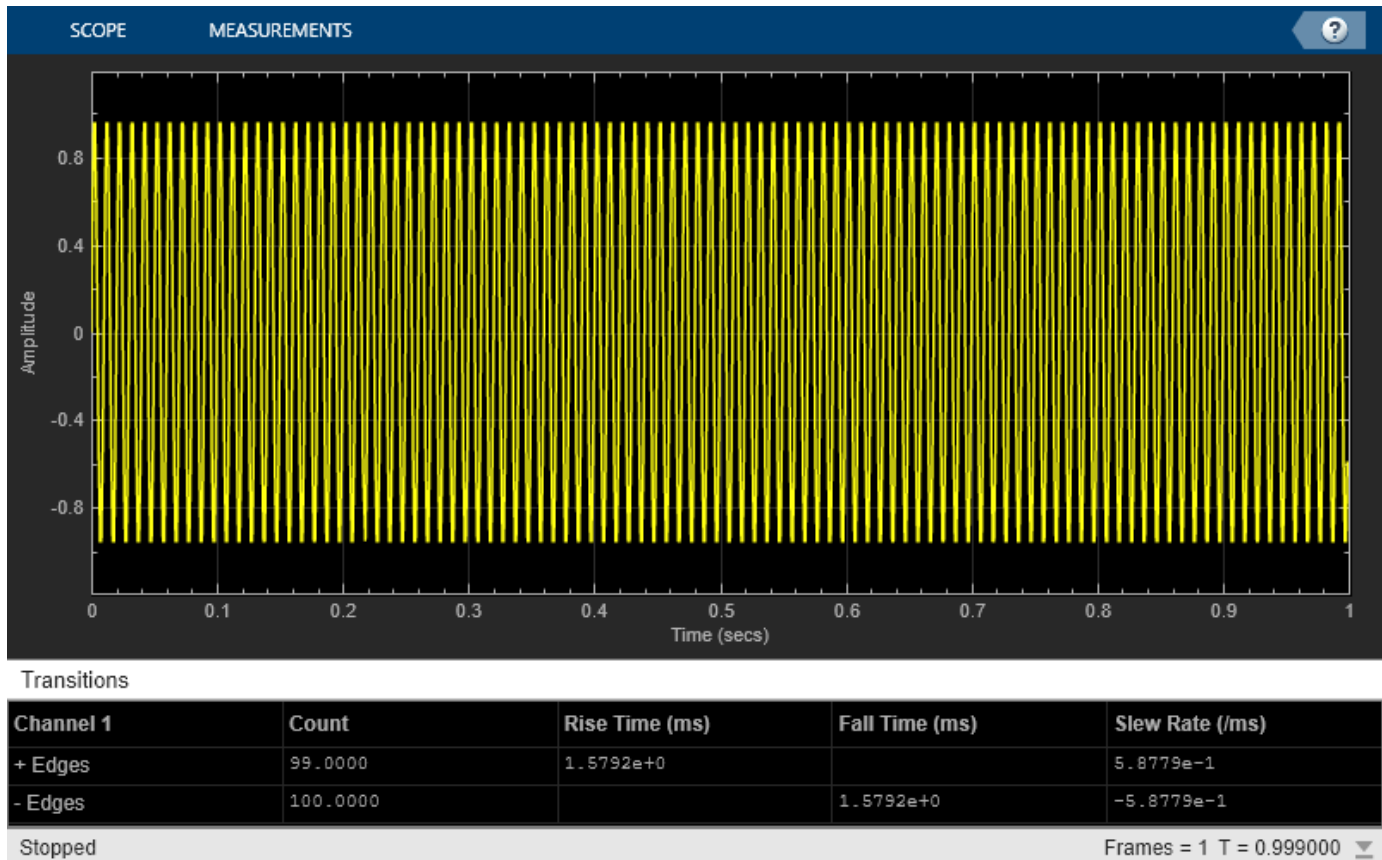
```
f = 100;
fs = 1000;
swv = sin(2.*pi.*f.*(0:1/fs:1-1/fs)).';
scope = timescope(SampleRate=fs,...
    TimeSpanSource="property",...
    TimeSpan=1);
```

### Transition Measurements

Enable the scope to show transition measurements programmatically by setting the `ShowTransitions` property to `true`. Display the sine wave in the scope.

Transition measurements such as rise time, fall time, and slew rate appear in the **Transitions** pane at the bottom of the scope.

```
scope.BilevelMeasurements.ShowTransitions = true;
scope(swv);
release(scope);
```

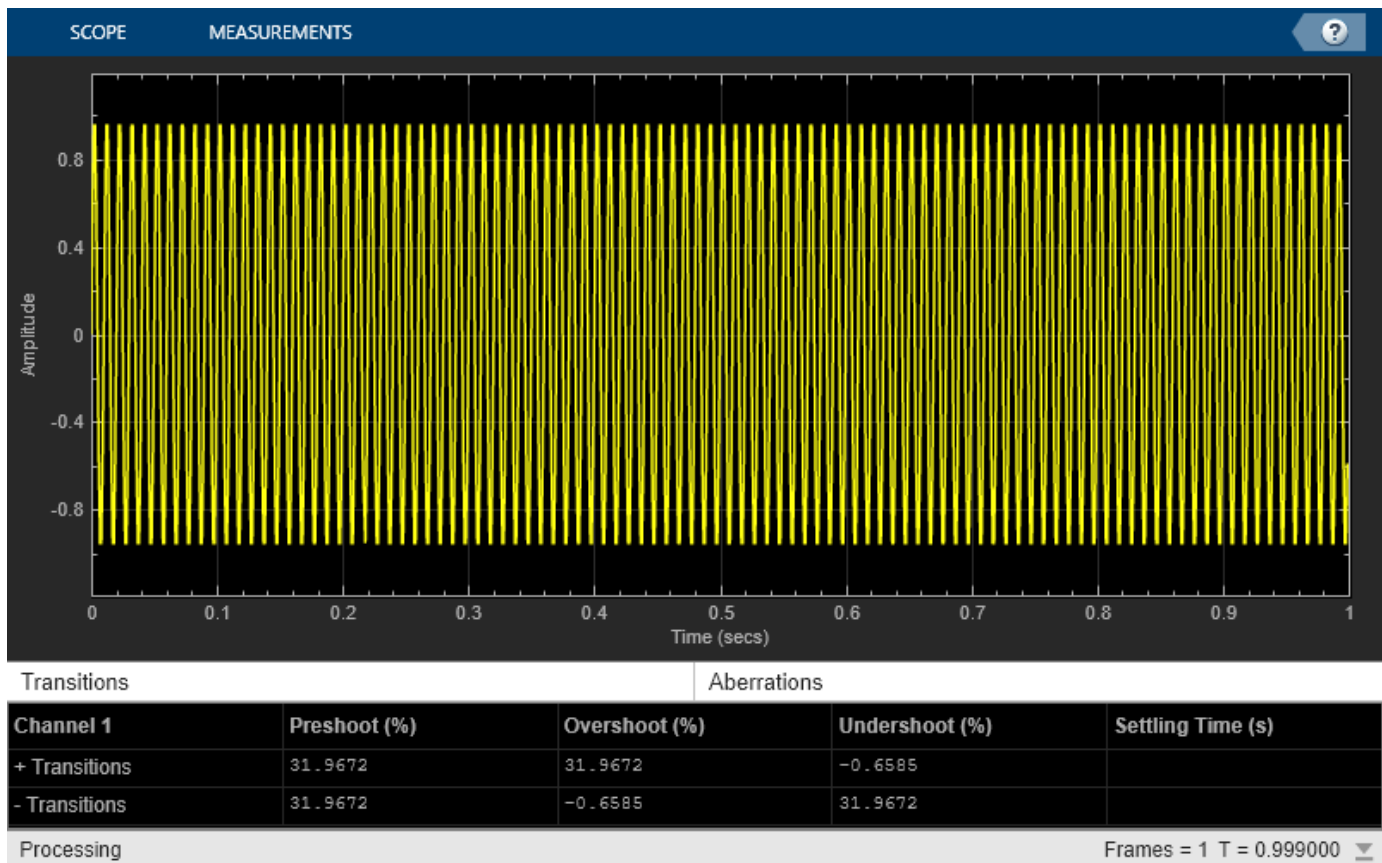


### Aberration Measurements

Enable the scope to show aberration measurements programmatically by setting the `ShowAberrations` property to `true`. Display the sine wave in the scope.

Aberration measurements such as preshoot, overshoot, undershoot, and settling time appear in the **Aberrations** pane at the bottom of the scope.

```
scope.BilevelMeasurements.ShowAberrations = true;
scope(swv);
release(scope);
```

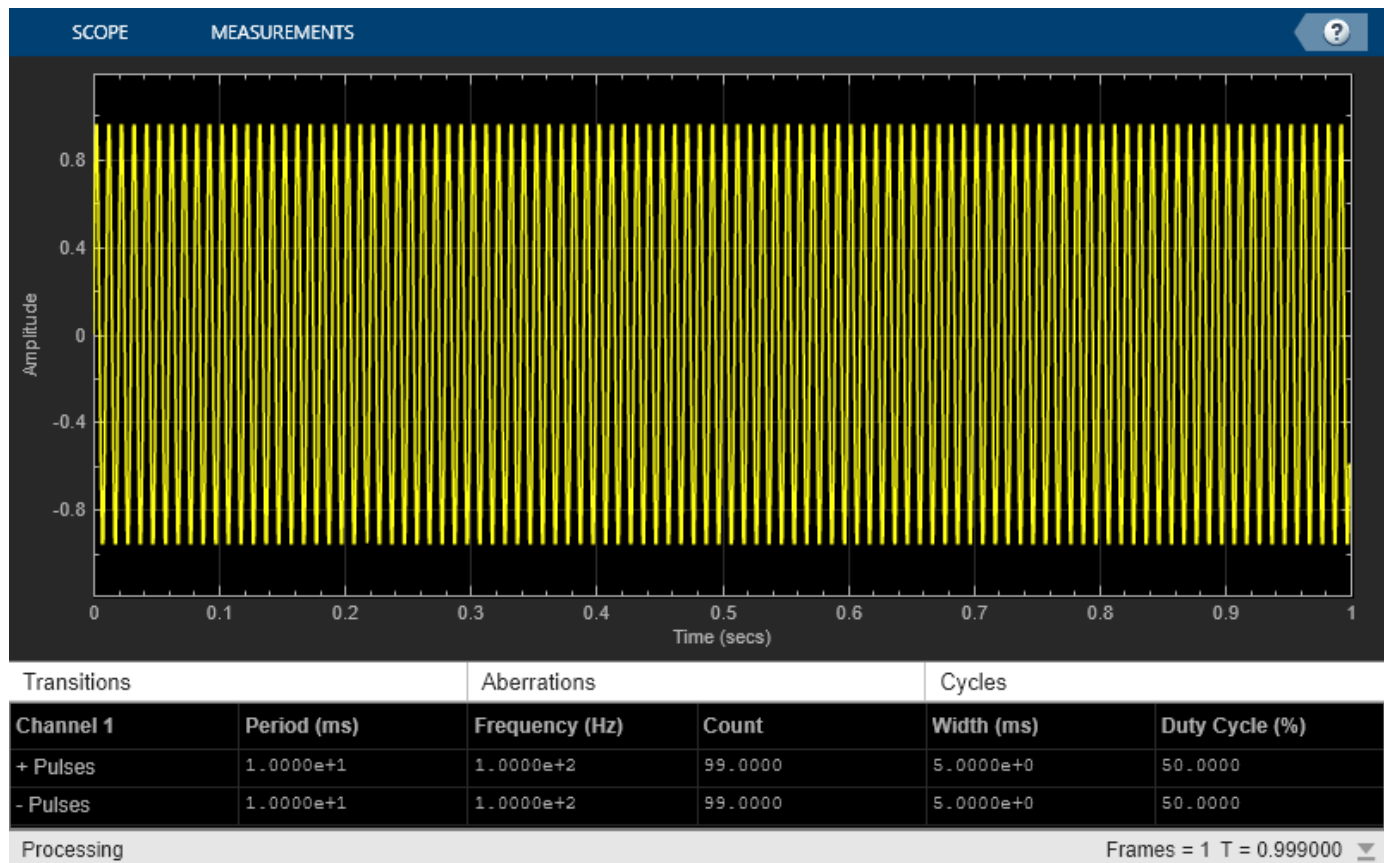


### Cycle Measurements

Enable the scope to show cycles measurements programmatically by setting the `ShowCycles` property to `true`. Display the sine wave in the scope.

Cycle measurements such as period, frequency, pulse width, and duty cycle appear in the **Cycles** pane at the bottom of the scope.

```
scope.BilevelMeasurements.ShowCycles = true;
scope(svw);
release(scope);
```



### Configure Signal Statistics Programmatically in Time Scope MATLAB Object

Create a sine wave and view it in the Time Scope. Enable the scope programmatically to compute the signal statistics.

The object supports these signal statistics:

- Maximum
- Minimum
- Mean
- Median
- RMS
- Peak to peak
- Variance
- Standard deviation
- Mean square

#### Initialization

Create the input sine wave using the `sin` function. Create a `timescope` MATLAB® object to display the signal. Set the `TimeSpan` property to 1 second.

```
f = 100;
fs = 1000;
swv = sin(2.*pi.*f.*(0:1/fs:1-1/fs)).';
scope = timescope(SampleRate=fs,...
    TimeSpanSource="property",...
    TimeSpan=1);
```

### Signal Statistics

Enable the scope to show signal statistics programmatically by setting the `SignalStatistics > Enabled` property to `true`.

```
scope.SignalStatistics.Enabled = true;
```

By default, the scope enables the following measurements.

```
scope.SignalStatistics
```

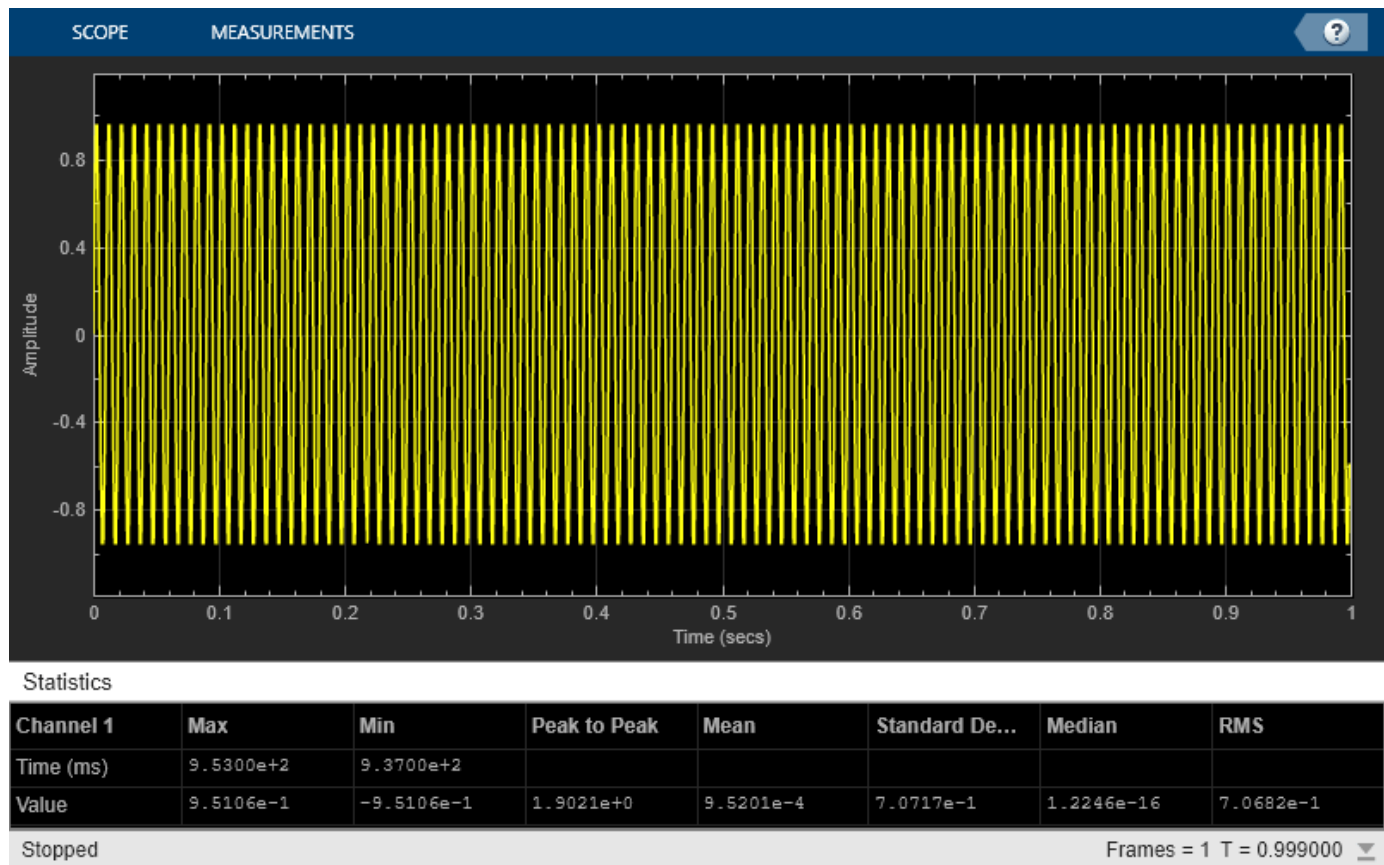
```
ans =
    SignalStatisticsConfiguration with properties:

        ShowMax: 1
        ShowMin: 1
    ShowPeakToPeak: 1
        ShowMean: 1
        ShowVariance: 0
    ShowStandardDeviation: 1
        ShowMedian: 1
        ShowRMS: 1
    ShowMeanSquare: 0
        Enabled: 1
```

Display the sine wave in the scope. A Statistics pane appears at the bottom of the scope window displaying the statistics for the portion of the signal that you can see in the scope.

If you use the zoom options on the scope, the statistics automatically adjust to the time range in the display.

```
scope(swv);
release(scope);
```



### Visualize Multiple Inputs with Different Sample Rates

This example shows how to visualize multiple inputs with different sample rates and plot the signals on multiple axes.

Generate three different sine waves and plot them in the Time Scope.

```

freq = 1/500;
t    = (0:100)'/freq;
t2   = (0:0.5:100)'/freq;
xin1 = sin(1/2*t);
xin2 = sin(1/4*t2);
xin  = sin(1/2*t2)+sin(1/4*t2);

scope = timescope(SampleRate=[freq freq/2 freq],...
    TimeSpanSource="property", ...
    TimeSpan=0.1,...
    LayoutDimensions=[2,1]);
scope(xin,xin1,xin2)

release(scope)

```





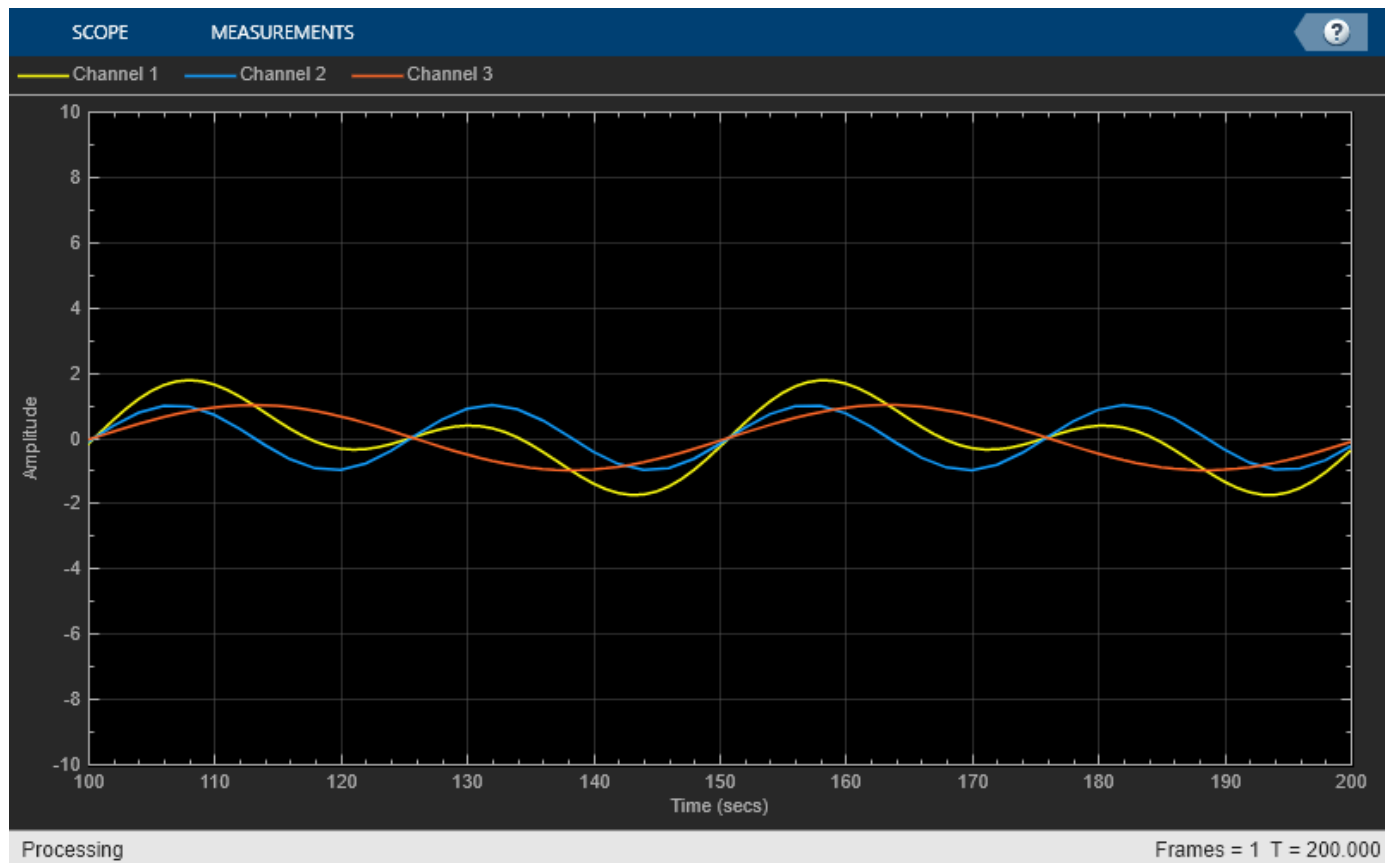
### Use Multiple Axes on Scope

This example shows how to add titles, set y-axis limits, and modify properties when you have multiple axes on your timescope object.

Use the timescope object to visualize three sine waves with two different sample rates.

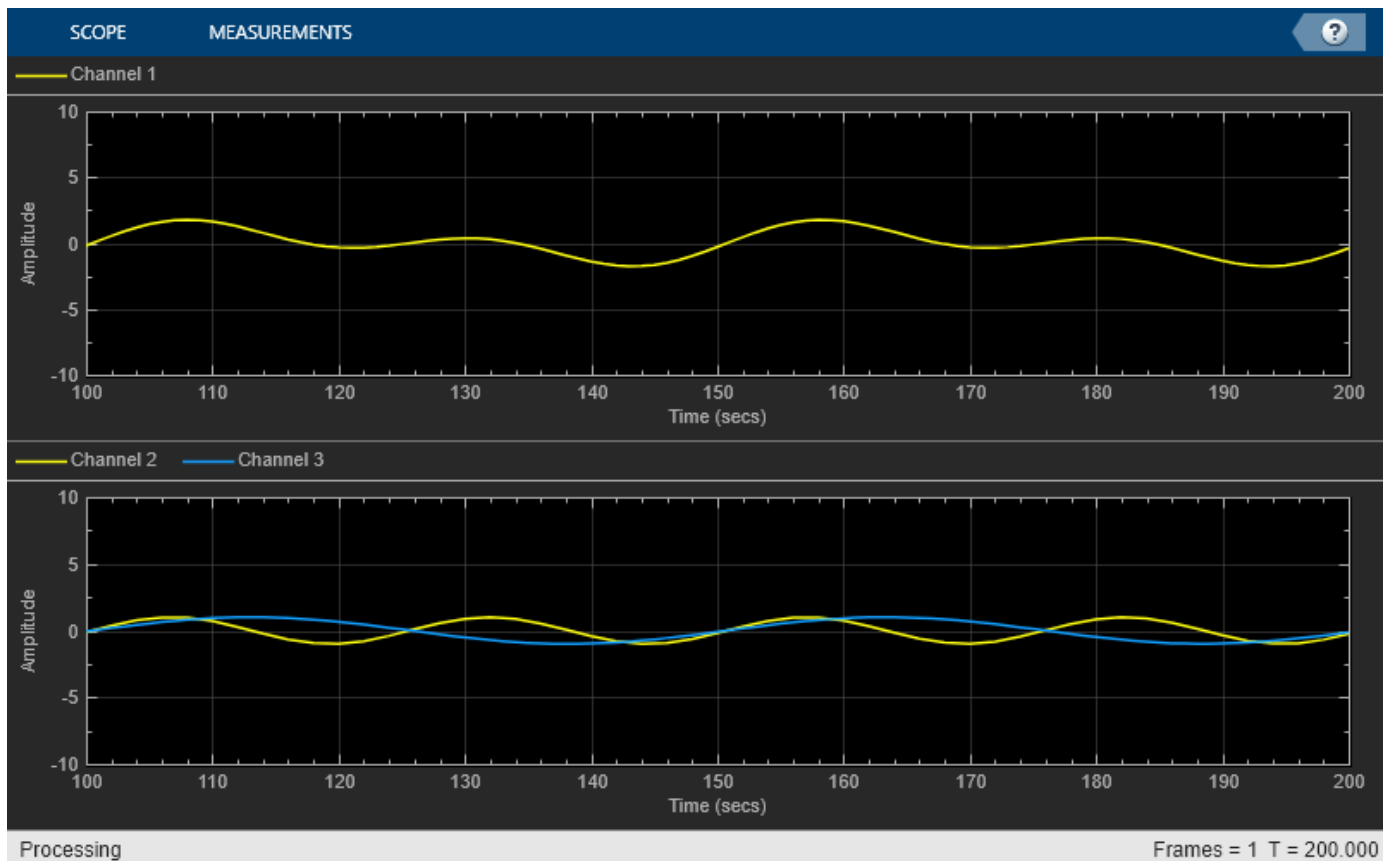
```
freq = 1;
t = (0:100)'/freq;
t2 = (0:0.5:100)'/freq;
xin1 = sin(1/2*t);
xin2 = sin(1/4*t2);
xin = sin(1/2*t2)+sin(1/4*t2);

scope = timescope(SampleRate=[freq freq/2 freq],...
    TimeSpanSource="property",...
    TimeSpan=100);
scope(xin, xin1, xin2)
```



Change the layout to add a second axis. The second and third inputs automatically move to the new second axis.

```
scope.LayoutDimensions = [2,1];
```

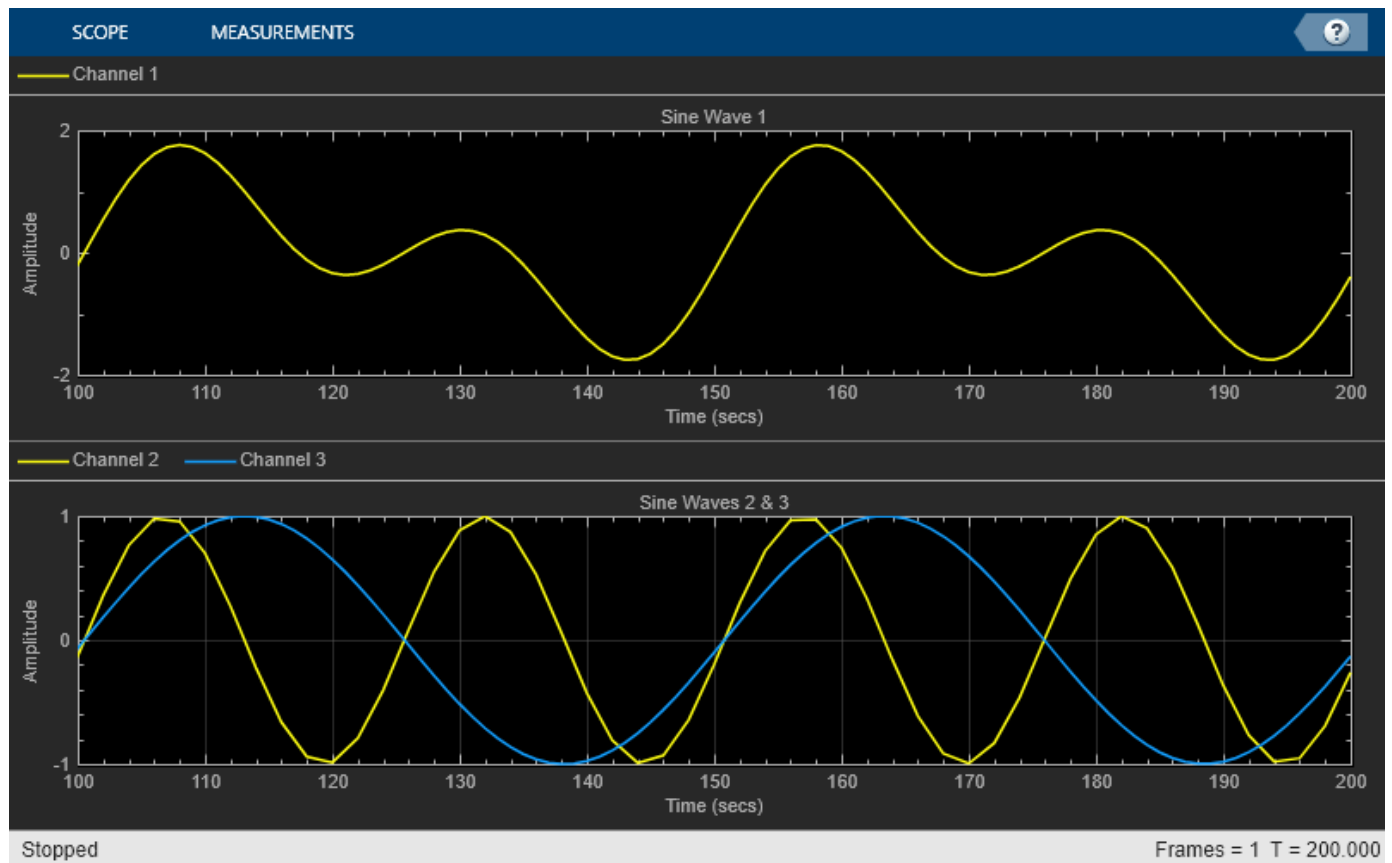


Modify the properties of the first axis.

```
scope.ActiveDisplay = 1;  
scope.ShowGrid = false;  
scope.Title = "Sine Wave 1";  
scope.YLimits = [-2,2];
```

Repeat this process to modify the second axis.

```
scope.ActiveDisplay = 2;  
scope.Title = "Sine Waves 2 & 3";  
scope.YLimits = [-1,1];  
release(scope)
```



### View Sine Wave Input Signals at Different Sample Rates and Offsets

Create a `dsp.SineWave` object. Create a `dsp.FIRDecimator` object to decimate the sine wave by 2. Create a `timescope` object with two input ports.

```
Fs = 1000; % Sample rate
sine = dsp.SineWave(Frequency=50,...
    SampleRate=Fs,...
    SamplesPerFrame=100);
decimate = dsp.FIRDecimator; % To decimate sine by 2
scope = timescope(SampleRate=[Fs Fs/2],...
    TimeDisplayOffset=[0 38/Fs],...
    TimeSpanSource="Property",...
    TimeSpan=0.25,...
    YLimits=[-1 1],...
    ShowLegend=true);
```

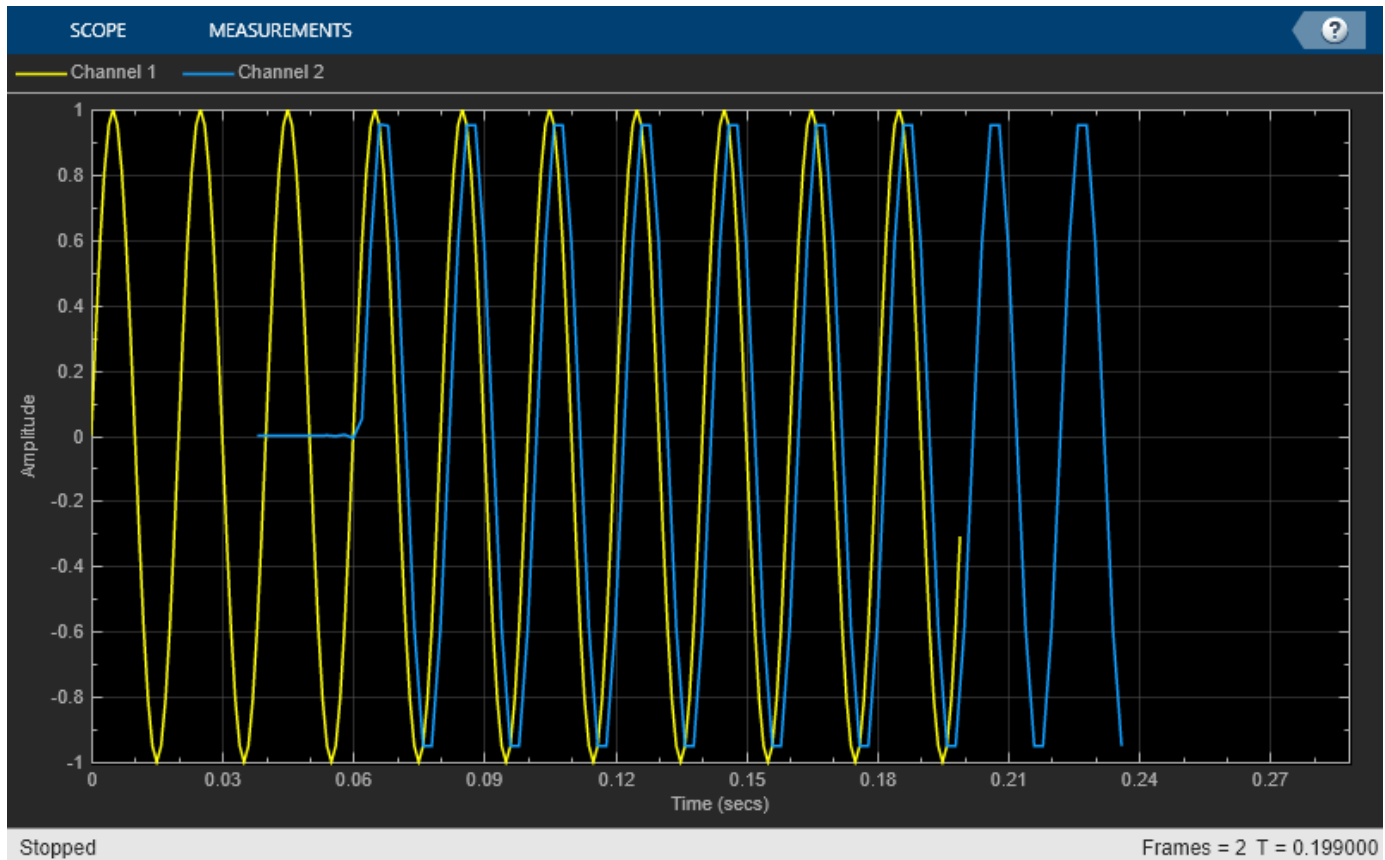
Call the `dsp.SineWave` object to create a sine wave signal. Use the `dsp.FIRDecimator` object to create a second signal that equals the original signal and decimate it by a factor of 2. Display the signals by calling the `timescope` object.

```
for ii = 1:2
    xsine = sine();
    xdec = decimate(xsine);
```

```

    scope(xsine,xdec)
end
release(scope)

```



Close the Time Scope window and clear the variables.

```
clear scope Fs sine decimate ii xsine xdec
```

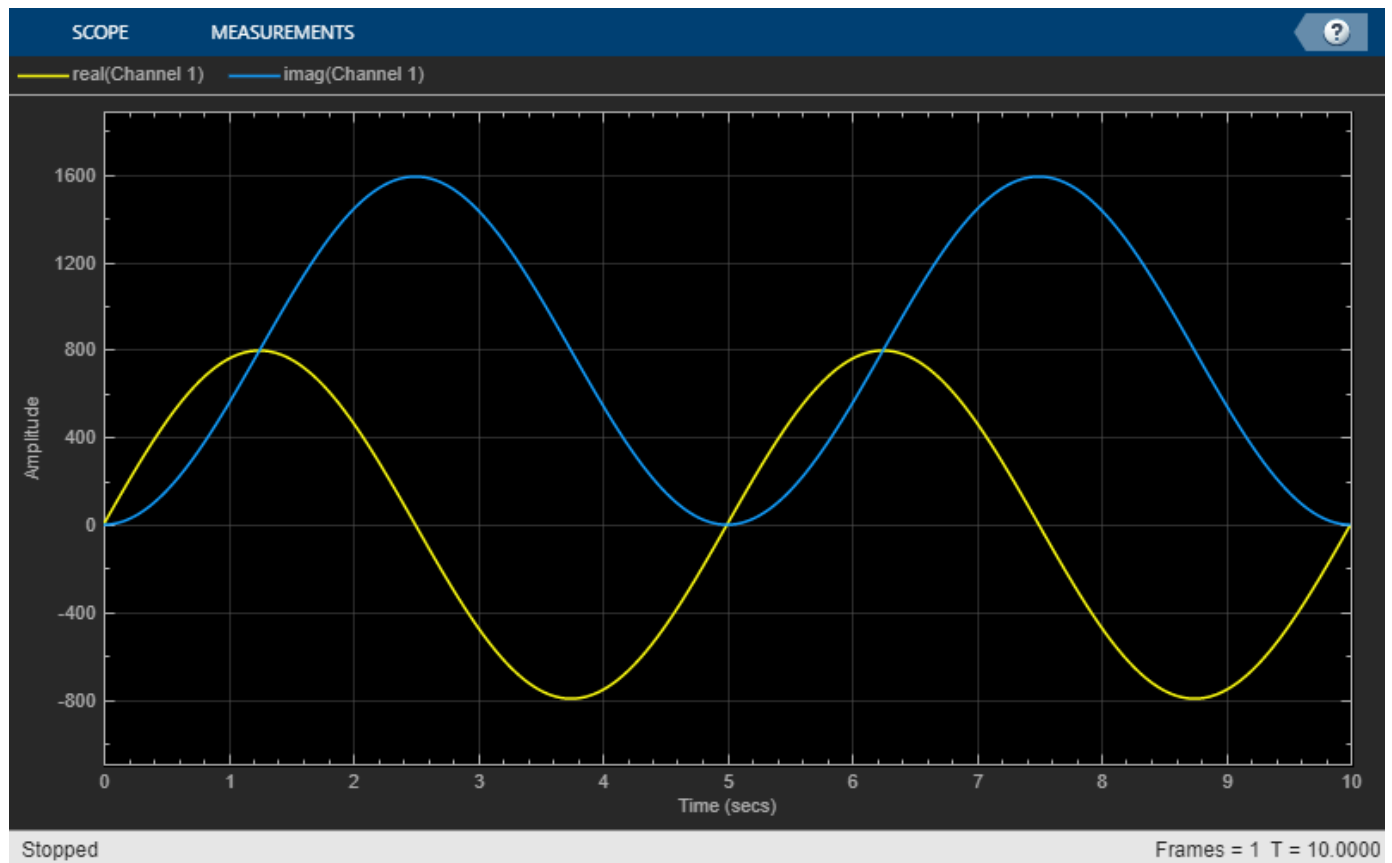
### Display Complex-Valued Input Signal

Create a vector representing a complex-valued sinusoidal signal and a `timescope` object. Call the `scope` to display the signal.

```

fs = 1000;
t = (0:1/fs:10)';
CxSine = cos(2*pi*0.2*t) + 1i*sin(2*pi*0.2*t);
CxSineSum = cumsum(CxSine);
scope = timescope(SampleRate=fs,...
    TimeSpanSource="auto", ShowLegend=1);
scope(CxSineSum);
release(scope)

```

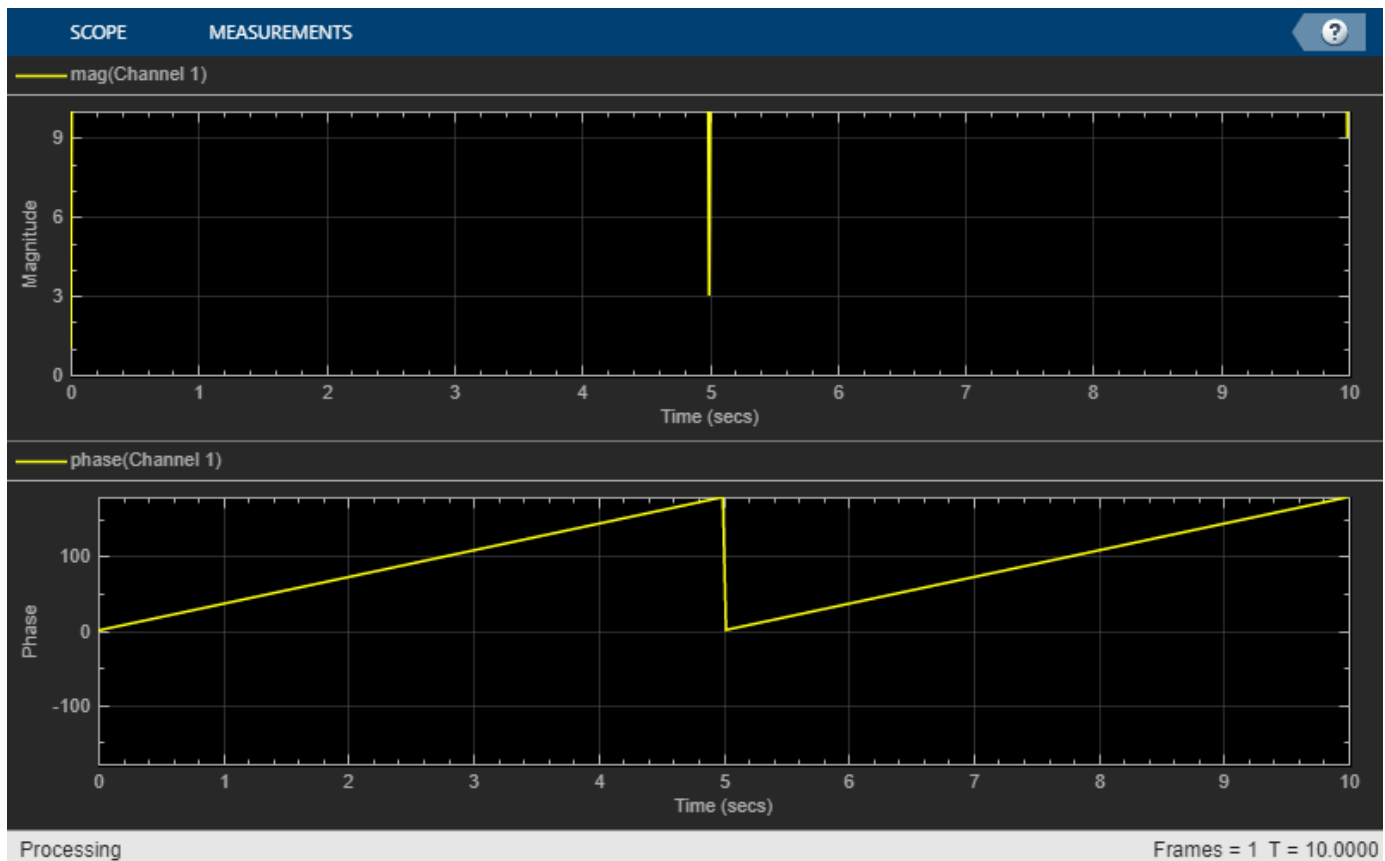


By default, when the input is a complex-valued signal, the Time Scope plots the real and imaginary portions on the same axes. These portions appear as different-colored lines on the same axes within the same active display.

Change the `PlotAsMagnitudePhase` property to `true` and call `release`.

The Time Scope now plots the magnitude and phase of the input signal on two separate axes within the same active display. The top axes display magnitude and the bottom axes display the phase in degrees.

```
scope.PlotAsMagnitudePhase = true;
scope(CxSineSum);
release(scope)
```

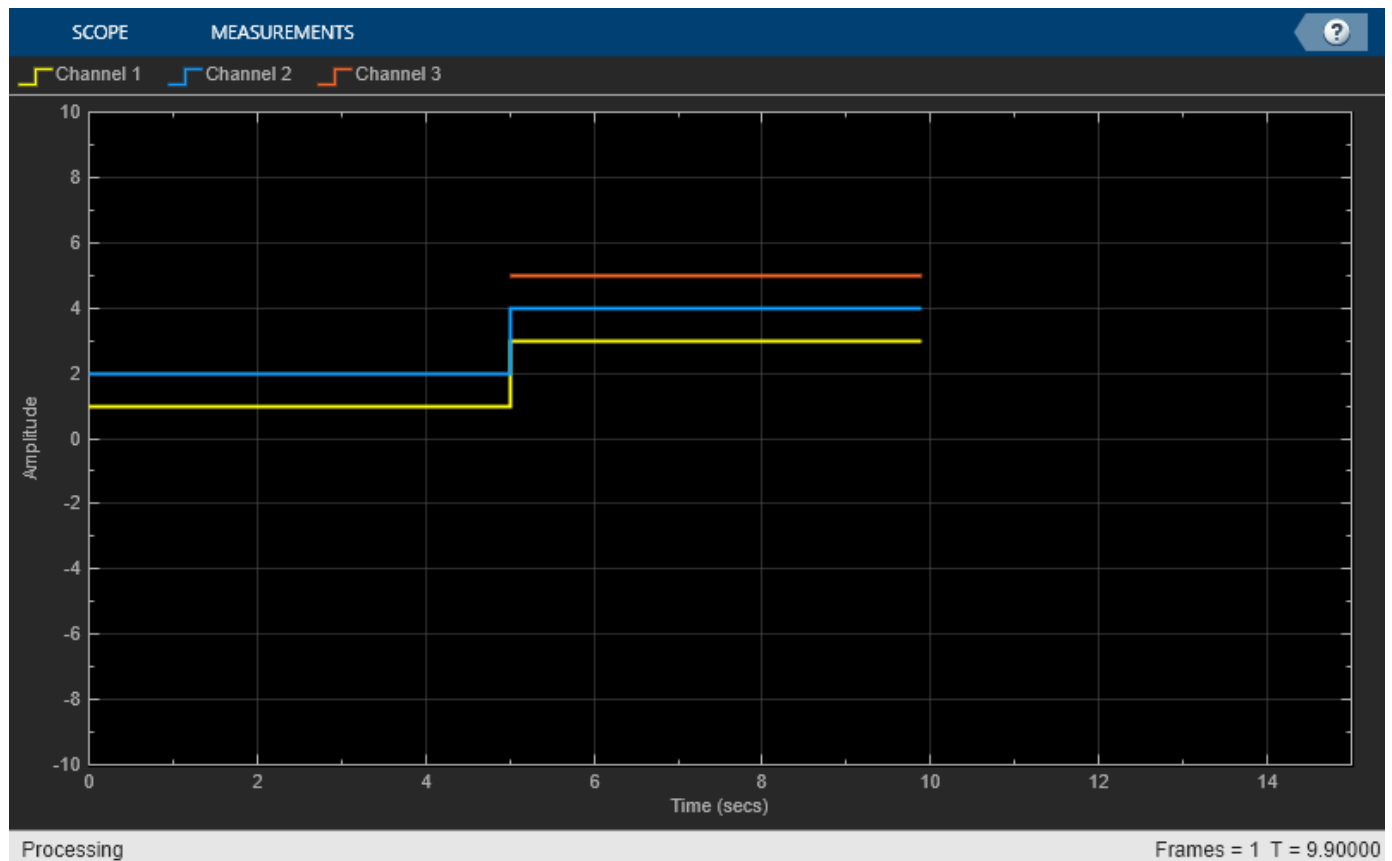


### Display Input Signal of Changing Size

This example shows how the `timescope` object visualizes inputs that change dimensions halfway through.

Create a vector that represents a two-channel constant signal. Create another vector that represents a three-channel constant signal. Create a `timescope` object and call the scope with two inputs to display the signal.

```
fs = 10;
sigdim2 = [ones(5*fs,1) 1+ones(5*fs,1)]; % 2-dim 0-5 s
sigdim3 = [2+ones(5*fs,1) 3+ones(5*fs,1) 4+ones(5*fs,1)]; % 3-dim 5-10 s
scope = timescope(SampleRate=fs,TimeSpanSource="property");
scope.PlotType = "stairs";
scope.TimeSpanOvverrunAction = "scroll";
scope.TimeDisplayOffset = [0 5];
scope([sigdim2; sigdim3(:,1:2)], sigdim3(:,3));
```

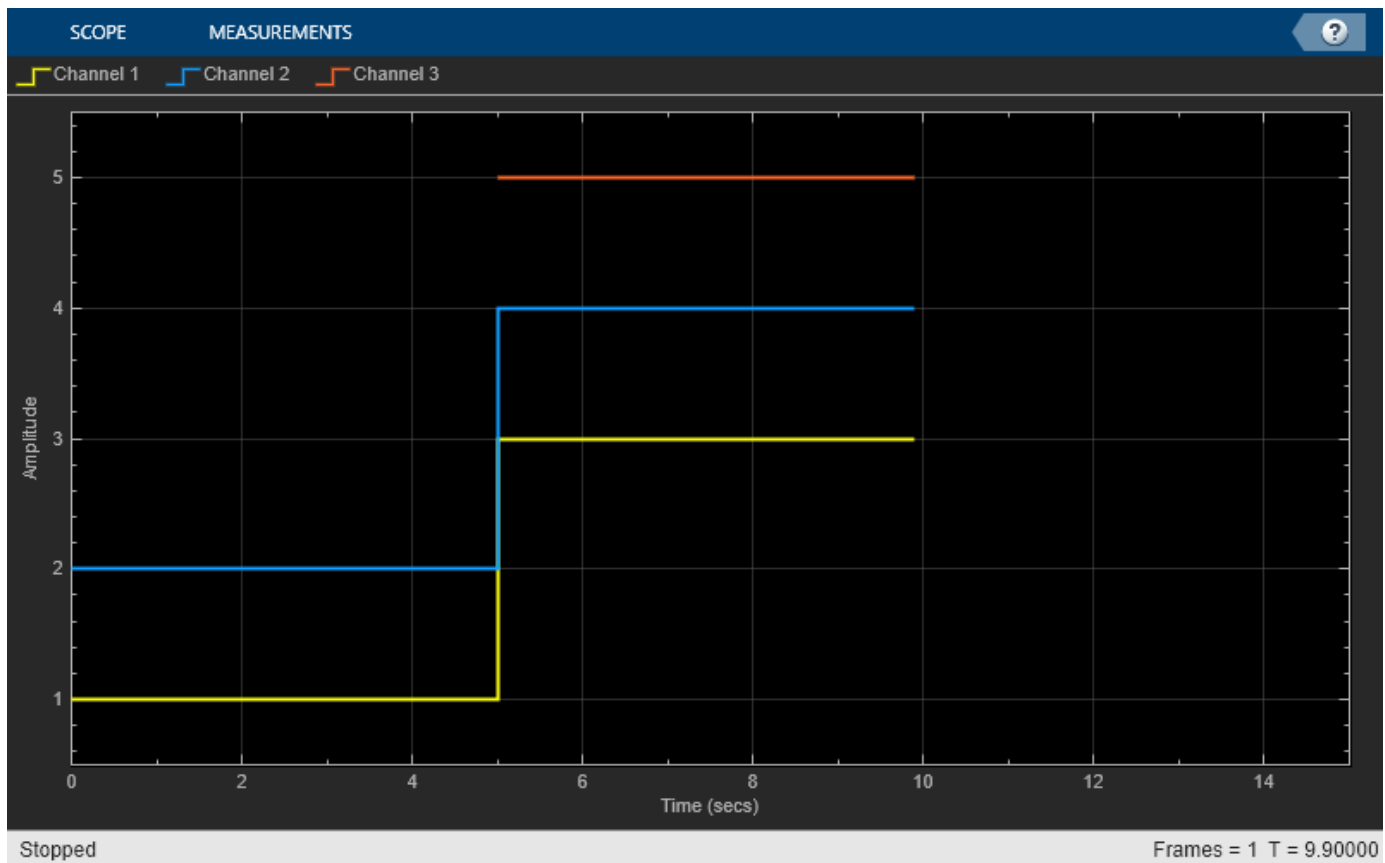


The size of the input signal to the Time Scope changes as the simulation progresses. When the simulation time is less than 5 seconds, the Time Scope plots only the two-channel signal `sigdim2`. After 5 seconds, the Time Scope also plots the three-channel signal `sigdim3`.

Run the `release` method to enable changes to property values and input characteristics. The scope automatically scales the axes.

```
release(scope)
```





## Find Heart Rate Using Peak Finder Panel with ECG Input Signal

Use Peak Finder pane in the Time Scope to measure heart rate.

### Create and Display ECG Signal

Use the custom ecg function to generate an electrocardiogram (ECG) signal.

type `ecg.m`

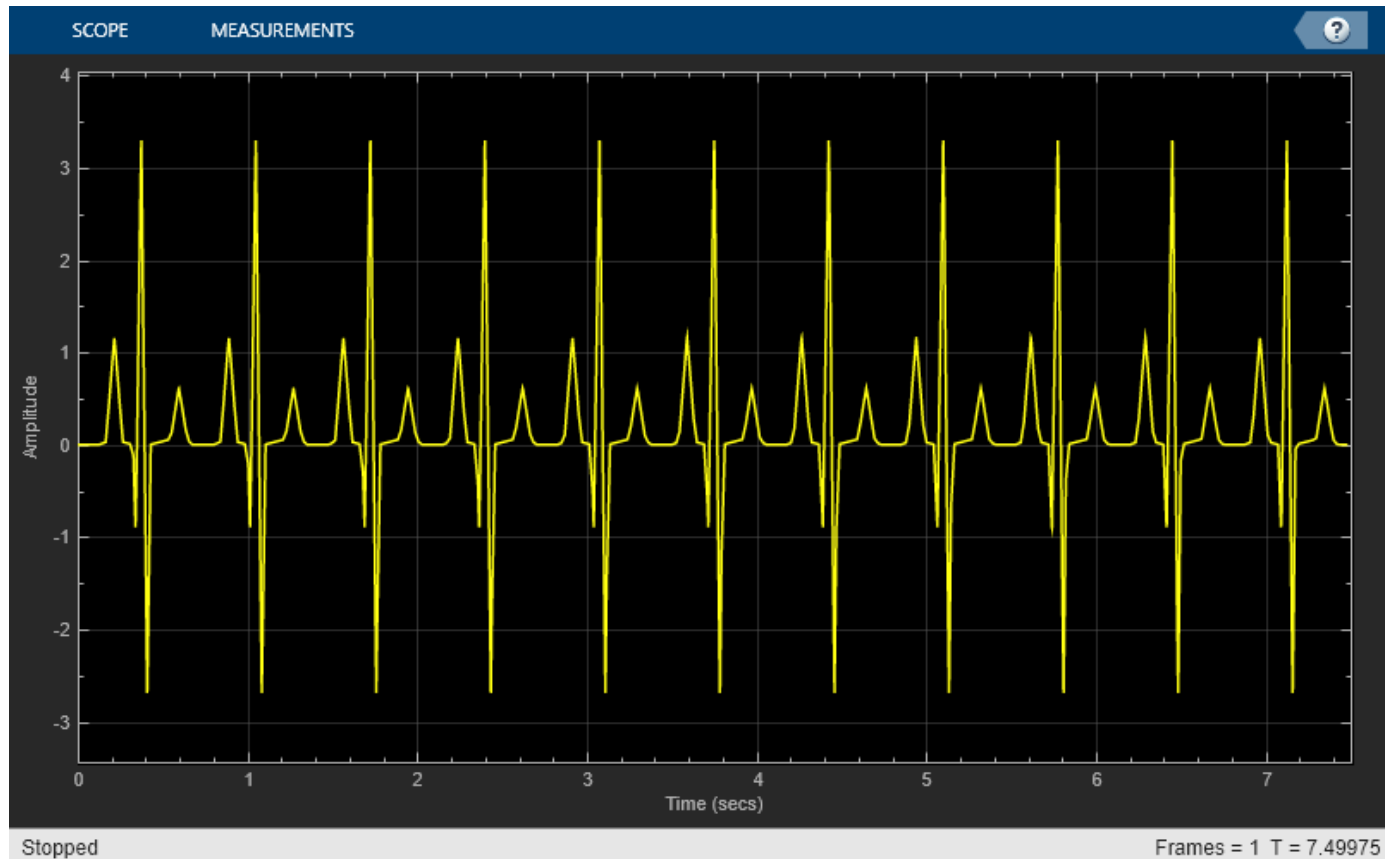
```
function x = ecg(L)
a0 = [0, 1, 40, 1, 0, -34, 118, -99, 0, 2, 21, 2, 0, 0, 0];
d0 = [0, 27, 59, 91, 131, 141, 163, 185, 195, 275, 307, 339, 357, 390, 440];
a = a0 / max(a0);
d = round(d0 * L / d0(15));
d(15) = L;
for i = 1:14
    m = d(i) : d(i+1) - 1;
    slope = (a(i+1) - a(i)) / (d(i+1) - d(i));
    x(m+1) = a(i) + slope * (m - d(i));
end

x1 = 3.5*ecg(2700).';
y1 = sgolayfilt(kron(ones(1,13),x1),0,21);
n = (1:30000)';
```

```
del = round(2700*rand(1));
mhb = y1(n + del);
ts = 0.00025;
```

Create a `timescope` object and call the object to display the signal. To autoscale the axes and enable changes to property values and input characteristics, call `release`.

```
scope = timescope(SampleRate=1/ts);
scope(mhb);
release(scope)
```



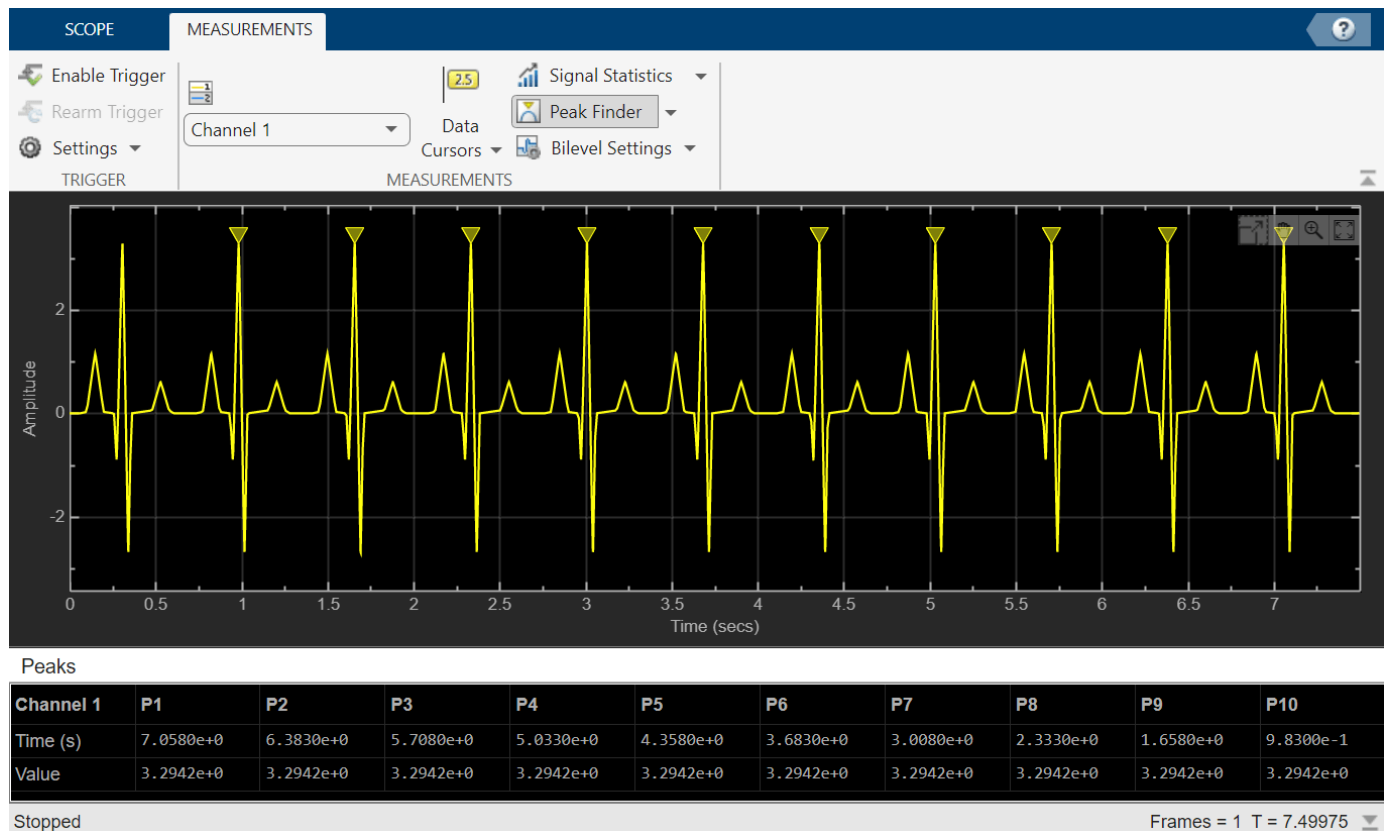
### Find Heart Rate

Use the Peak Finder measurements to measure the time between heartbeats.

In the **Measurements** tab, select **Peak Finder** to enable the peak finder measurements.

Click the **Peak Finder** arrow and set the **Num Peaks** property to 10 and hit enter.

In the **Peaks** pane at the bottom of the window, the Time Scope displays a list of ten peak amplitude values and the times at which they occur.



The list of peak values shows a constant time difference of 0.675 seconds between each heartbeat. Based on this equation, the heart rate of this ECG signal is about 89 beats per minute.

$$\frac{60 \text{ s/min}}{0.675 \text{ s/beat}} = 88.89 \text{ bpm}$$

Close the Time Scope window and remove the variables you created from the workspace.

```
clear scope x1 y1 n del mhb ts
```

## Tips

- To close the scope window and clear its associated data, use the MATLAB `clear` function.
- To hide or show the scope window, use the `hide` and `show` functions.
- Use the MATLAB `mcc` function to compile code containing a scope. You cannot open scope configuration dialogs if you have more than one compiled component in your application.

## Version History

Introduced in R2020a

**R2022b: Channel names support array of strings**

Starting in R2022b, you can specify the `ChannelNames` property of the `timescope` object as an array of strings.

```
ts = timescope(SampleRate=Fs, ChannelNames=["Input", "Lowpass Output"]);
```

### See Also

#### Topics

“Configure Time Scope MATLAB Object”

# dsp.TransferFunctionEstimator

**Package:** dsp

Estimate transfer function

## Description

The `dsp.TransferFunctionEstimator` System object computes the transfer function of a system, using the Welch's averaged periodogram method.

To implement the transfer function estimation object:

- 1 Create the `dsp.TransferFunctionEstimator` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
tfe = dsp.TransferFunctionEstimator
tfe = dsp.TransferFunctionEstimator(Name,Value)
```

### Description

`tfe = dsp.TransferFunctionEstimator` returns a transfer function estimator object, that computes the transfer function of real or complex signals. This System object uses the periodogram method and Welch's averaged, modified periodogram method.

`tfe = dsp.TransferFunctionEstimator(Name,Value)` returns a transfer function estimator object with each specified property set to the specified value. Unspecified properties have default values.

### Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

#### FFTLengthSource — Source of FFT length value

'Auto' (default) | 'Property'

Specify the source of the FFT length value as either 'Auto' or 'Property'. If you set this property to 'Auto', the transfer function estimator sets the FFT length to the input frame size. If you set this property to 'Property', then specify the number of FFT points using the FFTLength property.

**FFTLength — FFT Length**

128 (default) | positive integer

Specify the length of the FFT that the transfer function estimator uses to compute spectral estimates as a positive, integer scalar.

**Dependencies**

This property applies when you set the FFTLengthSource property to 'Property'.

Data Types: double

**Window — Window function**

'Hann' (default) | 'Rectangular' | 'Chebyshev' | 'Flat Top' | 'Hamming' | 'Kaiser'

Specify a window function for the transfer function estimator as one of 'Rectangular', 'Chebyshev', 'Flat Top', 'Hamming', 'Hann', or 'Kaiser'.

**SidelobeAttenuation — Side lobe attenuation of window**

60 (default) | positive scalar

Specify the side lobe attenuation of the window as a real, positive scalar, in decibels (dB).

**Dependencies**

This property applies when you set the Window property to 'Chebyshev' or 'Kaiser'.

Data Types: double

**FrequencyRange — Frequency range of the transfer function estimate**

'Twosided' (default) | 'onesided' | 'centered'

Specify the frequency range of the transfer function estimator as one of 'twosided', 'onesided', or 'centered'.

If you set the FrequencyRange to 'onesided', the transfer function estimator computes the one-sided transfer function of real input signals,  $x$  and  $y$ . If the FFT length,  $N_{FFT}$ , is even, the length of the transfer function estimate is  $N_{FFT}/2+1$  and is computed over the interval  $[0, SampleRate/2]$ . If  $N_{FFT}$  is odd, the length of the transfer function estimate is equal to  $(N_{FFT}+1)/2$ , and the interval is  $[0, SampleRate/2]$ .

If FrequencyRange is set to 'twosided', the transfer function estimator computes the two-sided transfer function of complex or real input signals,  $x$  and  $y$ . The length of the transfer function estimate is equal to  $N_{FFT}$  and is computed over  $[0, SampleRate]$ .

If you set the FrequencyRange to 'centered', the transfer function estimator computes the centered two-sided transfer function of complex or real input signals,  $x$  and  $y$ . The length of the transfer function estimate is equal to  $N_{FFT}$  and it is computed over  $[-SampleRate/2, SampleRate/2]$  for even lengths, and  $[-SampleRate/2, SampleRate/2]$  for odd lengths.

**AveragingMethod — Averaging method**

'Running' (default) | 'Exponential'

Specify the averaging method as 'Running' or 'Exponential'. In the running averaging method, the object computes an equally weighted average of a specified number of spectrum estimates defined by the `SpectralAverages` property. In the exponential method, the object computes the average over samples weighted by an exponentially decaying forgetting factor.

### **SpectralAverages — Number of spectral averages**

8 (default) | positive integer

Specify the number of spectral averages as a positive, integer scalar. The transfer function estimator computes the current estimate by averaging the last  $N$  estimates, where  $N$  is the number of spectral averages defined in the `SpectralAverages` property.

#### **Dependencies**

This property applies when you set `AveragingMethod` to 'Running'.

Data Types: double

### **ForgettingFactor — Forgetting factor**

0.9 (default) | scalar in the range (0,1]

Specify the exponential weighting forgetting factor as a scalar value greater than zero and smaller than or equal to one.

**Tunable:** Yes

#### **Dependencies**

This property applies when you set `AveragingMethod` to 'Exponential'.

Data Types: single | double

### **OutputCoherence — Magnitude squared coherence estimate**

false (default) | true

Specify `true` to compute and output the magnitude squared coherence estimate using Welch's averaged, modified periodogram method. The magnitude squared coherence estimate has values between 0 and 1 that indicate the correspondence at each frequency between two input signals. If you specify `false`, the magnitude squared coherence estimate is not computed.

## **Usage**

### **Syntax**

```
tfeEst = tfe(x,y)
[tfeEst,cxy] = tfe(x,y)
```

#### **Description**

`tfeEst = tfe(x,y)` computes the transfer function estimate, `tfeEst`, of the system with input  $x$  and output  $y$  using Welch's averaged periodogram method.

`[tfeEst,cxy] = tfe(x,y)` also computes the magnitude squared coherence estimate, `cxy`, of the system.

### Input Arguments

#### **x — First data input**

vector | matrix

First data input, specified as a vector or a matrix. *x* and *y* must have the same size and data type.

Data Types: single | double

#### **y — Second data input**

vector | matrix

Second data input, specified as a vector or a matrix. *x* and *y* must have the same size and data type.

Data Types: single | double

### Output Arguments

#### **tfeEst — Transfer function estimate**

vector | matrix

Transfer function estimate of the system for which *x* and *y* are the input and output signals, respectively.

The estimate, *tfeEst*, is equal to  $p_{xy}/p_{xx}$ , where  $p_{xy}$  is the cross-power spectral density of *x* and *y*, and  $p_{xx}$  is the power spectral density of *x*.

The transfer function estimate has the same size and data type as the input.

Data Types: single | double

#### **cxy — Coherence estimate**

vector | matrix

Magnitude squared coherence estimate of the system, returned as a vector or a matrix.

The coherence estimate, *cxy*, is equal to  $(\text{abs}(p_{xy}) . ^2) ./ (p_{xx} . *p_{yy})$ , where  $p_{xy}$  is the cross power spectral density of *x* and *y*,  $p_{xx}$  is the power spectral density of *x*, and  $p_{yy}$  is the power spectral density of *y*. For coherence to be estimated, the `OutputCoherence` property must be set to `true`.

The coherence estimate has the same size and data type as the input.

Data Types: single | double

### Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named *obj*, use this syntax:

```
release(obj)
```

### Specific to dsp.TransferFunctionEstimator

`getFrequencyVector`    Vector of frequencies at which estimation is done  
`getRBW`                Resolution bandwidth of spectrum



## Common to All System Objects

|         |  |
|---------|--|
| step    | Run System object algorithm  |
| release | Release resources and allow changes to System object property values and input characteristics |
| reset   | Reset internal states of System object   |

## Examples

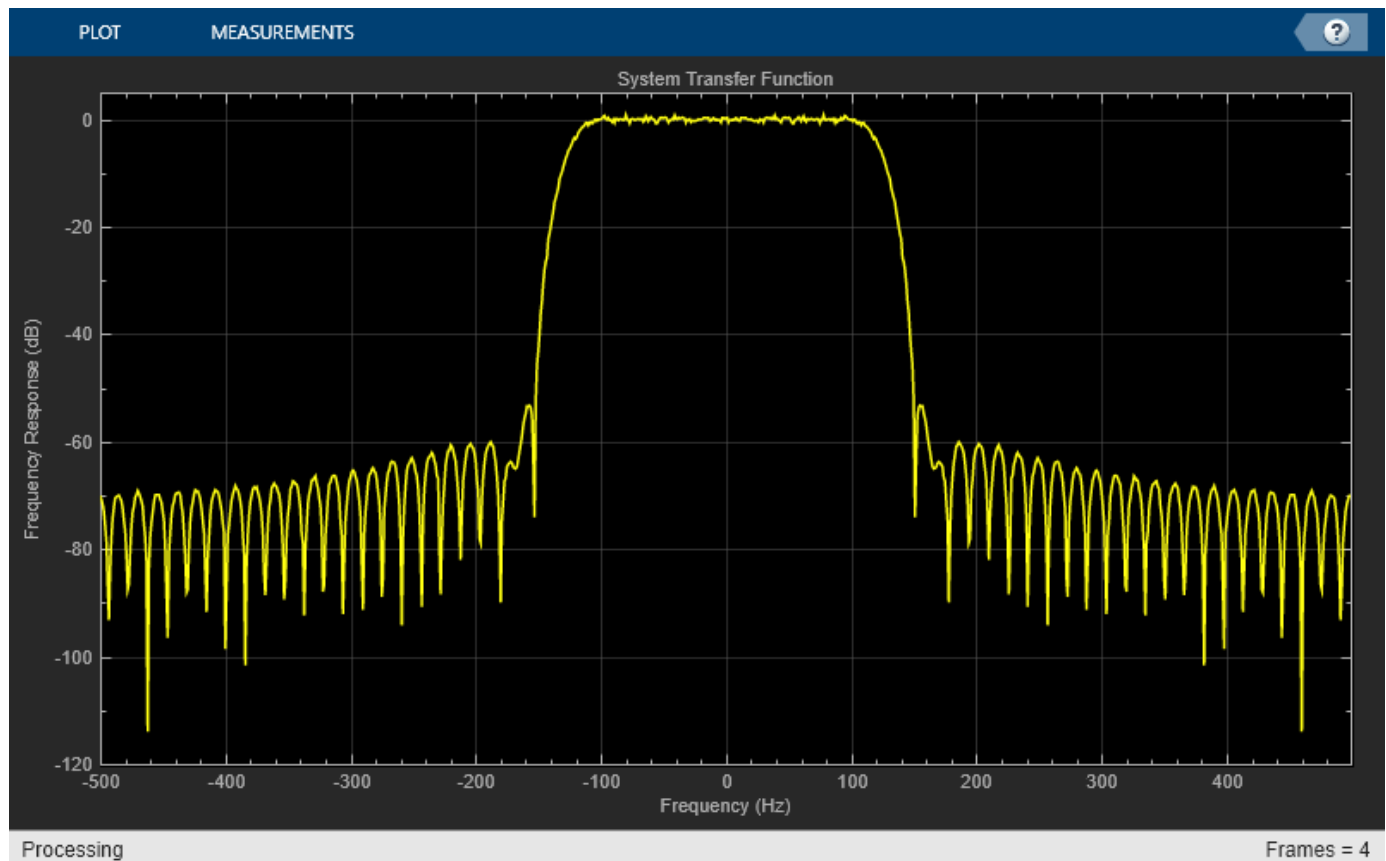
### Estimate Transfer Function of a System Represented by an Order-64 FIR Filter

Generate a sine wave. Use the dsp.TransferFunctionEstimator System object™ to estimate the system transfer function and the dsp.ArrayPlot System object to display it.

```
sin = dsp.SineWave('Frequency',100,'SampleRate',1000);
sin.SamplesPerFrame = 1000;
tfe = dsp.TransferFunctionEstimator('FrequencyRange','centered');
aplot = dsp.ArrayPlot('PlotType','Line','XOffset',-500,'YLimits',...
    [-120 5],'YLabel','Frequency Response (dB)',...
    'XLabel','Frequency (Hz)',...
    'Title','System Transfer Function');
```

Create an FIR Filter System object of order 64 and (normalized) cutoff frequency of 1/4. Add random noise to the sine wave. Step through the System objects to obtain the data streams, and plot the log of the magnitude of the transfer function.

```
firFilt = dsp.FIRFilter('Numerator',fir1(64,1/4));
for ii = 1:100
x = sin() + 0.05*randn(1000,1);
y = firFilt(x);
Txy = tfe(x,y);
aplot(20*log10(abs(Txy)))
end
```



## Algorithms

### Welch's Method of Averaged Modified Periodograms

Give two signal inputs,  $x$  and  $y$ :

- 1 Multiply the inputs by the window and scale the result by the window power.
- 2 Take FFT of the signals,  $X$  and  $Y$ .
- 3 Compute the current power spectral density estimates,  $P_{xx}$ ,  $P_{yy}$ , and the current cross power spectral density estimate,  $P_{yx}$ , by taking the moving average of last  $N$  number of  $Z_1$ ,  $Z_2$ , and  $Z_3$  vectors, respectively:
  - $Z_1 = X \cdot \text{conj}(X)$
  - $Z_2 = Y \cdot \text{conj}(Y)$
  - $Z_3 = Y \cdot \text{conj}(X)$

For details on the moving average methods, see "Averaging Method" on page 4-1141.

The transfer function estimate is calculated by dividing  $P_{yx}$  by  $P_{xx}$ .

The magnitude squared coherence,  $C_{xy}$ , is defined by the following equation:

$$C_{xy} = \frac{(\text{abs}(P_{xy}) \cdot \wedge 2)}{(P_{xx} \cdot * P_{yy})}$$

## Version History

Introduced in R2013b

## References

- [1] Hayes, Monson H. *Statistical Digital Signal Processing and Modeling*. Hoboken, NJ: John Wiley & Sons, 1996
- [2] Kay, Steven M. *Modern Spectral Estimation: Theory and Application*. Englewood Cliffs, NJ: Prentice Hall, 1999
- [3] Stoica, Petre and Randolph L. Moses. *Spectral Analysis of Signals*. Englewood Cliffs, NJ: Prentice Hall, 2005
- [4] Welch, P. D. ``The use of fast Fourier transforms for the estimation of power spectra: A method based on time averaging over short modified periodograms," *IEEE Transactions on Audio and Electroacoustics*, Vol. 15, pp. 70-73, 1967.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- "System Objects in MATLAB Code Generation" (MATLAB Coder)

## See Also

### Objects

spectrumAnalyzer | dsp.SpectrumEstimator | dsp.CrossSpectrumEstimator

## **dsp.TransitionMetrics**

**Package:** dsp

(Removed) Transition metrics of bilevel waveforms

---

**Note** `dsp.TransitionMetrics` has been removed. Use functions from “Pulse and Transition Metrics” instead. Functions in this set include `falltime`, `overshoot`, `risetime`, `settlingtime`, `slewrates`, and `undershoot` among others.

---

### **Description**

The `dsp.TransitionMetrics` object extracts information such as duration, slew rate, and reference-level crossings for each transition found in the bilevel waveform. The `dsp.TransitionMetrics` object can additionally return preshoot, postshoot and settling metrics for the regions immediately before and after each transition.

To obtain transition metrics for a bilevel waveform:

- 1 Create the `dsp.TransitionMetrics` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

### **Creation**

#### **Syntax**

```
tm = dsp.TransitionMetrics  
tm = dsp.TransitionMetrics(Name,Value)
```

#### **Description**

`tm = dsp.TransitionMetrics` creates a transition metrics System object, `tm`. The object computes the rise time, fall time, and width of a pulse. `TransitionMetrics` additionally computes cycle metrics such as pulse separations, periods, and duty cycles.

`tm = dsp.TransitionMetrics(Name,Value)` returns a `TransitionMetrics` System object, `tm`, with each specified property set to the specified value.

#### **Properties**

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

### **MaximumRecordLength — Maximum samples to preserve**

1000 (default) | positive integer

Maximum samples to preserve between calls to the algorithm. This property requires a positive integer that specifies the maximum number of samples to save between calls to the algorithm. When the number of samples to be saved exceeds this length, the oldest excess samples are discarded.

#### **Dependencies**

This property applies when `RunningMetrics` is `true` and is tunable.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **PercentReferenceLevels — Reference levels**

[10 50 90] (default) | three-element row vector

Lower-, middle-, and upper-percent reference levels. This property contains a three-element numeric row vector of the lower-, middle-, and upper-percent reference levels. These reference levels are used as an offset between the low and high states of the waveform when computing the duration of each transition.

Data Types: `double`

### **PercentStateLevelTolerance — Tolerance of state level**

2 (default) | positive scalar

Tolerance of the state level (in percent). This property requires a scalar that specifies the maximum deviation from either the low or high state before it is considered to be outside that state. The tolerance is expressed as a percentage of the waveform amplitude.

Data Types: `double`

### **PostshootOutputPort — Enable posttransition aberration metrics**

`false` (default) | `true`

Enable posttransition aberration metrics. If this property is set to `true`, overshoot and undershoot metrics are reported for a region defined immediately after each transition. The posttransition aberration region is defined as the waveform interval that begins at the end of each transition and whose duration is the value of `PostshootSeekFactor` times the computed transition duration. If a complete subsequent transition is detected before the interval is over, the region is truncated at the start of the subsequent transition. The metrics are computed for each transition that has a complete posttransition aberration region.

### **PostshootSeekFactor — Postshoot seek factor**

3 (default) | positive scalar

Corresponds to the duration of time to search for the overshoot and undershoot metrics immediately following each transition. The duration is expressed as a factor of the duration of the transition.

**Tunable:** Yes

**Dependencies**

This property is enabled only when the `PostshootOutputPort` property is set to `true` and is tunable.

Data Types: `double`

**PreshootOutputPort — Enable pretransition aberration metrics**

`false` (default) | `true`

Enable pretransition aberration metrics. If the `PreshootOutputPort` property is set to `true`, overshoot and undershoot metrics are reported for a region defined immediately before each transition. The pretransition aberration region is defined as the waveform interval that ends at the start of each transition and whose duration is `PreshootSeekFactor` times the computed transition duration.

**PreshootSeekFactor — Preshoot seek factor**

3 (default) | positive scalar

Corresponds to the duration of time to search for the overshoot and undershoot metrics immediately preceding each transition. The duration is expressed as a factor of the duration of the transition.

**Tunable:** Yes

**Dependencies**

This property is enabled only when the `PreshootOutputPort` property is set to `true` and is tunable.

Data Types: `double`

**RunningMetrics — Enable metrics over calls**

`false` (default) | `true`

Enable metrics over all calls to the algorithm. If `RunningMetrics` is set to `false`, metrics are computed for each call to the algorithm independently. If `RunningMetrics` is set to `true`, metrics are computed across subsequent calls to the algorithm. If there are not enough samples to compute metrics associated with the last transition, posttransition aberration region, or settling seek duration in the current record, the object defers reporting all transition, aberration, and settling metrics associated with the last transition until a subsequent call to the algorithm is made with enough data to compute all enabled metrics for that transition.

**SampleRate — Sampling rate**

1 (default) | positive scalar

Sampling rate of uniformly sampled signal. Specify the sample rate in hertz as a positive scalar. This property is used to construct the internal time values that correspond to the input sample values. Time values start with zero.

**Dependencies**

This property applies when the `TimeInputPort` property is set to `false`.

Data Types: `double`

**SettlingOutputPort — Enable settling metrics**

`false` (default) | `true`

Enable settling metrics. If `SettlingOutputPort` is set to `true`, settling metrics are reported for each transition. The region used to compute the settling metrics starts at the midcrossing and lasts until the `SettlingSeekDuration` has elapsed. If an intervening transition occurs, or the signal has not settled within the `PercentStateLevelTolerance` of the final level, `NaN` is returned for each metric. If there are not enough samples after the last transition to complete the `SettlingSeekDuration`, no metrics are reported for the last transition. The metrics are reported for the transition the next time the algorithm is called if the `RunningMetrics` property is set to `true`.

### **SettlingSeekDuration — Settling seek duration**

`0.02` (default) | positive scalar

Duration of time over which to search for settling. This property is a scalar that specifies the amount of time to inspect from the mid-reference level crossing (in seconds). If the transition has not yet settled, or a subsequent complete transition is detected within this duration, the `TransitionMetrics` object reports `NaN` for all settling metrics.

**Tunable:** Yes

#### **Dependencies**

This property applies only when you set the `SettlingOutputPort` property to `true`.

Data Types: `double`

### **StateLevels — State levels**

`[0 2.3]` (default) | two-element row vector

Low- and high-state levels. This property is a two-element numeric row vector that contains the low and high state levels respectively. These state levels correspond to the nominal logic low and high levels of the pulse waveform.

**Tunable:** Yes

Data Types: `double`

### **StateLevelsSource — State level source**

`'Property'` (default) | `'Auto'`

Auto or manual state level computation. If the `StateLevelsSource` property is set to `'Auto'`, the first record sent to the algorithm uses the `dsp.StateLevels` object with the default settings to determine the state levels of the incoming waveform. If this property is set to `'Property'`, the object uses the values the user specifies in the `StateLevels` property.

### **TimeInputPort — Time input**

`false` (default) | `true`

Add input to specify sample instants. Set `TimeInputPort` to `true` to enable an additional real input column vector to the algorithm to specify the sample instants that correspond to the sample values. If this property is `false`, the sample instants are built internally. The sample instants start at zero and increment by the reciprocal of the `SampleRate` property for subsequent samples. The sample instants continue to increment if the `RunningMetrics` property is set to `true` and no intervening calls to the `reset` or `release` methods are encountered.

## Usage

### Syntax

```
pulse = tm(x)
[pulse,cycle] = tm(x)
[pulse,transition] = tm(x)
[pulse,preshoot] = tm(x)
[pulse,postshoot] = tm(x)
[pulse,settling] = tm(x)
[pulse,cycle,transition,preshoot,postshoot,settling] = tm(x)
[ ___ ] = tm(x,T)
```

### Description

`pulse = tm(x)` returns a structure array, `pulse`, whose fields contain real-valued column vectors. The number of rows of each field corresponds to the number of complete pulses found in the real-valued column vector input, `x`.

`[pulse,cycle] = tm(x)` returns a structure array, `cycle`, when you set the `CycleOutputPort` property to `true`. The fields of `cycle` contain real-valued column vectors. The number of rows of each field corresponds to the number of complete pulse periods found in the real-valued column vector input, `x`.

`[pulse,transition] = tm(x)` returns a structure array, `transition`, when you set the `TransitionOutputPort` property to `true`. The fields of `transition` contain real-valued matrices with two columns, which correspond to the metrics of the first and second transitions. The number of rows corresponds to the number of pulses found in the input waveform.

`[pulse,preshoot] = tm(x)` returns a structure array, `preshoot`, when you set the `PreshootOutputPort` property to `true`. The fields of `preshoot` contain real-valued two-column matrices whose row length corresponds to the number of transitions found in the input waveform. The field names are identical to those of the `postshoot` structure.

`[pulse,postshoot] = tm(x)` returns a structure array, `postshoot`, when you set the `PostshootOutputPort` property to `true`. The fields of `postshoot` contain real-valued two-column matrices whose row length corresponds to the number of transitions found in the input waveform.

`[pulse,settling] = tm(x)` returns a structure, `settling`, when you set the `SettlingOutputPort` property to `true`. The fields of `settling` correspond to the settling metrics for each transition. Each field is a column vector whose elements correspond to the individual settling durations, levels, and instants.

`[pulse,cycle,transition,preshoot,postshoot,settling] = tm(x)` which returns the `pulse`, `cycle`, `transition`, `preshoot`, `postshoot`, and `settling` structure arrays when the `CycleOutputPort`, `PreshootOutputPort`, `PostshootPort`, and `SettlingOutputPort` properties are `true`. You may enable or disable any combination of output ports. However, the output arguments are defined in the order shown here.

`[ ___ ] = tm(x,T)` performs the above metrics with respect to a sampled signal, whose sample values, `x`, and sample instants, `T`, are real-valued column vectors of the same length. The additional input `T` applies only when you set the `TimeInputPort` property to `true`.



## Input Arguments

### **x** — Input signal

column vector

Input signal, specified as a real-valued column vector.

Data Types: `double`

### **T** — Sampling instants

column vector

Sampling instants, specified as a real-valued column vector. Set `TimeInputPort` to `true` to enable an additional real input column vector to the object algorithm to specify the sample instants that correspond to the sample values. If this property is `false`, the sample instants are built internally. The sample instants start at zero and increment by the reciprocal of the `SampleRate` property for subsequent samples. The sample instants continue to increment if the `RunningMetrics` property is set to `true` and no intervening calls to the `reset` or `release` methods are encountered.

### Dependencies

This input is applicable when you set the `TimeInputPort` property to `true`.

Data Types: `double`

## Output Arguments

### **pulse** — Complete pulses

structure

Complete pulses, returned as a structure whose fields contain real-valued column vectors. The number of rows of each field corresponds to the number of complete pulses found in the real-valued column vector input, `x`. Each pulse starts with a transition of the polarity specified by the `Polarity` property and ends with a transition of the opposite polarity.

The `pulse` output contains the following fields:

- `PositiveCross` — Instants where the positive-going transitions cross the mid-reference level of each pulse
- `NegativeCross` — Instants where the negative-going transitions cross the mid-reference level of each pulse
- `Width` — Absolute difference between `PositiveCross` and `NegativeCross` of each pulse
- `RiseTime` — Duration between the linearly-interpolated instants when the positive-going (rising) transition of each pulse crosses the lower- and upper-reference levels
- `FallTime` — Duration between the linearly-interpolated instants when the negative-going (falling) transition of each pulse crosses the upper- and lower-reference levels

Data Types: `struct`

### **cycle** — Complete pulse periods

structure

Complete pulse periods, returned when you set the `CycleOutputPort` property to `true`. The pulse periods are returned as a structure whose fields contain real-valued column vectors. The number of rows of each field corresponds to the number of complete pulse periods found in the real-valued

column vector input, `x`. You need at least three consecutive alternating polarity transitions that start and end with the same polarity as the value of the `Polarity` property if you want to compute cycle metrics. If the last transition found in the input `x` does not match the polarity of the `Polarity` property, the pulse separation, period, frequency, and duty cycle are not reported for the last pulse. If the `RunningMetrics` property is set to `true` when this occurs, all pulse, cycle, transition, preshoot, postshoot, and settling metrics associated with the last pulse are deferred until a subsequent call to the algorithm detects the next transition.

The `cycle` output contains the following fields:

- `Period` — Duration between the first transition of the current pulse and the first transition of the next pulse.
- `Frequency` — Reciprocal of the period.
- `Separation` — Durations between the mid-reference level crossings of the first and second transitions of each pulse.
- `Width` — Durations between the mid-reference level crossings of the first and second transitions of each pulse. This is equivalent to the `width` parameter of the `pulse` structure.
- `DutyCycle` — Ratio of the width to the period for each pulse.

Data Types: `struct`

### **transition — Transition metrics**

`structure`

Transition metrics, returned as a structure array, when you set the `TransitionOutputPort` property to `true`. The fields of `transition` contain real-valued matrices with two columns which correspond to the metrics of the first and second transitions. The number of rows corresponds to the number of pulses found in the input waveform.

The `transition` output contains the following fields:

- `Duration` — Amount of time between the interpolated instants where the transition crosses the lower- and upper-reference levels
- `SlewRate` — Ratio of absolute difference between the upper and lower reference levels to the transition duration
- `MiddleCross` — Linearly interpolated instant where the transition first crosses the mid-reference level
- `LowerCross` — Linearly interpolated instant where the signal crosses the lower-reference level
- `UpperCross` — Linearly interpolated instant where the signal crosses the upper-reference level

Data Types: `struct`

### **preshoot — Preshoot metrics**

`structure`

Preshoot metrics, returned as a structure array, when you set the `PreshootOutputPort` property to `true`. The fields of `preshoot` contain real-valued two-column matrices whose row length corresponds to the number of transitions found in the input waveform.

The `preshoot` output contains the following fields:

- **Overshoot** — Overshoot of the region of interest expressed as a percentage of the waveform amplitude
- **Undershoot** — Undershoot of the region of interest expressed as a percentage of the waveform amplitude
- **OvershootLevel** — Level of the overshoot
- **UndershootLevel** — Level of the undershoot
- **OvershootInstant** — Instant that corresponds to the overshoot
- **UndershootInstant** — Instant that corresponds to the undershoot

Data Types: `struct`

### **postshoot — Postshoot metrics**

structure

Postshoot metrics, returned as a structure array, when you set the `PostshootOutputPort` property to `true`. The fields of `postshoot` contain real-valued two-column matrices whose row length corresponds to the number of transitions found in the input waveform.

The `postshoot` output contains the following fields:

- **Overshoot** — Overshoot of the region of interest expressed as a percentage of the waveform amplitude
- **Undershoot** — Undershoot of the region of interest expressed as a percentage of the waveform amplitude
- **OvershootLevel** — Level of the overshoot
- **UndershootLevel** — Level of the undershoot
- **OvershootInstant** — Instant that corresponds to the overshoot
- **UndershootInstant** — Instant that corresponds to the undershoot

Data Types: `struct`

### **settling — Settling metrics**

structure

Settling metrics for each transition, returned as a structure array, when you set the `SettlingOutputPort` property to `true`. The fields of `settling` correspond to the settling metrics for each transition. Each field is a column vector whose elements correspond to the individual settling durations, levels, and instants.

The `settling` output contains the following fields:

- **Duration** — Amount of time from when the signal crosses the mid-reference level to the time where the signal enters and remains within the specified `PercentStateLevelTolerance` of the waveform amplitude over the specified settling seek duration
- **Instant** — Instant in time where the signal enters and remains within the specified tolerance.
- **Level** — Level of the waveform where it enters and remains within the specified tolerance.

Data Types: `struct`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

### Specific to `dsp.TransitionMetrics`

`plot` (To be removed) Plot pulse signal and metrics

### Common to All System Objects

`step` Run System object algorithm

`release` Release resources and allow changes to System object property values and input characteristics

`reset` Reset internal states of System object

## Examples

### Transition and Preshoot Information of a 2.3 V Step Waveform

**Note:** If you are using R2016a or an earlier release, replace each call to the object with the equivalent step syntax. For example, `obj(x)` becomes `step(obj,x)`.

Compute transition and preshoot information of a 2.3 V step waveform sampled at 4 MHz. Load the data.

```
load('transitionex.mat','x');
```

Construct the `dsp.TransitionMetrics` object. Set the `SampleRate` property to 4 MHz and the `StateLevelsSource` property to 'Auto' to estimate the state levels from the data. Set the `PreshootOutputPort` property to `true` to output pretransition aberration metrics when you call the object.

```
tm = dsp.TransitionMetrics('SampleRate',4e6, ...  
                           'StateLevelsSource','Auto', ...  
                           'PreshootOutputPort',true)
```

```
tm =  
dsp.TransitionMetrics with properties:
```

```
    StateLevelsSource: 'Auto'  
        StateLevels: [0 2.3000]  
PercentStateLevelTolerance: 2  
    PercentReferenceLevels: [10 50 90]  
        RunningMetrics: false  
        TimeInputPort: false  
        SampleRate: 4000000  
    PreshootOutputPort: true  
    PreshootSeekFactor: 3  
    PostshootOutputPort: false  
    SettlingOutputPort: false
```

Call the object to compute the transition and preshoot information. Plot the result.

```
[transition,preshoot] = tm(x)

transition = struct with fields:
    Duration: 1.7800e-07
    Polarity: 1
    SlewRate: 1.0310e+07
    MiddleCross: 5.1250e-06
    LowerCross: 5.0360e-06
    UpperCross: 5.2140e-06

preshoot = struct with fields:
    Overshoot: 4.8050
    Undershoot: 6.1798
    OvershootLevel: 0.1020
    UndershootLevel: -0.1500
    OvershootInstant: 4.7500e-06
    UndershootInstant: 5.0000e-06

plot(tm)
```

## Version History

### Introduced in R2012a

#### **R2021b: dsp.TransitionMetrics System object will be removed**

*Warns starting in R2021b*

dsp.TransitionMetrics System object will be removed in a future release. Use functions from “Pulse and Transition Metrics” instead. Functions in this set include `falltime`, `overshoot`, `risetime`, `settlingtime`, `slewrate`, and `undershoot` among others.

#### **R2023a: dsp.TransitionMetrics System object has been removed**

*Errors starting in R2023a*

The dsp.TransitionMetrics System object has been removed. Use functions from “Pulse and Transition Metrics” instead. This set includes functions such as `falltime`, `overshoot`, `risetime`, `settlingtime`, `slewrate`, and `undershoot` among others.

## References

[1] *IEEE Standard on Transitions, Pulses, and Related Waveforms*, IEEE Standard 181, 2003.

## See Also

### Functions

`falltime` | `overshoot` | `risetime` | `settlingtime` | `slewrate` | `undershoot`

## dsp.UDPReceiver

**Package:** dsp

Receive UDP packets from the network

### Description

The `dsp.UDPReceiver` System object receives UDP packets over a UDP network from an IP address specified in the `RemoteIPAddress` property. The object then saves the data to its internal buffer. The amount of data (number of elements) received in each UDP packet can vary. The maximum number of bytes the object can receive without losing data is set by the `ReceiveBufferSize` property. The `MaximumMessageLength` property specifies the maximum number of samples each data packet can contain. The `LocalIPPort` on which the object receives the data is tunable in generated code but not tunable during simulation. For an example, see “Tune the UDP Port Number in MATLAB” on page 4-1227.

Due to the nature of the UDP transmission protocol, the receiver is not guaranteed to receive all the data packets that you send using the `dsp.UDPSender` object.

To receive UDP packets from the network:

- 1 Create the `dsp.UDPReceiver` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

### Creation

#### Syntax

```
udpr = dsp.UDPReceiver  
udpr = dsp.UDPReceiver(Name,Value)
```

#### Description

`udpr = dsp.UDPReceiver` returns a UDP receiver object that receives UDP packets from a specified port.

`udpr = dsp.UDPReceiver(Name,Value)` returns a UDP receiver object with each specified property set to the specified value. Enclose each property name in single quotes.

#### Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

### **LocalIPPort — Local port on which to receive data**

25000 (default) | [1, 65535]

Port on which to receive the data, specified as a scalar in the range [1, 65535]. This property is tunable in generated code but not tunable during simulation.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### **RemoteIPAddress — Address from which to accept data**

'0.0.0.0' (default) | character vector containing a valid IP address | string scalar

Address from which to accept data, specified as a character vector or a string scalar containing a valid IP address. Entering a specific IP address blocks UDP packets from other addresses. The default, '0.0.0.0', indicates that the data can be accepted from any remote IP address.

Data Types: char

### **ReceiveBufferSize — Size of internal buffer**

8192 bytes (default) | [1, 67108864]

Size of the internal buffer that receives UDP packets, specified in bytes as a scalar in the range [1, 67108864]. If the number of bytes received exceeds this value, the buffer overflows and the contents are truncated.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### **MaximumMessageLength — Maximum size of output message**

255 (default) | [1, 65507]

Maximum length of the output message, specified in samples as a positive scalar in the range [1, 65507]. Set this property to a value equal to or greater than the data size of the UDP packet. If you receive more samples than specified in this property, the excess data is truncated.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### **MessageDataType — Data type of the message**

'uint8' (default) | 'double' | 'single' | 'int8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'logical'

Data type of the vector elements in the message output, specified as a MATLAB built-in data type.

Match the data type with the data input used to create the UDP packets.

Data Types: char

### **IsMessageComplex — Complexity of message**

false (default) | true

Complexity of the message, specified as either true or false.

Set this property to true if the received message is complex. Set the property to false if the received message is real.

Data Types: logical

## Usage

### Syntax

```
dataR = udpr()
```

### Description

`dataR = udpr()` receives one UDP packet from the network.

### Output Arguments

#### **dataR — Data received**

scalar | vector

Data received from the network, returned as one packet. The “MaximumMessageLength” on page 4-0 property specifies the maximum number of bytes each data packet can contain. Length of the data received is the number of bytes received from the network.

The data is received as complex data if the `IsMessageComplex` property is set to `true`.

If the packet gets dropped during transmission and the receiver does not receive the packet, the `dataR` output is empty.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `logical`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Common to All System Objects

|                      |  |
|----------------------|--|
| <code>step</code>    | Run System object algorithm  |
| <code>release</code> | Release resources and allow changes to System object property values and input characteristics |
| <code>reset</code>   | Reset internal states of System object   |

## Examples

### Byte Transmission Using UDP

Send and receive UDP packets using the `dsp.UDPSender` and `dsp.UDPReceiver` System objects. Calculate the number of bytes successfully transmitted.

Set the `RemoteIPPort` of UDP sender and the `LocalIPPort` of the UDP receiver to 31000. Set the length of the data vector to 128 samples, which is less than the value of the `MaximumMessageLength` property of the receiver. To prevent the loss of packets, call the `setup` method on the receiver object before the first call to the object algorithm.

```
udpr = dsp.UDPReceiver('LocalIPPort',31000);  
udps = dsp.UDPSender('RemoteIPPort',31000);
```



```
setup(udpr);
```

```
bytesSent = 0;
bytesReceived = 0;
dataLength = 128;
```

In each loop of iteration, send and receive a packet of data. At the end of the loop, use the `fprintf` function to print the number of bytes sent by the sender and the number of bytes received by the receiver.

```
for k = 1:20
    dataSent = uint8(255*rand(1,dataLength));
    bytesSent = bytesSent + dataLength;
    udps(dataSent);
    dataReceived = [];
    while (isempty(dataReceived))
        dataReceived = udpr();
    end
    bytesReceived = bytesReceived + length(dataReceived);
end
```

```
release(udps);
release(udpr);
```

```
fprintf('Bytes sent:      %d\n', bytesSent);
```

```
Bytes sent:      2560
```

```
fprintf('Bytes received: %d\n', bytesReceived);
```

```
Bytes received: 2560
```

### Tune the UDP Port Number in MATLAB

The local IP port number of the `dsp.UDPReceiver` object and the remote IP port number of the `dsp.UDPSender` object are tunable in the generated code. Generate a MEX file from the receiver function which contains the algorithm to receive sine wave data over a UDP network. Change the remote IP port number of the UDP receiver without regenerating the MEX file. Verify the number of bytes sent and received over the network.

**Note:** This example runs only in R2017a or later.

The input to the receiver function is the local IP port number of the `dsp.UDPReceiver` System object™. The output of this function is the number of bytes received from the UDP network.

```
type receiver
```

```
function [bytesReceived] = receiver(portnumber)
```

```
persistent udpRx
```

```
if isempty(udpRx)
    udpRx = dsp.UDPReceiver('MessageDataType','double');
```

```
end
```

```
udpRx.LocalIPPort = portnumber;  
dataReceived = udpRx();  
bytesReceived = length(dataReceived);
```

The `dsp.UDPSender` object with `remoteIPPort` number set to 65000 sends the data over the UDP network. The `dsp.UDPReceiver` object with `LocalIPPort` number set to 65000 receives the data from the UDP network. The data is a sine wave containing 250 samples per frame.

```
portnumber = 65000;  
udpSend = dsp.UDPSender('RemoteIPPort',portnumber);  
sine = dsp.SineWave('SamplesPerFrame',250);
```

```
bytesSent = 0;  
bytesReceived = 0;  
dataLength = 250;
```

```
for i = 1:10  
dataSent = sine();  
bytesSent = bytesSent + dataLength;  
udpSend(dataSent);  
bytesReceived = bytesReceived + receiver(portnumber);  
end  
fprintf('Number of bytes sent: %d', bytesSent);
```

```
Number of bytes sent: 2500
```

```
fprintf('Number of bytes received: %d', bytesReceived);
```

```
Number of bytes received: 2250
```

The data is sent and received successfully over the UDP network. The initial data is dropped due to overhead.

Generate a MEX file from the `receiver.m` function.

```
codegen receiver -args {65000}
```

```
Code generation successful.
```

Release the sender and change the `RemotePort` number to 25000. The `LocalIPPort` number of the receiver continues to be 65000. Since the port numbers are different, the data is not transmitted successfully.

```
release(udpSend)  
portnumberTwo = 25000;  
udpSend.RemoteIPPort = portnumberTwo;  
bytesReceived = 0;  
bytesSent = 0;  
for i = 1:10  
dataSent = sine();  
bytesSent = bytesSent + dataLength;  
udpSend(dataSent);  
bytesReceived = bytesReceived + receiver_mex(portnumber);  
end  
fprintf('Number of bytes sent: %d', bytesSent);
```

```
Number of bytes sent: 2500
```

```
fprintf('Number of bytes received: %d', bytesReceived);
```

```
Number of bytes received: 0
```

Clear the MEX file and change the local IP port number of the receiver to 25000. Clearing the MEX enables the receiver port number to change without having to regenerate the MEX. The port numbers of the sender and receiver match. Verify if the data is transmitted successfully.

```
clear mex %#ok
bytesReceived = 0;
bytesSent = 0;
for i = 1:10
    dataSent = sine();
    bytesSent = bytesSent + dataLength;
    udpSend(dataSent);
    bytesReceived = bytesReceived + receiver_mex(portnumberTwo);
end
fprintf('Number of bytes sent: %d', bytesSent);
```

```
Number of bytes sent: 2500
```

```
fprintf('Number of bytes received: %d', bytesReceived);
```

```
Number of bytes received: 2250
```

The data is transmitted successfully over the UDP network. The initial data is dropped due to overhead.

### Transmit Complex Data over UDP Network

Compute the STFT of a sine wave and transmit the complex STFT data over a UDP network. At the receiver side, compute the ISTFT of the received data. Visualize the data sent and the data received using a time scope.

The `dsp.UDPSender` object can send complex data. In order to enable the `dsp.UDPReceiver` object to receive complex data, set the `IsMessageComplex` property to `true`.

```
udps = dsp.UDPSender('RemoteIPPort',31000);
udpr = dsp.UDPReceiver('LocalIPPort',31000,...
    'IsMessageComplex',true,...
    'MessageDataType','double');
```

```
setup(udpr);
```

```
bytesSent = 0;
bytesReceived = 0;
dataLength = 128;
```

Initialize the `dsp.STFT` and `dsp.ISTFT` System objects with a periodic hann window of length 120 samples and an overlap length of 60 samples. Set the FFT length to 128.

```
winLen = 120;
overlapLen = 60;
```

```
frameLen = winLen-overlapLen;
```

```
stf = dsp.STFT(...  
    'Window',hann(winLen,'periodic'),...  
    'OverlapLength',overlapLen,'FFTLength',128);  
istf = dsp.ISTFT(...  
    'Window',hann(winLen,'periodic'),...  
    'OverlapLength',overlapLen,...  
    'WeightedOverlapAdd',0);
```

The input is a sinusoidal signal with a frequency of 100 Hz, a sample rate of 1000 Hz, and with 60 samples per each signal frame.

```
sine = dsp.SineWave(...  
    'SamplesPerFrame',winLen-overlapLen,...  
    'Frequency',100);
```

Initialize a timescope object with a sample rate of 1000 Hz and a time span of 0.09. The Delay object corrects the overlap length while comparing the input with the reconstructed output signal.

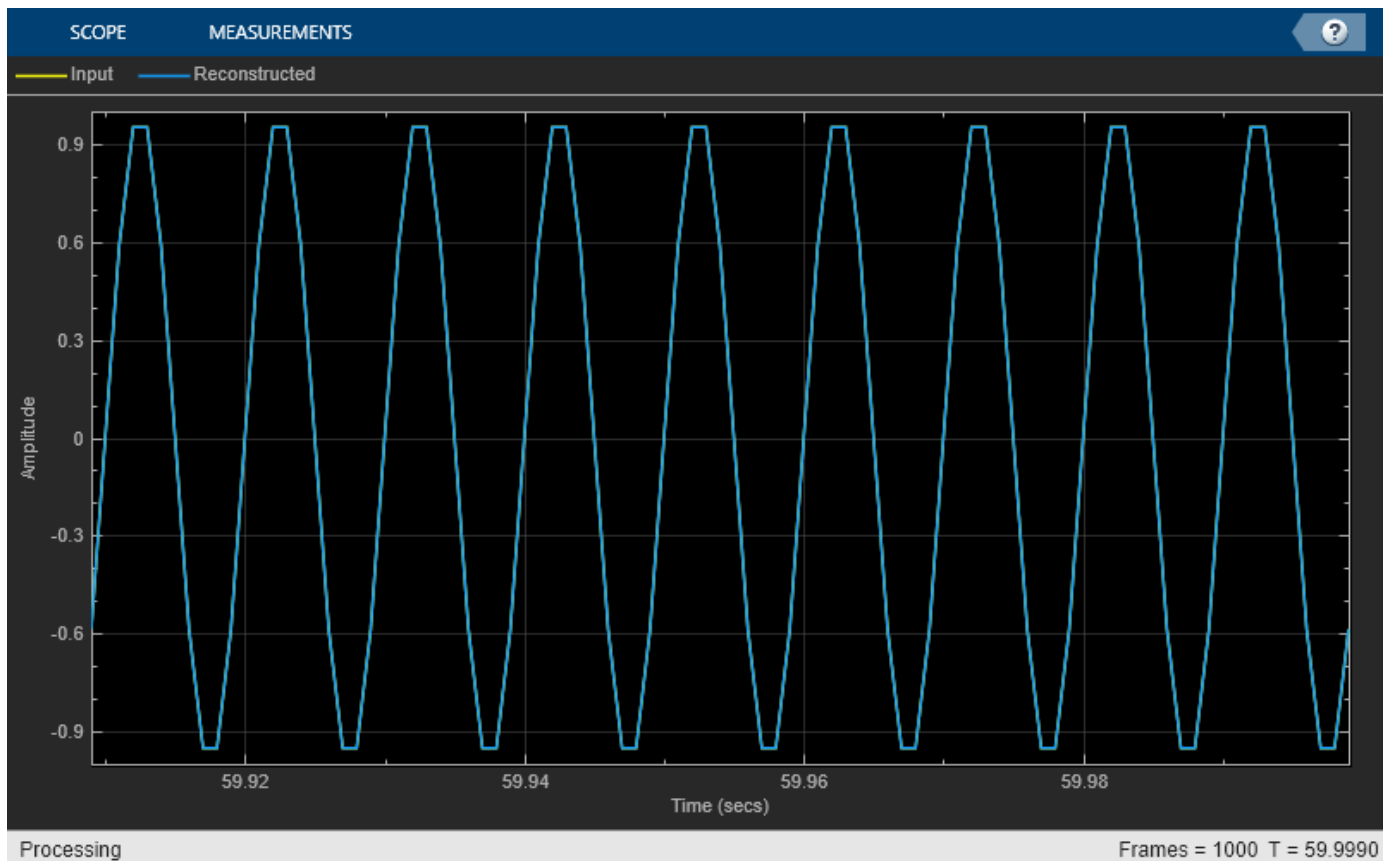
```
ts = timescope('SampleRate',1000,...  
    'ShowLegend',true,...  
    'YLimits',[-1 1],...  
    'TimeSpanSource','Property',...  
    'TimeSpan',.09,...  
    'ChannelNames',{'Input','Reconstructed'});  
dly = dsp.Delay('Length',overlapLen);
```

Transmit complex STFT data of the sine wave over the UDP network. Due to the undependable nature of the UDP protocol, the data packets can get lost during transmission. The receiver is not guaranteed to receive all the data that you send.

Compute the ISTFT of the received data. Compare the input  $x$  to the reconstructed output  $y$ . Due to the latency introduced by the objects, the reconstructed output is shifted in time compared to the input. Therefore, to compare, take the norm of the difference between the reconstructed output  $y$  and the previous input,  $x_{prev}$ .

Visualize the signals using a time scope. You can see that the reconstructed signal overlaps very closely with the input signal.

```
n = zeros(1,1e3);  
xprev = 0;  
for k = 1:1e3  
    x = sine();  
    X = stf(x);  
    bytesSent = bytesSent + length(X);  
    udps(X);  
    dataReceived = [];  
    while (isempty(dataReceived))  
        dataReceived = udpr();  
    end  
    if (~isempty(dataReceived))  
        y = istf(dataReceived);  
    end  
    n(1,k) = norm(y-xprev);  
    xprev = x;  
    bytesReceived = bytesReceived + length(dataReceived);  
    ts([dly(x),y]);  
end
```



The norm of the difference is very small, indicating that the output signal is a perfectly reconstructed version of the input signal.

```
max(abs(n))
```

```
ans = 4.0270e-14
```

Release the UDP objects.

```
release(udps);
release(udpr);
```

Some of the packets sent can be lost during transmission due to the lossy nature of the UDP protocol. To check for loss, compare the bytes sent to the bytes received.

```
fprintf('Bytes sent:    %d\n', bytesSent);
```

```
Bytes sent:    128000
```

```
fprintf('Bytes received: %d\n', bytesReceived);
```

```
Bytes received: 128000
```

## Version History

Introduced in R2012a

### Extended Capabilities

#### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- “System Objects in MATLAB Code Generation” (MATLAB Coder)
- The executable generated from this System object relies on prebuilt dynamic library files (.dll files) included with MATLAB. Use the `packNGo` function to package the code generated from this object and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see “How To Run a Generated Executable Outside MATLAB”.
- The `LocalIPPort` property is tunable in generated code but not tunable during simulation.

### See Also

#### Objects

`dsp.UDPSEnder`

#### Blocks

UDP Receive | UDP Send

#### Topics

“How To Run a Generated Executable Outside MATLAB”

“Variable-Size Signal Support DSP System Objects”

# dsp.UDPSender

**Package:** dsp

Send UDP packets to network

## Description

The `dsp.UDPSender` object sends data packets over the network using the User Datagram Protocol (UDP). UDP is a simple message-based connectionless protocol. The protocol sends data packets in one direction from source to destination without verifying the readiness of the receiver. The protocol has no handshaking mechanism. The data packets can get dropped for several reasons. There is no acknowledgement, retransmission, or timeout in UDP. However, UDP is a very simple transmission protocol and is suitable for time-sensitive applications where dropping packets is preferable to waiting for packets delayed due to retransmission.

To send UDP packets to the network:

- 1 Create the `dsp.UDPSender` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
udps = dsp.UDPSender  
udps = dsp.UDPSender(Name, Value)
```

### Description

`udps = dsp.UDPSender` returns a UDP sender object that sends UDP packets to a specified port.

`udps = dsp.UDPSender(Name, Value)` returns a UDP sender object, `udps`, with each property set to the specified value.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

### RemoteIPAddress — Remote address to which to send data

'127.0.0.1' (default) | character vector containing a valid IP address | string scalar

Specify the remote (that is, host) IP address to which the data is sent. The default is '127.0.0.1', which is the local host.

Data Types: char

### **RemoteIPPort — Remote port to which to send data**

25000 (default) | integer in the range [1, 65535]

Specify the port at the remote IP address to which the data is sent. This property is tunable in generated code but not tunable during simulation.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### **LocalIPPortSource — Source of local IP port**

Auto (default) | Property

Specify how to determine the local IP port on the host as Auto or Property. If you specify Auto, the object selects the port dynamically from the available ports. If you specify Property, the object uses the source specified in the LocalIPPort property.

### **LocalIPPort — Local port from which to send data**

25000 (default) | integer in the range [1, 65535]

Specify the port from which to send data.

### **Dependencies**

This property applies when you set the LocalIPPortSource property to Property.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### **SendBufferSize — Size of internal buffer**

8192 bytes (default) | integer in the range [1, 67108864]

Size of the internal buffer that sends UDP packets, specified in bytes as an integer in the range [1, 67108864].

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

## **Usage**

### **Syntax**

y = udps(Packet)

### **Description**

y = udps(Packet) sends one UDP packet, Packet, to the network.

Since the object sends the data over the UDP network, the data packets can get lost during transmission and the receiver is not guaranteed to receive all the data that you send. You can receive the data using the dsp.UDPReceiver object.

### **Input Arguments**

#### **Packet — Data sent**

scalar | vector



The object sends one UDP packet to the network per call.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `logical`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Common to All System Objects

`step` Run System object algorithm  
`release` Release resources and allow changes to System object property values and input characteristics  
`reset` Reset internal states of System object

## Examples

### Byte Transmission Using UDP

Send and receive UDP packets using the `dsp.UDPsender` and `dsp.UDPReceiver` System objects. Calculate the number of bytes successfully transmitted.

Set the `RemoteIPPort` of UDP sender and the `LocalIPPort` of the UDP receiver to 31000. Set the length of the data vector to 128 samples, which is less than the value of the `MaximumMessageLength` property of the receiver. To prevent the loss of packets, call the `setup` method on the receiver object before the first call to the object algorithm.

```
udpr = dsp.UDPReceiver('LocalIPPort',31000);
udps = dsp.UDPsender('RemoteIPPort',31000);
```

```
setup(udpr);
```

```
bytesSent = 0;
bytesReceived = 0;
dataLength = 128;
```

In each loop of iteration, send and receive a packet of data. At the end of the loop, use the `fprintf` function to print the number of bytes sent by the sender and the number of bytes received by the receiver.

```
for k = 1:20
    dataSent = uint8(255*rand(1,dataLength));
    bytesSent = bytesSent + dataLength;
    udps(dataSent);
    dataReceived = [];
    while (isempty(dataReceived))
        dataReceived = udpr();
    end
    bytesReceived = bytesReceived + length(dataReceived);
end
```

```
release(udps);
release(udpr);

fprintf('Bytes sent:      %d\n', bytesSent);
Bytes sent:      2560

fprintf('Bytes received: %d\n', bytesReceived);
Bytes received: 2560
```

### Tune the UDP Port Number in MATLAB

The local IP port number of the `dsp.UDPReceiver` object and the remote IP port number of the `dsp.UDPSender` object are tunable in the generated code. Generate a MEX file from the `receiver` function which contains the algorithm to receive sine wave data over a UDP network. Change the remote IP port number of the UDP receiver without regenerating the MEX file. Verify the number of bytes sent and received over the network.

**Note:** This example runs only in R2017a or later.

The input to the `receiver` function is the local IP port number of the `dsp.UDPReceiver` System object™. The output of this function is the number of bytes received from the UDP network.

```
type receiver

function [bytesReceived] = receiver(portnumber)

persistent udpRx

if isempty(udpRx)
    udpRx = dsp.UDPReceiver('MessageDataType','double');
end

udpRx.LocalIPPort = portnumber;
dataReceived = udpRx();
bytesReceived = length(dataReceived);
```

The `dsp.UDPSender` object with `remoteIPPort` number set to 65000 sends the data over the UDP network. The `dsp.UDPReceiver` object with `LocalIPPort` number set to 65000 receives the data from the UDP network. The data is a sine wave containing 250 samples per frame.

```
portnumber = 65000;
udpSend = dsp.UDPSender('RemoteIPPort',portnumber);
sine = dsp.SineWave('SamplesPerFrame',250);

bytesSent = 0;
bytesReceived = 0;
dataLength = 250;

for i = 1:10
    dataSent = sine();
    bytesSent = bytesSent + dataLength;
    udpSend(dataSent);
    bytesReceived = bytesReceived + receiver(portnumber);
```

```
end
fprintf('Number of bytes sent: %d', bytesSent);
```

```
Number of bytes sent: 2500
```

```
fprintf('Number of bytes received: %d', bytesReceived);
```

```
Number of bytes received: 2250
```

The data is sent and received successfully over the UDP network. The initial data is dropped due to overhead.

Generate a MEX file from the receiver.m function.

```
codegen receiver -args {65000}
```

```
Code generation successful.
```

Release the sender and change the RemotePort number to 25000. The LocalIPPort number of the receiver continues to be 65000. Since the port numbers are different, the data is not transmitted successfully.

```
release(udpSend)
portnumberTwo = 25000;
udpSend.RemoteIPPort = portnumberTwo;
bytesReceived = 0;
bytesSent = 0;
for i = 1:10
    dataSent = sine();
    bytesSent = bytesSent + dataLength;
    udpSend(dataSent);
    bytesReceived = bytesReceived + receiver_mex(portnumber);
end
fprintf('Number of bytes sent: %d', bytesSent);
```

```
Number of bytes sent: 2500
```

```
fprintf('Number of bytes received: %d', bytesReceived);
```

```
Number of bytes received: 0
```

Clear the MEX file and change the local IP port number of the receiver to 25000. Clearing the MEX enables the receiver port number to change without having to regenerate the MEX. The port numbers of the sender and receiver match. Verify if the data is transmitted successfully.

```
clear mex %#ok
bytesReceived = 0;
bytesSent = 0;
for i = 1:10
    dataSent = sine();
    bytesSent = bytesSent + dataLength;
    udpSend(dataSent);
    bytesReceived = bytesReceived + receiver_mex(portnumberTwo);
end
fprintf('Number of bytes sent: %d', bytesSent);
```

```
Number of bytes sent: 2500
```

```
fprintf('Number of bytes received: %d', bytesReceived);
```

Number of bytes received: 2250

The data is transmitted successfully over the UDP network. The initial data is dropped due to overhead.

### Transmit Complex Data over UDP Network

Compute the STFT of a sine wave and transmit the complex STFT data over a UDP network. At the receiver side, compute the ISTFT of the received data. Visualize the data sent and the data received using a time scope.

The `dsp.UDPSender` object can send complex data. In order to enable the `dsp.UDPReceiver` object to receive complex data, set the `IsMessageComplex` property to `true`.

```
udps = dsp.UDPSender('RemoteIPPort',31000);  
udpr = dsp.UDPReceiver('LocalIPPort',31000,...  
    'IsMessageComplex',true,...  
    'MessageDataType','double');
```

```
setup(udpr);
```

```
bytesSent = 0;  
bytesReceived = 0;  
dataLength = 128;
```

Initialize the `dsp.STFT` and `dsp.ISTFT` System objects with a periodic hann window of length 120 samples and an overlap length of 60 samples. Set the FFT length to 128.

```
winLen = 120;  
overlapLen = 60;  
  
frameLen = winLen-overlapLen;  
stf = dsp.STFT(...  
    'Window',hann(winLen,'periodic'),...  
    'OverlapLength',overlapLen,'FFTLength',128);  
istf = dsp.ISTFT(...  
    'Window',hann(winLen,'periodic'),...  
    'OverlapLength',overlapLen,...  
    'WeightedOverlapAdd',0);
```

The input is a sinusoidal signal with a frequency of 100 Hz, a sample rate of 1000 Hz, and with 60 samples per each signal frame.

```
sine = dsp.SineWave(...  
    'SamplesPerFrame',winLen-overlapLen,...  
    'Frequency',100);
```

Initialize a `timescope` object with a sample rate of 1000 Hz and a time span of 0.09. The `Delay` object corrects the overlap length while comparing the input with the reconstructed output signal.

```
ts = timescope('SampleRate',1000,...  
    'ShowLegend',true,...  
    'YLimits',[-1 1],...  
    'TimeSpanSource','Property',...  
    'TimeSpan',.09,...
```

```

    'ChannelNames',{ 'Input', 'Reconstructed'});
dly = dsp.Delay('Length',overlapLen);

```

Transmit complex STFT data of the sine wave over the UDP network. Due to the undependable nature of the UDP protocol, the data packets can get lost during transmission. The receiver is not guaranteed to receive all the data that you send.

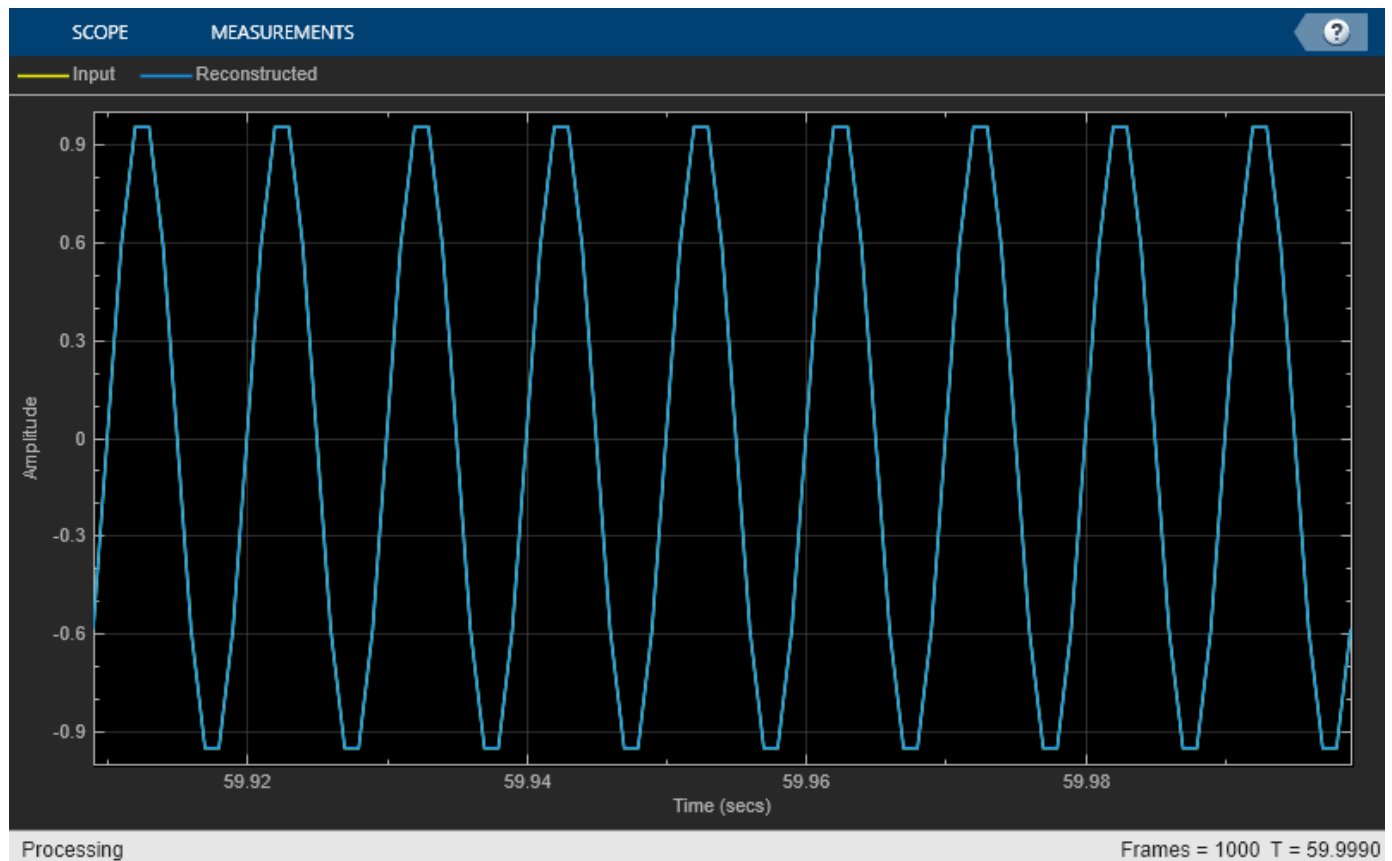
Compute the ISTFT of the received data. Compare the input  $x$  to the reconstructed output  $y$ . Due to the latency introduced by the objects, the reconstructed output is shifted in time compared to the input. Therefore, to compare, take the norm of the difference between the reconstructed output  $y$  and the previous input,  $x_{\text{prev}}$ .

Visualize the signals using a time scope. You can see that the reconstructed signal overlaps very closely with the input signal.

```

n = zeros(1,1e3);
xprev = 0;
for k = 1:1e3
    x = sine();
    X = stf(x);
    bytesSent = bytesSent + length(X);
    udps(X);
    dataReceived = [];
    while (isempty(dataReceived))
        dataReceived = udpr();
    end
    if (~isempty(dataReceived))
        y = istf(dataReceived);
    end
    n(1,k) = norm(y-xprev);
    xprev = x;
    bytesReceived = bytesReceived + length(dataReceived);
    ts([dly(x),y]);
end

```



The norm of the difference is very small, indicating that the output signal is a perfectly reconstructed version of the input signal.

```
max(abs(n))
```

```
ans = 4.0270e-14
```

Release the UDP objects.

```
release(udps);
release(udpr);
```

Some of the packets sent can be lost during transmission due to the lossy nature of the UDP protocol. To check for loss, compare the bytes sent to the bytes received.

```
fprintf('Bytes sent:    %d\n', bytesSent);
```

```
Bytes sent:    128000
```

```
fprintf('Bytes received: %d\n', bytesReceived);
```

```
Bytes received: 128000
```

## Version History

Introduced in R2012a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- “System Objects in MATLAB Code Generation” (MATLAB Coder)
- The executable generated from this System object relies on prebuilt dynamic library files (.dll files) included with MATLAB. Use the `packNGo` function to package the code generated from this object and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see “How To Run a Generated Executable Outside MATLAB”.

The `RemoteIPPort` property is tunable in generated code but not tunable during simulation.

## See Also

### Objects

`dsp.UDPReceiver`

### Blocks

UDP Send | UDP Receive

## **dsp.UniformDecoder**

**Package:** dsp

(Removed) Decode integer input into floating-point output

---

**Note** `dsp.UniformDecoder` has been removed. Use `udecode` instead. For more information, see “Compatibility Considerations”.

---

### **Description**

The `dsp.UniformDecoder` System object decodes integer input into floating-point output. The decoder adheres to the definition for uniform decoding specified in ITU-T Recommendation G.701.

To decode an integer input into a floating-point output:

- 1 Create the `dsp.UniformDecoder` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

### **Creation**

#### **Syntax**

```
ud = dsp.UniformDecoder
ud = dsp.UniformDecoder(peakvalue,numbits)
ud = dsp.UniformDecoder( ____,Name,Value)
```

#### **Description**

`ud = dsp.UniformDecoder` returns a uniform decoder, `ud`, that performs the inverse operation of the `dsp.UniformEncoder` object, reconstructing quantized floating-point values from encoded integer input.

`ud = dsp.UniformDecoder(peakvalue,numbits)` returns a uniform decoder, `ud`, with the `PeakValue` property set to `peakvalue` and the `NumBits` property set to `numbits`.

`ud = dsp.UniformDecoder( ____,Name,Value)` returns a uniform decoder, `ud`, with the `PeakValue` property set to `peakvalue`, `NumBits` property set to `numbits`, and other properties set to the specified values.

#### **Properties**

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.



For more information on changing property values, see System Design in MATLAB Using System Objects.

### **PeakValue — Largest amplitude represented in encoded input**

1 (default) | nonnegative scalar

Specify the largest amplitude represented in the encoded input as a nonnegative numeric scalar. To correctly decode values encoded with the `dsp.UniformEncoder` object, set the `PeakValue` property in both objects to the same value. For more information on setting this property, see the `PeakValue` property description on the `dsp.UniformEncoder` page.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **NumBits — Number of input bits used to encode data**

3 (default) | integer in the range [2, 32]

Specify the number of bits used to encode the input data as an integer value between 2 and 32. The value of this property can be less than the total number of bits supplied by the input data type. To correctly decode values encoded with the `dsp.UniformEncoder` object, set the `NumBits` property in both objects to the same value. For more information on setting this property, see the `NumBits` property description on the `dsp.UniformEncoder` page.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **OverflowAction — Action to take when integer input is out of range**

'Saturate' (default) | 'Wrap'

Specify the behavior of the uniform decoder when the integer input is out of range as 'Saturate' or 'Wrap'. The value of the `NumBits` property specifies the representable range of the input.

### **OutputDataType — Output data type**

'double' (default) | 'single'

Data type of the output, specified as 'single' or 'double'.

## **Usage**

### **Syntax**

$Y = ud(X)$

### **Description**

$Y = ud(X)$  reconstructs quantized floating-point output  $Y$  from the encoded integer input  $X$ .

### **Input Arguments**

#### **X — Encoded integer input**

vector | matrix

Encoded integer input, specified as a vector or a matrix.

Data Types: `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

## Output Arguments

### Y — Decoded output

vector | matrix

Decoded floating-point output, returned as a vector or a matrix. The data type of the output is determined by the `OutputDataType` property.

Data Types: `single` | `double`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Common to All System Objects

`step` Run System object algorithm

`release` Release resources and allow changes to System object property values and input characteristics

`reset` Reset internal states of System object

## Examples

### Decode Sequence

```
ue = dsp.UniformEncoder('PeakValue',2,'NumBits',4,...  
                        'OutputDataType','Signed integer');  
x = (0:0.25:2)'; % Create an input sequence  
ud = dsp.UniformDecoder('PeakValue',2,'NumBits',4);  
x_encoded = ue(x);
```

Check that the last element has been saturated.

```
x_decoded = ud(x_encoded);
```

## Algorithms

This object implements the algorithm, inputs, and outputs described on the Uniform Decoder block reference page. The object properties correspond to the block parameters.

## Version History

**Introduced in R2012a**

**R2021b: `dsp.UniformDecoder` System object will be removed**

*Warns starting in R2021b*

`dsp.UniformDecoder` System object will be removed in a future release. Use the `udcode` function instead.

**R2023a: dsp.UniformDecoder System object has been removed***Errors starting in R2023a*

The dsp.UniformDecoder System object has been removed. Use the udecode function instead.

**Update Code**

This table shows how to update existing code to use the udecode function.

| <b>Discouraged Usage</b>   | <b>Recommended Replacement</b>   |
|--|--|
| <p><b>Encode a sequence</b></p> <pre>ue = dsp.UniformEncoder(PeakValue=2,NumBits=4,OutputDataType='Signed'); x = (0:0.25:2)'; x_encoded = ue(x);</pre> <p><b>Decode the encoded sequence</b></p> <pre>ud = dsp.UniformDecoder(PeakValue=2,NumBits=4); x_decoded = ud(x_encoded);</pre> | <p><b>Encode a sequence</b></p> <p><i>Number of bits is 4, and peak value is 2.</i></p> <pre>x_encodedFn = uencode(x,4,2,'signed');</pre> <p><b>Decode the encoded sequence</b></p> <pre>x_decodedFn = udecode(x_encodedFn,4,2);</pre> |

**See Also**

udecode

## dsp.UniformEncoder

**Package:** dsp

(Removed) Quantize and encode floating-point input into integer output

---

**Note** `dsp.UniformEncoder` has been removed. Use `uencode` instead. For more information, see “Compatibility Considerations”.

---

### Description

The `dsp.UniformEncoder` System object quantizes floating-point input, using the precision you specify in the `NumBits` property, and encodes the quantized input into integer output. The operations of the uniform encoder adhere to the definition for uniform encoding specified in ITU-T Recommendation G.701.

To quantize and encode a floating-point input into an integer output:

- 1 Create the `dsp.UniformEncoder` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

### Creation

#### Syntax

```
ue = dsp.UniformEncoder
ue = dsp.UniformEncoder(peakvalue,numbits)
ue = dsp.UniformEncoder( ____,Name,Value)
```

#### Description

`ue = dsp.UniformEncoder` returns a uniform encoder, `ue`, that quantizes floating-point input samples and encodes them as integers using  $2N$ -level quantization, where  $N$  is an integer.

`ue = dsp.UniformEncoder(peakvalue,numbits)` returns a uniform encoder, `ue`, with the `PeakValue` property set to `peakvalue` and the `NumBits` property set to `numbits`.

`ue = dsp.UniformEncoder( ____,Name,Value)` returns a uniform encoder, `ue`, with the `PeakValue` property set to `peakvalue`, the `NumBits` property set to `numbits`, and other specified properties set to the specified values.

### Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

### PeakValue — Largest input amplitude to be encoded

1 (default) | nonnegative scalar

Specify the largest input amplitude to be encoded, as a nonnegative numeric scalar. If the real or imaginary input are outside of the interval  $[-P, (1 - 2^{(1-B)})P]$ , where  $P$  is the peak value and  $B$  is the value of the NumBits property, the uniform encoder saturates (independently for complex inputs) at those limits.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### NumBits — Number of bits needed to represent output

8 (default) | integer in the range [2, 32]

Specify the number of bits needed to represent the integer output as an integer value between 2 and 32. The number of levels at which the uniform encoder quantizes the floating-point input is  $2^B$ , where  $B$  is the number of bits.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### OutputDataType — Data type of output

'Unsigned integer' (default) | 'Signed integer'

Specify the data type of the output as 'Unsigned integer' or 'Signed integer'. Unsigned outputs are uint8, uint16, or uint32, and signed outputs are int8, int16, or int32. The quantized inputs are linearly (uniformly) mapped to the intermediate integers in the interval  $[0, 2^{(B-1)}]$  when you set this property to 'Unsigned integer', and in the interval  $[-2^{(B-1)}, 2^{(B-1)} - 1]$  when you set this property to 'Signed integer'. The variable  $B$  in both expressions corresponds to the value of the NumBits property.

## Usage

### Syntax

$Y = \text{ue}(X)$

### Description

$Y = \text{ue}(X)$  quantizes and encodes the input  $X$  to output  $Y$ .

### Input Arguments

#### X — Data input

vector | matrix

Data input, specified as a vector or a matrix. The input  $X$  can be real or complex and double-, or single-precision values.

Data Types: single | double

## Output Arguments

### Y — Encoded data

vector | matrix

Quantized and encoded output, returned as a vector or a matrix.

The uniform encoder chooses the output data type according to the table.

| Number of Bits | Unsigned Integer | Signed Integer |
|----------------|------------------|----------------|
| 2 to 8         | uint8            | int8           |
| 9 to 16        | uint16           | int16          |
| 17 to 32       | uint32           | int32          |

The row in the table corresponds to the value of the NumBits property, and the column in the table corresponds to the value of the OutputDataType property.

Data Types: int8 | int16 | int32 | uint8 | uint16 | uint32

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named obj, use this syntax:

```
release(obj)
```

## Common to All System Objects

step     Run System object algorithm  
 release   Release resources and allow changes to System object property values and input characteristics  
 reset     Reset internal states of System object

## Examples

### Encode Number Sequence

```
ue = dsp.UniformEncoder;
ue.PeakValue = 2;
ue.NumBits = 4;
ue.OutputDataType = 'Signed integer';
x = (-1:0.01:1)'; % Create an input sequence
x_encoded = ue(x);
plot(x, x_encoded, '.')
xlabel('Input')
ylabel('Encoded Output')
grid
```

## Algorithms

This object implements the algorithm, inputs, and outputs described on the Uniform Encoder block reference page. The object properties correspond to the block parameters.

## Version History

### Introduced in R2012a

#### R2021b: dsp.UniformEncoder System object will be removed

*Warns starting in R2021b*

dsp.UniformEncoder System object will be removed in a future release. Use the uencode function instead.

#### R2023a: dsp.UniformEncoder System object has been removed

*Errors starting in R2023a*

The dsp.UniformEncoder System object has been removed. Use the uencode function instead.

### Update Code

This table shows how to update existing code to use the uencode function.

| Discouraged Usage  | Recommended Replacement  |
|--|--|
| <p><b>Encode a sequence</b></p> <pre>ue = dsp.UniformEncoder(PeakValue=2,NumBits=4,OutputDataType='Signed'); x = (0:0.25:2)'; x_encoded = ue(x);</pre> <p><b>Decode the encoded sequence</b></p> <pre>ud = dsp.UniformDecoder(PeakValue=2,NumBits=4); x_decoded = ud(x_encoded);</pre> | <p><b>Encode a sequence</b></p> <pre>x_encodedFn = uencode(x,4,2,'signed');</pre> <p><b>Decode the encoded sequence</b></p> <pre>x_decodedFn = udecode(x_encodedFn,4,2);</pre> |

### See Also

uencode

## dsp.UpperTriangularSolver

**Package:** dsp

(Removed) Solve upper-triangular matrix equation

---

**Note** `dsp.UpperTriangularSolver` has been removed. Use the `mldivide` function or the `\` operator instead. For more information, see “Compatibility Considerations”.

---

### Description

The `UpperTriangularSolver` object solves  $UX = B$  for  $X$  when  $U$  is a square, upper-triangular matrix with the same number of rows as  $B$ .

To solve  $UX = B$ :

- 1 Create the `dsp.UpperTriangularSolver` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

### Creation

#### Syntax

```
uptriang = dsp.UpperTriangularSolver
uptriang = dsp.UpperTriangularSolver(Name, Value)
```

#### Description

`uptriang = dsp.UpperTriangularSolver` returns a linear system solver, `uptriang`, used to solve  $UX = B$  where  $U$  is an upper (or unit-upper) triangular matrix.

`uptriang = dsp.UpperTriangularSolver(Name, Value)` returns a linear system solver, `uptriang`, with each specified property set to the specified value. Enclose each property name in single quotes. Unspecified properties have default values.

### Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

#### **OverwriteDiagonal — Replace diagonal elements of input with ones**

false (default) | true



When you set this property to `true`, the linear system solver replaces the elements on the diagonal of the input,  $U$ , with ones. This property is useful when matrix  $U$  is the result of another operation, such as an LDL decomposition, that uses the diagonal elements to represent the  $D$  matrix.

### **RealDiagonalElements — Indicate that diagonal of complex input is real**

`false` (default) | `true`

When you set this property to `true`, the linear system solver optimizes computation speed if the input  $U$  is complex, but its diagonal elements are real. Set this property to either `true` or `false`.

#### **Dependencies**

This property applies only when you set the `OverwriteDiagonal` property to `false`.

### **Fixed-Point Properties**

#### **RoundingMethod — Rounding method for fixed-point operations**

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

Specify the rounding method as `Ceiling`, `Convergent`, `Floor`, `Nearest`, `Round`, `Simplest`, or `Zero`.

#### **OverflowAction — Overflow action for fixed-point operations**

`Wrap` (default) | `Saturate`

Specify the overflow action as `Wrap` or `Saturate`.

#### **ProductDataType — Data type of product**

`Full precision` (default) | `Same as input` | `Custom`

Specify the product data type as `Full precision`, `Same as input`, or `Custom`.

#### **CustomProductDataType — Product word and fraction lengths**

`numerictype([],32,30)` (default) | `numerictype`

Specify the product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`.

#### **Dependencies**

This property applies only when you set the `ProductDataType` property to `Custom`.

#### **AccumulatorDataType — Data type of accumulator**

`Full precision` (default) | `Same as first input` | `Same as product` | `Custom`

Specify the accumulator data type as `Full precision`, `Same as first input`, `Same as product`, or `Custom`.

#### **CustomAccumulatorDataType — Accumulator word and fraction lengths**

`numerictype([],32,30)` (default) | `numerictype`

Specify the accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`.

#### **Dependencies**

This property applies only when you set the `AccumulatorDataType` property to `Custom`.

**OutputDataType — Data type of output**

Same as first input (default) | Custom

Specify the output data type as Same as first input or Custom.

**CustomOutputDataType — Output word and fraction lengths**

numericType([],16,15) (default) | numericType

Specify the output fixed-point type as a scaled numericType object with a Signedness of Auto.

**Dependencies**

This property applies only when you set the “OutputDataType” on page 4-0 property to Custom.

**Usage****Syntax** $X = \text{uptriang}(U,B)$ **Description** $X = \text{uptriang}(U,B)$  computes the solution,  $X$ , of the matrix equation  $UX = B$ , where  $U$  is a square, upper-triangular matrix with the same number of rows as the matrix  $B$ .**Input Arguments****U — Upper-triangular matrix**

matrix

Upper-triangular square matrix of size  $M$ -by- $M$ .

If the matrix is of fixed-point data type, it must be signed fixed point.

Data Types: single | double | int8 | int16 | int32 | int64 | fi

**B — Input B**

vector | matrix

Input  $B$  in the equation  $UX = B$ , where  $B$  is an  $M$ -by- $N$  matrix.

If the matrix is of fixed-point data type, it must be signed fixed point.

Data Types: single | double | int8 | int16 | int32 | int64 | fi

**Output Arguments****X — Solution of equation**

vector | matrix

Solution of the  $UX = B$  equation, returned as an  $M$ -by- $N$  output matrix. The object uses only the elements in the upper triangle of input  $U$  and ignores the lower elements. When you set `OverwriteDiagonal` to `true`, the object replaces the elements on the diagonal of the input,  $U$ , with ones.

If the matrix is of fixed-point data type, it must be signed fixed point.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `fi`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Common to All System Objects

|                      |  |
|----------------------|--|
| <code>step</code>    | Run System object algorithm  |
| <code>release</code> | Release resources and allow changes to System object property values and input characteristics |
| <code>reset</code>   | Reset internal states of System object   |

## Examples

### Solve an Upper Triangular Matrix

```
rng(1)
uptriang = dsp.UpperTriangularSolver;
u = triu(rand(4,4));
b = rand(4,1);
```

Check that result is the solution to the linear equations.

```
x1 = u\b
```

```
x1 = 4×1
```

```
0.0632
1.3991
0.3156
0.2955
```

```
x = uptriang(u,b)
```

```
x = 4×1
0.0632
1.3991
0.3156
0.2955
```

## Algorithms

This object implements the algorithm, inputs, and outputs described on the Backward Substitution block reference page. The object properties correspond to the block parameters.

## Version History

**Introduced in R2012a**

**R2021b: dsp.UpperTriangularSolver System object will be removed***Warns starting in R2021b*

`dsp.UpperTriangularSolver` System object will be removed in a future release. Use the `mldivide` function or the `\` operator instead.

**R2023a: dsp.UpperTriangularSolver System object has been removed***Errors starting in R2023a*

The `dsp.UpperTriangularSolver` System object has been removed. Use the `mldivide` function or the `\` operator instead.

**Update Code**

This table shows how to update existing code to use the `mldivide` function or the `\` operator.

| Discouraged Usage   | Recommended Replacement  |
|---|--|
| <pre>rng(1); uptriang = dsp.UpperTriangularSolver; u = triu(rand(4,4)); b = rand(4,1); x = uptriang(u,b)  x = 4x1     0.0632     1.3991     0.3156     0.2955</pre> | <pre>x1 = u\b  x = 4x1     0.0632     1.3991     0.3156     0.2955</pre> <p>Alternatively, you can also use the <code>mldivide</code> function.</p> <pre>x1 = mldivide(u,b)  x = 4x1     0.0632     1.3991     0.3156     0.2955</pre> |

**Extended Capabilities****C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

**See Also****Functions**

`mldivide`

# dsp.VariableBandwidthFIRFilter

**Package:** dsp

Variable bandwidth FIR filter

## Description

The `dsp.VariableBandwidthFIRFilter` object filters each channel of the input using FIR filter implementations. It does so while having the capability of tuning the bandwidth.

To filter each channel of the input:

- 1 Create the `dsp.VariableBandwidthFIRFilter` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
vbw = dsp.VariableBandwidthFIRFilter  
vbw = dsp.VariableBandwidthFIRFilter(Name=Value)
```

### Description

`vbw = dsp.VariableBandwidthFIRFilter` returns a variable bandwidth FIR filter object which independently filters each channel of the input over successive calls to the object. The filter cutoff frequency can be tuned during the filtering operation. The variable bandwidth FIR filter is designed using the window method.

`vbw = dsp.VariableBandwidthFIRFilter(Name=Value)` returns a variable bandwidth FIR filter with additional properties specified by one or more `Name-Value` pair arguments. `Name` is the property name and `Value` is the corresponding value. For example, `FilterOrder=50` sets the filter order to 50.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

### **FilterOrder — FIR filter order**

30 (default) | even positive integer

FIR filter order, specified as an even positive integer. This property is nontunable.

Data Types: `double` | `single`

### **FilterType — Type of filter response**

'Lowpass' (default) | 'Highpass' | 'Bandpass' | 'Bandstop'

Type of filter response, specified as one of these options:

- 'Lowpass'
- 'Highpass'
- 'Bandpass'
- 'Bandstop'

### **CutoffFrequency — Filter cutoff frequency**

512 (default) | positive scalar

Filter cutoff frequency, specified as a real positive scalar in Hz or in normalized frequency units.

If you set the `NormalizedFrequency` property to:

- `false` -- The value of the cutoff frequency is in Hz. The value must be less than half the `SampleRate` property value.
- `true` -- The value of the cutoff frequency is in normalized frequency units. The value must be a positive scalar less than `1.0`.

**Tunable:** Yes

### **Dependencies**

To enable this property, set `FilterType` to 'Lowpass' or 'Highpass'.

Data Types: `double` | `single`

### **Window — Window function**

'Hann' (default) | 'Hamming' | 'Chebyshev' | 'Kaiser'

Window function used to design the FIR filter, specified as one of these options:

- 'Hann'
- 'Hamming'
- 'Chebyshev'
- 'Kaiser'

### **KaiserWindowParameter — Kaiser window parameter**

0.5 (default) | real scalar

Kaiser window parameter, specified as a real scalar.

### **Dependencies**

To enable this property, set the `Window` property to 'Kaiser'.

Data Types: `double` | `single`

**CenterFrequency — Center frequency of filter**

11025 (default) | positive scalar

Center frequency of filter, specified as a real positive scalar in Hz or in normalized frequency units.

If you set the `NormalizedFrequency` property to:

- `false` -- The value of the center frequency is in Hz. The value must be less than half the `SampleRate` property value.
- `true` -- The value of the center frequency is in normalized frequency units. The value must be a positive scalar less than `1.0`.

**Tunable:** Yes

**Dependencies**

To enable this property, set the `FilterType` property to 'Bandpass' or 'Bandstop'.

Data Types: double | single

**Bandwidth — Filter bandwidth**

7680 (default) | positive scalar

Filter bandwidth, specified as a real positive scalar in Hz or in normalized frequency units.

If you set the `NormalizedFrequency` property to:

- `false` -- The value of the filter bandwidth is in Hz. The value must be less than half the `SampleRate` property value.
- `true` -- The value of the filter bandwidth is in normalized frequency units. The value must be a positive scalar less than `1.0`.

**Tunable:** Yes

**Dependencies**

To enable this property, set the `FilterType` property to 'Bandpass' or 'Bandstop'.

Data Types: double | single

**SidelobeAttenuation — Chebyshev window sidelobe attenuation**

60 (default) | positive scalar

Chebyshev window sidelobe attenuation, specified as a real positive scalar in dB.

**Dependencies**

To enable this property, set the `Window` property to 'Chebyshev'.

Data Types: double | single

**NormalizedFrequency — Flag to set frequencies in normalized units**

false (default) | true

Flag to set frequencies in normalized units, specified as one of these values:

- `true` -- The filter cutoff frequency, center frequency, and the filter bandwidth must be in the normalized frequency units and less than `1.0`.

- `false` -- The filter cutoff frequency, center frequency, and the filter bandwidth are in Hz. You can specify the input sample rate through the `SampleRate` property.

Data Types: `logical`

### **SampleRate** — Input sample rate

44100 (default) | positive scalar

Input sample rate, specified as a positive scalar in Hz.

#### **Dependency**

To enable this property, set `NormalizedFrequency` to `false`.

Data Types: `double` | `single`

## **Usage**

### **Syntax**

`y = vbw(x)`

#### **Description**

`y = vbw(x)` filters the input signal `x` using the variable bandwidth FIR filter to produce the output `y`. The variable bandwidth FIR filter object operates on each channel, which means the object filters every column of the input signal independently over successive calls to the algorithm.

#### **Input Arguments**

##### **x** — Data input

vector | matrix

Data input, specified as a vector or a matrix. This object also accepts variable-size inputs. Once the object is locked, you can change the size of each input channel, but you cannot change the number of channels.

Data Types: `double` | `single`

#### **Output Arguments**

##### **y** — Filtered output

vector | matrix

Filtered output, returned as a vector or a matrix. The size, data type, and complexity of the output signal matches that of the input signal.

Data Types: `double` | `single`

## **Object Functions**

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

`release(obj)`



## Specific to dsp.VariableBandwidthFIRFilter

|          |  |
|----------|--|
| freqz    | Frequency response of discrete-time filter System object     |
| fvtool   | Visualize frequency response of DSP filters                  |
| impz     | Impulse response of discrete-time filter System object       |
| info     | Information about filter System object                       |
| coeffs   | Returns the filter System object coefficients in a structure |
| cost     | Estimate cost of implementing filter System object           |
| grpdelay | Group delay response of discrete-time filter System object   |

## Common to All System Objects

|         |  |
|---------|--|
| step    | Run System object algorithm  |
| release | Release resources and allow changes to System object property values and input characteristics |
| reset   | Reset internal states of System object   |

## Examples

### Filtering Through a Variable Bandwidth Bandpass FIR Filter

Tune the center frequency and the bandwidth of the FIR bandpass filter. Filter a sinusoidal signal through this filter.

Define a bandpass variable bandwidth FIR filter. Specify an input sample rate of 44100 Hz. Initialize a dsp.TransferFunctionEstimator object to estimate the transfer function of the filter from the input and output signals. To visualize the transfer function, initialize a dsp.ArrayPlot object.

```
Fs = 44100;
vbw = dsp.VariableBandwidthFIRFilter(FilterType='Bandpass',...
    FilterOrder=100,...
    SampleRate=Fs,...
    CenterFrequency=1e4,...
    Bandwidth=4e3);
tfe = dsp.TransferFunctionEstimator(FrequencyRange='onesided');
aplot = dsp.ArrayPlot(PlotType='Line',...
    Xoffset=0,...
    YLimits=[-120 5], ...
    SampleIncrement=44100/1024,...
    YLabel='Frequency Response (dB)',...
    XLabel='Frequency (Hz)',...
    Title='System Transfer Function');
```

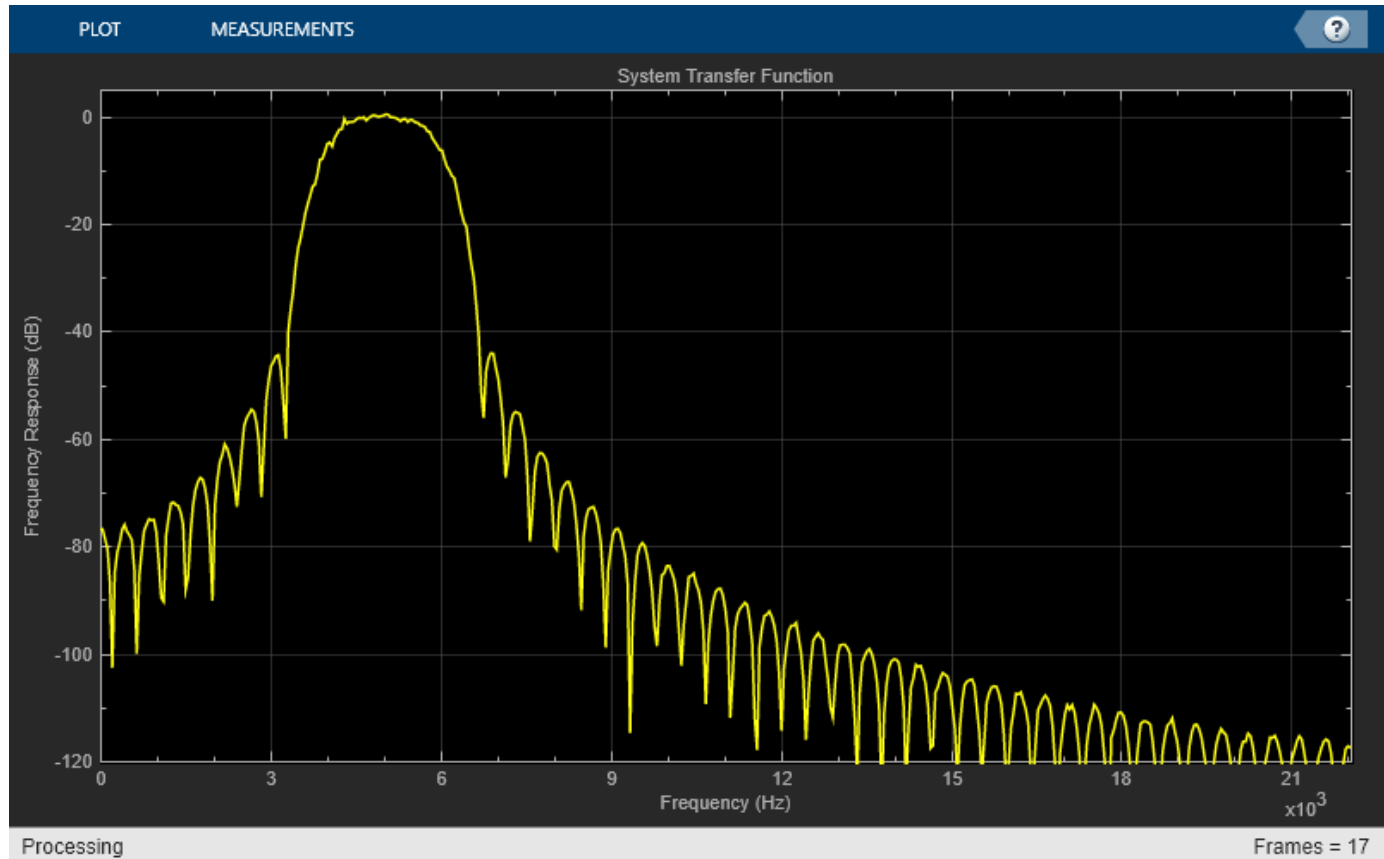
Generate a sine wave signal with a frame length of 1024. Tune the bandwidth and the center frequency of the filter. Pass the signal through this filter. Estimate the transfer function of the filter using the input and the generated output. Plot the system transfer function using an array plot.

```
FrameLength = 1024;
sine = dsp.SineWave(SamplesPerFrame=FrameLength);
for i=1:500
    % Generate input
    x = sine() + randn(FrameLength,1);
    % Pass input through the filter
    y = vbw(x);
    % Transfer function estimation
```

```

h = tfe(x,y);
% Plot transfer function
aplot(20*log10(abs(h)))
% Tune bandwidth and center frequency of the FIR filter
if (i==250)
    vbw.CenterFrequency = 5000;
    vbw.Bandwidth = 2000;
end
end
end

```



## Algorithms

### FIR Transformations

All transformations assume a lowpass filter of length  $2N+1$ .

#### Lowpass to Lowpass

Consider an ideal lowpass brickwall filter with normalized cutoff frequency  $\omega_{c1}$ . By taking the inverse discrete Fourier transform of the ideal frequency response, and clipping the resulting sequence to length  $2N+1$ , the impulse response is:

for  $n = 0$

$$h_{LP}(n) = \frac{\omega_c}{\pi} \cdot w(0)$$

for  $1 \leq |n| \leq N$

$$h_{LP}(n) = \frac{\sin(\omega_c n)}{\pi n} \cdot w(n)$$

where  $w(n)$  is the window vector. Tune the lowpass filter coefficients to a new cutoff frequency  $\omega_{c2}$  as follows:

for  $n = 0$

$$h_{LP}(n) = \frac{\omega_{c2}}{\pi} \cdot w(0)$$

for  $1 \leq |n| \leq N$

$$h_{LP}(n) = \frac{\sin(\omega_{c2} n)}{\pi n} \cdot w(n)$$

You do not need to recompute the window every time you tune the cutoff frequency.

### Lowpass to Highpass

Assuming a lowpass filter with normalized 6-dB cutoff frequency  $\omega_c$ , you can obtain a highpass filter with the same cutoff frequency by taking the complementary of the lowpass frequency response:

$$H_{HP}(e^{j\omega}) = 1 - H_{LP}(e^{j\omega})$$

Taking the inverse discrete Fourier transform of the above response, you have the following highpass filter coefficients:

for  $n = 0$

$$h_{hp}(n) = 1 - h_{LP}(n)$$

for  $1 \leq |n| \leq N$

$$h_{hp}(n) = -h_{LP}(n)$$

### Lowpass to Bandpass

Obtain a bandpass filter centered at frequency  $\omega_0$  by shifting the lowpass response:

$$H_{BP}(e^{j\omega}) = H_{LP}(e^{j(\omega-\omega_0)}) + H_{LP}(e^{j(\omega+\omega_0)})$$

The bandwidth of the resulting bandpass filter is  $2\omega_c$ , as measured between the two cutoff frequencies of the bandpass filter. The equivalent bandpass filter coefficients are then:

$$h_{BP}(n) = (e^{j\omega_0 n} + e^{-j\omega_0 n})h_{LP}(n)$$

which can be written as:

$$h_{BP}(n) = 2\cos(\omega_0 n)h_{LP}(n)$$

### Lowpass to Bandstop

You can transform a lowpass filter to a bandstop filter by combining the highpass and bandpass transformations. First make the filter bandpass by shifting the lowpass response, and then invert it to get a bandstop response centered at  $\omega_0$ .

$$H_{BS}(e^{j\omega}) = 1 - (H_{LP}(e^{j(\omega-\omega_0)}) + H_{LP}(e^{j(\omega+\omega_0)}))$$

This yields the following coefficients:

for  $n = 0$

$$h_{BS}(n) = 1 - 2\cos(\omega_0 n)h_{LP}(n)$$

for  $1 \leq |n| \leq N$

$$h_{BS}(n) = -2\cos(\omega_0 n)h_{LP}(n)$$

### Generalized Transformation

You can combine these transformations to transform a lowpass filter to a lowpass, highpass, bandpass, or bandstop filter with arbitrary cutoffs.

For example, to transform a lowpass filter with cutoff  $\omega_{c1}$  to a highpass with cutoff  $\omega_{c2}$ , you first apply the lowpass-to-lowpass transformation to get a lowpass filter with cutoff  $\omega_{c2}$ , and then apply the lowpass-to-highpass transformation to get the highpass with cutoff  $\omega_{c2}$ .

To get a bandpass filter with center frequency  $\omega_0$  and bandwidth  $\beta$ , we first apply the lowpass-to-lowpass transformation to go from a lowpass with cutoff  $\omega_c$  to a lowpass with cutoff  $\beta/2$ , and then apply the lowpass-to-bandpass transformation to get the desired bandpass filter. You can use the same approach for a bandstop filter.

## Version History

### Introduced in R2014a

#### R2023a: Support for normalized frequencies

When you set the `NormalizedFrequency` property to `true`, you can specify the filter cutoff frequency, center frequency, and the filter bandwidth in normalized frequency units (0 to 1).

When you set the `NormalizedFrequency` property to `true` while creating the object, the frequency values are automatically set to normalized frequency units using the default sample rate of 44100 Hz.

```
vbFIRFilter = dsp.VariableBandwidthFIRFilter(NormalizedFrequency=true)
```

```
vbFIRFilter =  
    dsp.VariableBandwidthFIRFilter with properties:
```

```
        FilterOrder: 30  
        FilterType: 'Lowpass'  
        CutoffFrequency: 0.0232  
        Window: 'Hann'  
        NormalizedFrequency: true
```

When you set the `NormalizedFrequency` property to `true` after you create the object, the frequencies must be manually set to the normalized frequency values before you run the object algorithm.

```
vbFIRFilter = dsp.VariableBandwidthFIRFilter
```

```
vbFIRFilter =  
    dsp.VariableBandwidthFIRFilter with properties:
```

```

    FilterOrder: 30
    FilterType: 'Lowpass'
    CutoffFrequency: 512
    Window: 'Hann'
    NormalizedFrequency: false
    SampleRate: 44100

```

To specify the normalized frequency values, set `NormalizedFrequency` to `true` and manually convert the frequency values in Hz to normalized values using the input sample rate in Hz. For example, if the input sample rate is 44100 Hz, the corresponding values in normalized units are computed using these equations.

```

vbFIRFilter.NormalizedFrequency = true;
vbFIRFilter.CutoffFrequency = 512/(44100/2)

```

```

vbFIRFilter =
    dsp.VariableBandwidthFIRFilter with properties:

```

```

    FilterOrder: 30
    FilterType: 'Lowpass'
    CutoffFrequency: 0.0232
    Window: 'Hann'
    NormalizedFrequency: true

```

## References

- [1] Jarske, P.Y. Neuvo, and S. K. Mitra, "A simple approach to the design of linear phase FIR digital filters with variable characteristics." *Signal Processing*. Vol. 14, Issue 4, June 1988, pp. 313-326.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

This object also supports SIMD code generation using Intel AVX2 technology when the input signal has a data type of `single` or `double`.

The SIMD technology significantly improves the performance of the generated code.

## See Also

### Functions

`freqz` | `fvtool` | `impz` | `info` | `coeffs` | `cost` | `grpdelay`

### Objects

`dsp.SOSFilter` | `dsp.IIRFilter` | `dsp.FIRFilter` | `dsp.AllpoleFilter` | `dsp.VariableBandwidthIIRFilter`

### **Blocks**

Variable Bandwidth FIR Filter | Variable Bandwidth IIR Filter

### **Topics**

“Analysis Methods for Filter System Objects” on page 3-2

# dsp.VariableBandwidthIIRFilter

**Package:** dsp

Variable bandwidth IIR filter

## Description

The `dsp.VariableBandwidthIIRFilter` object filters each channel of the input using IIR filter implementations. It does so while having the capability of tuning the bandwidth.

To filter each channel of the input:

- 1 Create the `dsp.VariableBandwidthIIRFilter` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
vbwIIR = dsp.VariableBandwidthIIRFilter  
vbwIIR = dsp.VariableBandwidthIIRFilter(Name=Value)
```

### Description

`vbwIIR = dsp.VariableBandwidthIIRFilter` returns a variable bandwidth IIR filter object which independently filters each channel of the input over successive calls to the algorithm. This System object uses a specified IIR filter implementation. The filter passband frequency can be tuned during the filtering operation. The variable bandwidth IIR filter is designed using the elliptical method. The filter is tuned using IIR spectral transformations based on allpass filters.

`vbwIIR = dsp.VariableBandwidthIIRFilter(Name=Value)` returns a variable bandwidth IIR filter with additional properties specified by one or more `Name=Value` pair arguments. `Name` is the property name and `Value` is the corresponding value. For example, `PassbandFrequency=512` sets the passband frequency to 512 Hz.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

### **FilterType — Type of filter response**

'Lowpass' (default) | 'Highpass' | 'Bandpass' | 'Bandstop'

Type of filter response, specified as one of these options:

- 'Lowpass'
- 'Highpass'
- 'Bandpass'
- 'Bandstop'

### **FilterOrder — IIR filter order**

8 (default) | positive integer

IIR filter order, specified as a positive integer.

Data Types: double | single

### **PassbandFrequency — Filter passband frequency**

512 (default) | positive scalar

Filter passband frequency, specified as a positive scalar in Hz or in normalized frequency units.

If you set the `NormalizedFrequency` property to:

- `false` -- The value of the passband frequency is in Hz. The value must be less than half the `SampleRate` property value.
- `true` -- The value of the passband frequency is in normalized frequency units. The value must be a positive scalar less than 1.0.

**Tunable:** Yes

#### **Dependencies**

To enable this property, set the `FilterType` property to 'Lowpass' or 'Highpass'.

Data Types: double | single

### **CenterFrequency — Filter center frequency**

11025 (default) | positive scalar

Filter center frequency, specified as a real positive scalar in Hz or in normalized frequency units.

If you set the `NormalizedFrequency` property to:

- `false` -- The value of the center frequency is in Hz. The value must be less than half the `SampleRate` property value.
- `true` -- The value of the center frequency is in normalized frequency units. The value must be a positive scalar less than 1.0.

**Tunable:** Yes

#### **Dependencies**

To enable this property, set the `FilterType` property to 'Bandpass' or 'Bandstop'.

Data Types: double | single



**Bandwidth — Filter bandwidth**

7680 (default) | positive scalar

Filter bandwidth, specified as a real positive scalar in Hz or in normalized frequency units.

If you set the `NormalizedFrequency` property to:

- `false` -- The value of the filter bandwidth is in Hz. The value must be less than half the `SampleRate` property value.
- `true` -- The value of the filter bandwidth is in normalized frequency units. The value must be a positive scalar less than `1.0`.

**Tunable:** Yes

**Dependencies**

To enable this property, set the `FilterType` property to 'Bandpass' or 'Bandstop'.

Data Types: double | single

**PassbandRipple — Filter passband ripple**

1 (default) | positive scalar

Filter passband ripple, specified as a real positive scalar in dB.

Data Types: double | single

**StopbandAttenuation — Filter Stopband attenuation**

60 (default) | positive scalar

Filter stopband attenuation, specified as a real positive scalar in dB.

Data Types: double | single

**NormalizedFrequency — Specify frequencies in normalized units**`false` (default) | `true`

When you set this property to:

- `true` -- The filter passband frequency, center frequency, and the filter bandwidth must be in the normalized frequency units and less than `1.0`.
- `false` -- The filter passband frequency, center frequency, and the filter bandwidth are in Hz. You can specify the input sample rate through the `SampleRate` property.

Data Types: logical

**SampleRate — Input sample rate**

44100 (default) | positive scalar

Input sample rate, specified as a positive scalar in Hz.

**Dependency**

To enable this property, set `NormalizedFrequency` to `false`.

Data Types: double | single

## Usage

### Syntax

```
y = vbwIIR(x)
```

### Description

`y = vbwIIR(x)` filters the real or complex input signal `x` using a variable bandwidth IIR filter to produce the output `y`. The variable bandwidth IIR filter object operates on each channel, which means the object filters every column of the input signal independently over successive calls to the algorithm.

### Input Arguments

#### **x** — Data input

vector | matrix

Data input, specified as a vector or a matrix. This object also accepts variable-size inputs. Once the object is locked, you can change the size of each input channel, but you cannot change the number of channels.

Data Types: double | single

### Output Arguments

#### **y** — Filtered output

vector | matrix

Filtered output, returned as a vector or a matrix. The size, data type, and complexity of the output signal matches that of the input signal.

Data Types: double | single

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

### Specific to `dsp.VariableBandwidthIIRFilter`

|                       |  |
|-----------------------|--|
| <code>freqz</code>    | Frequency response of discrete-time filter System object     |
| <code>fvtool</code>   | Visualize frequency response of DSP filters                  |
| <code>impz</code>     | Impulse response of discrete-time filter System object       |
| <code>info</code>     | Information about filter System object                       |
| <code>coeffs</code>   | Returns the filter System object coefficients in a structure |
| <code>cost</code>     | Estimate cost of implementing filter System object           |
| <code>grpdelay</code> | Group delay response of discrete-time filter System object   |

### Common to All System Objects

`step` Run System object algorithm

release Release resources and allow changes to System object property values and input characteristics

reset Reset internal states of System object

## Examples

### Filtering Through a Variable Bandwidth Bandpass IIR Filter

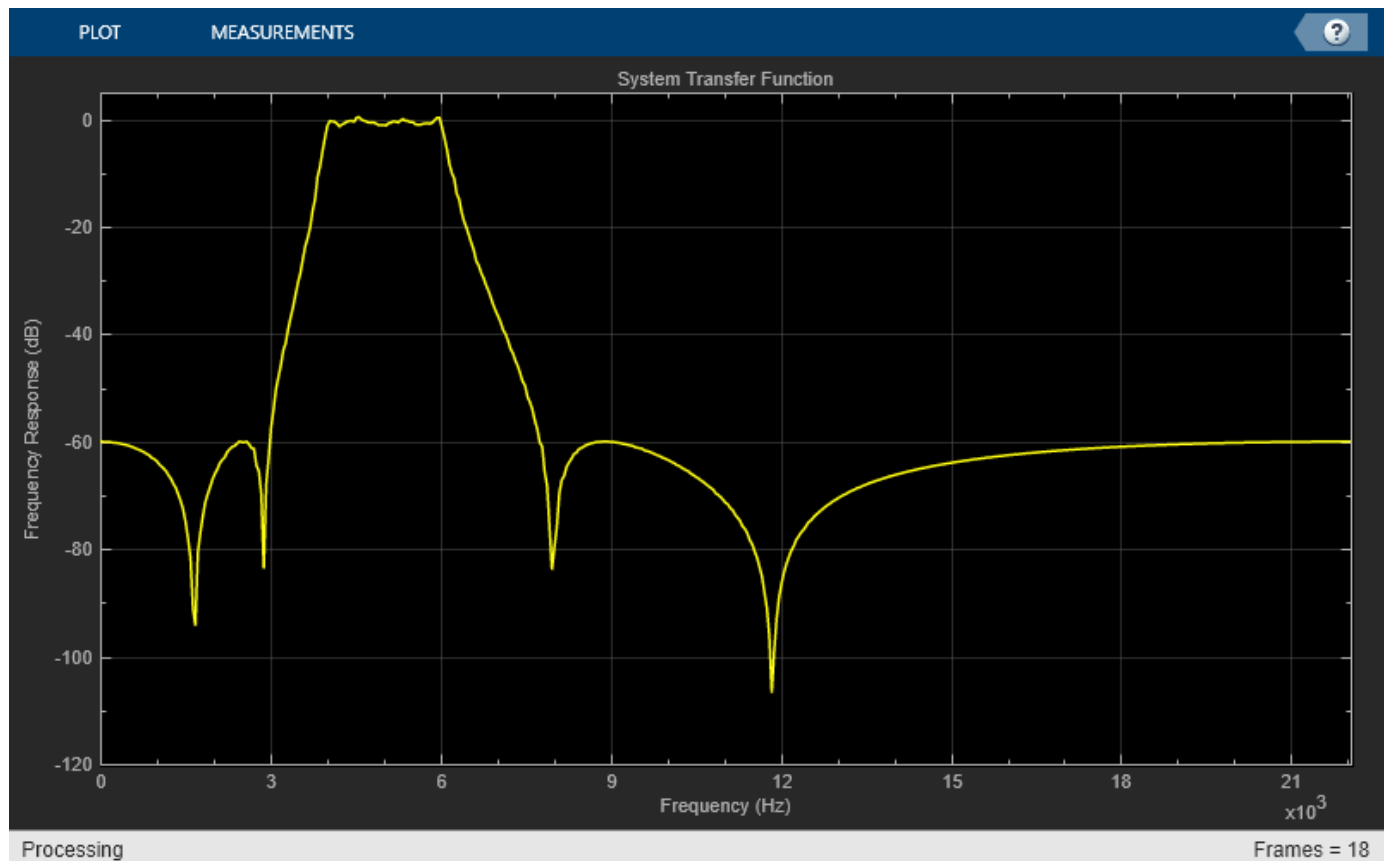
Tune the center frequency and the bandwidth of the IIR bandpass filter. Filter a sinusoidal signal through this filter.

Define a bandpass variable bandwidth IIR filter. Specify an input sample rate of 44100 Hz. Initialize a `dsp.TransferFunctionEstimator` object to estimate the transfer function of the filter from the input and output signals. To visualize the transfer function, initialize a `dsp.ArrayPlot` object.

```
Fs = 44100;
vbwiir = dsp.VariableBandwidthIIRFilter(FilterType='Bandpass',...
    FilterOrder=8,...
    SampleRate=Fs,...
    CenterFrequency=1e4,...
    Bandwidth=4e3);
tfe = dsp.TransferFunctionEstimator(FrequencyRange='onesided');
aplot = dsp.ArrayPlot(PlotType='Line',...
    XOffset=0,...
    YLimits=[-120 5], ...
    SampleIncrement=44100/1024,...
    YLabel='Frequency Response (dB)',...
    XLabel='Frequency (Hz)',...
    Title='System Transfer Function');
```

Generate a sine wave signal with a frame length of 1024. Tune the bandwidth and the center frequency of the filter. Pass the signal through this filter. Estimate the transfer function of the filter using the input and the generated output. Plot the system transfer function on Array Plot.

```
FrameLength = 1024;
sine = dsp.SineWave(SamplesPerFrame=FrameLength);
for i = 1:500
    % Generate input
    x = sine() + randn(FrameLength,1);
    % Pass input through the filter
    y = vbwiir(x);
    % Transfer function estimation
    h = tfe(x,y);
    % plot transfer function
    aplot(20*log10(abs(h)))
    % Tune bandwidth and center frequency of the IIR filter
    if (i==250)
        vbwiir.CenterFrequency = 5000;
        vbwiir.Bandwidth = 2000;
    end
end
```



## Algorithms

This filter covers frequency transformations. The algorithm designs a lowpass IIR prototype using the elliptical method by specifying its order, passband frequency, passband ripple, and stopband attenuation. The passband ripple and stopband attenuation are equal to the values of the `PassbandRipple` and `StopbandAttenuation` properties. The algorithm sets the prototype passband frequency to 0.5. If the `FilterType` property is 'Lowpass' or 'Highpass', the order of the prototype filter is equal to the value of `FilterOrder`. If the `FilterType` property is 'Bandpass' or 'Bandstop', the order of the prototype filter is equal to `FilterOrder/2`. The prototype is a Direct Form II Transposed cascade of second-order sections (Biquad filter). The prototype is transformed into the desired filter using the algorithms used in "Digital Frequency Transformations". Each prototype SOS section is transformed separately. When `FilterType` is 'Lowpass' or 'Highpass', the resulting filter remains a Direct Form II Transposed cascade of second order sections. If the `FilterType` is 'Bandpass' or 'Bandstop', the resulting filter is a cascade of Direct Form II Transposed cascade of fourth order sections.

## Version History

Introduced in R2014a

**R2023a: Support for normalized frequencies**

When you set the `NormalizedFrequency` property to `true`, you can specify the filter passband frequency, center frequency, and the filter bandwidth in normalized frequency units (0 to 1).

When you set the `NormalizedFrequency` property to `true` while creating the object, the frequency values are automatically set to normalized frequency units using the default sample rate of 44100 Hz.

```
vbIIRFilter = dsp.VariableBandwidthIIRFilter(NormalizedFrequency=true)
```

```
vbIIRFilter =  
    dsp.VariableBandwidthIIRFilter with properties:
```

```
        FilterType: 'Lowpass'  
        FilterOrder: 8  
        PassbandFrequency: 0.0232  
        PassbandRipple: 1  
        StopbandAttenuation: 60  
        NormalizedFrequency: true
```

When you set the `NormalizedFrequency` property to `true` after you create the object, the frequencies must be manually set to the normalized frequency values before you run the object algorithm.

```
vbIIRFilter = dsp.VariableBandwidthIIRFilter
```

```
vbIIRFilter =  
    dsp.VariableBandwidthIIRFilter with properties:
```

```
        FilterType: 'Lowpass'  
        FilterOrder: 8  
        PassbandFrequency: 512  
        PassbandRipple: 1  
        StopbandAttenuation: 60  
        NormalizedFrequency: false  
        SampleRate: 44100
```

To specify the normalized frequency values, set `NormalizedFrequency` to `true` and manually convert the frequency values in Hz to normalized values using the input sample rate in Hz. For example, if the input sample rate is 44100 Hz, the corresponding values in normalized units are computed using these equations.

```
vbIIRFilter.NormalizedFrequency = true;  
vbIIRFilter.PassbandFrequency = 512/(44100/2)
```

```
vbIIRFilter =  
    dsp.VariableBandwidthIIRFilter with properties:
```

```
        FilterType: 'Lowpass'  
        FilterOrder: 8  
        PassbandFrequency: 0.0232  
        PassbandRipple: 1  
        StopbandAttenuation: 60  
        NormalizedFrequency: true
```

## References

- [1] A. G. Constantinides. "Spectral transformations for digital filters", Proc. Inst. Elect. Eng. Vol. 117, No. 8, 1970, pp. 1585-1590.

### Extended Capabilities

#### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

### See Also

#### Functions

`freqz` | `fvtool` | `impz` | `info` | `coeffs` | `cost` | `grpdelay`

#### Objects

`dsp.SOSFilter` | `dsp.IIRFilter` | `dsp.FIRFilter` | `dsp.AllpoleFilter` | `dsp.VariableBandwidthFIRFilter`

#### Blocks

Variable Bandwidth IIR Filter | Variable Bandwidth FIR Filter

#### Topics

“Analysis Methods for Filter System Objects” on page 3-2

# dsp.VariableFractionalDelay

**Package:** dsp

Delay input by time-varying fractional number of sample periods

## Description

The `dsp.VariableFractionalDelay` System object delays the input signal by a specified number of fractional samples along each channel of the input. The object can also concurrently compute multiple delayed versions (taps) of the same signal. For an example, see “Signal Delay Using Multitap Fractional Delay” on page 4-1285.

The object interpolates the input signal to obtain new samples at noninteger sampling intervals. You can set the “InterpolationMethod” on page 4-0 property to 'Linear', 'FIR', or 'Farrow'. The object supports time-varying delay values. That is, the delay value can vary within a frame from sample to sample. You can also specify the maximum value of the delay by using the “MaximumDelay” on page 4-0 property. Delay values greater than the maximum are clipped to the maximum.

To delay the input by a time-varying fractional number of sample periods:

- 1 Create the `dsp.VariableFractionalDelay` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
vfd = dsp.VariableFractionalDelay
vfd = dsp.VariableFractionalDelay(Name,Value)
```

### Description

`vfd = dsp.VariableFractionalDelay` creates a variable fractional delay System object that delays a discrete-time input by a time-varying fractional number of sample periods, as specified by the second input.

`vfd = dsp.VariableFractionalDelay(Name,Value)` creates a variable fractional delay System object with each specified property set to the specified value. Enclose each property name in single quotes.

Example: `dsp.VariableFractionalDelay('MaximumDelay',50);`

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see *System Design in MATLAB Using System Objects*.

**InterpolationMethod — Method of interpolation**

'Linear' (default) | 'FIR' | 'Farrow'

Method of interpolation, specified as one of the following. Using this method, the object interpolates the signal to obtain new samples at noninteger sampling intervals.

- 'Linear' -- The object uses linear interpolation.
- 'FIR' -- The object implements a polyphase FIR interpolation filter to interpolate values.
- 'Farrow' -- The object uses the LaGrange method to interpolate values.

For more details on these methods, see “Algorithms” on page 4-1295.

**FilterHalfLength — Half-length of FIR interpolation filter**

4 (default) | positive integer in the range [1 65535]

Half-length of FIR interpolation filter, specified as a positive integer in the range [1 65535].

For periodic signals, a larger value of this property, which indicates a higher order filter, yields a better estimate of the delayed output sample. A property value of 4 to 6, which corresponds to a 7th-order to 11th-order filter, is usually adequate.

**Dependencies**

This property applies only when you set the “InterpolationMethod” on page 4-0 property to 'FIR'.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical | fi

**FilterLength — Length of Farrow filter**

4 (default) | integer greater than or equal to 2

Length of the FIR filter implemented using the Farrow structure, specified as an integer greater than or equal to 2. If the length equals 2, the filter performs linear interpolation. The filter length value determines the order of the polynomial used for lagrange interpolation.

Example: 4

Example: 10

**Dependencies**

This property applies only when you set the “InterpolationMethod” on page 4-0 property to 'Farrow'.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**InterpolationPointsPerSample — Number of interpolation points per input sample**

10 (default) | positive integer in the range [2, 65,535]

Number of interpolation points per input sample at which a unique FIR interpolation filter is computed, specified as a positive integer in the range [2 65535].



Example: 20

Example: 5

### Dependencies

This property applies only when you set the “InterpolationMethod” on page 4-0 property to 'FIR'.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### Bandwidth — Normalized input bandwidth

1 (default) | real scalar in the range (0 1]

Normalized input bandwidth at which to constrain the interpolated output samples, specified as a real scalar in the range (0 1]. A value of 1 equals the Nyquist frequency, or half the sampling frequency,  $F_s$ . Use this property to take advantage of the bandlimited frequency content of the input. For example, if the input signal does not have frequency content above  $F_s/4$ , you can specify a value of 0.5.

Example: 0.5

Example: 0.8

### Dependencies

This property applies only when you set the “InterpolationMethod” on page 4-0 property to 'FIR'.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical | fi

### InitialConditions — Initial values in the memory

0 (default) | scalar | 1-by- $N$ -by- $D$  array | 1-by- $N$ -by- $(D+L)$  array

Initial values in the memory of the object, specified as a scalar or an array. The dimensions of this property can vary depending on whether you want fixed or time-varying initial conditions. The object treats each of the  $N$  input columns as a frame containing  $M$  sequential time samples from an independent channel.

For an  $M$ -by- $N$  input matrix,  $U$ , you can set the InitialConditions property as follows :

- To specify fixed initial conditions, specify InitialConditions as a scalar value. The object initializes every sample of every channel in memory using the value you specify.
- The dimensions you specify for time-varying initial conditions depend on the value of the “InterpolationMethod” on page 4-0 property.
  - When InterpolationMethod is set to 'Linear', specify InitialConditions as a 1-by- $N$ -by- $D$ , where  $D$  is the value of the “MaximumDelay” on page 4-0 property.
  - When InterpolationMethod is set to 'FIR' or 'Farrow', specify InitialConditions as a 1-by- $N$ -by- $(D+L)$  array, where  $D$  is the value of the MaximumDelay property. For FIR interpolation,  $L$  is the value of the FilterHalfLength property. For Farrow interpolation,  $L$  equals the floor of half the value of the FilterLength property:  $\text{floor}(\text{FilterLength}/2)$ .

Example: 1

Example: randn(1,3,104)

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**MaximumDelay — Maximum delay**

`100` (default) | integer in the range [0 65535]

Maximum delay that the object can produce for any sample, specified as an integer in the range [0 65535]. The object clips input delay values greater than `MaximumDelay` to that maximum value.

Example: `100`

Example: `10`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**FIRSmallDelayAction — Action to take for small input delay values when object uses FIR interpolation method**

`'Clip to the minimum value necessary for centered kernel'` (default) | `'Switch to linear interpolation if kernel cannot be centered'`

Action taken for small input delay values when the object uses the FIR interpolation method.

**Dependencies**

This property applies only when you set the `InterpolationMethod` property to `'FIR'`.

**FarrowSmallDelayAction — Action to take for small input delay values when object uses Farrow interpolation method**

`'Clip to the minimum value necessary for centered kernel'` (default) | `'Use off-centered kernel'`

Action taken for small input delay values when the object uses the farrow interpolation method.

**Dependencies**

This property applies only when you set the `InterpolationMethod` property to `'Farrow'`.

**Fixed-Point Properties****RoundingMethod — Rounding method for fixed-point operations**

`'Zero'` (default) | `'Ceiling'` | `'Convergent'` | `'Floor'` | `'Nearest'` | `'Round'` | `'Simplest'`

Rounding mode for fixed-point operations, specified as one of the following:

- `'Zero'`
- `'Ceiling'`
- `'Convergent'`
- `'Floor'`
- `'Nearest'`
- `'Round'`
- `'Simplest'`

For more details, see rounding mode.

**OverflowAction — Overflow action for fixed-point operations**

`'Wrap'` (default) | `'Saturate'`

Overflow action for fixed-point operations, specified as one of the following:

- 'Wrap' -- The object wraps the result of its fixed-point operations.
- 'Saturate' -- The object saturates the result of its fixed-point operations.

For more details on overflow actions, see overflow mode for fixed-point operations.

### **CoefficientsDataType — Data type of the coefficients**

'Same word length as input' (default) | 'Custom'

Data type of the coefficients in this object, specified as one of the following:

- 'Same word length as input' -- The object specifies the coefficients word length to be the same as that of the input. The fraction length is computed to get the best possible precision.
- 'Custom' -- The coefficients data type is specified as a custom numeric type through the "CustomCoefficientsDataType" on page 4-0 property.

For more information on the coefficients data type this object uses, see the "Fixed Point" on page 4-1291 section.

### **CustomCoefficientsDataType — Coefficient word and fraction lengths**

numericType([],32) (default)

Coefficient word and fraction lengths, specified as an autosigned numeric type with a word length of 32.

Example: numericType([],16)

#### **Dependencies**

This property applies only when you set "CoefficientsDataType" on page 4-0 to 'Custom'.

### **ProductPolynomialValueDataType — Data type of the product polynomial value**

'Same as first input' (default) | 'Custom'

Data type of the product polynomial value, specified as one of the following:

- 'Same as first input' -- The object specifies the product polynomial value data type to be the same as that of the data input.
- 'Custom' -- The product polynomial value data type is specified as a custom numeric type through the "CustomProductPolynomialValueDataType" on page 4-0 property.

For more information on the product polynomial value data type this object uses, see the "Fixed Point" on page 4-1291 section.

#### **Dependencies**

This property applies when you set "InterpolationMethod" on page 4-0 to 'Farrow'.

### **CustomProductPolynomialValueDataType — Word and fraction lengths of product polynomial value**

numericType([],32,10) (default)

Word and fraction lengths of the product polynomial value, specified as an autosigned numeric type with a word length of 32 and a fraction length of 10.

Example: `numericity([],30,5)`

**Dependencies**

This property applies only when you set “InterpolationMethod” on page 4-0 to 'Farrow' and “ProductPolynomialValueDataType” on page 4-0 to 'Custom'.

**AccumulatorPolynomialValueDataType — Data type of the accumulator polynomial value**

'Same as first input' (default) | 'Custom'

Data type of the accumulator polynomial value, specified as one of the following:

- 'Same as first input' -- The object specifies the accumulator polynomial value data type to be the same as that of the data input.
- 'Custom' -- The accumulator polynomial value data type is specified as a custom numeric type through the “CustomAccumulatorPolynomialValueDataType” on page 4-0 property.

For more information on the accumulator polynomial value data type that this object uses, see the “Fixed Point” on page 4-1291 section.

**Dependencies**

This property applies when you set “InterpolationMethod” on page 4-0 to 'Farrow'.

**CustomAccumulatorPolynomialValueDataType — Word and fraction lengths of accumulator polynomial value**

`numericity([],32,10)` (default)

Word and fraction lengths of the accumulator polynomial value, specified as an autosigned numeric type with a word length of 32 and a fraction length of 10.

Example: `numericity([],30,5)`

**Dependencies**

This property applies only when you set “InterpolationMethod” on page 4-0 to 'Farrow' and “AccumulatorPolynomialValueDataType” on page 4-0 to 'Custom'.

**MultiplicandPolynomialValueDataType — Data type of multiplicand polynomial value**

'Same as first input' (default) | 'Custom'

Data type of multiplicand polynomial value, specified as one of the following:

- 'Same as first input' -- The object specifies the multiplicand polynomial value data type to be the same as that of the data input.
- 'Custom' -- The multiplicand polynomial value data type is specified as a custom numeric type through the “CustomMultiplicandPolynomialValueDataType” on page 4-0 property.

For more information on the multiplicand polynomial value data type that this object uses, see the “Fixed Point” on page 4-1291 section.

**Dependencies**

This property applies when you set “InterpolationMethod” on page 4-0 to 'Farrow'.

### CustomMultiplicandPolynomialValueDataType — Word and fraction lengths of multiplicand polynomial value

`numerictype([],32,10)` (default)

Word and fraction lengths of the multiplicand polynomial value, specified as an autosigned numeric type with a word length of 32 and a fraction length of 10.

Example: `numerictype([],30,5)`

#### Dependencies

This property applies only when you set “InterpolationMethod” on page 4-0 to 'Farrow' and “MultiplicandPolynomialValueDataType” on page 4-0 to 'Custom'.

### ProductDataType — Data type of product output

'Same as first input' (default) | 'Custom'

Data type of the product output in this object, specified as one of the following:

- 'Same as first input' -- The object specifies the product output data type to be the same as that of the data input.
- 'Custom' -- The product output data type is specified as a custom numeric type through the “CustomProductDataType” on page 4-0 property.

For more information on the product output data type, see “Multiplication Data Types” and the “Fixed Point” on page 4-1291 section.

### CustomProductDataType — Word and fraction lengths of product data type

`numerictype([],32,10)` (default)

Word and fraction lengths of the product data type, specified as an autosigned numeric type with a word length of 32 and a fraction length of 10.

Example: `numerictype([],30,5)`

#### Dependencies

This property applies only when you set “ProductDataType” on page 4-0 to 'Custom'.

### AccumulatorDataType — Data type of accumulation operation

'Same as product' (default) | 'Same as first input' | 'Custom'

Data type of an accumulation operation in this object, specified as one of the following:

- 'Same as product' -- The object specifies the accumulator data type to be the same as that of the product output data type.
- 'Same as first input' -- The object specifies the accumulator data type to be the same as that of the data input.
- 'Custom' -- The accumulator data type is specified as a custom numeric type through the “CustomAccumulatorDataType” on page 4-0 property.

For more information on the accumulator data type this object uses, see the “Fixed Point” on page 4-1291.

### CustomAccumulatorDataType — Word and fraction lengths of accumulator data type

`numerictype([],32,10)` (default)

Word and fraction lengths of the accumulator data type, specified as an autosigned numeric type with a word length of 32 and a fraction length of 10.

Example: `numerictype([],30,5)`

### Dependencies

This property applies only when you set “AccumulatorDataType” on page 4-0 to 'Custom'.

### OutputDataType — Data type of object output

'Same as accumulator' (default) | 'Same as first input' | 'Custom'

Data type of the object output, specified as one of the following:

- 'Same as accumulator' -- The object specifies the output data type to be the same as that of the accumulator output data type.
- 'Same as first input' -- The object specifies the output data type to be the same as that of the data input.
- 'Custom' -- The output data type is specified as a custom numeric type through the “CustomOutputDataType” on page 4-0 property.

For more information on the output data type this object uses, see the “Fixed Point” on page 4-1291 section.

### CustomOutputDataType — Word and fraction lengths of output data type

`numerictype([],32,10)` (default)

Word and fraction lengths of the output data type, specified as an autosigned numeric type with a word length of 32 and a fraction length of 10.

Example: `numerictype([],30,5)`

### Dependencies

This property applies only when you set “OutputDataType” on page 4-0 to 'Custom'.

## Usage

### Syntax

```
vfdOut = vfd(input,d)
```

### Description

`vfdOut = vfd(input,d)` delays the input to the variable fractional delay System object by `d` samples. `d` must be less than or equal to the value you specify in the “MaximumDelay” on page 4-0 property of the object.

Delay values greater than the specified maximum delay are clipped appropriately. Each column of the input is treated as an independent channel.

### Input Arguments

#### input — Data input

vector | matrix

Data input, specified as a vector or matrix. The data input must have the same data type as the delay input.

This object supports variable-size input signal, that is, you can change the input frame size (number of rows)  $N$  even after calling the algorithm. However, the number of channels  $L$  must remain constant. For an example, see “Variable-Size Signal Support for Input and Delay Signals” on page 4-1290.

Example: [1 2 3 4;5 1 4 2;2 6 2 3;1 2 3 2;3 4 5 6;1 2 3 1]

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fi

### **d** — Delay input

scalar | vector | matrix |  $N$ -D array

Delay input, specified as a scalar, vector, matrix, or  $N$ -D array. The delay can be an integer or a fractional value. When the delay input is a fractional value, the object interpolates the signal to obtain new samples at noninteger sampling intervals. The delay input must have the same data type as the data input.

This object supports variable-size delay signal, that is, you can change the first dimension of the delay signal even after calling the algorithm. However, the number of channels  $L$  and the number of taps per channel  $P$  must remain constant. For an example, see “Variable-Size Signal Support for Input and Delay Signals” on page 4-1290.

The table shows the effect of the dimension of the delay input on the data input.

| Data Input   | Delay Input  | Output       | Effect of Delay Input on Data Input   |
|--|--------------|--------------|---|
| $N$ -by-1 (one channel with frame size equal to $N$ )      | scalar       | $N$ -by-1    | One delay value applied to the input channel  |
| $N$ -by-1 (one channel with frame size equal to $N$ )      | $N$ -by-1    | $N$ -by-1    | Delay value varies within the frame from sample to sample   |
| $N$ -by-1 (one channel with frame size equal to $N$ )      | 1-by- $P$    | $N$ -by- $P$ | $P$ taps per channel. Each column in the output is a delayed version of the input. The delay value is specified by the corresponding element in the delay input vector. |
| $N$ -by-1 (one channel with frame size equal to $N$ )      | $N$ -by- $P$ | $N$ -by- $P$ | $P$ taps per channel. In addition, delay varies within each frame from sample to sample.  |
| $N$ -by- $L$ ( $L$ channels with frame size equal to $N$ ) | scalar       | $N$ -by- $L$ | One delay value applied to all input channels   |

| Data Input   | Delay Input           | Output                | Effect of Delay Input on Data Input  |
|--|-----------------------|-----------------------|--|
| $N$ -by- $L$ ( $L$ channels with frame size equal to $N$ ) | 1-by- $L$             | $N$ -by- $L$          | Unique delay value for each input channel  |
| $N$ -by- $L$ ( $L$ channels with frame size equal to $N$ ) | $N$ -by-1             | $N$ -by- $L$          | Delay value varies within the frame from sample to sample. Same set of delay values for all channels.                                    |
| $N$ -by- $L$ ( $L$ channels with frame size equal to $N$ ) | $N$ -by- $L$          | $N$ -by- $L$          | Delay value varies within the frame from sample to sample. Different delay values for each input channel.                                |
| $N$ -by- $L$ ( $L$ channels with frame size equal to $N$ ) | 1-by-1-by- $P$        | $N$ -by- $L$ -by- $P$ | $L$ channels. $P$ taps per channel. Same delay for all channels.   |
| $N$ -by- $L$ ( $L$ channels with frame size equal to $N$ ) | 1-by- $L$ -by- $P$    | $N$ -by- $L$ -by- $P$ | $L$ channels. $P$ taps per channel. Delay varies across channels.  |
| $N$ -by- $L$ ( $L$ channels with frame size equal to $N$ ) | $N$ -by-1-by- $P$     | $N$ -by- $L$ -by- $P$ | $L$ channels. $P$ taps per channel. Delay varies within the frame from sample to sample. Same set of delay values for each channel.      |
| $N$ -by- $L$ ( $L$ channels with frame size equal to $N$ ) | $N$ -by- $L$ -by- $P$ | $N$ -by- $L$ -by- $P$ | $L$ channels. $P$ taps per channel. Delay varies within the frame from sample to sample. Different set of delay values for each channel. |

Example: [2 3 4 5]

Example: [2.5]

Example: [5.6]

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fi

### Output Arguments

#### vfdOut — Delayed output

vector | matrix

Delayed output, returned as a vector or matrix. The size, data type, and complexity of the output match the size, data type, and complexity of the data input.

The table shows how the data input and delay input dimensions affect the output dimensions:



| <b>Data Input</b>  | <b>Delay Input</b> | <b>Output</b>         | <b>Effect of Delay Input on Data Input</b>  |
|--|--------------------|-----------------------|---|
| $N$ -by-1 (one channel with frame size equal to $N$ )      | scalar             | $N$ -by-1             | One delay value applied to the input channel  |
| $N$ -by-1 (one channel with frame size equal to $N$ )      | $N$ -by-1          | $N$ -by-1             | Delay value varies within the frame from sample to sample   |
| $N$ -by-1 (one channel with frame size equal to $N$ )      | 1-by- $P$          | $N$ -by- $P$          | $P$ taps per channel. Each column in the output is a delayed version of the input. The delay value is specified by the corresponding element in the delay input vector. |
| $N$ -by-1 (one channel with frame size equal to $N$ )      | $N$ -by- $P$       | $N$ -by- $P$          | $P$ taps per channel. In addition, delay varies within each frame from sample to sample.  |
| $N$ -by- $L$ ( $L$ channels with frame size equal to $N$ ) | scalar             | $N$ -by- $L$          | One delay value applied to all input channels   |
| $N$ -by- $L$ ( $L$ channels with frame size equal to $N$ ) | 1-by- $L$          | $N$ -by- $L$          | Unique delay value for each input channel   |
| $N$ -by- $L$ ( $L$ channels with frame size equal to $N$ ) | $N$ -by-1          | $N$ -by- $L$          | Delay value varies within the frame from sample to sample. Same set of delay values for all channels.   |
| $N$ -by- $L$ ( $L$ channels with frame size equal to $N$ ) | $N$ -by- $L$       | $N$ -by- $L$          | Delay value varies within the frame from sample to sample. Different delay values for each input channel.   |
| $N$ -by- $L$ ( $L$ channels with frame size equal to $N$ ) | 1-by-1-by- $P$     | $N$ -by- $L$ -by- $P$ | $L$ channels. $P$ taps per channel. Same tap for all channels.  |
| $N$ -by- $L$ ( $L$ channels with frame size equal to $N$ ) | 1-by- $L$ -by- $P$ | $N$ -by- $L$ -by- $P$ | $L$ channels. $P$ taps per channel. Taps vary across channels.  |
| $N$ -by- $L$ ( $L$ channels with frame size equal to $N$ ) | $N$ -by-1-by- $P$  | $N$ -by- $L$ -by- $P$ | $L$ channels. $P$ taps per channel. Delay varies within the frame from sample to sample. Same set of delay values for each channel.                                     |

| Data Input   | Delay Input           | Output                | Effect of Delay Input on Data Input  |
|--|-----------------------|-----------------------|--|
| $N$ -by- $L$ ( $L$ channels with frame size equal to $N$ ) | $N$ -by- $L$ -by- $P$ | $N$ -by- $L$ -by- $P$ | $L$ channels. $P$ taps per channel. Delay varies within the frame from sample to sample. Different set of delay values for each channel. |

Example: [0 0 0 0;0 0 0 0;1 0 0 0;5 2 0 0;2 1 3 0;1 6 4 4]

Example: [0 0 0 0;0 0 0 0;0.5 1.0 1.5 2.0;3 1.5 3.5 3.0;3.5 3.5 3.0 2.5;1.5 4.0 2.5 2.5]

Example: [0 0 0 0;0 0 0 0;0 0 0 0;0 0 0 0;0.4 0.8 1.2 1.6]

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fi

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

### Specific to dsp.VariableFractionalDelay

`info` Characteristic information about valid delay range

`generatehdl` Generate HDL code for quantized DSP filter (requires Filter Design HDL Coder)

### Common to All System Objects

`step` Run System object algorithm

`release` Release resources and allow changes to System object property values and input characteristics

`reset` Reset internal states of System object

## Examples

### Signal Delay using Variable Fractional Delay

Delay a signal by a varying fractional number of sample periods.

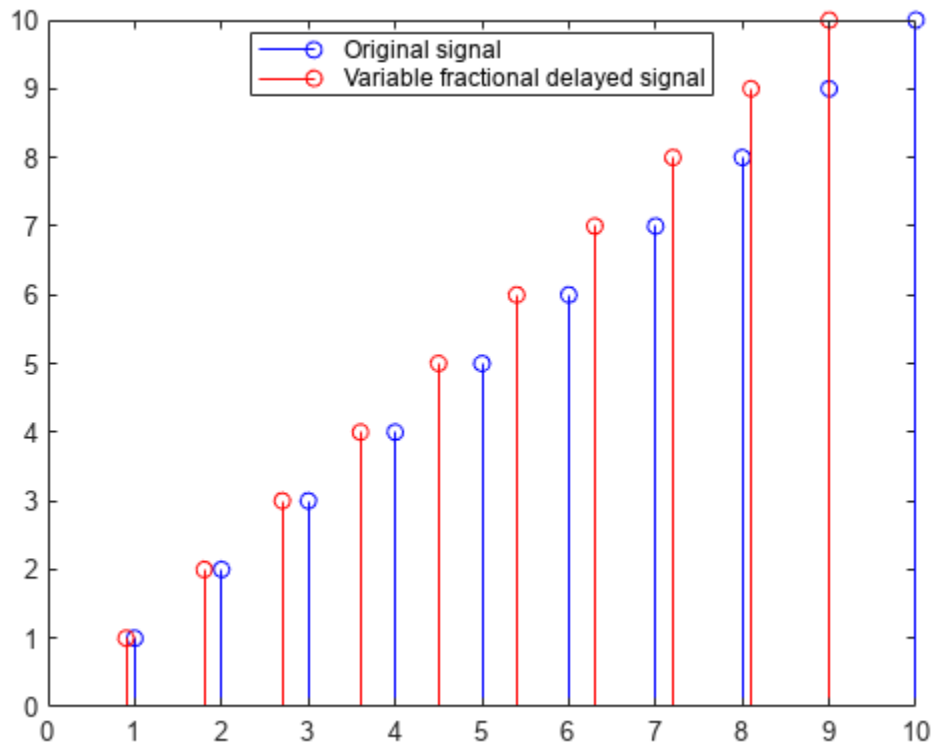
```
sr = dsp.SignalSource;
vfd = dsp.VariableFractionalDelay;
sink = dsp.SignalSink;

for ii = 1:10
    delayedsig = vfd(sr(), ii/10);
    sink(delayedsig);
end

sigd = sink.Buffer;
```

The output `sigd` corresponds to the values of the delayed signal that are sampled at fixed-time intervals. To plot the time instants at which the amplitudes of signal samples are constant, treat the signals as the sampling instants.

```
stem(sr.Signal,1:10,'b')
hold on;
stem(sigd.',1:10,'r');
legend('Original signal', ...
      'Variable fractional delayed signal', ...
      'Location','best')
```



### Signal Delay Using Multitap Fractional Delay

Delay the input signal using the `dsp.VariableFractionalDelay` object. Each delay value is unique and can vary from sample to sample within a frame, and can vary across channels. You can compute multiple delayed versions of the same input signal concurrently by passing a delay input with the appropriate dimension.

Consider the input to be a random signal with one channel and a frame size of 10. Apply a delay of 4.8 and 8.2 samples concurrently.

```
vfd = dsp.VariableFractionalDelay
vfd =
  dsp.VariableFractionalDelay with properties:
```

```
InterpolationMethod: 'Linear'  
InitialConditions: 0  
MaximumDelay: 100
```

Show all properties

```
in = randn(10,1)
```

```
in = 10×1
```

```
0.5377  
1.8339  
-2.2588  
0.8622  
0.3188  
-1.3077  
-0.4336  
0.3426  
3.5784  
2.7694
```

```
delayVec = [4.8 8.2];  
outcase1 = vfd(in,delayVec)
```

```
outcase1 = 10×2
```

```
0 0  
0 0  
0 0  
0 0  
0.1075 0  
0.7969 0  
1.0153 0  
-1.6346 0  
0.7535 0.4301  
-0.0065 1.5746
```

Each channel in the output is delayed by 4.8 and 8.2 samples, respectively. The object uses the 'Linear' interpolation method to compute the delayed value. For more details, see 'Algorithms' in the `dsp.VariableFractionalDelay` object page.

For the same delay vector, if the input has 2 channels, each element of the delay vector is applied on the corresponding channel in the input.

```
release(vfd);  
in = randn(10,2)
```

```
in = 10×2
```

```
-1.3499 0.6715  
3.0349 -1.2075  
0.7254 0.7172  
-0.0631 1.6302  
0.7147 0.4889  
-0.2050 1.0347
```

```
-0.1241    0.7269
 1.4897   -0.3034
 1.4090    0.2939
 1.4172   -0.7873
```

```
outcase2 = vfd(in,delayVec)
```

```
outcase2 = 10x2
```

```
         0         0
         0         0
         0         0
         0         0
    -0.2700         0
    -0.4729         0
     2.5730         0
     0.5677         0
     0.0925     0.5372
     0.5308    -0.8317
```

To compute multiple delayed versions of the two-dimensional input signal, pass the delay vector as a three-dimensional array. The third dimension contains the taps or delays to apply on the signal. If you pass a non-singleton third dimension (1-by-1-by- $P$ ), where  $P$  represents the number of taps, the same tap is applied across all the channels. Pass the delays [4.8 8.2] in the third dimension.

```
clear delayVec;
delayVec(1,1,1) = 4.8;
delayVec(1,1,2) = 8.2;
whos delayVec
```

| Name     | Size  | Bytes | Class  | Attributes |
|----------|-------|-------|--------|------------|
| delayVec | 1x1x2 | 16    | double |            |

`delayVec` is a 1-by-1-by-2 array. Pass the two-dimensional input to the `dsp.VariableFractionalDelay` object with this delay vector.

```
release(vfd);
outcase3 = vfd(in,delayVec)
```

```
outcase3 =
outcase3(:,:,1) =
```

```
         0         0
         0         0
         0         0
         0         0
    -0.2700     0.1343
    -0.4729     0.2957
     2.5730    -0.8225
     0.5677     0.8998
     0.0925     1.4020
     0.5308     0.5981
```

```
outcase3(:,:,2) =
```

```
      0      0
      0      0
      0      0
      0      0
      0      0
      0      0
      0      0
      0      0
-1.0799  0.5372
 2.1580 -0.8317
```

```
whos outcase3
```

| Name     | Size   | Bytes | Class  | Attributes |
|----------|--------|-------|--------|------------|
| outcase3 | 10x2x2 | 320   | double |            |

`outcase3(:, :, 1)` represents the input signal delayed by 4.8 samples. `outcase3(:, :, 2)` represents the input signal delayed by 8.2 samples. The same delay is applied across all the channels.

In addition, if you pass a non-singleton second dimension (1-by- $L$ -by- $P$ ), where  $L$  is the number of input channels, taps vary across channels. Apply the delay vectors [2.3 3.5] and [4.4 5.6] to compute the two delayed versions of the input signal.

```
clear delayVec;
delayVec(1,1,1) = 2.3;
delayVec(1,2,1) = 3.5;
delayVec(1,1,2) = 4.4;
delayVec(1,2,2) = 5.6;
whos delayVec
```

| Name     | Size  | Bytes | Class  | Attributes |
|----------|-------|-------|--------|------------|
| delayVec | 1x2x2 | 32    | double |            |

```
release(vfd);
outcase4 = vfd(in,delayVec)
```

```
outcase4 =
outcase4(:, :, 1) =

      0      0
      0      0
-0.9449      0
 1.7195  0.3357
 1.4183 -0.2680
 0.1735 -0.2451
 0.4814  1.1737
 0.0709  1.0596
-0.1484  0.7618
 1.0055  0.8808
```

```
outcase4(:, :, 2) =

      0      0
      0      0
```

```

         0         0
         0         0
    -0.8099         0
     1.2810     0.2686
     1.6492    -0.0801
     0.2523    -0.4376
     0.4036     1.0824
     0.1629     1.1737

```

whos outcase4

| Name     | Size   | Bytes | Class  | Attributes |
|----------|--------|-------|--------|------------|
| outcase4 | 10x2x2 | 320   | double |            |

`outcase4(:, :, 1)` contains the input signal delayed by the vector [2.3 3.5]. `outcase4(:, :, 2)` contains the input signal delayed by the vector [4.4 5.6].

To vary the delay within a frame from sample to sample, the first dimension of the delay vector ( $N$ -by-1-by- $P$  or  $N$ -by- $L$ -by- $P$ ) must equal the frame size of the input ( $N$ -by- $L$ ). Pass a delay vector of size 10-by-1-by-2.

```

clear delayVec;
delayVec(:,1,1) = 3.1:0.1:4;
delayVec(:,1,2) = 0.1:0.1:1;
whos delayVec

```

| Name     | Size   | Bytes | Class  | Attributes |
|----------|--------|-------|--------|------------|
| delayVec | 10x1x2 | 160   | double |            |

```

release(vfd);
outcase5 = vfd(in,delayVec)

```

```

outcase5 =
outcase5(:, :, 1) =

```

```

         0         0
         0         0
         0         0
    -0.8099     0.4029
     0.8425    -0.2680
     2.1111    -0.4376
     0.4889     0.9911
     0.0925     1.4020
     0.6228     0.5435
    -0.2050     1.0347

```

```

outcase5(:, :, 2) =

```

```

    -1.2149     0.6043
     2.1580    -0.8317
     1.4183     0.1398
     0.2523     1.2650
     0.3258     1.0596
     0.3469     0.7072

```

```
-0.1807    0.9424  
0.1986    0.5208  
1.4816   -0.2437  
1.4090    0.2939
```

Delay varies across each element in a channel. Same set of delay values apply across all channels. `delayVec(:,1,1)` applies to the first delayed signal and `delayVec(:,1,2)` applies to the second delayed signal.

### Variable-Size Signal Support for Input and Delay Signals

`dsp.VariableFractionalDelay` System object supports variable-size input and delay signals. That is, you can change the dimension of the input signal and the delay signal even after calling the algorithm. You can change the dimensions of one or both the signals simultaneously. Together, they must make sure that the number of output channels (number of columns) remains constant.

#### Variable-Size Support for Input Signal

The number of samples in each frame of the input signal can change. However, the number of input channels must remain constant.

Create a `dsp.VariableFractionalDelay` object. Pass an input signal of size [256 1] and a delay of 1.4 to the object algorithm. In subsequent calls to the algorithm, change the input frame size to 128, 512, and 64, respectively

```
vfd = dsp.VariableFractionalDelay;  
vfd(randn(256,1),1.4);  
vfd(randn(128,1),1.4);  
vfd(randn(512,1),1.4);  
vfd(randn(64,1),1.4);
```

The output frame size (number of rows) changes according to the input frame size. The number of output channels in each of these cases is 1.

To change the number of input channels, release the object.

```
release(vfd);
```

Call the algorithm with a two-channel input and vary the input frame size in subsequent calls.

```
vfd(randn(256,2),1.4);  
vfd(randn(128,2),1.4);
```

#### Variable-Size Support for Delay Signal

In addition to the input, the delay signal can also vary. That is, you can change one or both of the dimensions of the delay signal after calling the algorithm. However, the object must make sure that the resulting number of output channels remains constant. The delay signal can be a scalar, vector, matrix, or an N-D array.

```
release(vfd);  
vfd(randn(512,2),randn(512,2));  
vfd(randn(128,2),[1.4 1.7]);
```



```
vfd( randn(256,2) , randn(256,1) );
vfd( randn(128,2) , 1.4 );
```

In each of these cases, the number of output channels is 2. To apply different delays on the input signal, release the object.

```
release(vfd);
vfd( randn(256,1) , randn(256,7) );
vfd( randn(512,1) , randn(512,7) );
vfd( randn(100,1) , randn(100,7) );
vfd( randn(100,1) , randn(1,7) );
```

The output in each of these cases is [256 7], [512 7], [100 7], and [100 7], respectively.

## More About

### Fixed Point

The diagrams in the following sections show the data types used within the dsp.VariableFractionalDelay object for fixed-point signals.

Although you can specify most of these data types, the following data types are computed internally by the object and cannot be directly specified on the object interface.

| Data Type             | Word Length                               | Fraction Length         |
|-----------------------|---|-------------------------|
| vf data type          | Same word length as the coefficients      | Same as the word length |
| HoldInteger data type | Same word length as the input delay value | 0 bits                  |
| Integer data type     | 32 bits                                   | 0 bits                  |

**Note** When the input is fixed point, all internal data types are signed fixed point.

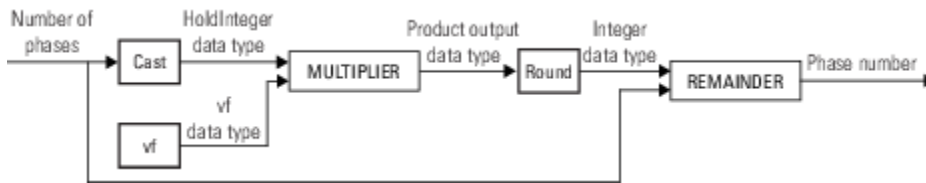
### Linear Interpolation Mode

The following diagram shows the fixed-point data types used by the Linear interpolation mode of the variable fractional delay algorithm.

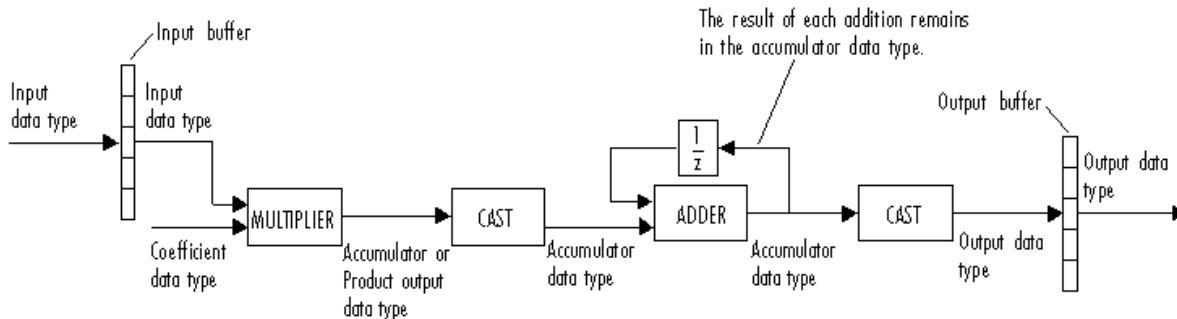


### FIR Interpolation Mode

The following diagram illustrates how the variable fractional delay object selects the arm of the polyphase filter structure that most closely matches the fractional delay value ( $v_f$ ).



The following diagram shows the fixed-point data types used by the variable fractional delay algorithm in the FIR interpolation mode.



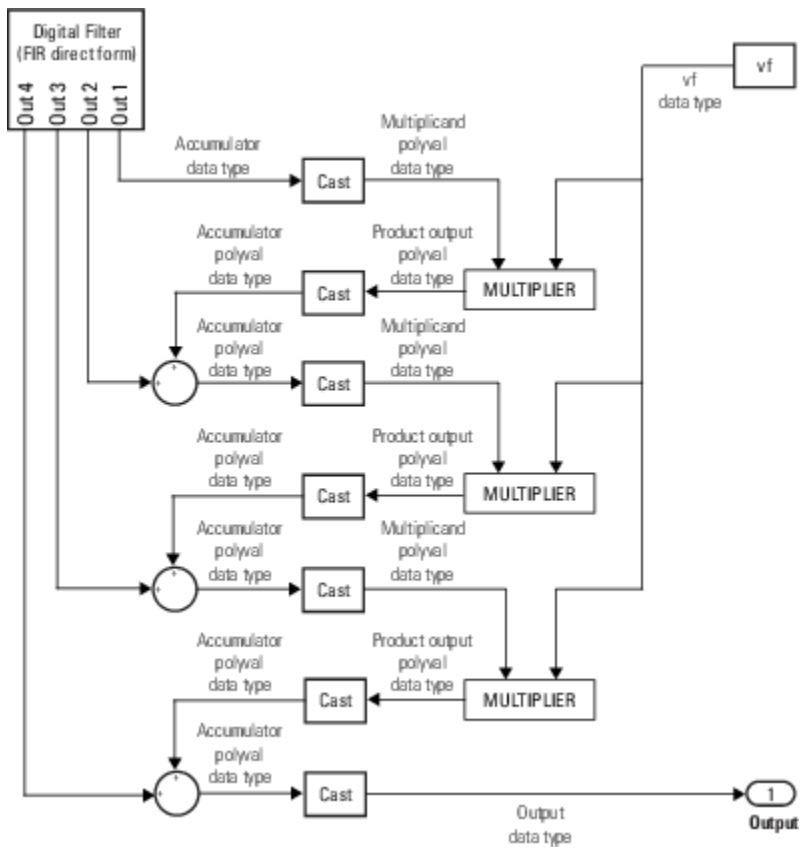
You can set the coefficient, product output, accumulator, and output data types in the object. This diagram shows that input data is stored in the input buffer with the same data type and scaling as the input. The object stores filtered data and any initial conditions in the output buffer using the output data type and scaling that you set.

When at least one of the inputs to the multiplier is real, the output of the multiplier is in the product output data type. When both inputs to the multiplier are complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication, see "Multiplication Data Types".

**Farrow Interpolation Mode**

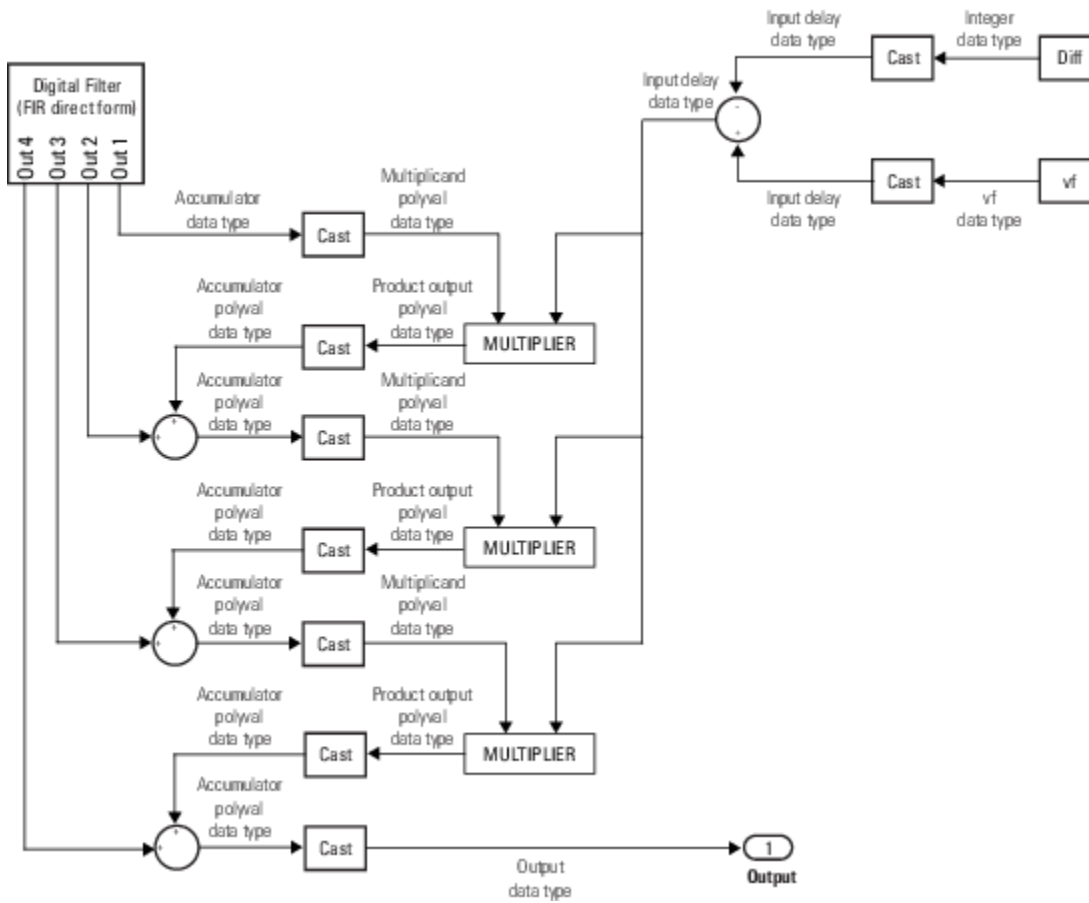
The following diagram shows the fixed-point data types used by the Farrow interpolation mode when:

- Farrow filter length is set to 4
- Farrow small delay action is set to 'Clip to the minimum value necessary for centered kernel'



The following diagram shows the fixed-point data types used by the Farrow interpolation mode when:

- Farrow filter length is set to 4
- Farrow small delay action is set to 'Use off-centered kernel'



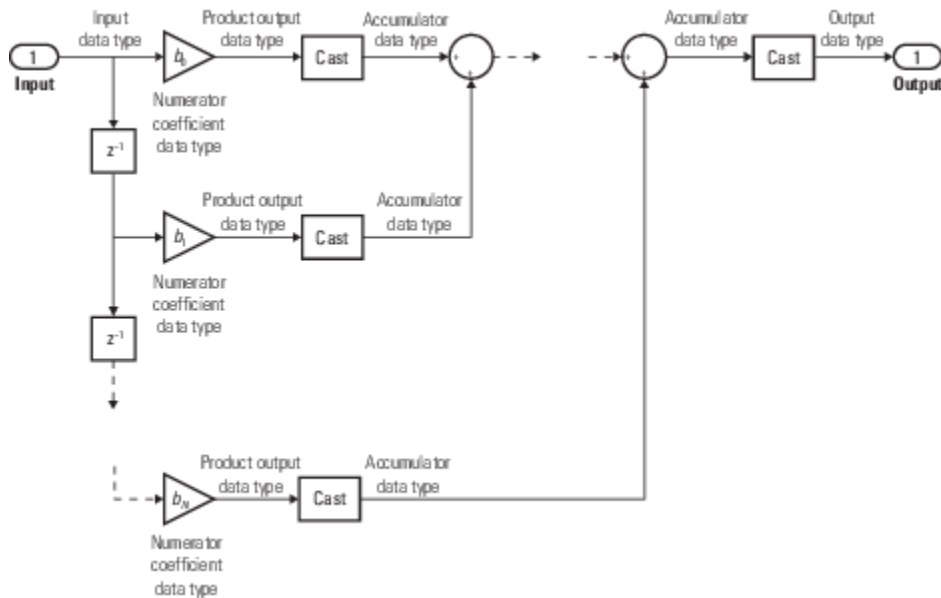
Diff is computed from the integer part of the delay value ( $v_i$ ) and the farrow filter length ( $N$ ) according to the following equation:

$$Diff = v_i - \left( \frac{N - 1}{2} \right)$$

$$Diff \geq 0 \Rightarrow Diff = 0$$

$$Diff < 0 \Rightarrow Diff = -Diff$$

The following diagram shows the fixed-point data types used by the FIR direct form filter.



## Algorithms

When you specify a fractional delay value, the algorithm uses a linear, FIR, or Farrow interpolation method to interpolate signal values at noninteger sample intervals.

### Linear Interpolation Mode

For noninteger delays, at each sample time, the linear interpolation method uses the two samples in memory nearest to the specified delay to compute a value for the sample at that time.

For a vector data input, the output vector,  $y$ , is computed using the following relation:

$$\begin{aligned} v_i &= \text{floor}(v) \\ v_f &= v - v_i \\ y(i) &= U(i - v_i - 1) * v_f + U(i - v_i) * (1 - v_f) \end{aligned}$$

where,

- $i$  -- Index of the current sample
- $v$  -- Fractional delay
- $v_i$  -- Integer part of the delay
- $v_f$  -- Fractional part of the delay
- $U$  -- Input data vector
- $y$  -- Output data vector
- $U(i - v_i)$ ,  $U(i - v_i - 1)$  -- Two samples in memory nearest to the specified delay
- $i - v_i$  -- Distance, in samples, between the current index and the nearest point in the interpolation line.

The variable fractional delay stores the  $D_{max} + 1$  most recent samples received at the input for each channel, where  $D_{max}$  is the maximum delay specified.  $U$  represents the stored samples.

### FIR Interpolation Mode

In the FIR interpolation mode, the variable fractional delay stores the  $D_{max}+P+1$  most recent samples received at the input for each channel, where  $P$  is the specified interpolation filter half-length.

In this mode, the object provides a discrete set of fractional delays:

$$v + \frac{i}{L}, \quad v \geq P - 1, \quad i = 0, 1, \dots, L - 1$$

If  $v$  is less than  $P - 1$ , the behavior depends on the FIR small delay value action setting. You can specify the object's behavior when the input delay value is too small to center the kernel (less than  $P-1$ ), by setting the FIR small delay value action setting:

- **Clip to the minimum value necessary for centered kernel** -- The FIR interpolation method remains in use. The small input delay values are clipped to the smallest value necessary to center the kernel.
- **Switch to linear interpolation if kernel cannot be centered** -- Fractional delays are computed using linear interpolation when the input delay value is less than  $P-1$ .

In the FIR interpolation mode, the algorithm implements a polyphase structure to compute a value for each sample at the specified delay. Each arm of the structure corresponds to a different delay value. The output computed for each sample corresponds to the output of the arm with a delay value nearest to the specified input delay. Therefore, only a discrete set of delays is actually possible. The number of coefficients in each of the  $L$  filter arms of the polyphase structure is  $2P$ . In most cases, using values of  $P$  between 4 and 6 provides you with reasonably accurate interpolation values.

The `designMultirateFIR` function designs the FIR interpolation filter.

For example, when you set the following values:

- Interpolation filter half-length ( $P$ ) to 4
- Interpolation points per input sample to 10
- Normalized input bandwidth to 1
- Stopband attenuation to 80 dB

The filter coefficients are given by:

```
b = designMultirateFIR(10,1,4,80);
```

The algorithm then implements this filter as a polyphase structure.

Increasing the filter half length ( $P$ ) increases the accuracy of the interpolation, but also increases the number of computations performed per input sample. The amount of memory needed to store the filter coefficients increases too. Increasing the interpolation points per sample ( $L$ ) increases the number of representable discrete delay points, but also increases the simulation's memory requirements. The computational load per sample is not affected.

The normalized input bandwidth from 0 to 1 allows you to take advantage of the bandlimited frequency content of the input. For example, if you know that the input signal does not have frequency content above  $F_s/4$ , you can specify 0.5 normalized bandwidth to constrain the frequency content of the output to that range.

---

**Note** You can consider each of the  $L$  interpolation filters to correspond to one output phase of an upsample-by- $L$  FIR filter. Therefore, the normalized input value improves the stopband in critical regions and relaxes the stopband requirements in frequency regions without signal energy.

---

### Farrow Interpolation Mode

In Farrow interpolation mode, the algorithm uses the LaGrange method to interpolate values.

The order of the polynomial used for the interpolation is based on the number of data points used in the Lagrange interpolation. This value is specified in the FilterLength property.

To specify the behavior when the input delay value is too small to center the kernel (less than  $\frac{N}{2} - 1$ ), use the Farrow small delay action setting.

- **Clip to the minimum value necessary for centered kernel** -- The algorithm clips small input delay values to the smallest value necessary to keep the kernel centered. This option yields more accurate interpolation values.
- **Use off-centered kernel** -- The fractional delays are computed using a Farrow filter with an off-centered kernel. The results for input delay values less than  $\frac{N}{2} - 1$  are less accurate than the results achieved by keeping the kernel centered.

When the length of the farrow filter is 2, the filter performs linear interpolation.

## Version History

Introduced in R2012a

### Extended Capabilities

#### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

#### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

This object supports HDL code generation with the Filter Design HDL Coder product. For workflows and limitations, see “Generate HDL Code for Filter System Objects” (Filter Design HDL Coder).

### See Also

#### Functions

info | generatehdl | designFracDelayFIR

#### Objects

dsp.VariableIntegerDelay | dsp.Delay

### **Blocks**

Variable Fractional Delay

### **Topics**

“Fractional Delay Filters Using Farrow Structures”

“Variable-Size Signal Support DSP System Objects”

“System Objects Supported by Fixed-Point Converter App”



# dsp.VariableIntegerDelay

**Package:** dsp

Delay input by time-varying integer number of sample periods

## Description

The `dsp.VariableIntegerDelay` System object delays input by time-varying integer number of sample periods.

To delay the input by a time-varying integer number of sample periods:

- 1 Create the `dsp.VariableIntegerDelay` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
vid = dsp.VariableIntegerDelay
vid = dsp.VariableIntegerDelay(Name,Value)
```

### Description

`vid = dsp.VariableIntegerDelay` returns a variable integer delay System object, `vid`, that delays discrete-time input by a time-varying integer number of sample periods.

`vid = dsp.VariableIntegerDelay(Name,Value)` returns a variable integer delay System object with each specified property set to the specified value.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

### MaximumDelay — Maximum delay

100 (default) | positive integer scalar

Specify the maximum delay the object can produce for any sample. The object clips input delay values greater than the `MaximumDelay` to the `MaximumDelay`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**InitialConditions — Initial values in memory**

0 (default) | scalar | 1-by- $N$ -by- $D$  array

Specify the values with which the object's memory is initialized. The dimensions of this property can vary depending on whether you want fixed or time-varying initial conditions.

For an  $M$ -by- $N$  frame-based input matrix  $U$ , you can set the `InitialConditions` property as follows:

- To specify fixed initial conditions, set the `InitialConditions` property to a scalar value. The object initializes every sample of every channel in memory using the value you specify.
- To specify different time-varying initial conditions for each channel, set the `InitialConditions` property to an array of size 1-by- $N$ -by- $D$ , where  $D$  is the value of the `MaximumDelay` property.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**Usage****Syntax**

```
vidOut = vid(input,d)
```

**Description**

`vidOut = vid(input,d)` delays the input by  $d$  samples, where  $d$  should be less than or equal to the value specified in the `MaximumDelay` property and greater than or equal to 0. Delay values outside this range are clipped appropriately and non-integer delays are rounded to the nearest integer values. Each column of the input is treated as an independent channel

**Input Arguments****input — Data input**

vector | matrix

Data input, specified as a vector or matrix.

This object supports variable-size input signal. That is, you can change the input frame size (number of rows) even after calling the algorithm. However, the number of channels (number of columns) must remain constant.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

**d — Delay input**

scalar | vector | matrix

Delay input, specified as a scalar, vector, or matrix. The delay is an integer value.

For an  $M$ -by-1 or a 1-by- $N$  data input vector, the delay can be a:

- Scalar
- Vector -- The length and the orientation of the delay vector match the length and the orientation of the data input.

For an  $M$ -by- $N$  matrix data input, the delay can be a:

- Column vector -- The length of the vector is  $M$ .
- Row vector -- The length of the vector is  $N$ .
- Matrix -- Delay must be an  $M$ -by- $N$  matrix.

The dimensions of the delay signal can change according to the supported dimensions listed in the table. The table also shows how delay signal is applied to the input signal.

| Data Input   | Delay Input  | Output       | Effect of Delay Input on Data Input   |
|--|--------------|--------------|---|
| $M$ -by-1 (one channel with frame size equal to $M$ )      | scalar       | $M$ -by-1    | One delay value applied to the input channel  |
| $M$ -by-1 (one channel with frame size equal to $M$ )      | $M$ -by-1    | $M$ -by-1    | Delay value varies within the frame from sample to sample   |
| 1-by- $N$ ( $N$ channels with frame size equal to 1)       | scalar       | 1-by- $N$    | One delay value applied to all the $N$ channels   |
| 1-by- $N$ ( $N$ channels with frame size equal to 1)       | 1-by- $N$    | 1-by- $N$    | Unique delay value for each input channel   |
| $M$ -by- $N$ ( $N$ channels with frame size equal to $M$ ) | scalar       | $M$ -by- $N$ | One delay value applied to all input channels   |
| $M$ -by- $N$ ( $N$ channels with frame size equal to $M$ ) | 1-by- $N$    | $M$ -by- $N$ | Unique delay value for each input channel   |
| $M$ -by- $N$ ( $N$ channels with frame size equal to $M$ ) | $M$ -by-1    | $M$ -by- $N$ | Delay value varies within the frame from sample to sample. Same set of delay values for all channels. |
| $M$ -by- $N$ ( $N$ channels with frame size equal to $M$ ) | $M$ -by- $N$ | $M$ -by- $N$ | Unique delay value for each element in the matrix   |

Example: [2 3 4 5]

Example: [2;3;4;5]

Example: [5]

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32

### Output Arguments

#### vidOut – Delayed output

vector | matrix

Delayed output, returned as a vector or matrix. The size, data type, and complexity of the output match the size, data type, and complexity of the data input, “input” on page 4-0 .

The table shows how the data input and delay input dimensions affect the output dimensions:

| Data Input   | Delay Input  | Output       | Effect of Delay Input on Data Input   |
|--|--------------|--------------|---|
| $M$ -by-1 (one channel with frame size equal to $M$ )      | scalar       | $M$ -by-1    | One delay value applied to the input channel  |
| $M$ -by-1 (one channel with frame size equal to $M$ )      | $M$ -by-1    | $M$ -by-1    | Delay value varies within the frame from sample to sample   |
| 1-by- $N$ ( $N$ channels with frame size equal to 1)       | scalar       | 1-by- $N$    | One delay value applied to all the $N$ channels   |
| 1-by- $N$ ( $N$ channels with frame size equal to 1)       | 1-by- $N$    | 1-by- $N$    | Unique delay value for each input channel   |
| $M$ -by- $N$ ( $N$ channels with frame size equal to $M$ ) | scalar       | $M$ -by- $N$ | One delay value applied to all input channels   |
| $M$ -by- $N$ ( $N$ channels with frame size equal to $M$ ) | 1-by- $N$    | $M$ -by- $N$ | Unique delay value for each input channel   |
| $M$ -by- $N$ ( $N$ channels with frame size equal to $M$ ) | $M$ -by-1    | $M$ -by- $N$ | Delay value varies within the frame from sample to sample. Same set of delay values for all channels. |
| $M$ -by- $N$ ( $N$ channels with frame size equal to $M$ ) | $M$ -by- $N$ | $M$ -by- $N$ | Unique delay value for each element in the matrix   |

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fi

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

### Common to All System Objects

- step     Run System object algorithm
- release   Release resources and allow changes to System object property values and input characteristics
- reset     Reset internal states of System object

## Examples

## Delay a Signal with VariableIntegerDelay Object

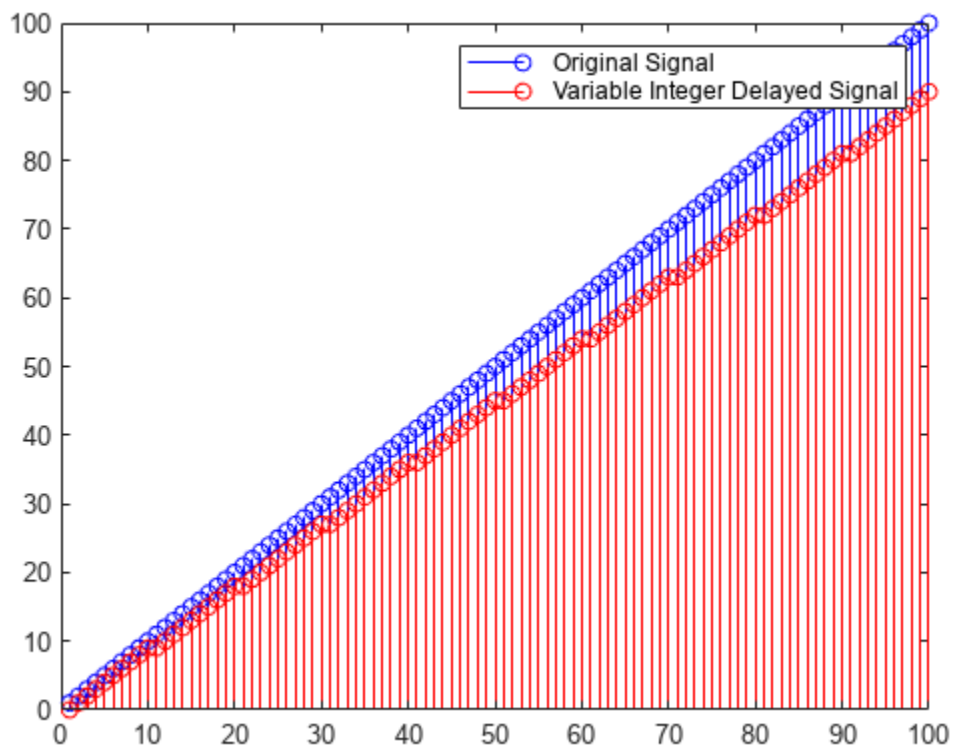
Delay a signal by a varying number of integer sample periods.

```
vid = dsp.VariableIntegerDelay;

yout = zeros(100,1);
x     = (1:100).';

for k=1:10
    range = (k-1)*10+1:k*10;
    yout(range) = vid(x(range),k);
end

stem(x,'b')
hold on;
stem(yout,'r')
legend('Original Signal', 'Variable Integer Delayed Signal')
```



## Algorithms

This object implements the algorithm, inputs, and outputs described on the Variable Integer Delay block reference page.

## **Version History**

**Introduced in R2012a**

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

## **See Also**

### **Objects**

`dsp.VariableFractionalDelay` | `dsp.Delay`

### **Blocks**

Variable Integer Delay

# dsp.VectorQuantizerDecoder

**Package:** dsp

(Removed) Vector quantizer codeword for given index value

---

**Note** dsp.VectorQuantizerDecoder has been removed.

---

## Description

The `VectorQuantizerDecoder` object associates each input index value with a codeword, a column vector of quantized output values defined in the `Codebook` property. Each column of the `Codebook` property is a codeword. When you input multiple index values into this object, the object outputs a matrix of quantized output vectors. This matrix is created by horizontally concatenating the codeword vectors that correspond to each index value.

You can select to enter the code book values via the `Codebook` property or as an input to the object.

To obtain the vector quantizer codeword for a given index value:

- 1 Create the `dsp.VectorQuantizerDecoder` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
vqdec = dsp.VectorQuantizerDecoder  
vqdec = dsp.VectorQuantizerDecoder(Name,Value)
```

### Description

`vqdec = dsp.VectorQuantizerDecoder` creates a vector quantizer decoder System object, `vqdec`, that returns a vector quantizer codeword corresponding to a given, zero-based index value.

`vqdec = dsp.VectorQuantizerDecoder(Name,Value)` returns a vector quantizer decoder, `vqdec`, with each specified property set to the specified value.

### Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

**CodebookSource — Source of codebook values**

Property (default) | Input port

Specify the codebook source as `Property` or `Input port`. When you select `Property`, the object reads the codebook from the `Codebook` property. When you select `Input port`, the object reads the codebook from the input argument `C`.

**Codebook — Matrix of codewords**

[1.5 13.3 136.4 6.8; 2.5 14.3 137.4 7.8; 3.5 15.3 138.4 8.8] (default) | matrix

Specify quantized output values as a  $k$ -by- $N$  matrix, where  $k \geq 1$  and  $N \geq 1$ . Each column of the codebook matrix is a codeword, and each codeword corresponds to an index value. The default is:

$$\begin{bmatrix} 1.5 & 13.3 & 136.4 & 6.8 \\ 2.5 & 14.3 & 137.4 & 7.8 \\ 3.5 & 15.3 & 138.4 & 8.8 \end{bmatrix}$$

The index values are zero based; therefore, the first codeword vector corresponds to an index value of 0, the second codeword vector corresponds to an index value of 1, and so on.

**Tunable:** Yes**Dependencies**

This property applies when you set the `CodebookSource` property to `Property`.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**OutputDataType — Data type of codebook and quantized output**

double (default) | Same as input | single | Custom

Specify the data type of the codebook and quantized output values as: `Same as input`, `double`, `single`, or `Custom`. If this property is set to `Custom`, the data type is specified by the `CustomOutputDataType` property.

**Dependencies**

This property applies only when you set `CodebookSource` to `Property`.

**Fixed-Point Properties****CustomOutputDataType — Output word and fraction lengths**

numerictype(true,16) (default) | numerictype

Specify the output fixed-point type as a signed or unsigned `numerictype` object.

**Dependencies**

This property applies only when you set the `OutputDataType` property to `Custom`.

**Usage****Syntax**
$$Q = \text{vqdec}(I)$$



$Q = \text{vqdec}(I, C)$

### Description

$Q = \text{vqdec}(I)$  returns the quantized output values  $Q$  corresponding to the input indices  $I$ .

$Q = \text{vqdec}(I, C)$  uses input  $C$  as the codebook values when the CodebookSource property is Input port.

### Input Arguments

#### I — Indices

scalar | row vector

Input indices, specified as a scalar or a row vector.

The input to this object is a vector of index values, where  $0 \leq \text{index} < N$  and  $N$  is the number of columns of the codebook matrix. The object sets any index values less than 0 to 0 and any index values greater than or equal to  $N$  to  $N - 1$ .

Data Types: int8 | int16 | int32 | uint8 | uint16 | uint32

#### C — Codebook values

scalar | vector | matrix

Codebook values, specified as a scalar, vector, or matrix.

### Dependencies

This input is enabled only when the CodebookSource property is set to 'Input port'.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fi

### Output Arguments

#### Q — Quantized output values

scalar | vector | matrix

Quantized output values, returned as a scalar, vector, or matrix. Each column of  $Q$  is a codeword whose index in the codebook matches the element specified in the  $I$  matrix.

The codebook is zero based. The first codeword vector corresponds to an index value of 0, the second codeword vector corresponds to an index value of 1, and so on.

For example, if the codebook is

$$\begin{bmatrix} 1.5 & 13.3 & 136.4 & 6.8 \\ 2.5 & 14.3 & 137.4 & 7.8 \\ 3.5 & 15.3 & 138.4 & 8.8 \end{bmatrix}$$

and the  $I$  vector is  $[1 \ 0 \ 3 \ 2 \ 1 \ 0]$ , the output  $Q$  matrix is

$$\begin{bmatrix} 13.3 & 1.5 & 6.8 & 136.4 & 13.3 & 1.5 \\ 14.3 & 2.5 & 7.8 & 137.4 & 14.3 & 2.5 \\ 15.3 & 3.5 & 8.8 & 138.4 & 15.3 & 3.5 \end{bmatrix}$$

If the CodebookSource is set to 'Property', the data type of Q is determined by the OutputDataType property.

If the CodebookSource is set to 'Input port', the output Q has the same data type as the codebook input C.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fi

### Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named obj, use this syntax:

```
release(obj)
```

### Common to All System Objects

|         |  |
|---------|--|
| step    | Run System object algorithm  |
| release | Release resources and allow changes to System object property values and input characteristics |
| reset   | Reset internal states of System object   |

### Examples

#### Determine Vector Quantizer Codeword

Given index values as an input, determine the corresponding vector quantized codewords for a specified codebook.

```
vqdec = dsp.VectorQuantizerDecoder;  
vqdec.Codebook = [1 10 100;2 20 200;3 30 300];  
indices = uint8([1 0 2 0]);  
qout = vqdec(indices)
```

```
qout = 3x4
```

```
    10     1    100     1  
    20     2    200     2  
    30     3    300     3
```

### Algorithms

This object implements the algorithm, inputs, and outputs described on the Vector Quantizer Decoder block reference page. The object properties correspond to the block parameters, except:

- There is no object property that directly corresponds to the **Action for out of range index value** block parameter. The object sets any index values less than 0 to 0 and any index values greater than or equal to N to N-1.

### Version History

Introduced in R2012a

**R2021b: dsp.VectorQuantizerDecoder System object will be removed**

*Warns starting in R2021b*

`dsp.VectorQuantizerDecoder` System object will be removed in a future release.

**R2023a: dsp.VectorQuantizerDecoder System object has been removed**

*Errors starting in R2023a*

The `dsp.VectorQuantizerDecoder` System object has been removed.

**Extended Capabilities****C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

## dsp.VectorQuantizerEncoder

**Package:** dsp

(Removed) Vector quantization encoding

---

**Note** dsp.VectorQuantizerEncoder has been removed.

---

### Description

The VectorQuantizerEncoder object performs vector quantization encoding. The object finds the nearest codeword by computing a distortion based on Euclidean or weighted Euclidean distance.

To perform vector quantization encoding:

- 1 Create the dsp.VectorQuantizerEncoder object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

### Creation

#### Syntax

```
vgenc = dsp.VectorQuantizerEncoder  
vgenc = dsp.VectorQuantizerEncoder(Name,Value)
```

#### Description

vgenc = dsp.VectorQuantizerEncoder returns a vector quantizer encoder System object, vgenc. This object finds a zero-based index of the nearest codeword for each given input column vector.

vgenc = dsp.VectorQuantizerEncoder(Name,Value) returns a vector quantizer encoder System object, vgenc, with each specified property set to the specified value.

### Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

#### CodebookSource — Source of codebook values

Property (default) | Input port

Specify how to determine the codebook values as `Property` or `Input port`.

### Codebook — Matrix of codewords

[1.5 13.3 136.4 6.8; 2.5 14.3 137.4 7.8; 3.5 15.3 138.4 8.8] (default) | matrix

Specify the codebook to which the input column vector or matrix is compared, as a  $k$ -by- $N$  matrix. Each column of the codebook matrix is a codeword, and each codeword corresponds to an index value. The codeword vectors must have the same number of rows as the input. The first codeword vector corresponds to an index value of 0, the second codeword vector corresponds to an index value of 1, and so on. The default is:

$$\begin{bmatrix} 1.5 & 13.3 & 136.4 & 6.8 \\ 2.5 & 14.3 & 137.4 & 7.8 \\ 3.5 & 15.3 & 138.4 & 8.8 \end{bmatrix}$$

**Tunable:** Yes

### Dependencies

This property applies when you set the `CodebookSource` property to `Property`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### DistortionMeasure — Distortion calculation method

Squared error (default) | Weighted squared error

Specify how to calculate the distortion as `Squared error` or `Weighted squared error`. If you set this property to `Squared error`, the object calculates the distortion by evaluating the Euclidean distance between the input column vector and each codeword in the codebook. If you set this property to `Weighted squared error`, the object calculates the distortion by evaluating a weighted Euclidean distance using a weighting factor to emphasize or deemphasize certain input values.

### WeightsSource — Source of weighting factor

`Property` (default) | `Input port`

Specify how to determine weighting factor as `Property` or `Input port`.

### Dependencies

This property applies when you set the `DistortionMeasure` property to `Weighted squared error`.

### Weights — Weighting factor

[1 1 1] (default) | vector

Specify the weighting factor as a vector of length equal to the number of rows of the input.

**Tunable:** Yes

### Dependencies

This property applies when you set the `DistortionMeasure` property to `Weighted squared error` and `WeightsSource` property is `Property`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**TiebreakerRule — Behavior when input column vector is equidistant from two codewords**

Choose the lower index (default) | Choose the higher index

Specify whether to represent the input column vector by the lower index valued codeword or higher indexed valued codeword when an input column vector is equidistant from two codewords. You can set this property to Choose the lower index or Choose the higher index.

**CodewordOutputPort — Enable output of codeword value**

false (default) | true

Set this property to true to output the codeword vectors nearest to the input column vectors.

**QuantizationErrorOutputPort — Enable output of quantization error**

false (default) | true

Set this property to true to output the quantization error value that results when the object represents the input column vector by the nearest codeword.

**OutputIndexDataType — Data type of index output**

int32 (default) | int8 | uint8 | int16 | uint16 | uint32

Specify the data type of the index output as: int8, uint8, int16, uint16, int32, uint32.

**Fixed-Point Properties****RoundingMethod — Rounding method for fixed-point operations**

Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Specify the rounding method as Ceiling, Convergent, Floor, Nearest, Round, Simplest or Zero.

**OverflowAction — Overflow action for fixed-point operations**

Wrap (default) | Saturate

Specify the overflow action as Wrap or Saturate.

**ProductDataType — Source of product word and fraction lengths**

Same as input (default) | Custom

Specify the product fixed-point data type as Same as input or Custom.

**CustomProductDataType — Product word and fraction lengths**

numericType([], 16, 15) (default) | numericType

Specify the product fixed-point type as a scaled numericType object with a Signedness of Auto.

**Dependencies**

This property applies when you set the ProductDataType property to Custom.

**AccumulatorDataType — Source of accumulator word and fraction lengths**

Same as product (default) | Same as input | Custom

Specify the accumulator fixed-point data type as Same as product, Same as input, or Custom.

**CustomAccumulatorDataType — Accumulator word and fraction lengths**

numericType([], 16, 15) (default) | numericType

Specify the accumulator fixed-point type as a scaled `numericType` object with a `Signedness` of `Auto`.

### Dependencies

This property applies when you set the `AccumulatorDataType` property to `Custom`.

## Usage

### Syntax

```
Index = vqenc(Input)
Index = vqenc(Input,Codebook)
Index = vqenc( ____,Weights)
[Index,Codeword] = vqenc( ____)
[Index,Qerr] = vqenc( ____)
```

### Description

`Index = vqenc(Input)` returns `Index`, a scalar or column vector representing the quantization region(s) to which `Input` belongs.

`Index = vqenc(Input,Codebook)` uses the codebook given in input `Codebook`, a  $k$ -by- $N$  matrix with  $N$  codewords each of length  $k$ . This option is available when the `CodebookSource` property is `Input port`.

`Index = vqenc( ____,Weights)` uses the input vector `Weights` to emphasize or de-emphasize certain input values when calculating the distortion measure. `Weights` must be a vector of length equal to the number of rows of `Input`. This option is available when the `DistortionMeasure` property is `Weighted squared error` and the `WeightsSource` property is `Input port`.

`[Index,Codeword] = vqenc( ____)` outputs the `Codeword` values that correspond to each index value when the `CodewordOutputPort` property is `true`. This syntax can be used with any of the previous input syntaxes.

`[Index,Qerr] = vqenc( ____)` outputs the quantization error `Qerr` for each input value when the `QuantizationErrorOutputPort` property is `true`.

### Input Arguments

#### Input — Data input

column vector | matrix

Data input, specified as a column vector of size  $k$ -by-1 or a matrix of size  $k$ -by- $M$ , where  $k$  is the length of each codeword in the codebook.

The number of rows in the data input, the length of the `Weights` vector, and the length of the codeword vector must all be the same value. All inputs to the object must have the same data type.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `fi`

#### Codebook — Codebook values

column vector | matrix

Codebook values, specified as a column vector of size  $k$ -by-1 or a matrix of size  $k$ -by- $N$ , where  $k$  is the length of each codeword and  $N$  is the number of codewords.

The length of the codeword vector, the number of rows in the data input, and the length of the `Weights` vector must all be the same value. All inputs to the object must have the same data type.

**Dependencies**

This input applies when the `CodebookSource` property is `Input port`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `fi`

**Weights — Measure of emphasis**

vector

The object uses the `Weights` vector to emphasize or de-emphasize certain input values when calculating the distortion measure.

The length of the `Weights` vector must equal the number of rows in the data input and the length of the codeword. All inputs to the object must have the same data type.

**Dependencies**

This input applies when the `DistortionMeasure` property is `Weighted squared error` and the `WeightsSource` property is `Input port`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `fi`

**Output Arguments****Index — Indices of nearest codeword vectors**

scalar | row vector

Indices of the nearest codeword vectors, returned as a scalar or a row vector. The object compares each input column vector to the codeword vectors in the codebook matrix. Each column of this codebook matrix is a codeword. The object finds the codeword vector nearest to the input column vector and returns its zero-based index. When the input is a matrix, the indices of the nearest codeword vectors are horizontally concatenated.

The object finds the nearest codeword by calculating the distortion using the method specified in `DistortionMeasure` property.

Data Types: `int32`

**Codeword — Codeword**

column vector | matrix

Codeword values that correspond to each index value, returned as a column vector or a matrix. When the input is a matrix, the corresponding codeword vectors are horizontally concatenated into a matrix.

**Dependencies**

This output is enabled when the `CodewordOutputPort` property is set to `true`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `fi`

**Qerr — Quantization error**

scalar | row vector



Quantization error, returned as a scalar or a row vector. The quantization error results when the object represents the input column vector by its nearest codeword. When the input is a matrix, the quantization error values are horizontally concatenated.

### Dependencies

This output is enabled when the `QuantizationErrorOutputPort` property is set to `true`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `fi`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Common to All System Objects

|                      |  |
|----------------------|--|
| <code>step</code>    | Run System object algorithm  |
| <code>release</code> | Release resources and allow changes to System object property values and input characteristics |
| <code>reset</code>   | Reset internal states of System object   |

## Examples

### Find Indices of Nearest Codewords

Find the indices of nearest codewords based on Euclidean distances.

```
vqenc = dsp.VectorQuantizerEncoder(...
    'Codebook', [-1 -1 1 1;1 -1 -1 1], ...
    'CodewordOutputPort', true, ...
    'QuantizationErrorOutputPort', true, ...
    'OutputIndexDataType', 'uint8');
```

Generate an input signal with some additive noise

```
x = sign(rand(2,40)-0.5) + 0.1*randn(2,40);
[ind, cw, err] = vqenc(x);
plot(cw(1,:), cw(2,:), 'r0', x(1,:), x(2,:), 'g.');
```

## Algorithms

This object implements the algorithm, inputs, and outputs described on the Vector Quantizer Encoder block reference page. The object properties correspond to the block parameters.

## Version History

**Introduced in R2012a**

**R2021b: `dsp.VectorQuantizerEncoder` System object will be removed**

*Warns starting in R2021b*

`dsp.VectorQuantizerEncoder` System object will be removed in a future release.

**R2023a: `dsp.VectorQuantizerEncoder` System object has been removed**

*Errors starting in R2023a*

The `dsp.VectorQuantizerEncoder` System object has been removed.

### **Extended Capabilities**

**C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

# dsp.Window

**Package:** dsp

(Removed) Apply window to input signal

---

**Note** dsp.Window has been removed. Use the window function instead. For more information, see “Compatibility Considerations”.

---

## Description

The Window object applies a window to an input signal.

To apply a window to an input signal:

- 1 Create the dsp.Window object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
win = dsp.Window
win = dsp.Window(WINDOW)
win = dsp.Window(WINDOW,Name,Value)
```

### Description

`win = dsp.Window` returns a window object, `win`, that applies a Hamming window with symmetric sampling.

`win = dsp.Window(WINDOW)` returns a window object with the `WindowFunction` property set to `WINDOW`.

`win = dsp.Window(WINDOW,Name,Value)` returns a window object with the `WindowFunction` property set to `WINDOW` and with other specified properties set to the specified values.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

**WindowFunction — Type of window**

'Hamming' (default) | 'Bartlett' | 'Blackman' | 'Boxcar' | 'Chebyshev' | 'Hann' | 'Hanning' | 'Kaiser' | 'Taylor' | 'Triang'

Specify the type of window to apply. If you run this object in simulation, this property is tunable. When you generate code from a function or script that contains this object and run the generated code, this property is not tunable.

**Tunable:** Yes

**WeightsOutputPort — Enable output of window weights**

false (default) | true

Set this property to `true` to output the window weights. The weights are an  $M$ -by-1 vector with  $M$  equal to the first dimension of the input.

**StopbandAttenuation — Level of stopband attenuation in decibels**

50 (default) | nonnegative scalar

Specify the level of stopband attenuation in decibels, specified as a nonnegative scalar.

**Tunable:** Yes

**Dependencies**

This property only applies when the `WindowFunction` property is 'Chebyshev'.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**Beta — Kaiser window parameter**

10 (default) | scalar

Specify the Kaiser window parameter as a real number. Increasing the absolute value of `Beta` widens the mainlobe and decreases the amplitude of the window sidelobes in the window's frequency magnitude response.

**Tunable:** Yes

**Dependencies**

This property only applies when `WindowFunction` property is 'Kaiser'.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**NumConstantSidelobes — Number of constant sidelobes**

4 (default) | positive integer

Specify the number of constant sidelobes as an integer greater than zero.

**Tunable:** Yes

**Dependencies**

This property only applies when `WindowFunction` property is 'Taylor'.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**MaximumSidelobeLevel — Maximum sidelobe level relative to mainlobe**

-30 (default) | nonpositive scalar

Specify, in decibels, the maximum sidelobe level relative to the mainlobe as a real number less than or equal to zero. The default is -30, which produces sidelobes with peaks 30 dB down from the mainlobe peak.

**Tunable:** Yes

**Dependencies**

This property only applies when WindowFunction property is 'Taylor'.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**Sampling — Window sampling for generalized-cosine windows**

'Symmetric' (default) | 'Periodic'

Specify the window sampling for generalized-cosine windows as 'Symmetric' or 'Periodic'. If you run this object in simulation, this property is tunable. When you generate code from a function or script that contains this object, and run the generated code, this property is not tunable.

**Tunable:** Yes

**Dependencies**

This property only applies when WindowFunction property is 'Blackman', 'Hamming', 'Hann', or 'Hanning'.

**Fixed-Point Properties**

**FullPrecisionOverride — Full precision override for fixed-point arithmetic**

true (default) | false

Specify whether to use full precision rules. If you set FullPrecisionOverride to true, which is the default, the object computes all internal arithmetic and output data types using full precision rules. These rules provide the most accurate fixed-point numerics. It also turns off the display of other fixed-point properties because they do not apply individually. These rules guarantee that no quantization occurs within the object. Bits are added, as needed, to ensure that no roundoff or overflow occurs. If you set FullPrecisionOverride to false, fixed-point data types are controlled through individual fixed-point property settings. For more information, see “Full Precision for Fixed-Point System Objects”.

**RoundingMethod — Rounding method for fixed-point operations**

'Floor' (default) | 'Ceiling' | 'Convergent' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'

Specify the rounding method.

**Dependencies**

This property applies only if the object is not in full precision mode.

**OverflowAction — Overflow action for fixed-point operations**

'Wrap' (default) | 'Saturate'

Specify the overflow action.

**Dependencies**

This property applies only if the object is not in full precision mode.

**WindowDataType — Window word and fraction lengths**

'Same word length as input' (default) | 'Custom'

Specify the window fixed-point data type.

**CustomWindowDataType — Window word and fraction lengths**

numerictype([],16,15) (default) | numerictype

Specify the window fixed-point type as a numerictype object with a Signedness of Auto.

**Dependencies**

This property applies when you set the WindowDataType property to 'Custom'.

**ProductDataType — Product word and fraction lengths**

'Full precision' (default) | 'Same as input' | 'Custom'

Specify the product fixed-point data type as one of 'Full precision', 'Same as input', or 'Custom'.

**CustomProductDataType — Product word and fraction lengths**

numerictype([],16,15) (default) | numerictype

Specify the product fixed-point type as a scaled numerictype object with a Signedness of Auto.

**Dependencies**

This property applies when you set the ProductDataType property to 'Custom'.

**OutputDataType — Output data type**

'Same as product' (default) | 'Same as input' | 'Custom'

Specify the output fixed-point data type as one of 'Same as product', 'Same as input', 'Custom'.

**CustomOutputDataType — Output word and fraction lengths**

numerictype([],16,15) (default) | numerictype

Specify the output fixed-point type as a numerictype object with a Signedness of Auto.

**Dependencies**

This property applies when you set the OutputDataType property to Custom.

**Usage****Syntax**

```
Y = win(X)
[Y,W] = win(X)
```

**Description**

Y = win(X) generates the windowed output, Y, of the input, X, using the specified window.

[Y,W] = win(X) returns the window values W when the WeightsOutputPort property is true.

## Input Arguments

### X — Data input

vector | matrix

Data input, specified as a vector or a matrix.

This object supports only frame-based processing. To see the effect of the window, the data must have a frame size of at least 2 in each channel.

When the input is fixed-point, it must be signed fixed point with power-of-two slope and zero bias.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `fi`

## Output Arguments

### Y — Windowed output

vector | matrix

Windowed output, returned as vector or a matrix.

When the input is an integer, the word length and fraction length of the output, Y is calculated using the following:

$$Y_{WL} = 2X_{WL}$$

$$Y_{FL} = X_{WL} + X_{FL} - 1$$

where,

- $Y_{WL}$  -- Output word length.
- $Y_{FL}$  -- Output fraction length.
- $X_{WL}$  -- Input word length.
- $X_{FL}$  -- Input fraction length. In case of signed integers, this value is 0.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `fi`

### W — Window values

column vector

Window values, returned as a column vector. The number of elements in the column vector is equal to the frame size (number of rows) of the input signal.

When the input is an integer, the word length and fraction length of the output, W is calculated using the following:

$$W_{WL} = X_{WL}$$

$$W_{FL} = Y_{FL} - X_{FL}$$

where,

- $W_{WL}$  -- Window word length.
- $W_{FL}$  -- Window fraction length.
- $Y_{FL}$  -- Output fraction length.

- $X_{FL}$  -- Input fraction length. In case of signed integers, this value is 0.

### Dependencies

This output appears only when the `WeightsOutputPort` property is set to `true`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `fi`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Common to All System Objects

|                      |  |
|----------------------|--|
| <code>step</code>    | Run System object algorithm  |
| <code>release</code> | Release resources and allow changes to System object property values and input characteristics |
| <code>reset</code>   | Reset internal states of System object   |

## Examples

### Apply Hamming Window

```
win = dsp.Window( ...  
    'WindowFunction','Hamming', ...  
    'WeightsOutputPort',true);  
x = rand(64,1);  
[y,w] = win(x);
```

View the window's time and frequency domain responses.

```
wvtool(w)
```

## Algorithms

This object implements the algorithm, inputs, and outputs described on the Window Function block reference page. The object properties correspond to the block parameters, except:

- **Operation** — The `dsp.Window` object does not support the `Generate window` option.
- **Operation** — The `Generate and apply window` option on the block corresponds to the `WeightsOutputPort` property set to `true` on the `dsp.Window` object.
- The `dsp.Window` object only supports frame-based processing.

## Version History

**Introduced in R2012a**

**R2021b: `dsp.Window` System object will be removed**

*Warns starting in R2021b*



`dsp.Window` System object will be removed in a future release. Use the `window` function instead.

### R2023a: `dsp.Window` System object has been removed

*Errors starting in R2023a*

The `dsp.Window` System object has been removed. Use the `window` function instead.

### Update Code

This table shows how to update existing code to use the `window` function.

| Discouraged Usage   | Recommended Replacement   |
|---|---|
| <p><b>Hamming window</b></p> <pre>N = 64; win = dsp.Window( ...     WindowFunction='Hamming', ...     WeightsOutputPort=true); x = rand(N,1); [y,w] = win(x);</pre> <p>The object designs the window and applies it to the input data. <code>y</code> stores the windowed data.</p> <p>Visualize the window using <b>WVTool</b>.</p> <pre>wvtool(w)</pre> | <p><b>Hamming window</b></p> <pre>wfn = window(@hamming,N); isequal(w,wfn)</pre> <pre>ans = 1x1     1</pre> <p><code>wvtool(wfn)</code></p> <p>Apply the window to the data.</p> <pre>yfn = wfn.*x; isequal(y,yfn)</pre> <pre>ans = 1x1     1</pre> |

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This object has no tunable properties for code generation.
- See “System Objects in MATLAB Code Generation” (MATLAB Coder).

## See Also

### Functions

`window`

### Objects

`dsp.FFT`

### Tools

**WVTool**

# dsp.ZeroCrossingDetector

**Package:** dsp

Detect zero crossings

## Description

The `dsp.ZeroCrossingDetector` System object counts the number of times the signal crosses zero, or changes sign. To detect if a signal in a given channel crosses zero, the object looks for the following conditions, where,  $x_{i-l}$  is the current signal value and  $x_{i-1}$  is the previous signal value:

- $x_i < 0$  and  $x_{i-1} > 0$
- $x_i > 0$  and  $x_{i-1} < 0$
- For some positive integer  $L$ ,  $x_i < 0$ ,  $x_{i-l} = 0$ , and  $x_{i-L-1} > 0$ , where  $0 \leq l \leq L$ .
- For some positive integer  $L$ ,  $x_i > 0$ ,  $x_{i-l} = 0$ , and  $x_{i-L-1} < 0$ , where  $0 \leq l \leq L$ .

For the first input value,  $x_{i-1}$  and  $x_{i-2}$  are zero.

To count the number of times a signal crosses zero or changes sign:

- 1 Create the `dsp.ZeroCrossingDetector` object.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
zcd = dsp.ZeroCrossingDetector
```

### Description

`zcd = dsp.ZeroCrossingDetector` returns a zero crossing detection object that calculates the number of times the signal crosses zero.

### Usage

### Syntax

```
zcdOut = zcd(input)
```

### Description

`zcdOut = zcd(input)` calculates the number of zero crossings of the input. Each column of the input is treated as an independent channel.

## Input Arguments

### input — Data input

vector | matrix

Data input whose zero crossings are counted by the object, specified as a vector or a matrix.

Example: `rand(20,1) - 0.3`

Example: `rand(20,2) - 0.3`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

## Output Arguments

### zcdOut — Number of zero crossings

scalar | row vector

Number of zero crossings in the data input, returned as one of the following:

- scalar -- When the data input is a column vector, the scalar output is the number of zero crossings in the data input..
- row vector -- When the data input is a matrix, each element in the row vector output is the number of zero crossings in the corresponding column of the data input.

Example: 10

Example: [9,6]

Data Types: `uint32`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Common to All System Objects

|                      |  |
|----------------------|--|
| <code>step</code>    | Run System object algorithm  |
| <code>release</code> | Release resources and allow changes to System object property values and input characteristics |
| <code>reset</code>   | Reset internal states of System object   |

## Examples

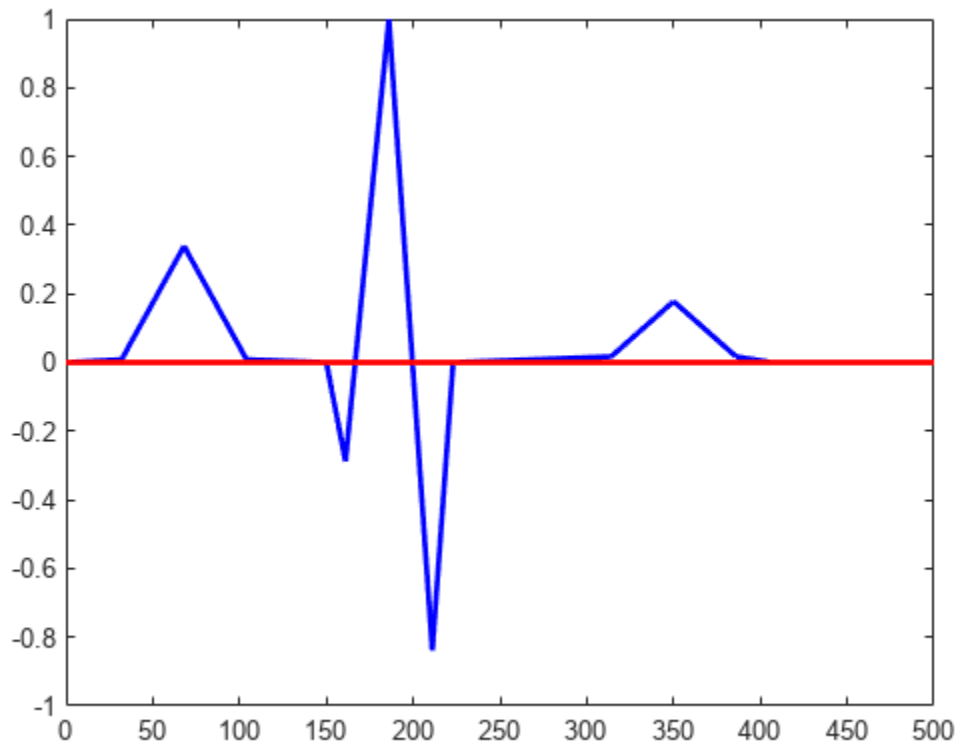
### Determine Number of Zero Crossings

Find the number of zero crossings in electrocardiogram data.

```
EcgData = ecg(500)';
zcd = dsp.ZeroCrossingDetector;
numZeroCross = zcd(EcgData)
```

```
numZeroCross = uint32  
4
```

```
plot(1:500,EcgData,'b',[0 500],[0 0],'r','linewidth',2)
```



## Version History

Introduced in R2012a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

## See Also

### Objects

`dsp.PhaseExtractor`

# dsp.ZoomFFT

**Package:** dsp

High-resolution FFT of a portion of a spectrum

## Description

The `dsp.ZoomFFT` System object computes the fast Fourier Transform (FFT) of a signal over a portion of frequencies in the Nyquist interval. By setting an appropriate decimation factor  $D$ , and sampling rate  $F_s$ , you can choose the bandwidth of frequencies to analyze  $BW$ , where  $BW = F_s/D$ . You can also select a specific range of frequencies to analyze in the Nyquist interval by choosing the center frequency of the desired band.

The resolution of a signal is the ratio of  $F_s$  and the FFT length ( $L$ ). Using zoom FFT, you can retain the same resolution you would achieve with a full-size FFT on your original signal by computing a small FFT on a shorter signal. The shorter signal comes from decimating the original signal. The savings come from being able to compute a much shorter FFT while achieving the same resolution. For a decimation factor of  $D$ , the new sampling rate,  $F_{sd}$ , is  $F_s/D$ , and the new frame size (and FFT length) is  $L_d = L/D$ . The resolution of the decimated signal is  $F_{sd}/L_d = F_s/L$ . To achieve a higher resolution of the shorter band, use the original FFT length,  $L$ , instead of the decimated FFT length,  $L_d$ .

To compute the FFT of a portion of the spectrum:

- 1 Create the `dsp.ZoomFFT` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
zfft = dsp.ZoomFFT
zfft = dsp.ZoomFFT(d)
zfft = dsp.ZoomFFT(d, Fc)
zfft = dsp.ZoomFFT(d, Fc, Fs)
zfft = dsp.ZoomFFT(Name, Value)
```

### Description

`zfft = dsp.ZoomFFT` creates a zoom FFT System object, `zfft`, that performs an FFT on a portion of the input signal's frequency range. The object determines the frequency range over which to perform the FFT using the specified center frequency and decimation factor values.

`zfft = dsp.ZoomFFT(d)` creates a zoom FFT object with the "DecimationFactor" on page 4-0 property set to `d`.

`zfft = dsp.ZoomFFT(d, Fc)` creates a zoom FFT object with the “DecimationFactor” on page 4-0 property set to `d`, and the “CenterFrequency” on page 4-0 property set to `Fc`.

`zfft = dsp.ZoomFFT(d, Fc, Fs)` creates a zoom FFT object with the “DecimationFactor” on page 4-0 property set to `d`, the “CenterFrequency” on page 4-0 property set to `Fc`, and the “SampleRate” on page 4-0 property set to `Fs`.

`zfft = dsp.ZoomFFT(Name, Value)` creates a zoom FFT object with each specified property set to the specified value. Enclose each property name in single quotes. You can use this syntax with any previous input argument combinations.

Example: `zfft = dsp.ZoomFFT(2, 2e3, 48e3, 'FFTLength', 64);`

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see *System Design in MATLAB Using System Objects*.

### **DecimationFactor — Decimation factor**

2 (default) | positive integer

Decimation factor  $D$ , specified as a positive integer. This value specifies the factor by which the object reduces the bandwidth of the input signal. The number of rows in the input signal must be a multiple of the decimation factor.

Example: 4

Example: 8

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **CenterFrequency — Center frequency**

0 (default) | real scalar

Center frequency of the desired band in Hz, specified as a real scalar in the range  $(-\text{SampleRate}/2, \text{SampleRate}/2)$ .

Example: 0.5

Example: 10

**Tunable:** Yes

Data Types: `single` | `double`

### **FFTLength — FFT length**

[] (default) | positive integer

FFT length, specified as a positive integer. The FFT length must be greater than or equal to the ratio of the frame size (number of input rows) and the decimation factor,  $L/D$ . The default, [], specifies an FFT length that equals the ratio,  $L/D$ .

Example: 24

Example: 52

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### SampleRate — Input sample rate

44100 (default) | positive real scalar

Input sample rate in Hz, specified as positive real scalar.

Example: 44100

Example: 48000

Data Types: `single` | `double`

## Usage

### Syntax

```
zfftOut = zfft(input)
```

### Description

`zfftOut = zfft(input)` computes the zoom FFT of the input. Each column of the input is treated as an independent channel. The object computes the FFT of each channel of the input signal independently over time.

### Input Arguments

#### input — Data input

vector | matrix

Data input for which the object computes the zoom FFT, specified as a vector or a matrix of size  $P$ -by- $Q$ . The number of input rows  $P$  can be arbitrary and does not have to be a multiple of the decimation factor  $D$ .

This object supports variable-size input signals, that is, the frame length (number of rows) of the signal can change even when the object is locked. However, the number of channels (columns) must remain constant.

Data Types: `single` | `double`

### Output Arguments

#### zfftOut — Zoom FFT output

vector | matrix

Zoom FFT output, returned as a vector or matrix.

If the FFT length is set to auto, for an input matrix of size  $P$ -by- $Q$ :

- If  $P$  is not a multiple of the decimation factor  $D$ , the output signal has an upper bound size of  $\text{ceil}(P/D)$ -by- $Q$ .
- If  $P$  is a multiple of the decimation factor, then the output is of size  $(P/D)$ -by- $Q$ .

The number of channels (columns) does not change.

If you specify a numeric value in the `FFTLength` property, the output frame size equals the specified FFT length. The output data type matches the input data type.

Data Types: `single` | `double`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Common to All System Objects

|                      |  |
|----------------------|--|
| <code>step</code>    | Run System object algorithm  |
| <code>release</code> | Release resources and allow changes to System object property values and input characteristics |
| <code>reset</code>   | Reset internal states of System object   |

## Examples

### Compute FFT of a Subband Using Zoom FFT

Compute FFT of the [1500 Hz 2500 Hz] subband using zoom FFT for a signal sampled at 48 kHz.

#### Initialization

Set the center frequency to 2 kHz and the bandwidth of interest to 1 kHz. The bandwidth is centered at the center frequency. The decimation factor is the ratio of the input sample rate, 48 kHz, and the bandwidth of interest, 1 kHz. Choose an FFT length of 64. Set the input frame size to be the decimation factor times the FFT length. Create a `dsp.ZoomFFT` object with the specified decimation factor, center frequency, sample rate, and FFT length.

```
Fs = 48e3;  
CF = 2e3;  
BW = 1e3;  
D = Fs/BW;  
fftlen = 64;  
L = D * fftlen;  
zfft = dsp.ZoomFFT(D,CF,Fs,'FFTLength',fftlen);
```

#### Frequencies

The FFT is computed over frequencies starting at 1500 Hz and spaced by  $Fs/(D * fftLen)$  Hz apart, which is the resolution or the minimum frequency that can be discriminated. The number of frequencies at which the zoom FFT is computed equals the FFT length.

```
Fsd = Fs/D;  
F = CF + (Fsd/fftlen)*(0:fftlen-1)-Fsd/2;
```

#### Initialize the Scope

Create an array plot to show the frequencies in `F`.



```

ap = dsp.ArrayPlot('XDataMode','Custom','CustomXData',F,...
    'YLabel','z .* conj(z)','XLabel','Frequency (Hz)','YLimits',[0 1.1e3],...
    'Title',sprintf('Decimation Factor = %d. Center Frequency = %d Hz. Resolution = %f Hz',D, CF

```

### Sine Wave Generator

Create a sine wave with frequencies at 1625 Hz, 2000 Hz, and 2125 Hz.

```

tones = [1625 2000 2125];
sine = dsp.SineWave('SampleRate',Fs,'Frequency',tones,'SamplesPerFrame',L);

```

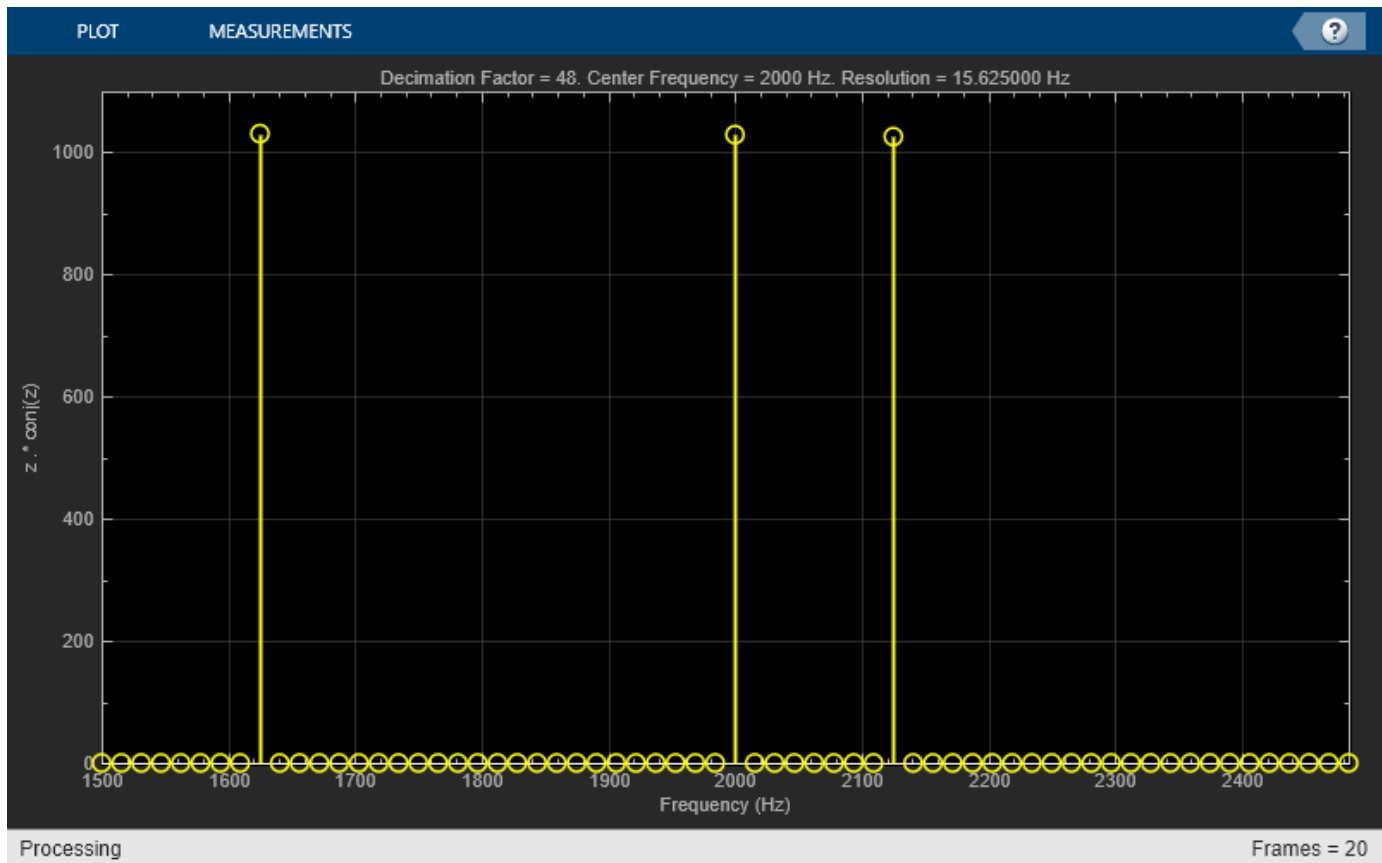
### Streaming

Pass a noisy sine wave with a sample rate of 48 kHz. Compute the zoom FFT of this sine wave in the subband [1500 Hz 2500 Hz]. Rearrange the Fourier transform by shifting the zero-frequency component to the center of the array. View the tones at 1625 Hz, 2000 Hz, and 2125 Hz in the array plot.

```

for i = 1:1000
    x = sum(sine(),2)+1e-1*randn(L,1);
    z = zfft(x);
    z = fftshift(z);
    ap(z.*conj(z));
end

```



### Compute Zoom FFT of Variable-Size Inputs

The `dsp.ZoomFFT` object accepts variable-size inputs as long as the input is a multiple of the decimation factor. The number of input channels cannot change.

Create a `dsp.ZoomFFT` object with a decimation factor of 4, center frequency of 2 kHz, and an input sample rate of 48 kHz. Pass a random input with  $4 \times 64$  rows and 2 columns. Vary the number of rows to  $4 \times 128$  and  $4 \times 32$ . The resulting FFT lengths are 64, 128, and 32, respectively. The size of the outputs is  $[64 \ 2]$ ,  $[128 \ 2]$ , and  $[32 \ 2]$ , respectively.

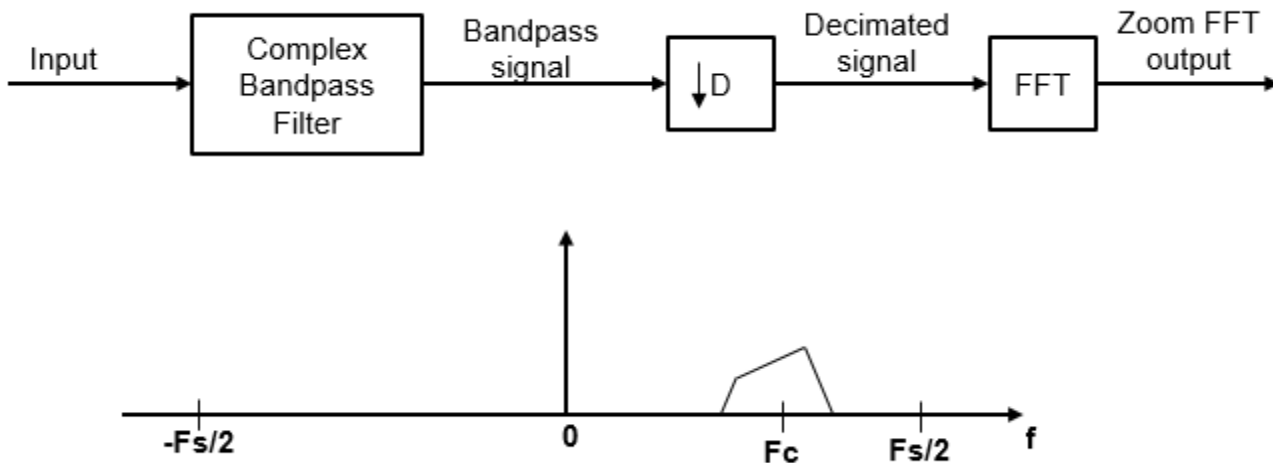
```
zfft = dsp.ZoomFFT(4,2e3,48e3);
y1 = zfft(randn(4*64,2));
y2 = zfft(randn(4*128,2));
y3 = zfft(randn(4*32,2));
```

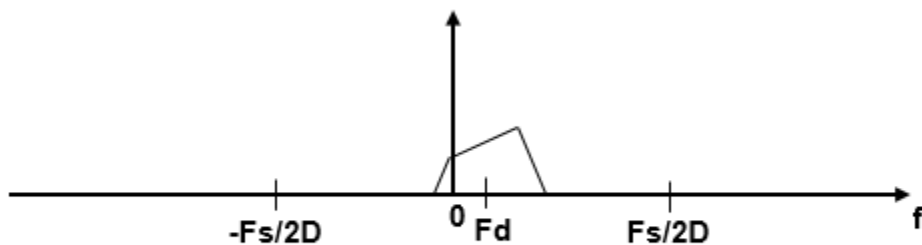
Set the FFT length as 256 and pass variable-size inputs. The size of all the outputs is  $[256 \ 2]$ .

```
release(zfft);
zfft.FFTLength = 256;
y4 = zfft(randn(4*64,2));
y5 = zfft(randn(4*128,2));
y6 = zfft(randn(4*32,2));
```

### Algorithms

The zoom FFT algorithm leverages bandpass filtering before computing the FFT of the signal. The concept of bandpass filtering is that suppose you are interested in the band  $[F1, F2]$  of the original input signal, sampled at the rate  $F_s$  Hz. If you pass this signal through a complex (one-sided) bandpass filter centered at  $F_c = (F1 + F2)/2$ , with the bandwidth  $BW = F2 - F1$ , and then downsample the signal by a factor of  $D = \text{floor}(F_s/BW)$ , the desired band comes down to the baseband.





If  $F_c$  cannot be expressed in the form of  $k \times F_s/D$ , where  $k$  is an integer, then the shifted, decimated spectrum is not centered at DC. In this case, the center frequency gets translated to  $F_d$ .

$$F_d = F_c - (F_s/D) \times \text{floor}((D \times F_c + F_s/2)/F_s)$$

The complex bandpass filter is obtained by first designing a lowpass filter prototype and then multiplying the lowpass coefficients with a complex exponential. This algorithm uses a multirate, multistage FIR filter as the lowpass filter prototype. To obtain the bandpass filter, the coefficients of each stage are frequency shifted. The decimation factor is the cumulative decimation factor of each stage. The complex bandpass filter followed by the decimator are implemented using an efficient polyphase structure. For more details on the design of the complex bandpass filter from the multirate multistage FIR filter prototype, see “Zoom FFT” and “Complex Bandpass Filter Design”.

## Version History

Introduced in R2017b

### R2023a: Support for arbitrary input frame length

This object supports an input signal with an arbitrary frame length, so the input frame length does not have to be a multiple of the decimation factor.

## References

- [1] Harris, F.J. *Multirate Signal Processing for Communication Systems*. Prentice Hall, 2004, pp. 208–209.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

In the code you generate from this object, the zoom FFT output is a fixed-size signal if the input signal is fixed size and is a multiple of the decimation factor. If the input is not a multiple of the decimation factor or if the input is a variable-size signal, then the zoom FFT output is of variable-size.

### See Also

#### Objects

`dsp.FFT` | `dsp.IFFT`

#### Blocks

Zoom FFT | FFT | Magnitude FFT | Short-Time FFT

#### Topics

“Zoom FFT”

“Complex Bandpass Filter Design”

# SpectrumAnalyzerConfiguration

Configure Spectrum Analyzer block

## Description

The `spbscopes.SpectrumAnalyzerConfiguration` object contains the scope configuration information for the Spectrum Analyzer block.

## Creation

`MyScopeConfiguration = get_param(gcbh, 'ScopeConfiguration')` constructs a new spectrum analyzer configuration object. You must first select the block in the model or provide the full path to the block.

## Properties

### Frequently Used

#### NumInputPorts — Number of input ports

"1" (default) | character vector | string scalar

Number of input ports on a scope block, specified as a character vector or string scalar. The maximum number of input ports that you can specify is 96.

#### Scope Window Use

This parameter is available only from the **Property Inspector**. To launch the **Property Inspector**, click the **Modeling** tab of the Simulink model window that contains the Spectrum Analyzer block. In the **Design** section, click the drop-down arrow and select **Property Inspector**. The **Property Inspector** window opens on the right. If you select the Spectrum Analyzer block, the **Property Inspector** window shows the corresponding parameters. In the parameters that show up, specify the **Number of input ports** parameter.

Data Types: char | string

#### InputDomain — Domain of input signal

"Time" (default) | "Frequency"

The domain of the input signal you want to visualize, specified as "Time" or "Frequency". If you visualize time-domain signals, the Spectrum Analyzer transforms the signal to the frequency spectrum based on the algorithm specified in the `Method` property.

#### Scope Window Use

In the **Estimation** tab on the Spectrum Analyzer toolstrip, set **Input Domain** to Time or Frequency.

Data Types: char | string

**SpectrumType — Spectrum type**

"Power" (default) | "Power density" | "RMS"

Spectrum type, specified as one of these:

"Power" — Power spectrum

"Power density" — Power spectral density. The power spectral density is the magnitude squared of the spectrum normalized to a bandwidth of 1 Hz.

"RMS" — Root mean square. The root mean square shows the square root of the mean square. Use this option to view the frequency of voltage or current signals.

**Tunable:** Yes

**Dependency**

To enable this property, set `InputDomain` to "Time".

**Scope Window Use**

In the **Analyzer** tab on the Spectrum Analyzer toolstrip, select **Spectrum**. Click **Spectrum** to select Power, Power Density, or RMS.

To enable these options, set the **Input Domain** on the **Estimation** tab to Time.

Data Types: char | string

**ViewType — Viewer type**

"Spectrum" (default) | "Spectrogram" | "Spectrum and spectrogram"

Viewer type, specified as one of these values:

- "Spectrum" — Show the power spectrum.
- "Spectrogram" — Show frequency content over time. Each line of the spectrogram is one periodogram. Time scrolls from the bottom to the top of the display. The most recent spectrogram update is at the bottom of the display.
- "Spectrum and Spectrogram" — Show spectrum and spectrogram.

To learn more about how the Spectrum Analyzer computes the spectrum and spectrogram, see the "Algorithms" on page 2-1335 section in the Spectrum Analyzer block reference page.

**Tunable:** Yes

**Scope Window Use**

In the **Analyzer** tab on the Spectrum Analyzer toolstrip, select **Spectrum**, **Spectrogram**, or both.

Data Types: char | string

**SampleRateSource — Source of input sample rate**

"Inherited" (default) | "Property"

Source of the input sample rate, specified as one of these:

- "Inherited" — Spectrum Analyzer inherits the input sample rate from the model.

- "Property" — Specify the sample rate input directly using the `SampleRate` property.

#### Scope Window Use

Click the **Analyzer** tab on the Spectrum Analyzer toolstrip. In the **Bandwidth** section, set **Sample Rate (Hz)** to `Inherited` or specify a finite scalar.

Data Types: `char` | `string`

#### SampleRate — Sample rate of input

`"10e3"` (default) | character vector of a positive scalar | string scalar of a positive scalar

Sample rate of the input signal in Hz, specified as a character vector or a string scalar of a positive scalar.

#### Dependency

To enable this property, set `SampleRateSource` to `"Property"`.

#### Scope Window Use

Click the **Analyzer** tab on the Spectrum Analyzer toolstrip. In the **Bandwidth** section, specify **Sample Rate (Hz)** to a finite scalar.

Data Types: `char` | `string`

#### Method — Spectrum estimation method

`"Filter Bank"` (default) | `"Welch"`

Spectrum estimation method, specified as one of the following:

- `"Filter bank"` -- Use an analysis filter bank to estimate the power spectrum. Compared to Welch's method, this method has a lower noise floor, better frequency resolution, lower spectral leakage, and requires fewer samples per update.
- `"Welch"` -- Use Welch's method of averaged modified periodograms.

For more information on these methods, see "Algorithms" on page 2-1335.

**Tunable:** Yes

#### Dependency

To enable this property, set `InputDomain` to `"Time"`.

#### Scope Window Use

In the **Estimation** tab of the Spectrum Analyzer toolstrip, set **Method** to `Filter bank` or `Welch`.

To enable this parameter, set **Input Domain** to `Time` in the **Estimation** tab.

Data Types: `char` | `string`

#### PlotAsTwoSidedSpectrum — Option to plot a two-sided spectrum

`true` (default) | `false`

Option to plot a two-sided spectrum, specified as one of the following:

- `true` — Compute and plot two-sided spectral estimates. When the input signal is complex valued, you must set this property to `true`.

- `false` — Compute and plot one-sided spectral estimates. If you set this property to `false`, then the input signal must be real valued.

When you set this property to `false`, the Spectrum Analyzer uses power-folding. The y-axis values are twice the amplitude that they would be if you were to set this property to `true`, except at 0 and the Nyquist frequency. A one-sided power spectral density (PSD) contains the total power of the signal in the frequency interval from DC to half the Nyquist rate. For more information, see `pwelch`.

**Tunable:** Yes

### Scope Window Use

Click the **Spectrum** tab or the **Spectrogram** tab (if enabled) of the Spectrum Analyzer toolstrip. In the **Trace Options** section, select **Two-Sided Spectrum** to compute and plot two-sided spectral estimates.

Data Types: `logical`

### FrequencyScale — Scale to display frequency

"Linear" (default) | "Log"

Scale to display frequency, specified as one of the following:

- "Linear" — Use a linear scale to display frequencies on the x-axis. To use the "Linear" option, you must also set the `PlotAsTwoSidedSpectrum` property to `true`.
- "Log" — Use a logarithmic scale to display frequencies on the x-axis. To use the "Log" option, you must also set the `PlotAsTwoSidedSpectrum` property to `false`.

**Tunable:** Yes

### Scope Window Use

Click the **Spectrum** tab or the **Spectrogram** tab (if enabled) on the Spectrum Analyzer toolstrip. In the **Scale** section, set the **Frequency Scale** to Linear or Log.

To set the **Frequency Scale** to Log, clear the **Two-Sided Spectrum** check box in the **Trace Options** section in the **Spectrum** or the **Spectrogram** tab (if enabled). If you select the **Two-Sided Spectrum** check box, then you must set the **Frequency Scale** to Linear.

Data Types: `char` | `string`

### PlotType — Plot type to display normal traces

"Line" (default) | "Stem"

Plot type to display normal traces, specified as "Line" or "Stem". Normal traces are traces that display free-running spectral estimates.

**Tunable:** Yes

### Dependencies

To enable this property, set:

- `ViewType` to "Spectrum" or "Spectrum and spectrogram".
- `PlotNormalTrace` to `true`.



**Scope Window Use**

Click the **Analyzer** tab on the Spectrum Analyzer toolstrip, navigate to the **Configuration** section and click **Settings**. In the Spectrum Analyzer Settings window, under **Display and Labels**, set **Plot Type** to Line or Stem.

To enable the **Plot Type**, you must:

- Select **Spectrum** in the **Views** section of the **Analyzer** tab.
- Enable the **Normal Trace** check box in the **Trace Options** section of the **Spectrum** tab.

Data Types: char | string

**AxesScaling — Axes scaling mode**

"Auto" (default) | "Manual" | "OnceAtStop" | "Updates"

Axes scaling mode, specified as one of these:

- "Auto" — The scope scales the axes to fit the data, both during and after simulation.
- "Manual" — The scope does not scale the axes automatically.
- "OnceAtStop" — The scope scales the axes when the simulation stops.
- "Updates" — The scope scales the axes after a specific number of visual updates. It determines the number of updates using the `AxesScalingNumUpdates` property.

**Tunable:** Yes

Data Types: char | string

**AxesScalingNumUpdates — Number of updates before scaling**

"100" (default) | character vector of a positive integer | string scalar of a positive integer

Number of updates before scaling, specified as a character vector or a string scalar of a positive integer.

**Tunable:** Yes

**Dependency**

To enable this property, set `AxesScaling` to "Updates".

Data Types: char | string

**Advanced****RBWSource — Source of resolution bandwidth value**

"Auto" (default) | "Property" | "InputPort"

Source of the resolution bandwidth (RBW) value, specified as one of these:

- "Auto" — The Spectrum Analyzer adjusts the spectral estimation resolution to ensure that there are 1024 RBW intervals over the defined frequency span.
- "Property" — Specify the resolution bandwidth directly using the RBW property.
- "InputPort" — An input port appears on the Spectrum Analyzer block through which you specify the RBW. You can use this option only for frequency inputs.

**Tunable:** Yes

**Scope Window Use**

Click the **Analyzer** tab on the Spectrum Analyzer toolstrip. In the **Bandwidth** section, set **RBW (Hz)** to Auto, Input port or to a positive scalar.

Data Types: char | string

**RBW — Resolution bandwidth**

"9.76" (default) | character vector of a positive scalar | string scalar of a positive scalar

Resolution bandwidth (RBW) in Hz, specified as a character vector or a string scalar of a positive scalar. Specify the value to ensure that there are at least two RBW intervals over the specified frequency span. The ratio of the overall span to RBW satisfies this condition:

$$\frac{span}{RBW} > 2$$

You can specify the overall span in different ways based on how you set the FrequencySpan property.

RBW controls the spectral resolution of the displayed signal. The RBW value determines the spacing between frequencies that can be resolved. A smaller value gives a higher spectral resolution and lowers the noise floor, that is, the Spectrum Analyzer can resolve frequencies that are closer to each other. However, this comes at the cost of a longer sweep time.

**Tunable:** Yes

**Dependency**

To enable this property, set RBWSource to "Property".

**Scope Window Use**

Click the **Analyzer** tab on the Spectrum Analyzer toolstrip. In the **Bandwidth** section, set **RBW (Hz)** to a positive scalar.

Data Types: char | string

**FilterSharpness — Sharpness of lowpass filter**

"0.3" (default) | character vector of a nonnegative scalar in the range [0,1] | string scalar of a nonnegative scalar in the range [0,1]

Sharpness of the prototype lowpass filter, specified as a character vector or a string scalar of a real nonnegative scalar in the range [0,1].

Increasing the filter sharpness decreases the spectral leakage and gives a more accurate power reading.

**Tunable:** Yes

**Dependencies**

To enable this property, set:

- InputDomain to "Time".

- Method to "Filter Bank".

#### Scope Window Use

Click the **Estimation** tab on the Spectrum Analyzer toolstrip. In the **Frequency Resolution** section, move the **Sharpness** slider.

To enable this parameter, set **Input Domain** to Time and **Method** to Filter bank in the **Estimation** tab.

Data Types: char | string

#### FrequencySpan — Frequency span mode

"Full" (default) | "Span and center frequency" | "Start and stop frequencies"

Frequency span mode, specified as one of the following:

- "Full" -- The Spectrum Analyzer computes and plots the spectrum over the entire "Nyquist Frequency Interval" on page 2-1340.
- "Span and center frequency" -- The Spectrum Analyzer computes and plots the spectrum over the interval specified by the Span and CenterFrequency properties.
- "Start and stop frequencies" -- The Spectrum Analyzer computes and plots the spectrum over the interval specified by the StartFrequency and StopFrequency properties.

**Tunable:** Yes

#### Dependency

To enable this property, set InputDomain to "Time".

#### Scope Window Use

Click the **Estimation** tab on the Spectrum Analyzer toolstrip. In the **Frequency Options** section, set **Frequency Span** to Full, Span and Center Frequency, or Start and Stop Frequencies.

To enable the **Frequency Span**, set **Input Domain** to Time.

Data Types: char | string

#### Span — Frequency span to compute spectrum

"10e3" (default) | character vector of a real positive scalar | string scalar of a real positive scalar

Frequency span over which the Spectrum Analyzer computes and plots the spectrum, specified as a character vector or a string scalar of a real positive scalar in Hz. The overall span, defined by this property and the CenterFrequency property, must fall within the "Nyquist Frequency Interval" on page 2-1340.

**Tunable:** Yes

#### Dependency

To enable this property, set:

- InputDomain to "Time".
- FrequencySpan to "Span and center frequency".

Data Types: char | string

### CenterFrequency — Center of frequency span

"0" (default) | character vector of a real scalar | string scalar of a real scalar

Center of the frequency span over which the Spectrum Analyzer computes and plots the spectrum, specified as a character vector or a string scalar of a real scalar in Hz. The overall frequency span, defined by Span and this property, must fall within the “Nyquist Frequency Interval” on page 2-1340.

**Tunable:** Yes

#### Dependency

To enable this property, set:

- InputDomain to "Time".
- FrequencySpan to "Span and center frequency".

#### Scope Window Use

Click the **Estimation** tab on the Spectrum Analyzer toolstrip. In the **Frequency Options** section, set **Frequency Span** to Span and Center Frequency and **Center Frequency (Hz)** to a real scalar.

To enable the **Frequency Span**, set **Input Domain** to Time.

Data Types: char | string

### StartFrequency — Start frequency to compute spectrum

"-5e3" (default) | character vector of a real scalar | string scalar of a real scalar

Starting frequency value in the frequency interval over which the Spectrum Analyzer computes and plots the spectrum, specified as a character vector or a string scalar of a real scalar in Hz. The overall span, which is defined by this property and StopFrequency, must fall within the “Nyquist Frequency Interval” on page 2-1340.

**Tunable:** Yes

#### Dependency

To enable this property, set:

- InputDomain to "Time".
- FrequencySpan to "Start and stop frequencies".

#### Scope Window Use

Click the **Estimation** tab on the Spectrum Analyzer toolstrip. In the **Frequency Options** section, set **Frequency Span** to Start and Stop Frequencies and **Start Frequency (Hz)** to a real scalar.

To enable the **Frequency Span**, set **Input Domain** to Time.

Data Types: char | string

### StopFrequency — Stop frequency to compute spectrum

"5e3" (default) | character vector of a real scalar | string scalar of a real scalar

Ending frequency value in the frequency interval over which the Spectrum Analyzer computes and plots the spectrum, specified as a character vector or a string scalar of a real scalar in Hz. The overall

span, which is defined by this property and the `StartFrequency` property, must fall within the "Nyquist Frequency Interval" on page 2-1340.

**Tunable:** Yes

#### Dependency

To enable this property, set:

- `InputDomain` to "Time".
- `FrequencySpan` to "Start and stop frequencies".

#### Scope Window Use

Click the **Estimation** tab on the Spectrum Analyzer toolstrip. In the **Frequency Options** section, set **Frequency Span** to Start and Stop Frequencies and **Stop Frequency (Hz)** to a real scalar.

To enable the **Frequency Span**, set **Input Domain** to Time.

Data Types: char | string

#### OverlapPercent — Overlap percentage

"0" (default) | character vector of a real scalar in the range [0 100] | string scalar of a real scalar in the range [0 100)

Overlap percentage between the previous and current buffered data segments, specified as a character vector or string scalar of a real scalar in the range [0 100). The overlap creates a window segment that the Spectrum Analyzer uses to compute a spectral estimate.

**Tunable:** Yes

#### Dependency

To enable this property, set:

- `InputDomain` to "Time".
- `Method` to "Welch"

#### Scope Window Use

Click the **Estimation** tab on the Spectrum Analyzer toolstrip. In the **Window Options** section, set the **Overlap (%)**.

To enable the **Overlap (%)**, set **Input Domain** to Time and **Method** to Welch in the **Estimation** tab on the Spectrum Analyzer toolstrip.

Data Types: char | string

#### Window — Window function

"Hann" (default) | "Rectangular" | "Chebyshev" | "Flat Top" | "Hamming" | "Kaiser" | "Blackman-Harris" | "Custom"

Specify a window function for the spectral estimator. You can select one of the window options in the following table or specify a custom window. For more information on the window functions, click the link to the function reference page in the Signal Processing Toolbox documentation.

| Window Option     | Corresponding Signal Processing Toolbox Function |
|-------------------|--|
| "Rectangular"     | rectwin  |
| "Chebyshev"       | chebwin  |
| "Flat Top"        | flattopwin                                       |
| "Hamming"         | hamming  |
| "Hann"            | hann   |
| "Kaiser"          | kaiser   |
| "Blackman-Harris" | blackmanharris                                   |

To set your own spectral estimation window, set this property to "Custom" and specify a custom window function in the CustomWindow property.

**Tunable:** Yes

#### Dependency

To enable this property, set:

- InputDomain to "Time".
- Method to "Welch"

#### Scope Window Use

Click the **Estimation** tab on the Spectrum Analyzer toolstrip. In the **Window Options** section, set the **Window**.

To enable the **Window**, set **Input Domain** to Time and **Method** to Welch in the **Estimation** tab on the Spectrum Analyzer toolstrip.

Data Types: char | string

#### CustomWindow — Name of custom window function

"hann" (default) | character vector | string scalar

Name of the custom window function, specified as a character vector or string scalar. The name of the custom window function must be on the MATLAB path. Use this property if you want to customize the window using additional properties available with the Signal Processing Toolbox version of the window function.

**Tunable:** Yes

#### Example

Define and use a custom window function.

```
function w = my_hann(L)
    w = hann(L, 'periodic')
end

scope.Window = 'Custom';
scope.CustomWindow = 'my_hann'
```

**Dependency**

To use this property, set Window to "Custom".

**Scope Window Use**

Click the **Estimation** tab on the Spectrum Analyzer toolstrip. In the **Window Options** section, for the **Window**, enter the name of the custom window function.

Data Types: char | string

**SidelobeAttenuation — Sidelobe attenuation of window**

"60" (default) | character vector of real positive scalar | string scalar of real positive scalar

Sidelobe attenuation of the window in decibels (dB), specified as a character vector or a string scalar of a positive scalar greater than or equal to 45.

**Tunable:** Yes

**Dependency**

To enable this property, set Window to "Chebyshev" or "Kaiser".

**Scope Window Use**

Click the **Estimation** tab on the Spectrum Analyzer toolstrip. In the **Window Options** section, set the **Attenuation (dB)**.

To enable the **Attenuation (dB)**, set:

- **Input Domain** to Time
- **Method** to Welch
- **Window** to either Chebyshev or Kaiser in the **Estimation** tab on the Spectrum Analyzer toolstrip.

Data Types: char | string

**AveragingMethod — Smoothing method**

"VBW" (default) | "Exponential"

Averaging method, specified as one of the following:

- "VBW" — Video bandwidth method. The object uses a lowpass filter to smooth the trace and decrease noise. Use the VBWSource and VBW properties to specify the VBW value.
- "Exponential" — Weighted average of samples. The object computes the average over samples weighted by an exponentially decaying forgetting factor. Use the ForgettingFactor property to specify the weighted forgetting factor.

For more information, see "Averaging Method" on page 2-1343.

**Tunable:** Yes

**Dependency**

To enable this property, set InputDomain to "Time".

**Scope Window Use**

Click the **Estimation** tab on the Spectrum Analyzer toolstrip. In the **Averaging** section, set **Averaging Method** to VBW or Exponential.

To enable the **Averaging Method**, set **Input Domain** to Time.

Data Types: char | string

**VBWSource — Source of video bandwidth**

"Auto" (default) | "Input port" | "Property"

Source of the video bandwidth (VBW), specified as one of these options:

- "Auto" — The Spectrum Analyzer adjusts the VBW such that the equivalent forgetting factor is 0.9.
- "Input port" -- An input port appears on the scope and you can specify the VBW value through this port.
- "Property" — The Spectrum Analyzer adjusts the VBW using the value specified in the VBW property.

For more details on the video bandwidth method, see "Averaging Method" on page 2-1343.

**Tunable:** Yes

**Dependency**

To enable this property, set InputDomain to "Time" and AveragingMethod to "VBW".

**Scope Window Use**

Click the **Estimation** tab on the Spectrum Analyzer toolstrip. In the **Averaging** section, set **VBW (Hz)** to either Auto or a positive real scalar less than or equal to **Sample Rate (Hz)/2**.

To enable the **VBW (Hz)**, set **Input Domain** to Time and **Averaging Method** to VBW.

Data Types: char | string

**VBW — Video bandwidth**

"9.76" (default) | character vector of a positive scalar | string scalar of a positive scalar

Video bandwidth, specified as a character vector or a string scalar of a positive scalar less than or equal to SampleRate/2. For more information on the video bandwidth method, see "Averaging Method" on page 2-1343.

**Tunable:** Yes

**Dependency**

To enable this property, set VBWSource to "Property".

**Scope Window Use**

Click the **Estimation** tab on the Spectrum Analyzer toolstrip. In the **Averaging** section, set **VBW (Hz)** to either Auto or enter a positive real scalar that is less than or equal to **Sample Rate (Hz)/2**.

To enable the **VBW (Hz)**, set **Input Domain** to Time and **Averaging Method** to VBW.



Data Types: double

### ForgettingFactor — Weighting forgetting factor

"0.9" (default) | string scalar of scalar in the range (0,1] | character vector of scalar in the range (0,1]

Forgetting factor of the exponential weighted averaging method, specified as a character vector or string scalar of a real scalar in the range (0,1].

**Tunable:** Yes

#### Dependency

To enable this property, set:

- InputDomain to "Time"
- AveragingMethod to "Exponential"

#### Scope Window Use

Click the **Estimation** tab on the Spectrum Analyzer toolstrip. In the **Averaging** section, adjust the **Forgetting Factor** slider.

To enable the **Forgetting Factor**, set **Input Domain** to Time and **Averaging Method** to Exponential.

Data Types: char | string

### FrequencyVectorSource — Source of frequency vector

"Auto" (default) | "Property" | "InputPort"

Source of the frequency vector, specified as one of the following:

- "Auto" — The Spectrum Analyzer computes the frequency vector based on the frame size of the input signal and the specified sample rate. For more details, see "Frequency Vector" on page 2-1341.
- "Property" — Enter a custom vector in the FrequencyVector property.
- "InputPort" — An input port appears on the Spectrum Analyzer block. Specify the frequency vector input through this port.

**Tunable:** Yes

#### Dependency

To enable this property, set InputDomain to "Frequency".

#### Scope Window Use

Click the **Estimation** tab on the Spectrum Analyzer toolstrip. In the **Domain** section, set **Frequency (Hz)** to Auto, Input port, or a monotonically increasing vector of length equal to the input signal frame size.

To enable the **Frequency (Hz)**, set **Input Domain** to Frequency.

Data Types: char | string

### FrequencyVector — Custom frequency vector

`[-5000 5000]` (default) | monotonically increasing vector

Custom frequency vector, specified as a monotonically increasing vector. This vector determines the x-axis in the display. The vector must be monotonically increasing and must have the same length as the input signal frame size.

**Tunable:** Yes

#### Dependency

To enable this property, set:

- `InputDomain` to "Frequency".
- `FrequencyVectorSource` to "Property".

#### Scope Window Use

Click the **Estimation** tab on the Spectrum Analyzer toolstrip. In the **Domain** section, set **Frequency (Hz)** to a monotonically increasing vector of length equal to the input signal frame size.

To enable the **Frequency (Hz)**, set **Input Domain** to Frequency.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### InputUnits — Units of frequency input

`"dBm"` (default) | `"dBV"` | `"dBW"` | `"Vrms"` | `"Watts"` | `"None"`

Units of the frequency-domain input, specified as `"dBm"`, `"dBV"`, `"dBW"`, `"Vrms"`, `"Watts"`, or `"none"`. This property allows the Spectrum Analyzer to scale frequency data if you choose a different display unit in the `SpectrumUnits` property.

**Tunable:** Yes

#### Dependency

To enable this property, set `InputDomain` to "Frequency".

#### Scope Window Use

Click the **Estimation** tab on the Spectrum Analyzer toolstrip. In the **Domain** section, set **Input Unit**.

To enable the **Input Unit**, set **Input Domain** to Frequency.

Data Types: `char` | `string`

### SpectrumUnits — Units of the spectrum

`"dBm"` (default) | `"dBFS"` | `"dBV"` | `"dBW"` | `"Vrms"` | `"Watts"` | `"dBm/Hz"` | `"dBW/Hz"` | `"dBFS/Hz"` | `"Watts/Hz"` | `"Auto"`

Units of the spectrum, specified as one of these:

- `"dBm"`
- `"dBFS"`
- `"dBV"`
- `"dBW"`

- "Vrms"
- "Watts"
- "dBm/Hz"
- "dBW/Hz"
- "dBFS/Hz"
- "Watts/Hz"
- "Auto"

The Spectrum Analyzer displays power values in the units you select in this property.

**Tunable:** Yes

**Dependency**

The available spectrum units depend on the value you specify in the SpectrumType property.

| InputDomain | SpectrumType    | Allowed SpectrumUnits                        |
|-------------|-----------------|--|
| "Time"      | "Power"         | "dBm", "dBW", "dBFS", "Watts"                |
|             | "Power density" | "dBm/Hz", "dBW/Hz", "dBFS/Hz", "Watts/Hz"    |
|             | "RMS"           | "dBV", "Vrms"                                |
| "Frequency" | —               | "Auto", "dBm", "dBV", "dBW", "Vrms", "Watts" |

If you set the InputDomain property to "Frequency" and the SpectrumUnits property to "Auto", the Spectrum Analyzer assumes the spectrum units to be equal to input units specified in the InputUnits property. If you set InputDomain to "Time" and SpectrumUnits to any option other than "Auto", then the Spectrum Analyzer converts the units specified in InputUnits to the units specified in SpectrumUnits.

**Scope Window Use**

Click the **Spectrum** tab on the Spectrum Analyzer toolstrip. In the **Scale** section, set **Spectrum Unit**.

Data Types: char | string

**FullScaleSource — Source of dBFS scaling factor**

"Auto" (default) | "Property"

Source of dBFS scaling factor, specified as one of these:

- "Auto" -- The Spectrum Analyzer adjusts the scaling factor based on the input data.
- "Property" -- Specify the full-scale scaling factor using the FullScale property.

**Tunable:** Yes

**Dependency**

To enable this property, set:

- InputDomain to "Time"

- SpectrumType to "Power" or "Power density"
- SpectrumUnits to "dBFS" or "dBFS/Hz" (when spectrum type is set to "Power density")

### Scope Window Use

Click the **Spectrum** tab on the Spectrum Analyzer toolstrip. In the **Scale** section, set the **Full Scale** to either **Auto** or a positive scalar.

To enable the **Full Scale**:

- In the **Analyzer** tab, set the spectrum type to Power or Power Density.
- In the **Estimation** tab, set **Input Domain** to Time.
- In the **Spectrum** tab, set **Spectrum Unit** to dBFS or dBFS/Hz (when spectrum type is set to Power Density).

Data Types: char | string

### FullScale — dBFS full scale value

"1" (default) | character vector of a positive scalar | string scalar of a positive scalar

dBFS full scale value, specified as a character vector or a string scalar of a real positive scalar.

**Tunable:** Yes

### Dependency

To enable this property, set:

- InputDomain to "Time"
- SpectrumType to "Power" or "Power density"
- SpectrumUnits to "dBFS" or "dBFS/Hz" (when spectrum type is set to "Power density")
- FullScaleSource to "Property"

### Scope Window Use

Click the **Spectrum** tab on the Spectrum Analyzer toolstrip. In the **Scale** section, set the **Full Scale** to either **Auto** or enter a positive scalar.

To enable the **Full Scale**:

- In the **Analyzer** tab, set the spectrum type to Power or Power Density.
- In the **Estimation** tab, set **Input Domain** to Time.
- In the **Spectrum** tab, set **Spectrum Unit** to dBFS or dBFS/Hz (when spectrum type is set to Power Density).

Data Types: char | string

### ReferenceLoad — Reference load to compute power levels

"1" (default) | character vector of a real positive scalar | string scalar of a real positive scalar

Reference load to compute the power levels, specified as a character vector or a string scalar of a positive scalar in Ohms.

**Tunable:** Yes

**Scope Window Use**

Click the **Spectrum** tab on the Spectrum Analyzer toolstrip. In the **Scale** section, set **Reference Load ( $\Omega$ )**.

Data Types: char | string

**FrequencyOffset — Frequency offset**

"0" (default) | numeric scalar character vector | numeric vector character vector | numeric scalar string scalar | numeric vector string scalar

Frequency offset, specified as one of these:

- Numeric scalar — Apply the same frequency offset to all channels, specified as a character vector or a string scalar in Hz.
- Numeric vector — Apply a specific frequency offset for each channel, specified as a character vector or a string scalar in Hz. The vector length must be equal to the number of input channels.

The frequency-axis values are offset by the values specified in this property. The overall span must fall within the "Nyquist Frequency Interval" on page 2-1340. You can control the overall span in different ways based on how you set the FrequencySpan property.

**Tunable:** Yes

**Scope Window Use**

Click the **Analyzer** tab on the Spectrum Analyzer toolstrip. In the **Bandwidth** section, set **Offset (Hz)**.

Data Types: char | string

**TreatMby1SignalsAsOneChannel — Option to treat unoriented input signal as a column vector**

true (default) | false

Option to treat unoriented input signal as a column vector, specified as true or false. Set this property to true to treat  $M$ -by-1 and unoriented inputs as a column vector or one channel. Set this property to false to treat  $M$ -by-1 and unoriented inputs as a 1-by- $M$  row vector.

Data Types: logical

**Spectrogram****SpectrogramChannel — Channel for which spectrogram is plotted**

"1" (default) | character vector of a positive scalar integer | string scalar of a positive scalar integer

Channel for which the Spectrum Analyzer plots the spectrogram, specified as a character vector or string scalar of a positive scalar integer in the range  $[1 N]$ , where  $N$  is the number of input channels.

**Tunable:** Yes

**Dependency**

To enable this property, set ViewType to "Spectrogram" or "Spectrum and spectrogram".

**Scope Window Use**

Click the **Spectrogram** tab on the Spectrum Analyzer toolstrip. In the **Channel** section, select a **Channel**.

Data Types: char | string

**TimeResolutionSource — Source of the time resolution value**

"Auto" (default) | "Property"

Source of the time resolution value of each spectrogram line, , specified as "Auto" or "Property". The TimeResolution property shows the time resolution for the different frequency resolution methods and time resolution properties.

**Tunable:** Yes

**Dependency**

To enable this property, set ViewType to "Spectrogram" or "Spectrum and spectrogram".

**Scope Window Use**

Click the **Spectrogram** tab on the Spectrum Analyzer toolstrip. In the **Time Options** section, set the **Time Resolution (s)** to Auto or enter a positive scalar.

To enable the **Time Resolution (s)**, select **Spectrogram** in the **Analyzer** tab.

Data Types: char | string

**TimeResolution — Time resolution**

"1e-3" (default) | character vector of a positive scalar | string scalar of a positive scalar

Time resolution of each spectrogram line, specified as a character vector or a string scalar of a positive scalar and expressed in seconds.

The Spectrum Analyzer determines the time resolution value based on the frequency resolution method, RBW, and time resolution properties.

| Method               | RBW              | Time Resolution  | Resulting Time Resolution in Seconds  |
|----------------------|------------------|------------------|---|
| Welch or Filter Bank | Auto             | Auto             | 1/RBW   |
| Welch or Filter Bank | Auto             | Manually entered | 1/Time Resolution Hz  |
| Welch or Filter Bank | Manually entered | Auto             | 1/RBW seconds   |
| Welch or Filter Bank | Manually entered | Manually entered | Equal to or greater than the minimum attainable time resolution 1/RBW. The Spectrum Analyzer combines several spectral estimates into one spectrogram line to obtain the desired time resolution. It uses interpolation to obtain time resolution values that are not integer multiples of 1/RBW. |

**Tunable:** Yes

**Dependency**

To enable this property, set:

- ViewType to "Spectrogram" or "Spectrum and spectrogram"
- TimeResolutionSource to "Property"

**Scope Window Use**

Click the **Spectrogram** tab on the Spectrum Analyzer toolstrip. In the **Time Options** section, set the **Time Resolution (s)** to Auto or enter a positive scalar.

To enable the **Time Resolution (s)**, select **Spectrogram** in the **Analyzer** tab.

Data Types: char | string

**TimeSpanSource — Source of time span value**

"Auto" (default) | "Property"

Source of the time span of the spectrogram, specified as "Auto" or "Property". If you set this property to "Auto", the spectrogram displays 100 spectrogram lines at any given time. If you set this property to "Property", the spectrogram uses the time duration you specify in seconds in the TimeSpan property.

**Tunable:** Yes

**Dependency**

To enable this property, set ViewType to "Spectrogram" or "Spectrum and spectrogram".

**Scope Window Use**

Click the **Spectrogram** tab on the Spectrum Analyzer toolstrip. In the **Time Options** section, set the **Time Span (s)** to Auto or enter a positive scalar.

Data Types: char | string

**TimeSpan — Time span**

"0.1" (default) | character vector of a positive scalar | string scalar of a positive scalar

Time span of the spectrogram display in seconds, specified as a character vector or string scalar of a positive scalar. You must set the time span to be at least twice as large as the duration of the number of samples required for a spectral update.

**Tunable:** Yes

**Dependency**

To enable this property, set:

- ViewType to "Spectrogram" or "Spectrum and spectrogram".
- TimeSpanSource to "Property".

**Scope Window Use**

Click the **Spectrogram** tab on the Spectrum Analyzer toolstrip. In the **Time Options** section, set the **Time Span (s)** to Auto or enter a positive scalar.

Data Types: char | string

### Measurements

#### MeasurementChannel — Channel for which to obtain measurements

"1" (default) | character vector | string scalar

Channel over which the measurements are obtained, specified as a character vector or a string scalar which evaluates to a positive integer less than or equal to 100. The highest number you can specify is equal to the number of channels (columns) in the input signal.

**Tunable:** Yes

#### Scope Window Use

Click the **Measurements** tab on the Spectrum Analyzer toolstrip. In the **Channel** section, select a **Channel**.

Data Types: char | string

#### ChannelMeasurements — Channel measurements

ChannelMeasurementsConfiguration object

Channel measurements, specified as a ChannelMeasurementsConfiguration object. Enable channel measurements to compute and display the occupied bandwidth or adjacent channel power ratio. All ChannelMeasurementsConfiguration properties are tunable.

**Tunable:** Yes

#### Dependency

To enable this property, set ViewType to "Spectrum" or "Spectrum and spectrogram".

#### Scope Window Use

Click the **Channel Measurements** tab on the Spectrum Analyzer toolstrip and modify the measurement settings.

The **Channel Measurements** tab appears when you select **Spectrum** in the **Analyzer** tab.

#### CursorMeasurements — Cursor measurements

CursorMeasurementsConfiguration object

Cursor measurements, specified as a CursorMeasurementsConfiguration object. Enable cursor measurements to display waveform cursors. All CursorMeasurementsConfiguration properties are tunable.

**Tunable:** Yes

#### Dependency

To enable this property, set ViewType to "Spectrum" or "Spectrum and spectrogram".

#### Scope Window Use

Click the **Measurements** tab on the Spectrum Analyzer toolstrip and modify the cursor measurements in the **Cursors** section.



The **Measurements** tab appears when you select **Spectrum** in the **Analyzer** tab.

### **DistortionMeasurements — Distortion measurements**

DistortionMeasurementsConfiguration object

Distortion measurements, specified as a DistortionMeasurementsConfiguration object. Enable distortion measurements to compute and display the harmonic distortion and intermodulation distortion. All DistortionMeasurementsConfiguration properties are tunable.

**Tunable:** Yes

#### **Dependency**

To enable this property, set ViewType to "Spectrum" or "Spectrum and spectrogram".

#### **Scope Window Use**

Click the **Measurements** tab on the Spectrum Analyzer toolstrip and modify the distortion measurements in the **Distortion** section.

The **Measurements** tab appears when you select **Spectrum** in the **Analyzer** tab.

### **PeakFinder — Peak finder measurement**

PeakFinderConfiguration object

Peak finder measurement, specified as a PeakFinderConfiguration object. Enable peak finder to compute and display the largest calculated peak values. All PeakFinderConfiguration properties are tunable.

**Tunable:** Yes

#### **Dependency**

To enable this property, set ViewType to "Spectrum" or "Spectrum and spectrogram".

#### **Scope Window Use**

Click the **Measurements** tab on the Spectrum Analyzer toolstrip and modify the peak finder measurements in the **Peaks** section.

The **Measurements** tab appears when you select **Spectrum** in the **Analyzer** tab.

### **SpectralMask — Spectral mask configuration**

SpectralMaskConfiguration object

Spectral mask configuration, specified as a SpectralMaskConfiguration object. Use the spectral mask configuration to draw upper, lower, or upper and lower mask lines in the power and power-density plots. All SpectralMaskConfiguration properties are tunable.

**Tunable:** Yes

#### **Dependency**

To enable this property, set:

- ViewType to "Spectrum" or "Spectrum and spectrogram".
- SpectrumType to either "Power" or "Power density".

**Scope Window Use**

Click the **Spectral Mask** tab on the Spectrum Analyzer toolstrip and modify the settings.

The **Spectral Mask** tab appears when you:

- Select **Spectrum** in the **Analyzer** tab.
- In the drop-down list under **Spectrum**, choose either Power or Power Density.

**Visualization****Name — Caption to display in spectrum Analyzer window**

"Spectrum Analyzer" (default) | character vector | string scalar

Caption to display in the scope window, specified as a character vector or string scalar.

**Tunable:** Yes

Data Types: char | string

**Position — Window position**

screen center (default) | [left bottom width height]

Spectrum Analyzer window position in pixels, specified by the size and location of the scope window as a four-element double vector of the form [left bottom width height]. You can place the scope window in a specific position on your screen by modifying the values to this property.

By default, the window appears in the center of your screen with a width of 800 pixels and height of 450 pixels. The exact center coordinates depend on your screen resolution.

**Tunable:** Yes

**MaximizeAxes — Maximize axes control**

"Auto" (default) | "On" | "Off"

Maximize axes control, specified as one of the following:

- "Auto" -- The Spectrum Analyzer maximizes axes only if the display does not contain any labels or title annotations.
- "On" -- The Spectrum Analyzer maximizes axes in all displays.
- "Off" -- The Spectrum Analyzer does not maximize axes in any display.

**Tunable:** Yes

**Scope Window Use**

Click the  control on the Spectrum Analyzer display to maximize the axes.

Data Types: char | string

**PlotNormalTrace — Normal trace flag**

true (default) | false

Normal trace flag, specified as true or false.

To remove normal traces from the display, set this property to `false`. These traces display the free-running spectral estimates. The Spectrum Analyzer continues its spectral computations even when you set this property to `false`.

**Tunable:** Yes

**Dependency**

To enable this property, set `ViewType` to "Spectrum" or "Spectrum and spectrogram".

**Scope Window Use**

Click the **Spectrum** tab on the Spectrum Analyzer toolstrip and select the **Normal Trace** check box in the **Trace Options** section.

To enable the **Normal Trace** check box, select **Spectrum** in the **Analyzer** tab.

Data Types: `logical`

**PlotMaxHoldTrace – Maximum-hold trace flag**

`false` (default) | `true`

Maximum-hold trace flag, specified as `false` or `true`.

To compute and plot the maximum-hold spectrum of each input channel, set this property to `true`. The Spectrum Analyzer computes the maximum-hold spectrum at each frequency bin by keeping the maximum value of all the power spectrum estimates. When you toggle this property, the Spectrum Analyzer resets its maximum-hold computations.

**Tunable:** Yes

**Dependency**

To enable this property, set `ViewType` to "Spectrum" or "Spectrum and spectrogram".

**Scope Window Use**

Click the **Spectrum** tab on the Spectrum Analyzer toolstrip and select the **Max-Hold Trace** check box in the **Trace Options** section.

To enable the **Max-Hold Trace** check box, select **Spectrum** in the **Analyzer** tab.

Data Types: `logical`

**PlotMinHoldTrace – Minimum-hold trace flag**

`false` (default) | `true`

Minimum-hold trace flag, specified as `false` or `true`.

To compute and plot the minimum-hold spectrum of each input channel, set this property to `true`. The Spectrum Analyzer computes the minimum-hold spectrum at each frequency bin by keeping the minimum value of all the power spectrum estimates. When you toggle this property, the Spectrum Analyzer resets its minimum-hold computations.

**Tunable:** Yes

**Dependency**

To enable this property, set `ViewType` to "Spectrum" or "Spectrum and spectrogram".

**Scope Window Use**

Click the **Spectrum** tab on the Spectrum Analyzer toolstrip and select the **Min-Hold Trace** check box in the **Trace Options** section.

To enable the **Min-Hold Trace** check box, select **Spectrum** in the **Analyzer** tab.

Data Types: `logical`

**Title — Display title**

`''` (default) | character vector | string scalar

Display title, specified as a character vector or a string scalar.

**Tunable:** Yes

**Scope Window Use**

Click the **Analyzer** tab on the Spectrum Analyzer toolstrip. In the **Configuration** section, click **Settings**. In the Spectrum Analyzer Settings window that opens up, under **Display and labels**, enter **Title**.

Data Types: `char` | `string`

**YLabel — Y-axis label**

`''` (default) | character vector | string scalar

y-axis label, specified as a character vector or a string scalar. The Spectrum Analyzer displays the label to the left of the y-axis.

Regardless of the value of this property, Spectrum Analyzer always displays power units as one of the `SpectrumUnits` values.

**Tunable:** Yes

**Dependency**

To enable this property, set `ViewType` to "Spectrum" or "Spectrum and spectrogram".

**Scope Window Use**

Click the **Analyzer** tab on the Spectrum Analyzer toolstrip. In the **Configuration** section, click **Settings**. In the Spectrum Analyzer Settings window that opens up, under **Display and labels**, enter **Y-Label**.

To enable the **Y-Label**, select **Spectrum** in the **Analyzer** tab.

Data Types: `char` | `string`

**YLimits — Y-axis limits**

`[-80, 20]` (default) | `[ymin ymax]`

y-axis limits, specified as a two-element numeric vector of the form `[ymin ymax]`. The units of the y-axis limits depend on the `SpectrumUnits` property.

Example: `scope.YLimits = [-10,20]`

**Tunable:** Yes

**Dependencies**

- To enable this property, set the `ViewType` property to "Spectrum" or "Spectrum and spectrogram".
- The units directly depend upon the `SpectrumUnits` property.

**Scope Window Use**

Click the **Analyzer** tab on the Spectrum Analyzer toolstrip. In the **Configuration** section, click **Settings**. In the Spectrum Analyzer Settings window that opens up, under **Display and Labels**, enter **Y-Axis Limits**.

To enable the **Y-Axis Limits**, select **Spectrum** in the **Analyzer** tab.

**ColorLimits — Color limits of the spectrogram**

`[-80, 20]` (default) | `[colorMin colorMax]`

Color limits of the spectrogram, specified as a two-element numeric vector of the form `[colorMin colorMax]`. The units of the color limits directly depend upon the `SpectrumUnits` property.

Example: `scope.ColorLimits = [-10,20]`

**Tunable:** Yes

**Dependencies**

- To enable this property, set the `ViewType` property to "Spectrogram" or "Spectrum and spectrogram".
- The units directly depend upon the `SpectrumUnits` property.

**Scope Window Use**

Click the **Analyzer** tab on the Spectrum Analyzer toolstrip. In the **Configuration** section, click **Settings**. In the Spectrum Analyzer Settings window that opens up, under **Display and Labels**, enter **Color Limits**.

To enable the **Color Limits**, select **Spectrogram** in the **Analyzer** tab.

**Colormap — Color lookup table**

`"jet"` (default) | `"bone"` | `"cool"` | `"copper"` | `"gray"` | `"hot"` | `"parula"` | three-column matrix

Color lookup table, specified as a valid colormap name or a three-column matrix with values in the range `[0,1]` defining RGB triplets.

**Tunable:** Yes

**Dependencies**

To enable this property, set the `ViewType` property to "Spectrogram" or "Spectrum and spectrogram".

**Scope Window Use**

Click the **Analyzer** tab on the Spectrum Analyzer toolstrip. In the **Configuration** section, click **Settings**. In the Spectrum Analyzer Settings window that opens up, under **Display and Labels**, enter **Color Map**.

To enable the **Color Map**, select **Spectrogram** in the **Analyzer** tab.

Data Types: `double` | `char` | `string`

### ShowGrid — Flag to show grid

`true` (default) | `false`

Flag to show the grid, specified as `true` or `false`. Set this property to `true` to show grid lines in the plot.

**Tunable:** Yes

#### Scope Window Use

Click the **Analyzer** tab on the Spectrum Analyzer toolbar. In the **Configuration** section, click **Settings**. In the Spectrum Analyzer Settings window that appears, under **Display and Labels**, select **Show Grid**.

Data Types: `logical`

### ChannelNames — Channel names

empty cell (default) | cell array of character vectors | array of strings

Channel names in the input data, specified as a cell array of character vectors or an array of strings. The names you specify in this property appear in the following locations:

- Legend
- **Spectrum Analyzer Settings > Color and styling** section
- **Measurements** and **Channel Measurements** tabs

If you do not specify channel names, the Spectrum Analyzer names the channels as `Channel 1`, `Channel 2`, and so on.

**Tunable:** Yes

#### Dependency

To see the channel names, set `ShowLegend` to `true`.

#### Scope Window Use

Click the **Analyzer** tab on the Spectrum Analyzer toolbar. To see the legend, click **Legend** in the **Configuration** section.

Data Types: `char`

### ShowLegend — Flag to show legend

`false` (default) | `true`

Flag to show the legend, specified as `true` or `false`. To show a legend with the input names, set this property to `true`.

Use the legend to control which signals are visible. In the scope legend, click a signal name to hide the signal in the scope. To show the signal, click the signal name again. To show only one signal, right-click the signal name. To show all signals, press **Esc**.

**Tunable:** Yes

**Dependencies**

To enable this property, set the `ViewType` property to "Spectrum" or "Spectrum and spectrogram".

**Scope Window Use**

Click the **Analyzer** tab on the Spectrum Analyzer toolstrip. To see the legend, click **Legend** in the **Configuration** section.

To enable the **Legend**, select **Spectrum** in the **Analyzer** tab.

Data Types: `logical`

**ShowColorbar — Flag to show color bar**

`true` (default) | `false`

Flag to show the color bar, specified as `true` or `false`.

**Tunable:** Yes

**Dependencies**

To enable this property, set the `ViewType` property to "Spectrogram" or "Spectrum and spectrogram".

**Scope Window Use**

Click the **Analyzer** tab on the Spectrum Analyzer toolstrip. To see the color bar, click **Colorbar** in the **Configuration** section.

To enable the **Colorbar**, select **Spectrogram** in the **Analyzer** tab.

Data Types: `logical`

**AxesLayout — Orientation of the spectrum and spectrogram**

`"Vertical"` (default) | `"Horizontal"`

Layout of the axes, specified as one of "Vertical" or "Horizontal". A vertical layout stacks the spectrum above the spectrogram. A horizontal layout places the spectrum next to the spectrogram.

**Tunable:** Yes

**Dependency**

To enable this property, set `ViewType` to "Spectrum and spectrogram".

**Scope Window Use**

Click the **Analyzer** tab on the Spectrum Analyzer toolstrip. Select **Spectrum** and **Spectrogram**. In the **Configuration** section, select and update **Layout**.

Data Types: `char` | `string`

**OpenAtSimulationStart — Flag to open scope when simulation starts**

`true` (default) | `false`

Flag to open scope when simulation starts, specified as `true` or `false`.

Set this property to `true` to open the scope when the simulation starts. Set this property to `false` to prevent the scope from opening when you simulate the model.

#### Scope Window Use

This parameter is available only from the **Property Inspector**. To launch the **Property Inspector**, click the **Modeling** tab of the Simulink model window that contains the Spectrum Analyzer block. In the **Design** section, click the drop-down arrow and select **Property Inspector**. The **Property Inspector** window opens on the right. If you select the Spectrum Analyzer block, the **Property Inspector** window shows the corresponding parameters. In the parameters that show up, select or clear the **Open at simulation start** parameter.

Data Types: `logical`

#### Visible — Visibility of the Spectrum Analyzer

`false` (default) | `true`

Set this property to `true` to display the spectrum analyzer window, and to `false` to hide the spectrum analyzer window.

Data Types: `logical`

## Examples

### Construct Spectrum Analyzer Configuration Object

Create a new Simulink® model with a randomly generated name.

```
sysname=char(randi(26,1,7)+96);  
new_system(sysname);
```

Add a new Spectrum Analyzer block to the model.

```
add_block('built-in/SpectrumAnalyzer',[sysname,'/SpectrumAnalyzer'])
```

Call the `get_param` function to retrieve the default Spectrum Analyzer block configuration properties.

```
config = get_param([sysname,'/SpectrumAnalyzer'],'ScopeConfiguration')
```

```
config =  
SpectrumAnalyzerBlockConfiguration with properties:
```

```
    NumInputPorts: '1'  
    InputDomain: 'Time'  
    SpectrumType: 'Power'  
    ViewType: 'Spectrum'  
    SampleRateSource: 'Inherited'  
    Method: 'Filter bank'  
    PlotAsTwoSidedSpectrum: 1  
    FrequencyScale: 'Linear'  
    PlotType: 'Line'  
    AxesScaling: 'Auto'
```

Advanced

```
    RBWSource: 'Auto'
```



```

        FilterSharpness: '0.3'
        FrequencySpan: 'Full'
        AveragingMethod: 'VBW'
            VBWSource: 'Auto'
        SpectrumUnits: 'dBm'
        ReferenceLoad: '1'
        FrequencyOffset: '0'
    TreatMby1SignalsAsOneChannel: 1

Spectrogram
    No properties.

Measurements
    MeasurementChannel: '1'
    ChannelMeasurements: [1x1 ChannelMeasurementsConfiguration]
    CursorMeasurements: [1x1 CursorMeasurementsConfiguration]
    DistortionMeasurements: [1x1 DistortionMeasurementsConfiguration]
    PeakFinder: [1x1 PeakFinderConfiguration]
    SpectralMask: [1x1 SpectralMaskConfiguration]

Visualization
    Name: 'SpectrumAnalyzer'
    Position: [240 262 800 500]
    MaximizeAxes: 'Auto'
    PlotNormalTrace: 1
    PlotMaxHoldTrace: 0
    PlotMinHoldTrace: 0
    Title: ''
    YLabel: ''
    YLimits: [-80 20]
    ShowGrid: 1
    ChannelNames: {' '}
    ShowLegend: 0
    OpenAtSimulationStart: 1
    Visible: 0

```

### Obtain Measurements Data Programmatically for Spectrum Analyzer Block

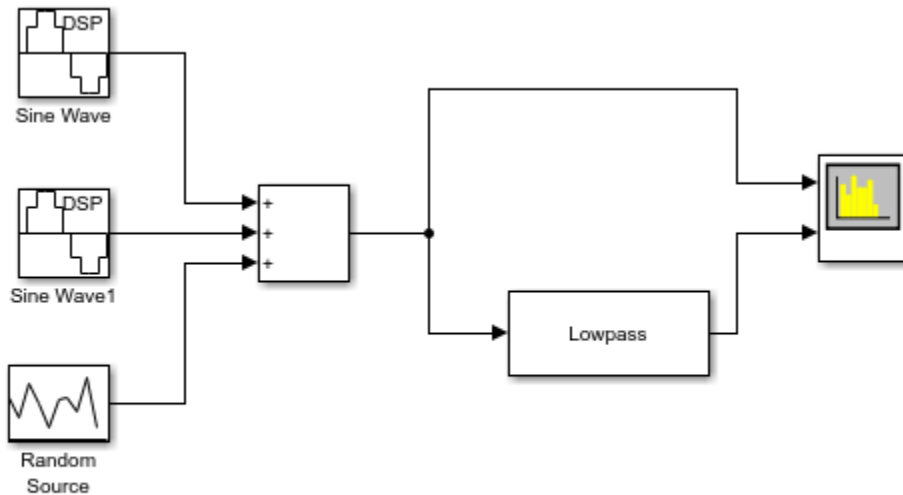
Compute and display the power spectrum of a noisy sinusoidal input signal using the Spectrum Analyzer block. Measure the cursor placements, adjacent channel power ratio, distortion, and peak values in the spectrum by enabling these block configuration properties:

- CursorMeasurements
- ChannelMeasurements
- DistortionMeasurements
- PeakFinder

### Open and Inspect the Model

Filter a streaming noisy sinusoidal input signal using a Lowpass Filter block. The input signal consists of two sinusoidal tones: 1 kHz and 15 kHz. The noise is white Gaussian noise with a mean of 0 and a variance of 0.05. The sampling frequency is 44.1 kHz. Open the model and inspect the parameter values in the blocks.

```
model = 'spectrumanalyzer_measurements.slx';  
open_system(model)
```



Copyright 2018-2022 The MathWorks, Inc.

Access the configuration properties of the Spectrum Analyzer block using the `get_param` function.

```
sablock = 'spectrumanalyzer_measurements/Spectrum Analyzer';  
cfg = get_param(sablock, 'ScopeConfiguration');
```

### Enable Measurements Data

To obtain the measurements, set the `Enabled` property to `true`.

```
cfg.CursorMeasurements.Enabled = true;  
cfg.ChannelMeasurements.Enabled = true;  
cfg.DistortionMeasurements.Enabled = true;  
cfg.PeakFinder.Enabled = true;
```

### Simulate the Model

Run the model. The Spectrum Analyzer block compares the original spectrum with the filtered spectrum.

```
sim(model)
```



The panes at the bottom of the spectrum analyzer window display the measurements that you have enabled.

### Use `getMeasurementsData` function

Use the `getMeasurementsData` function to obtain the measurements programmatically.

```
data = getMeasurementsData(cfg)
```

```
data =
```

```
1x5 table
```

```
SimulationTime
```

```
9.9962
```

```
PeakFinder
```

```
1x1 struct
```

```
CursorMeasurements
```

```
1x1 struct
```

```
ChannelMeasurements
```

```
1x1 struct
```

```
DistortionMeasurements
```

```
1x1 struct
```

The values shown in the measurement panels match the values shown in `data`. You can access the individual fields of `data` to obtain the various measurements programmatically.

### Compare Peak Values

As an example, compare the peak values. Verify that the peak values obtained by `data.PeakFinder` match with the values in the spectrum analyzer window.

```
peakvalues = data.PeakFinder.Value  
frequencieskHz = data.PeakFinder.Frequency/1000
```

```
peakvalues =
```

```
    26.8653  
    26.2638  
    -3.4329
```

```
frequencieskHz =
```

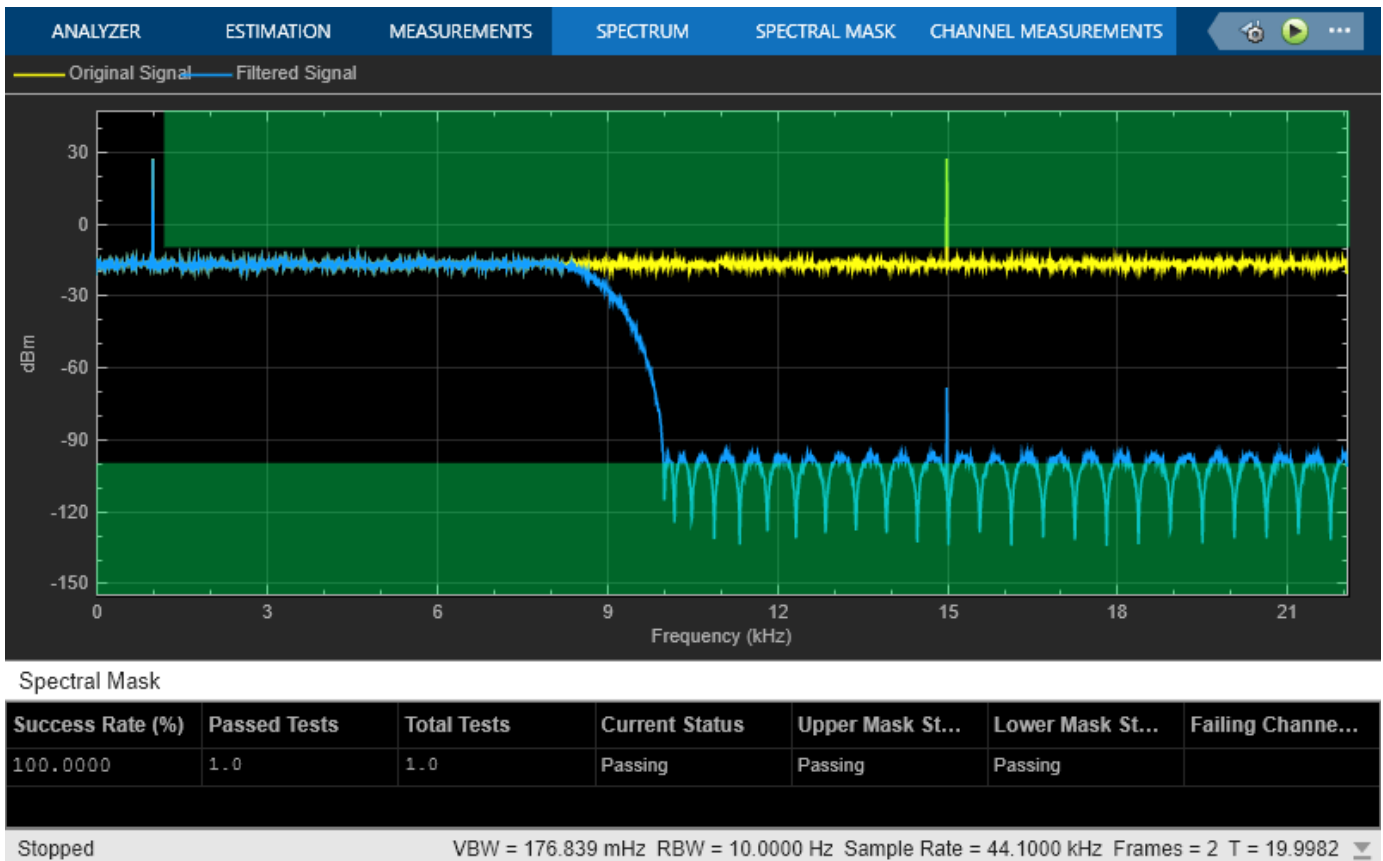
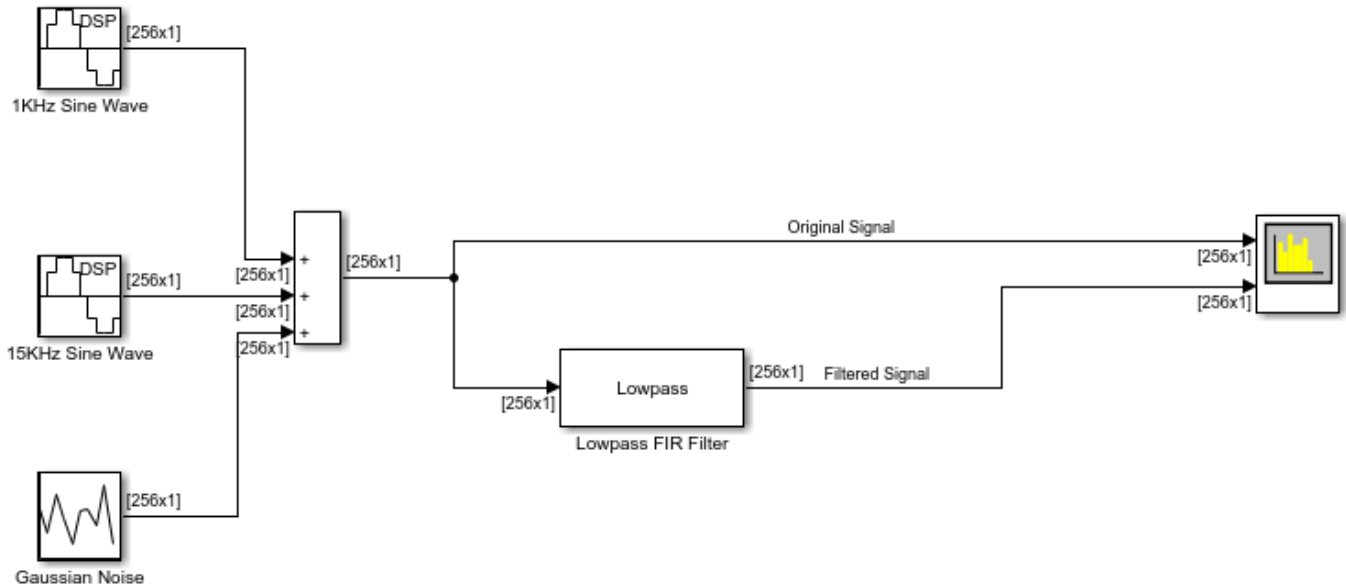
```
    15.0015  
     1.0049  
     3.7755
```

### Spectral Mask in Spectrum Analyzer Block

Create a new model based on the `dsp_basic_filter` template. Add a spectral mask to the Spectrum Analyzer block in the model. Run the model.

Masks are overlaid on the spectrum. If the mask is green, the signal is passing. If the mask is red, the signal is failing. The **Spectral Mask** panel shows what percentage of the time the mask is succeeding, which mask is failing, how many times the mask(s) failed, and which channels are causing the failure.

```
[~,mdl] = fileparts(tempname);  
open_system(new_system(mdl,'FromTemplate','dsp_basic_filter'));  
saBlock = find_system(mdl,'BlockType','SpectrumAnalyzer');  
  
scopeConfig = get_param(saBlock{1},'ScopeConfiguration');  
upperMask = [0 50; 1200 50; 1200 -10; 24000 -10];  
scopeConfig.SpectralMask.UpperMask = upperMask;  
scopeConfig.SpectralMask.LowerMask = -100;  
scopeConfig.SpectralMask.EnabledMasks = 'upper-and-lower';  
sim(mdl,StopTime='20');
```



## Version History

Introduced in R2013a

### **R2023a: Spectrum Analyzer with improved responsiveness and toolstrip interface in Simulink**

In R2023a, the Spectrum Analyzer block is more responsive and its toolstrip interface provides you easy access to spectral analysis, estimation, and measurements.

### **See Also**

#### **Functions**

`getMeasurementsData` | `getSpectrumData`

#### **Objects**

`spectrumAnalyzer`

#### **Blocks**

Spectrum Analyzer

#### **Topics**

“Obtain Measurements Data Programmatically for Spectrum Analyzer Block”

“Control Scope Blocks Programmatically” (Simulink)

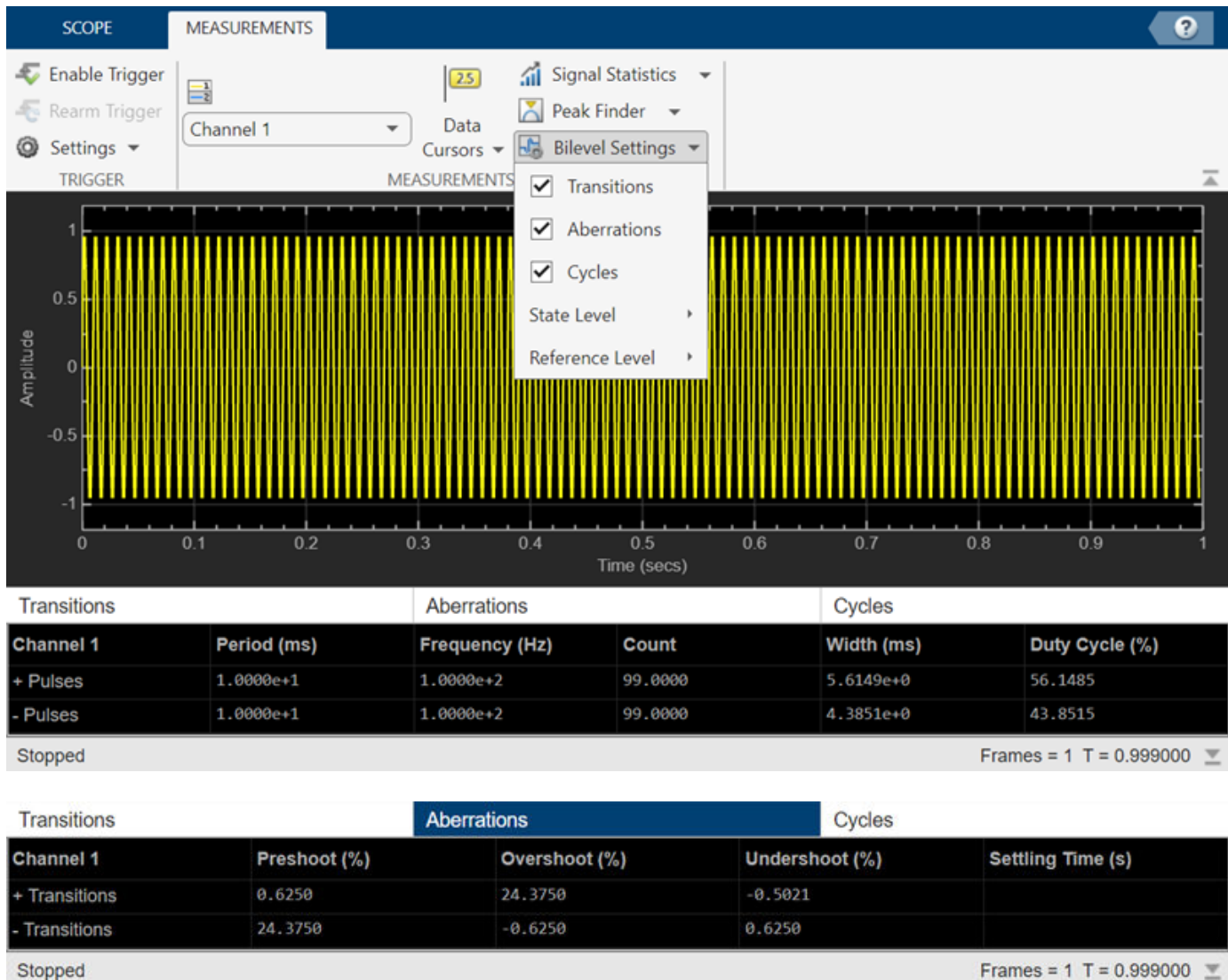
# BilevelMeasurementsConfiguration

Measure transitions, aberrations, and cycles of bilevel signals

## Description

Use the `BilevelMeasurementsConfiguration` object to measure transitions, aberrations, and cycles of bilevel signals. You can also specify the bilevel settings such as high-state level, low-state level, state-level tolerance, upper-reference level, mid-reference level, and lower-reference level.

You can control bilevel measurements from the toolbar or from the command line. To modify bilevel measurements from the toolbar, in the **Measurements** tab, click **Bilevel Settings** and select the measurements you want to display. A panel appears at the bottom of the Time Scope window showing all the measurements you enabled.



| Transitions |             | Aberrations    |         | Cycles     |                |
|-------------|-------------|----------------|---------|------------|----------------|
| Channel 1   | Period (ms) | Frequency (Hz) | Count   | Width (ms) | Duty Cycle (%) |
| + Pulses    | 1.0000e+1   | 1.0000e+2      | 99.0000 | 5.6149e+0  | 56.1485        |
| - Pulses    | 1.0000e+1   | 1.0000e+2      | 99.0000 | 4.3851e+0  | 43.8515        |

Stopped Frames = 1 T = 0.999000

## Creation

### Syntax

```
bilevelMeas = BilevelMeasurementsConfiguration()
```

### Description

`bilevelMeas = BilevelMeasurementsConfiguration()` creates a bilevel measurements configuration object.

### Properties

All properties are tunable.

#### AutoStateLevel — Automatic detection of high- and low-state levels

`true` (default) | `false`

Automatic detection of high- and low-state levels, specified as `true` or `false`. Set this property to `true` so that the scope automatically detects high- and low-state levels in the bilevel waveform. When you set this property to `false`, you can specify values for the high- and low- state levels manually using the `HighStateLevel` and `LowStateLevel` properties.

#### Scope Window Use

On the **Measurements** tab, in the **Measurements** section, click **Bilevel Settings**. Under **State Level**, and select **Auto State Level**.

Data Types: `logical`

#### HighStateLevel — High-state level

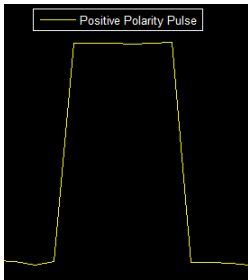
2.3 (default) | nonnegative scalar

High-state level, specified as a nonnegative scalar. The high-state level denotes a positive polarity.

If the initial transition of a pulse is positive-going, the pulse has positive polarity. The terminating state of a positive-polarity (positive-going) pulse is more positive than the originating state.

This figure shows a positive-polarity pulse.





### Dependency

To enable this property, set `AutoStateLevel` to `false`.

### Scope Window Use

On the **Measurements** tab, in the **Measurements** section, click **Bilevel Settings**. Under **State Level**, clear **Auto State Level** and specify a nonnegative scalar in the **High** box.

Data Types: `double`

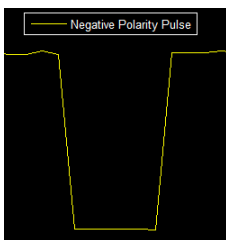
### LowStateLevel — Low-state level

0 (default) | nonnegative scalar

High-state level, specified as a nonnegative scalar. The low-state level denotes a negative polarity.

If the initial transition of a pulse is negative-going, the pulse has negative polarity. The terminating state of a negative-polarity (negative-going) pulse is more negative than the originating state.

This figure shows a negative-polarity pulse.



### Dependency

To enable this property, set `AutoStateLevel` to `false`.

### Scope Window Use

On the **Measurements** tab, in the **Measurements** section, click **Bilevel Settings**. Under **State Level**, clear **Auto State Level** and specify a nonnegative scalar in the **Low** box.

Data Types: `double`

### StateLevelTolerance — Tolerance level of state

2 (default) | positive scalar in the range (0 100)

Tolerance level of the state, specified as a positive scalar in the range (0 100).

This value determines how much a signal can deviate from the low- or high-state level before it is considered to be outside that state. Specify this value as a percentage of the difference between the high- and low-state levels. For more details, see “State-Level Tolerances” on page 4-1377.

**Scope Window Use**

On the **Measurements** tab, in the **Measurements** section, click **Bilevel Settings**. Under **State Level**, specify a positive scalar less than 100 in the **State Level Tol. (%)** box.

Data Types: double

**UpperReferenceLevel — Upper-reference level**

90 (default) | positive scalar in the range (0 100)

Upper-reference level, specified as a positive scalar in the range (0 100). The scope uses the upper-reference level to compute the start of a fall time or the end of a rise time. Specify this value as a percentage of the difference between the high- and low-state levels.

If  $S_1$  is the low-state level,  $S_2$  is the high-state level, and  $U$  is the upper-reference level, the waveform value corresponding to the upper-reference level is

$$S_1 + \frac{U}{100}(S_2 - S_1).$$

**Scope Window Use**

On the **Measurements** tab, in the **Measurements** section, click **Bilevel Settings**. Under **Reference Level**, specify a positive scalar less than 100 in the **Upper Ref. Level (%)** box.

Data Types: double

**MidReferenceLevel — Mid-reference level**

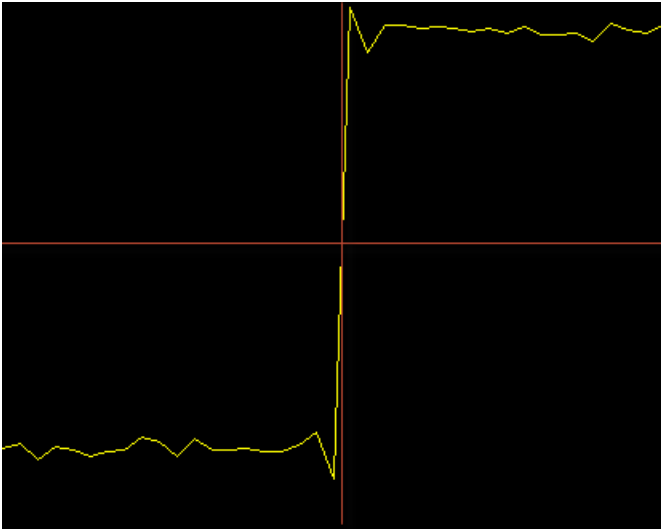
50 (default) | positive scalar in the range (0 100)

Mid-reference level, specified as a positive scalar in the range (0 100). The scope uses the mid-reference level to determine when a transition occurs. Specify this value as a percentage of the difference between the high- and low-state levels.

The mid-reference level in a bilevel waveform with low-state level  $S_1$  and high-state level  $S_2$  is

$$S_1 + \frac{1}{2}(S_2 - S_1)$$

This figure shows the mid-reference level as a horizontal line, and shows its corresponding mid-reference level instant as a vertical line.



#### Scope Window Use

On the **Measurements** tab, in the **Measurements** section, click **Bilevel Settings**. Under **Reference Level**, specify a positive scalar less than 100 in the **Mid Ref. Level (%)** box.

Data Types: double

#### LowerReferenceLevel<sub>L</sub> — Lower-reference level

10 (default) | positive scalar in the range (0 100)

Lower-reference level, specified as a positive scalar in the range (0 100). The scope uses the lower-reference level to compute the end of a fall time or the start of a rise time. Specify this value as a percentage of the difference between the high- and low-state levels.

If  $S_1$  is the low-state level,  $S_2$  is the high-state level, and  $L$  is the lower-reference level, the waveform value corresponding to the lower-reference level is

$$S_1 + \frac{L}{100}(S_2 - S_1).$$

#### Scope Window Use

On the **Measurements** tab, in the **Measurements** section, click **Bilevel Settings**. Under **Reference Level**, specify a positive scalar less than 100 in the **Lower Ref. Level (%)** box.

Data Types: double

#### SettleSeek — Time duration over which to search for a settling time

0.02 (default) | positive scalar

Time duration over which the scope searches for a settling time, specified as a positive scalar in seconds.

#### Scope Window Use

On the **Measurements** tab, in the **Measurements** section, click **Bilevel Settings**. Under **Reference Level**, specify a positive scalar in the **Settle Seek (s)** box.

Data Types: double

**ShowTransitions — Enable transition measurements**`false (default) | true`

Enable transition measurements, specified as `true` or `false`. For more information on the transition measurements that the scope displays, see “Transitions Pane” (Simulink).

**Scope Window Use**

On the **Measurements** tab, in the **Measurements** section, click **Bilevel Settings** and select **Transitions**. A **Transitions** pane opens at the bottom of the Time Scope window to show the transition measurements.

Data Types: `logical`

**ShowAberrations — Enable aberration measurements**`false (default) | true`

Enable aberration measurements, specified as `true` or `false`. Aberration measurements include distortion and damping measurements such as preshoot, overshoot, and undershoot. For more information on the aberration measurements that the scope displays, see “Overshoots / Undershoots Pane” (Simulink).

**Scope Window Use**

On the **Measurements** tab, in the **Measurements** section, click **Bilevel Settings** and select **Aberrations**. An **Aberrations** pane opens at the bottom of the Time Scope window to show the aberration measurements.

Data Types: `logical`

**ShowCycles — Enable cycle measurements**`false (default) | true`

Enable cycle measurements, specified as `true` or `false`. These measurements are related to repetitions or trends in the displayed portion of the input signal. For more information on the cycle measurements, see “Cycles Pane” (Simulink).

**Scope Window Use**

On the **Measurements** tab, in the **Measurements** section, click **Bilevel Settings** and select **Cycles**. A **Cycles** pane opens at the bottom of the Time Scope window to show the cycle measurements.

Data Types: `logical`

## Examples

**Configure Bilevel Measurements Programmatically in Time Scope MATLAB Object**

Create a sine wave and view it in the Time Scope. Programmatically compute the bilevel measurements related to signal transitions, aberrations, and cycles.

**Initialization**

Create the input sine wave using the `sin` function. Create a `timescope` MATLAB® object to display the signal. Set the `TimeSpan` property to 1 second.

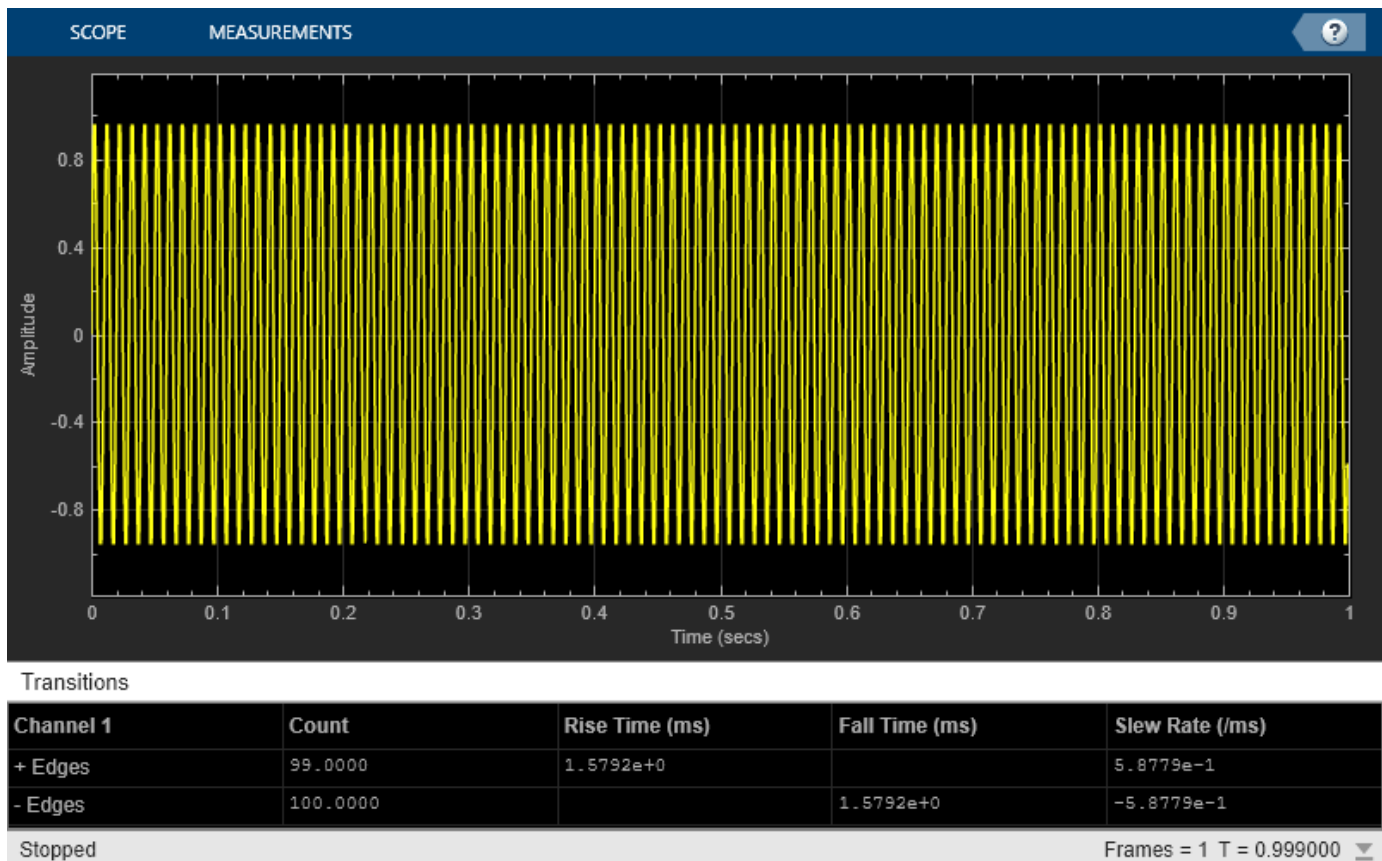
```
f = 100;
fs = 1000;
swv = sin(2.*pi.*f.*(0:1/fs:1-1/fs)).';
scope = timescope(SampleRate=fs,...
    TimeSpanSource="property",...
    TimeSpan=1);
```

### Transition Measurements

Enable the scope to show transition measurements programmatically by setting the `ShowTransitions` property to `true`. Display the sine wave in the scope.

Transition measurements such as rise time, fall time, and slew rate appear in the **Transitions** pane at the bottom of the scope.

```
scope.BilevelMeasurements.ShowTransitions = true;
scope(swv);
release(scope);
```

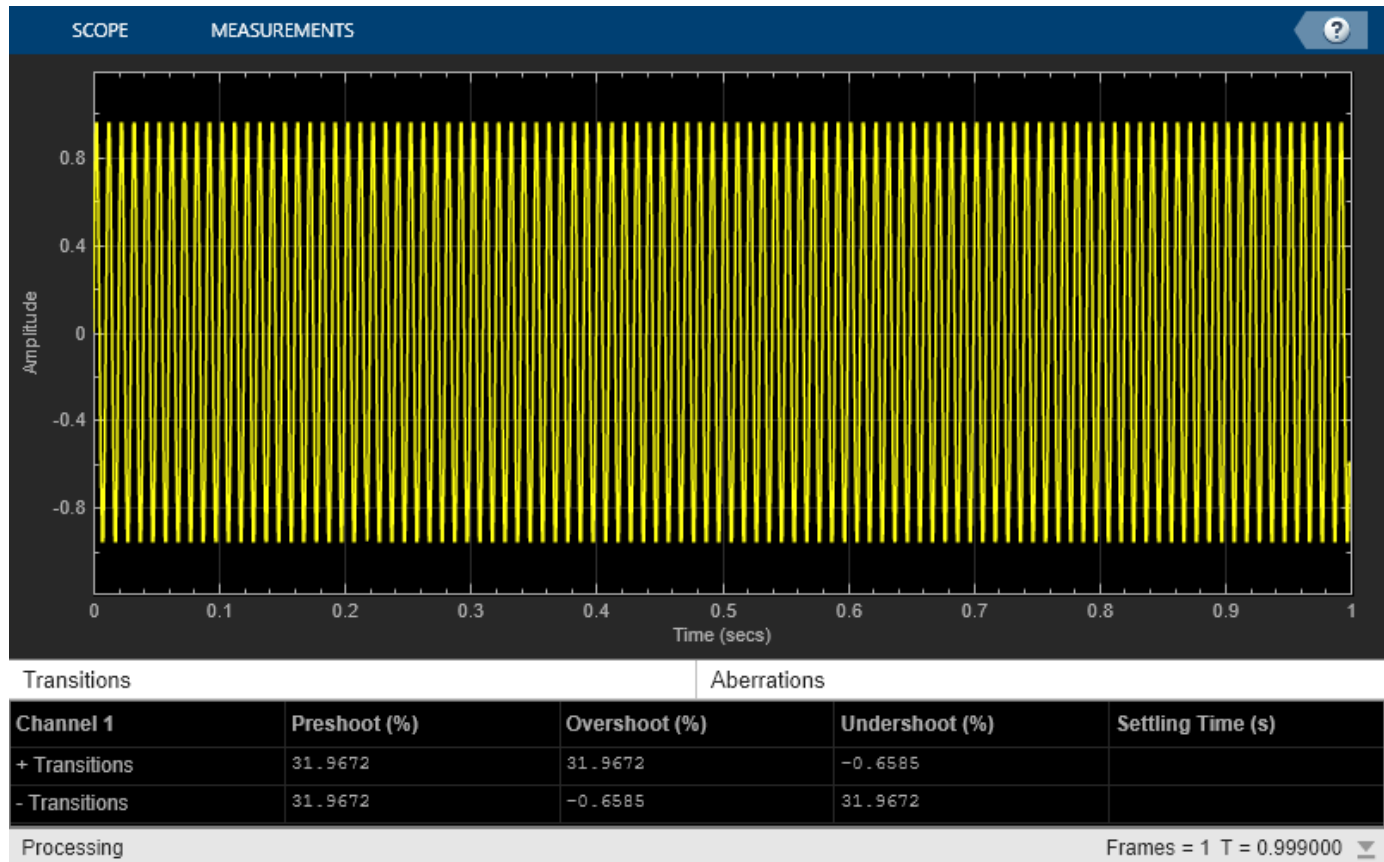


### Aberration Measurements

Enable the scope to show aberration measurements programmatically by setting the `ShowAberrations` property to `true`. Display the sine wave in the scope.

Aberration measurements such as preshoot, overshoot, undershoot, and settling time appear in the **Aberrations** pane at the bottom of the scope.

```
scope.BilevelMeasurements.ShowAberrations = true;
scope(svw);
release(scope);
```

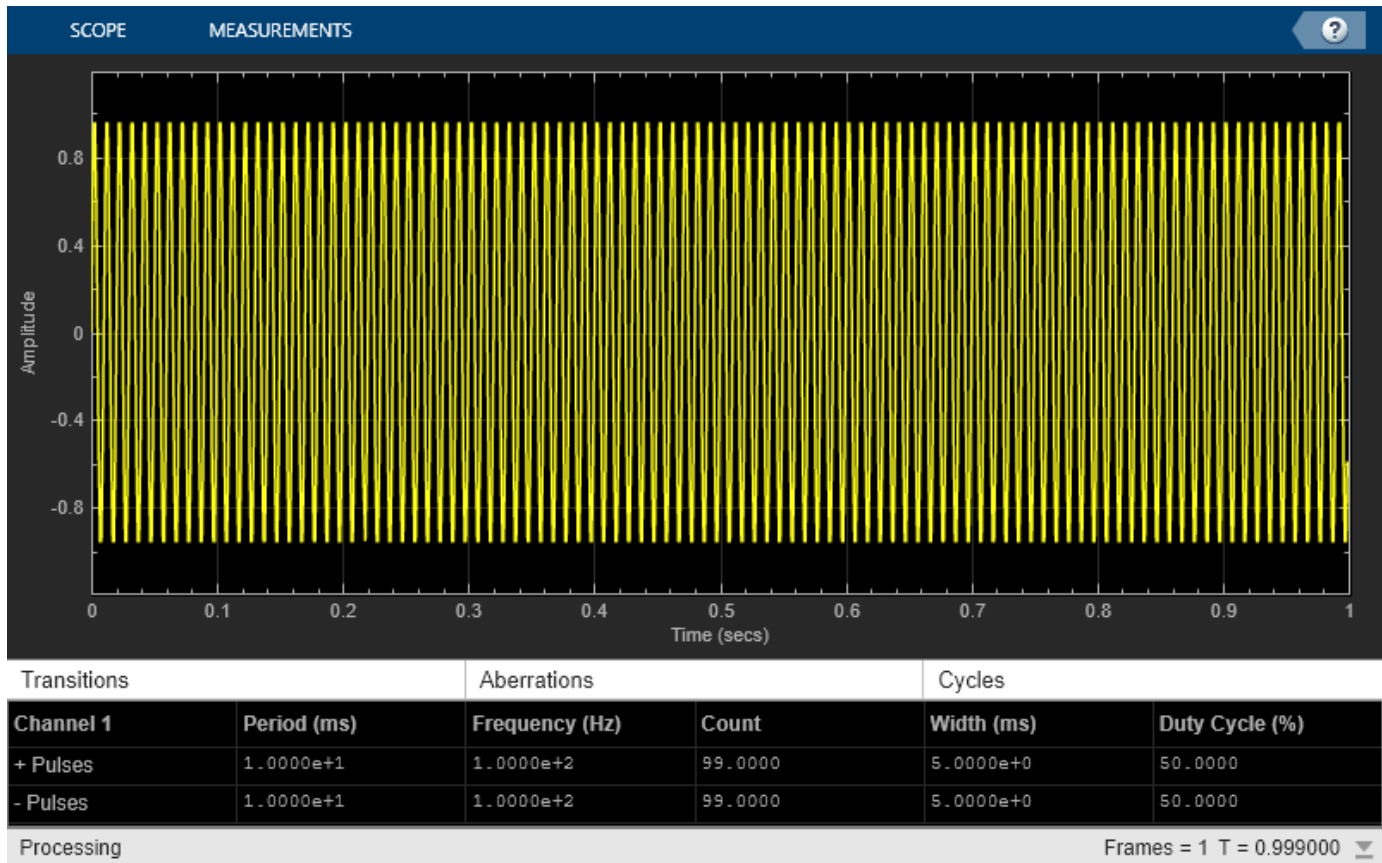


### Cycle Measurements

Enable the scope to show cycles measurements programmatically by setting the `ShowCycles` property to `true`. Display the sine wave in the scope.

Cycle measurements such as period, frequency, pulse width, and duty cycle appear in the **Cycles** pane at the bottom of the scope.

```
scope.BilevelMeasurements.ShowCycles = true;
scope(svw);
release(scope);
```



## More About

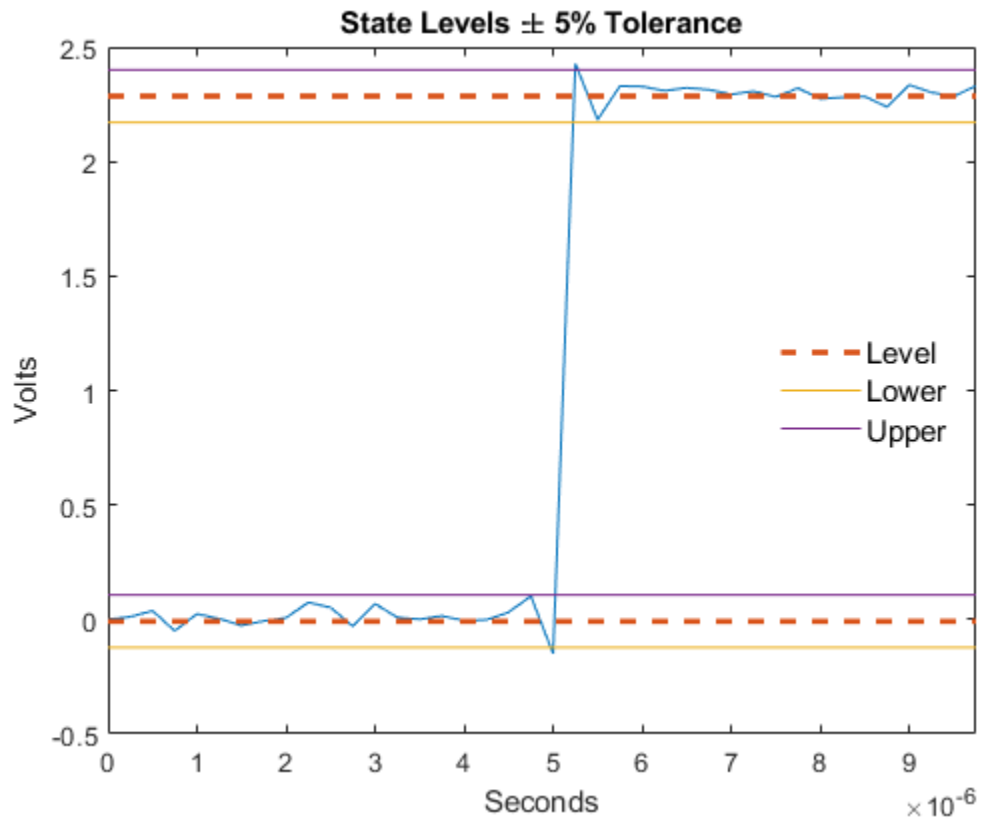
### State-Level Tolerances

You can specify lower- and upper-state boundaries for each state level. Define the boundaries as the state level plus or minus a scalar multiple of the difference between the high state and the low state. To provide a useful tolerance region, specify the scalar as a small number such as 2/100 or 3/100. In general, the  $\alpha\%$  region for the low state is defined as

$$S_1 \pm \frac{\alpha}{100}(S_2 - S_1),$$

where  $S_1$  is the low-state level and  $S_2$  is the high-state level. Replace the first term in the equation with  $S_2$  to obtain the  $\alpha\%$  tolerance region for the high state.

This figure shows lower and upper 5% state boundaries (tolerance regions) for a positive-polarity bilevel waveform. The thick dashed lines indicate the estimated state levels.



## Version History

Introduced in R2022a

## See Also

timescope

## Topics

“Configure Time Scope MATLAB Object”



# SignalStatisticsConfiguration

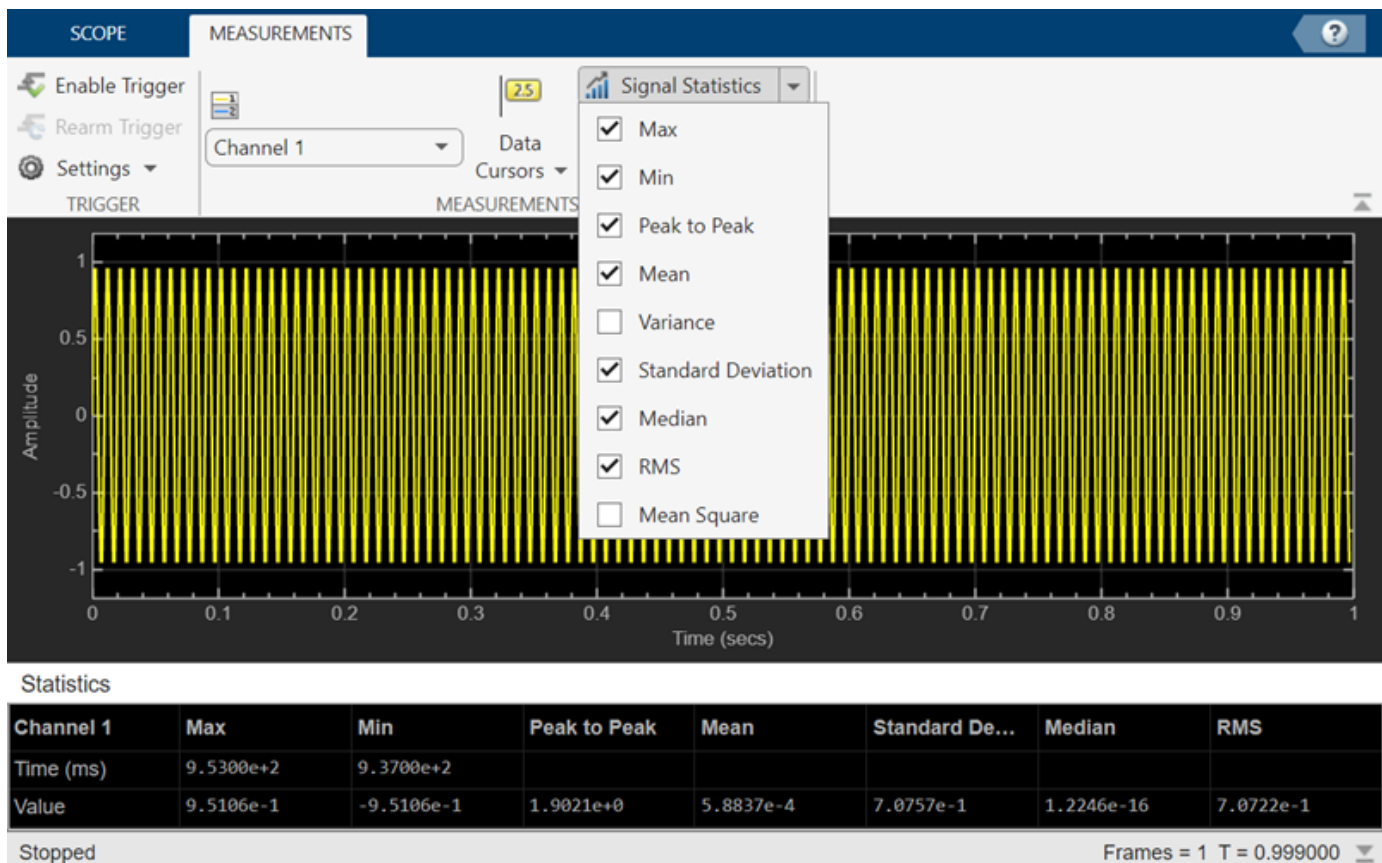
Compute and display signal statistics

## Description

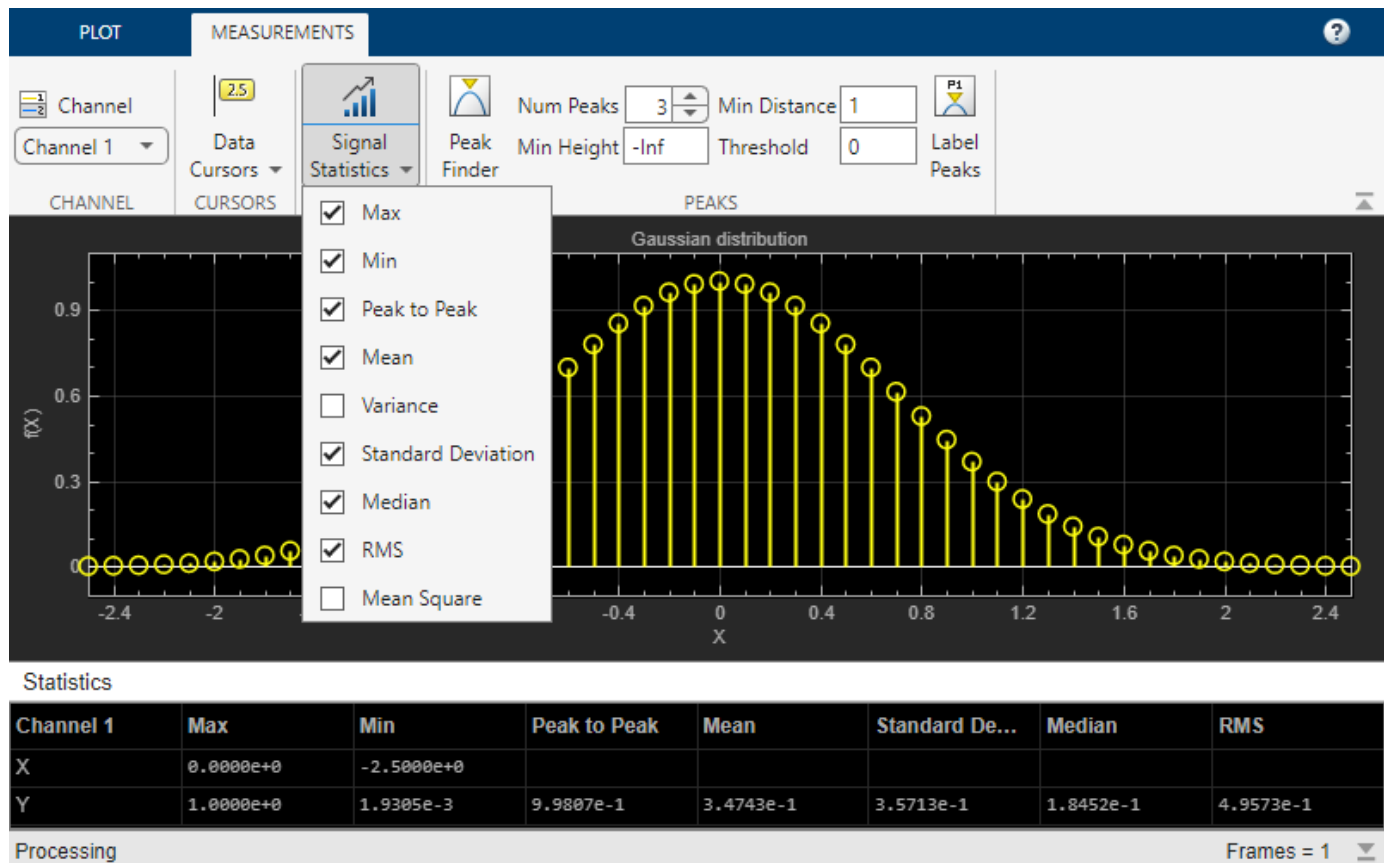
Use the `SignalStatisticsConfiguration` object to measure signal statistics such as maximum, minimum, peak-to-peak value, mean, variance, standard deviation, median, RMS, and mean square.

You can enable the scope to compute and display signal statistics from the toolstrip or from the command line. To enable from the scope interface, click the **Measurements** tab, and then click **Signal Statistics**. A statistics panel appears at the bottom of the scope window. To enable specific statistics, click the **Signal Statistics** drop-down list and select a statistic from the options. The **Statistics** panel shows those statistics.

### Time Scope



### Array Plot



## Creation

### Syntax

```
signalStats = SignalStatisticsConfiguration()
```

### Description

`signalStats = SignalStatisticsConfiguration()` creates a signal statistics configuration object `signalStats`.

### Properties

All properties are tunable.

#### ShowMax — Compute and display maximum

true (default) | false

Compute and display the maximum value, specified as `true` or `false`. The scope computes and displays the maximum value of the portion of the input signal that is currently on display in the scope.

**Scope Window Use**

On the **Measurements** tab, select **Signal Statistics** and then click **Signal Statistics** to select **Max**.

Data Types: `logical`

**ShowMin — Compute and display minimum**

`true` (default) | `false`

Compute and display the minimum value, specified as `true` or `false`. The scope computes and displays the minimum value of the portion of the input signal that is currently on display in the scope.

**Scope Window Use**

On the **Measurements** tab, select **Signal Statistics** and then click **Signal Statistics** to select **Min**.

Data Types: `logical`

**ShowPeakToPeak — Compute and display peak-to-peak values**

`true` (default) | `false`

Compute and display the peak-to-peak values, specified as `true` or `false`. The scope computes and displays the peak-to-peak values from the portion of the input signal that is currently on display in the scope.

**Scope Window Use**

On the **Measurements** tab, select **Signal Statistics** and then click **Signal Statistics** to select **Peak to Peak**.

Data Types: `logical`

**ShowMean — Compute and display mean**

`true` (default) | `false`

Compute and display the mean value, specified as `true` or `false`. The scope computes and displays the mean value of the portion of the input signal that is currently on display in the scope.

**Scope Window Use**

On the **Measurements** tab, select **Signal Statistics** and then click **Signal Statistics** to select **Mean**.

Data Types: `logical`

**ShowVariance — Compute and display variance**

`false` (default) | `true`

Compute and display the variance, specified as `true` or `false`. The scope computes and displays the variance of the portion of the input signal that is currently on display in the scope.

**Scope Window Use**

On the **Measurements** tab, select **Signal Statistics** and then click **Signal Statistics** to select **Variance**.

Data Types: `logical`

**ShowStandardDeviation — Compute and display standard deviation**

`true` (default) | `false`

Compute and display the standard deviation, specified as `true` or `false`. The scope computes and displays the standard deviation of the portion of the input signal that is currently on display in the scope.

### Scope Window Use

On the **Measurements** tab, select **Signal Statistics** and then click **Signal Statistics** to select **Standard Deviation**.

Data Types: `logical`

### ShowMedian — Compute and display median

`true` (default) | `false`

Compute and display the median, specified as `true` or `false`. The scope computes and displays the median of the portion of the input signal that is currently on display in the scope.

### Scope Window Use

On the **Measurements** tab, select **Signal Statistics** and then click **Signal Statistics** to select **Median**.

Data Types: `logical`

### ShowRMS — Compute and display RMS

`true` (default) | `false`

Compute and display the RMS, specified as `true` or `false`. The scope computes and displays the RMS of the portion of the input signal that is currently on display in the scope.

### Scope Window Use

On the **Measurements** tab, select **Signal Statistics** and then click **Signal Statistics** to select **RMS**.

Data Types: `logical`

### ShowMeanSquare — Compute and display mean square

`false` (default) | `true`

Compute and display the mean square, specified as `true` or `false`. The scope computes and displays the mean square of the portion of the input signal that is currently on display in the scope.

### Scope Window Use

On the **Measurements** tab, select **Signal Statistics** and then click **Signal Statistics** to select **Mean Square**.

Data Types: `logical`

### Enabled — Enable signal statistics measurements

`false` (default) | `true`

Enable signal statistics measurements, specified as `true` or `false`. Set this property to `true` to enable signal statistics measurements.

### Scope Window Use

On the **Measurements** tab, select **Signal Statistics**().

Data Types: `logical`

## Examples

### Configure Signal Statistics Programmatically in Time Scope MATLAB Object

Create a sine wave and view it in the Time Scope. Enable the scope programmatically to compute the signal statistics.

The object supports these signal statistics:

- Maximum
- Minimum
- Mean
- Median
- RMS
- Peak to peak
- Variance
- Standard deviation
- Mean square

### Initialization

Create the input sine wave using the `sin` function. Create a `timescope` MATLAB® object to display the signal. Set the `TimeSpan` property to 1 second.

```
f = 100;
fs = 1000;
swv = sin(2.*pi.*f.*(0:1/fs:1-1/fs)).';
scope = timescope(SampleRate=fs,...
    TimeSpanSource="property",...
    TimeSpan=1);
```

### Signal Statistics

Enable the scope to show signal statistics programmatically by setting the `SignalStatistics > Enabled` property to `true`.

```
scope.SignalStatistics.Enabled = true;
```

By default, the scope enables the following measurements.

```
scope.SignalStatistics
```

```
ans =
    SignalStatisticsConfiguration with properties:

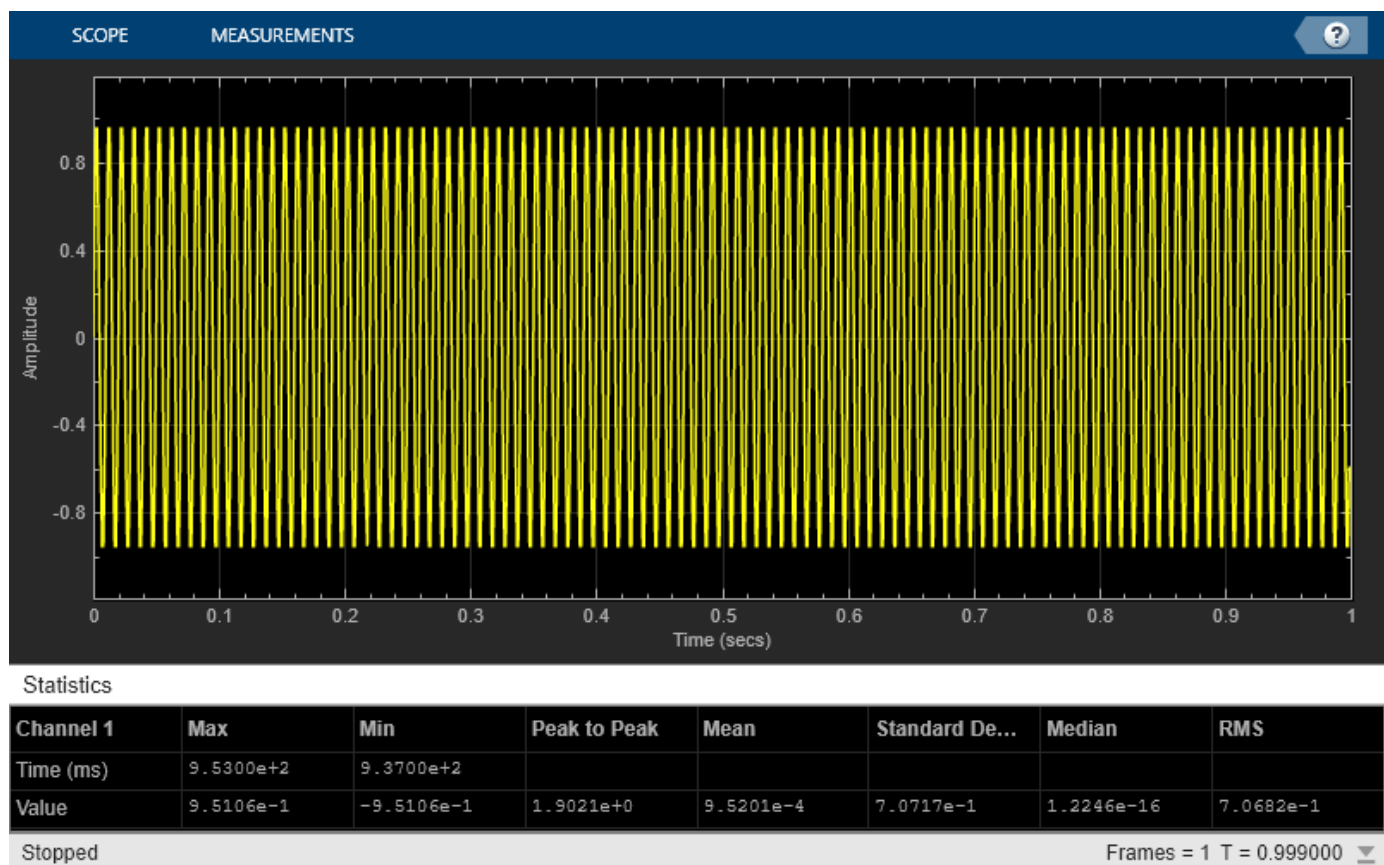
        ShowMax: 1
        ShowMin: 1
    ShowPeakToPeak: 1
        ShowMean: 1
        ShowVariance: 0
```

```
ShowStandardDeviation: 1
    ShowMedian: 1
    ShowRMS: 1
    ShowMeanSquare: 0
    Enabled: 1
```

Display the sine wave in the scope. A Statistics pane appears at the bottom of the scope window displaying the statistics for the portion of the signal that you can see in the scope.

If you use the zoom options on the scope, the statistics automatically adjust to the time range in the display.

```
scope(swv);
release(scope);
```



## Version History

Introduced in R2022a

## See Also

timescope | dsp.ArrayPlot

**Topics**

“Configure Time Scope MATLAB Object”

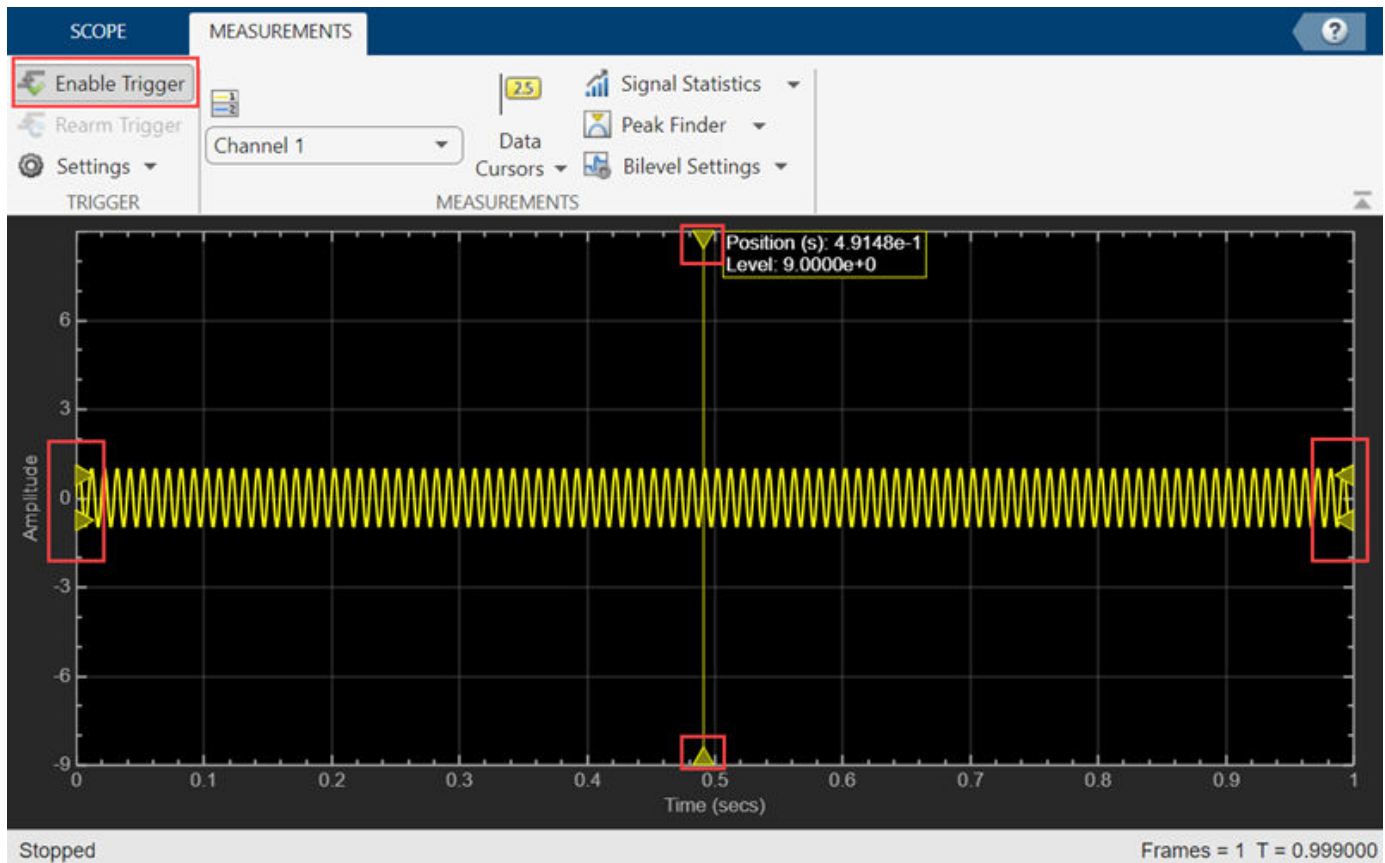
# TriggerConfiguration

Trigger measurements properties for scope

## Description

Use the TriggerConfiguration object to define a trigger event to identify the simulation time of specified input signal characteristics. You can use trigger events to stabilize periodic signals such as a sine wave or capture nonperiodic signals such as a pulse that occurs intermittently.

You can enable the trigger events either from the Time Scope toolstrip or from the command line. To enable a trigger event from the toolstrip, open the **Measurements** tab and click **Enable Trigger**.



## Creation

## Syntax

```
trigger = TriggerConfiguration()
```



## Description

`trigger = TriggerConfiguration()` creates a trigger configuration object `trigger`.

## Properties

All properties are tunable.

For more information on these triggers and the associated parameters, see “Source/Type and Levels/ Timing Panes” (Simulink).

### Mode — Display update mode

"auto" (default) | "normal" | "once"

Display update mode, specified as one of these:

- "auto" -- Display data from the last trigger event. If no event occurs after one time span, display the last available data.
- "normal" -- Display data from the last trigger event. If no event occurs, the display remains blank.
- "once" -- Display data from the last trigger event and freeze the display. If no event occurs, the

display remains blank. Click the **Rearm** button (  ) to look for the next trigger event.

### Scope Window Use

On the **Measurements** tab, in the **Trigger** section, click **Settings** and set **Mode** to one of the available options.

Data Types: char | string

### Type — Type of trigger

"edge" (default) | "pulse-width" | "transition" | "runt" | "window" | "timeout"

Type of trigger, specified as one of the following:

- "edge" -- Trigger when the signal crosses a threshold.
- "pulse-width" -- Trigger when the signal crosses a low threshold and a high threshold twice within a specified time.
- "transition" -- Trigger on the rising or falling edge of a signal that crosses the high and low levels within a specified time range.
- "runt" -- Trigger when a signal crosses a low threshold or a high threshold twice within a specified time.
- "window" -- Trigger when a signal stays within or outside a region defined by the high and low thresholds for a specified time.
- "timeout" -- Trigger when a signal stays above or below a threshold longer than a specified time.

### Scope Window Use

On the **Measurements** tab, in the **Trigger** section, click **Settings** and set **Type** to one of the available options.

Data Types: char | string

### **Polarity — Trigger polarity**

"rising" (default) | "falling" | "either" | "positive" | "negative" | "rise-time" | "fall-time" | "inside" | "outside"

Trigger polarity, specified as one of the following:

- "rising", "falling", or "either" -- When Type is set to "edge" or "timeout".
- "positive", "negative", or "either" -- When Type is set to "pulse-width" or "runt".
- "rise-time", "fall-time", or "either" -- When Type is set to "transition".
- "inside", "outside", or "either" -- When Type is set to "window".

### **Scope Window Use**

On the **Measurements** tab, in the **Trigger** section, click **Settings** and set **Polarity** to one of the available options.

Data Types: char | string

### **AutoLevel — Automatic thresholding**

true (default) | false

Automatic thresholding of edge-triggered signal, specified as true or false. When you set this property to false, specify the threshold manually using the **Level** property.

### **Scope Window Use**

On the **Measurements** tab, in the **Trigger** section, click **Settings**. Under **Level Settings**, select **Auto Level**.

Data Types: logical

### **Position — Horizontal position of trigger**

50 (default) | positive scalar in the range (0 100]

Horizontal position of the trigger on the screen, specified as a positive scalar in the range (0 100].

### **Scope Window Use**

On the **Measurements** tab, in the **Trigger** section, click **Settings** and specify a positive scalar less than or equal to 100 in the **Position (%)** box.

Data Types: double

### **Level — Threshold of edge-triggered signal**

0 (default) | real scalar

Threshold of an edge-triggered signal, specified as a finite real scalar.

### **Dependency**

To enable this property, set **AutoLevel** to false and **Type** to "edge" or "timeout".

### **Scope Window Use**

On the **Measurements** tab, in the **Trigger** section, click **Settings**. Under **Level Settings**, clear **Auto Level** and specify a real scalar in the **Level** box.

To enable this property, set **Type** to Edge or Timeout.

Data Types: double

### Hysteresis – Noise reject value

0 (default) | real scalar

Noise reject value, specified as a finite real scalar. For more information on hysteresis, see “Hysteresis of Trigger Signals” (Simulink).

#### Dependency

To enable this property, set `AutoLevel` to false and `Type` to "edge" or "timeout".

#### Scope Window Use

On the **Measurements** tab, in the **Trigger** section, click **Settings**. Under **Level Settings**, clear **Auto Level** and specify a real scalar in the **Hysteresis** box.

To enable this property, set **Type** to Edge or Timeout.

Data Types: double

### LowLevel – Lower trigger level of window-triggered signal

0.2 (default) | real scalar

Lower trigger level of window-triggered signal, specified as a finite real scalar.

#### Dependency

To enable this property, set `AutoLevel` to false and `Type` to "pulse-width", "transition", "runt", or "window".

#### Scope Window Use

On the **Measurements** tab, in the **Trigger** section, click **Settings**. Under **Level Settings**, clear **Auto Level** and specify a real scalar in the **Low** box.

To enable this property, set **Type** to Pulse Width, Transition, Runt, or Window.

Data Types: double

### HighLevel – Higher trigger level of window-triggered signal

2.3 (default) | real scalar

Higher trigger level of window-triggered signal, specified as a finite real scalar.

#### Dependency

To enable this property, set `AutoLevel` to false and `Type` to "pulse-width", "transition", "runt", or "window".

#### Scope Window Use

On the **Measurements** tab, in the **Trigger** section, click **Settings**. Under **Level Settings**, clear **Auto Level** and specify a real scalar in the **High** box.

To enable this property, set **Type** to Pulse Width, Transition, Runt, or Window.

Data Types: double

### **MinPulseWidth — Minimum pulse width for pulse or runt-triggered signal**

0 (default) | nonnegative scalar

Minimum pulse width for a pulse or runt-triggered signal, specified as a nonnegative scalar.

#### **Dependency**

To enable this property, set **Type** to "pulse-width" or "runt".

#### **Scope Window Use**

On the **Measurements** tab, in the **Trigger** section, click **Settings**. Under **Level Settings**, specify a nonnegative scalar in the **Min Width (s)** box.

To enable this property, set **Type** to Pulse Width or Runt.

Data Types: double

### **MaxPulseWidth — Maximum pulse width for pulse or runt-triggered signal**

Inf (default) | nonnegative scalar

Maximum pulse width for a pulse or runt-triggered signal, specified as a nonnegative scalar.

#### **Dependency**

To enable this property, set **Type** to "pulse-width" or "runt".

#### **Scope Window Use**

On the **Measurements** tab, in the **Trigger** section, click **Settings**. Under **Level Settings**, specify a nonnegative scalar in the **Max Width (s)** box.

To enable this property, set **Type** to Pulse Width or Runt.

Data Types: double

### **MinDuration — Minimum duration for transition or window-triggered signal**

0 (default) | nonnegative scalar

Minimum duration for a transition or window-triggered signal, specified as a nonnegative scalar.

#### **Dependency**

To enable this property, set **Type** to "transition" or "window".

#### **Scope Window Use**

On the **Measurements** tab, in the **Trigger** section, click **Settings**. Under **Level Settings**, specify a nonnegative scalar in the **Min Time (s)** box.

To enable this property, set **Type** to Transition or Window.

Data Types: double

### **MaxDuration — Maximum duration for transition or window-triggered signal**

Inf (default) | nonnegative scalar

Maximum time duration for a transition or window-triggered signal, specified as a nonnegative scalar.

**Dependency**

To enable this property, set **Type** to "transition" or "window".

**Scope Window Use**

On the **Measurements** tab, in the **Trigger** section, click **Settings**. Under **Level Settings**, specify a nonnegative scalar in the **Max Time (s)** box.

To enable this property, set **Type** to Transition or Window.

Data Types: double

**Timeout — Timeout duration**

0 (default) | nonnegative scalar

Timeout duration for a timeout-triggered signal, specified as a nonnegative scalar.

**Dependency**

To enable this property, set **Type** to "timeout".

**Scope Window Use**

On the **Measurements** tab, in the **Trigger** section, click **Settings**. Under **Level Settings**, specify a nonnegative scalar in the **Timeout (s)** box.

To enable this property, set **Type** to Timeout.

Data Types: double

**Delay — Trigger offset**

0 (default) | real scalar

Trigger offset in seconds, specified as a finite real scalar.

**Scope Window Use**

On the **Measurements** tab, in the **Trigger** section, click **Settings**. Under **Delay/Holdoff**, specify a real scalar in the **Delay (s)** box.

Data Types: double

**Holdoff — Minimum time between triggers**

0 (default) | nonnegative scalar

Minimum time between trigger events, specified as a finite nonnegative scalar.

**Scope Window Use**

On the **Measurements** tab, in the **Trigger** section, click **Settings**. Under **Delay/Holdoff**, specify a nonnegative scalar in the **Holdoff (s)** box.

Data Types: double

**Channel — Trigger channel**

1 (default) | positive integer

Trigger channel, specified as a positive integer.

**Scope Window Use**

On the **Measurements** tab, in the **Measurements** section, select a channel.

Data Types: double

**Enabled — Enable trigger**

false (default) | true

Enable trigger, specified as true or false. Set this property to true to enable trigger.

**Scope Window Use**

On the **Measurements** tab, in the **Trigger** section, click **Enable Trigger**.

Data Types: logical

**Examples****Enable Trigger Programmatically in Time Scope MATLAB Object**

View a sine wave in the Time Scope window. This sine wave is streaming constantly in the display and cannot be captured without stabilization. To stabilize the sine wave, enable a trigger event programmatically on the scope display using the Enabled property of the TriggerConfiguration object. Alternatively, you can enable the trigger by clicking the **Enable Trigger** button on the **Measurements** tab of the toolbar.

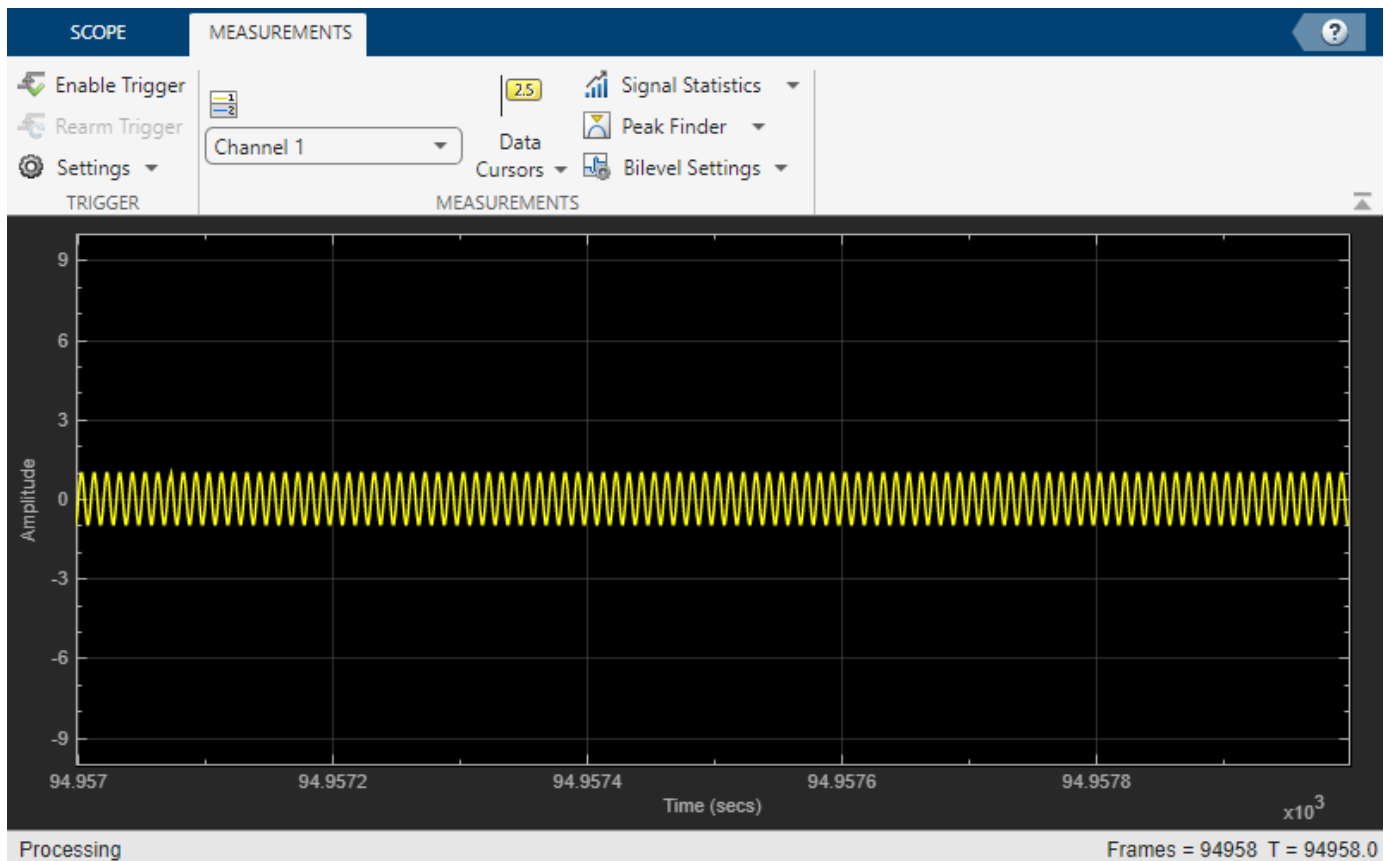
**Create Sine Wave**

Create the input sine wave using the sin function. Create a timescope MATLAB object to display the signal. Set the TimeSpan property to 1 second.

```
f = 100;  
fs = 1000;  
swv = sin(2.*pi.*f.*(0:1/fs:1-1/fs)).';  
scopeNoTrigger = timescope(SampleRate=fs,...  
    TimeSpanSource="property", ...  
    TimeSpan=1);
```

Display the sine wave in the scope. You can see that the signal in the scope is constantly moving.

```
while(1)  
    scopeNoTrigger(swv)  
end  
release(scopeNoTrigger)
```



## Enable Trigger

Now enable a trigger event to stabilize the signal.

You can enable the trigger event in the scope during simulation or enable the trigger event programmatically when creating the object.

To use the programmatic approach, create another `timescope` object and enable the trigger event programmatically while creating the object.

```
scope = timescope(SampleRate=fs,...
    TimeSpanSource="property",...
    TimeSpan=1);
scope.Trigger.Enabled = true;
scope.Trigger.Type = "transition";
scope.Trigger
```

TriggerConfiguration with properties:

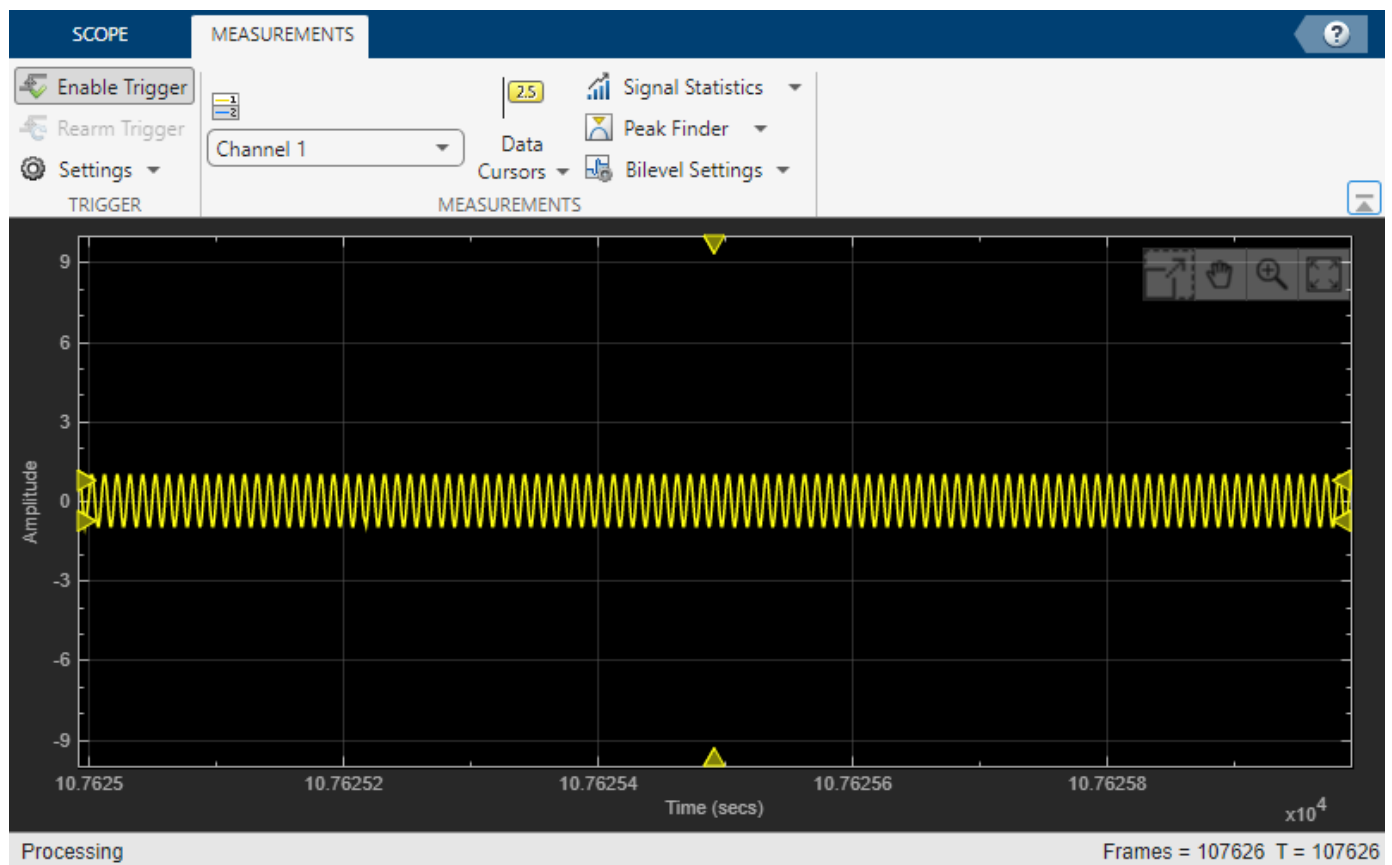
```
Mode: 'auto'
Type: 'transition'
Polarity: 'rise-time'
AutoLevel: 1
Position: 50
LowLevel: 0.2000
HighLevel: 2.3000
MinDuration: 0
```

```
MaxDuration: Inf
Delay: 0
Holdoff: 0
Channel: 1
Enabled: 1
```

Stream in the sine wave signal again.

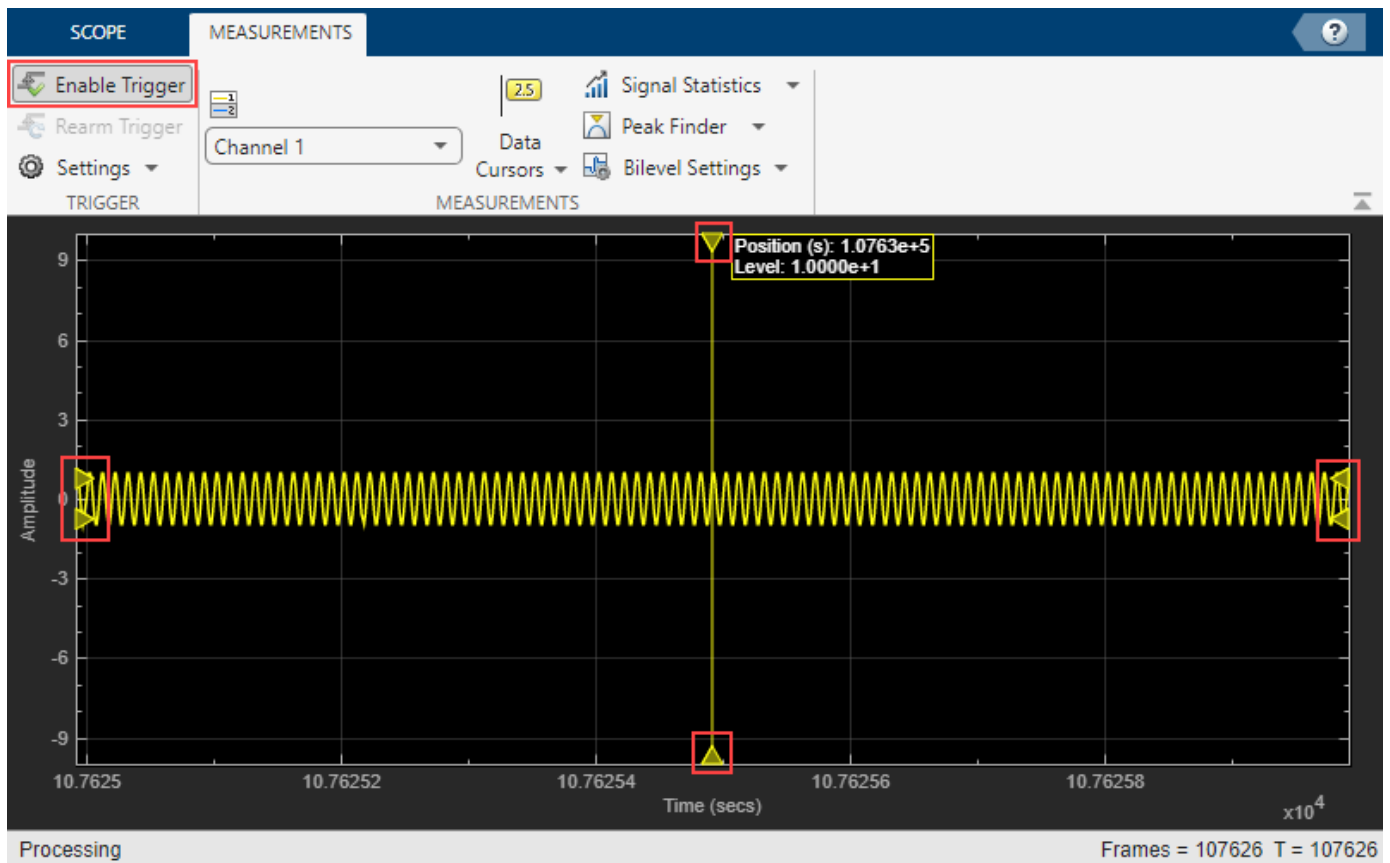
```
while(1)
    scope(svw)
end
release(scope)
```

The display freezes once you enable the trigger.



The triangle markers show the trigger positions and levels. For more information on the trigger, hover over the triangle.





## Version History

Introduced in R2022a

### See Also

timescope

### Topics

“Configure Time Scope MATLAB Object”

## ChannelMeasurementsConfiguration

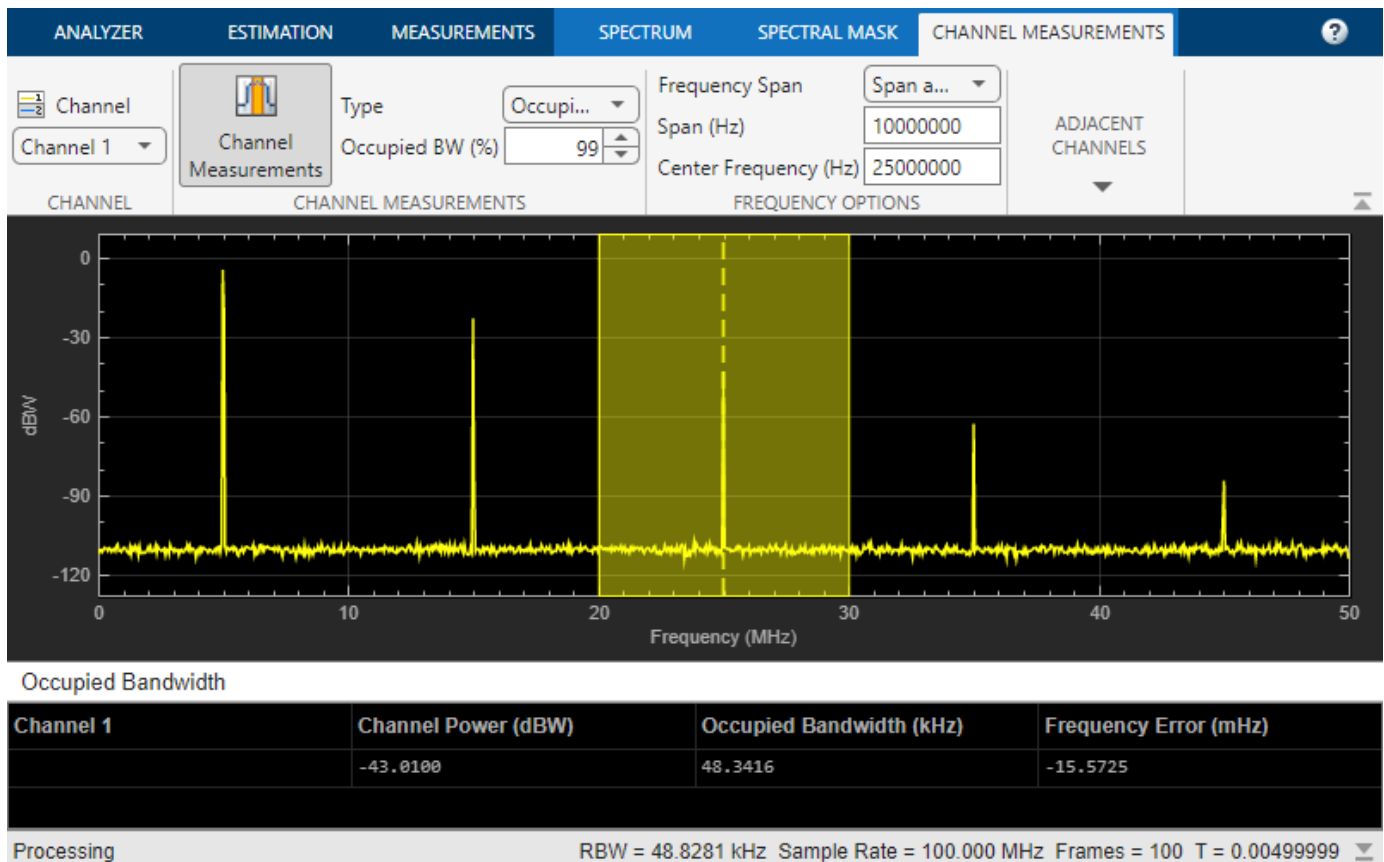
Compute and display occupied bandwidth or adjacent channel power ratio on the Spectrum Analyzer

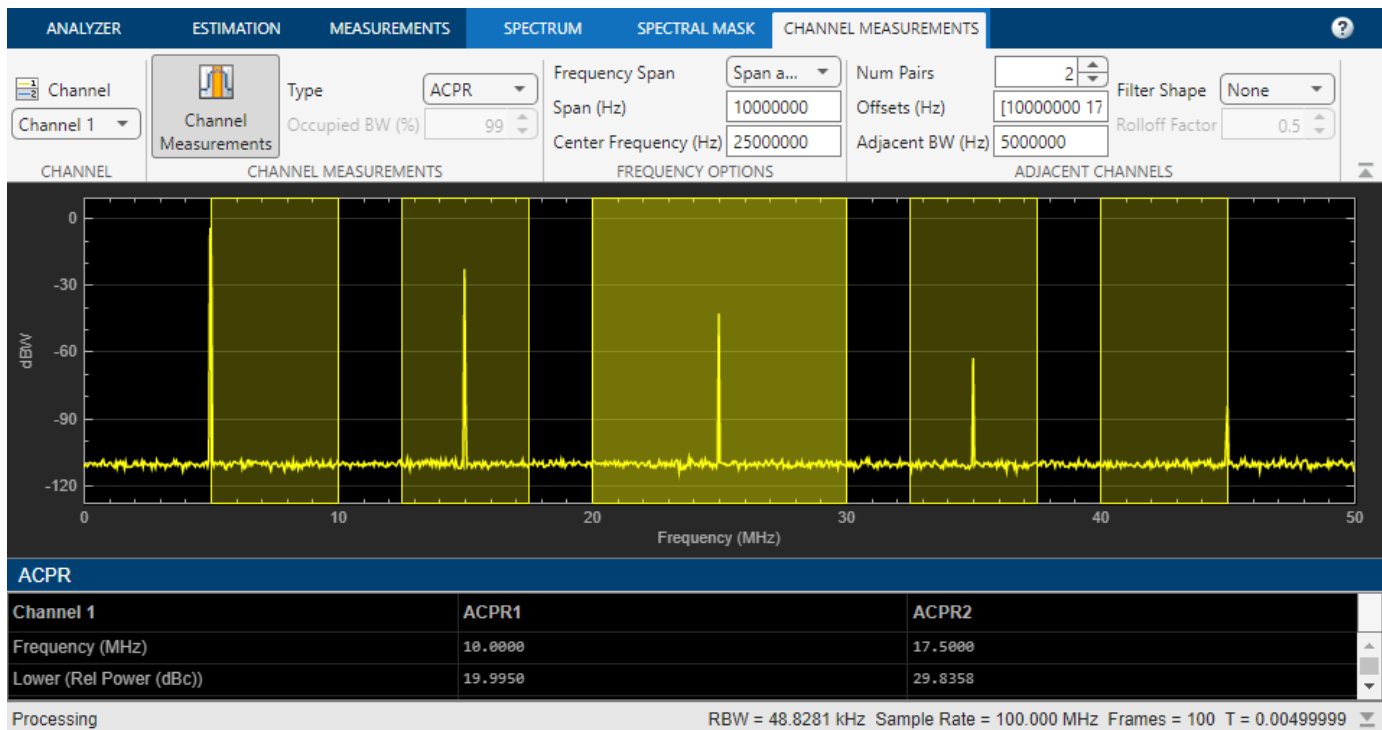
### Description

Use the ChannelMeasurementsConfiguration object to compute and display the occupied bandwidth or adjacent channel power ratio on the Spectrum Analyzer.

You can specify the occupied bandwidth or adjacent channel power ratio (ACPR) settings, frequency span, center frequency, and start and stop frequencies, and control the channel measurement settings from the Spectrum Analyzer toolstrip or from the command line. Both the spectrumAnalyzer object and the SpectrumAnalyzerConfiguration object support the ChannelMeasurementsConfiguration object in the command line.

To modify the channel measurement settings on the Spectrum Analyzer toolstrip, click the **Channel Measurements** tab.





## Creation

### Syntax

```
channMeas = ChannelMeasurementsConfiguration()
```

### Description

`channMeas = ChannelMeasurementsConfiguration()` creates a channel measurements configuration object `channMeas`.

### Properties

All properties are tunable.

#### Type — Type of measurement data to display

"occupied-bandwidth" (default) | "acpr"

Type of measurement data to display, specified as either "occupied-bandwidth" or "acpr".

#### Scope Window Use

Click the **Channel Measurements** tab on the Spectrum Analyzer toolstrip. In the **Channel Measurements** section, set **Type** to either Occupied BW or ACPR.

The **Channel Measurements** tab appears when you select **Spectrum** in the **Analyzer** tab.

Data Types: char | string

### FrequencySpan — Frequency span mode

"span-and-center-frequency" (default) | "start-and-stop-frequencies"

Frequency span mode, specified as one of the following:

- "span-and-center-frequency" -- Measure over a frequency range specified in Span Hz and around the frequency value specified in CenterFrequency property.
- "start-and-stop-frequencies" -- Measure over the frequency range [StartFrequency, StopFrequency].

#### Scope Window Use

Click the **Channel Measurements** tab on the Spectrum Analyzer toolstrip. In the **Frequency Options** section, set **Frequency Span** to either Span and Center Frequency or Start and Stop Frequencies.

The **Channel Measurements** tab appears when you select **Spectrum** in the **Analyzer** tab.

Data Types: char | string

### Span — Frequency span

2000 (default) | positive scalar

Frequency span over which the object computes the channel measurements, specified as a positive scalar in Hz.

#### Dependency

To enable this property, set the FrequencySpan property to "span-and-center-frequency".

#### Scope Window Use

Click the **Channel Measurements** tab on the Spectrum Analyzer toolstrip. In the **Frequency Options** section, set **Frequency Span** to Span and Center Frequency and **Span (Hz)** to a positive scalar.

The **Channel Measurements** tab appears when you select **Spectrum** in the **Analyzer** tab.

Data Types: double

### CenterFrequency — Center frequency of span

0 (default) | real scalar

Center frequency of the span over which the object computes the channel measurements, specified as a real scalar in Hz.

#### Dependency

To enable this property, set the FrequencySpan property to "span-and-center-frequency".

#### Scope Window Use

Click the **Channel Measurements** tab on the Spectrum Analyzer toolstrip. In the **Frequency Options** section, set **Frequency Span** to Span and Center Frequency and **Center Frequency (Hz)** to a real scalar.

The **Channel Measurements** tab appears when you select **Spectrum** in the **Analyzer** tab.

Data Types: double

### **StartFrequency — Start frequency**

-1000 (default) | real scalar

Start frequency over which the object computes the channel measurements, specified as a real scalar in Hz.

#### **Dependency**

To enable this property, set the FrequencySpan property to "start-and-stop-frequencies".

#### **Scope Window Use**

Click the **Channel Measurements** tab on the Spectrum Analyzer toolstrip. In the **Frequency Options** section, set **Frequency Span** to Start and Stop Frequencies and **Start Frequency (Hz)** to a real scalar.

The **Channel Measurements** tab appears when you select **Spectrum** in the **Analyzer** tab.

Data Types: double

### **StopFrequency — Stop frequency**

1000 (default) | real scalar

Stop frequency over which the object computes the channel measurements, specified as a real scalar in Hz.

#### **Dependency**

To enable this property, set the FrequencySpan property to "start-and-stop-frequencies".

#### **Scope Window Use**

Click the **Channel Measurements** tab on the Spectrum Analyzer toolstrip. In the **Frequency Options** section, set **Frequency Span** to Start and Stop Frequencies and **Stop Frequency (Hz)** to a real scalar.

The **Channel Measurements** tab appears when you select **Spectrum** in the **Analyzer** tab.

Data Types: double

### **PercentOccupiedBW — Percentage of power to compute occupied bandwidth**

99 (default) | positive scalar

Percentage of power over which the object computes the occupied bandwidth, specified as a positive scalar.

#### **Dependency**

To enable this property, set the Type property to "occupied-bandwidth".

#### **Scope Window Use**

Click the **Channel Measurements** tab on the Spectrum Analyzer toolstrip. In the **Channel Measurements** section, set **Type** to Occupied BW and **Occupied BW (%)** to a positive scalar.

The **Channel Measurements** tab appears when you select **Spectrum** in the **Analyzer** tab.

Data Types: double

### **AdjacentBW — Adjacent channel bandwidth**

1000 (default) | positive scalar

Adjacent channel bandwidth, specified as a positive scalar.

#### **Dependency**

To enable this property, set the Type property to "acpr".

#### **Scope Window Use**

Click the **Channel Measurements** tab on the Spectrum Analyzer toolstrip. In the **Channel Measurements** section, set **Type** to ACPR. In the **Adjacent Channels** section, set **Adjacent BW (Hz)** to a positive scalar.

The **Channel Measurements** tab appears when you select **Spectrum** in the **Analyzer** tab.

Data Types: double

### **FilterShape — Filter shape**

"none" (default) | "gaussian" | "rrc"

Filter shape for main and adjacent channels, specified as "none", "gaussian", or "rrc".

#### **Dependency**

To enable this property, set the Type property to "acpr".

#### **Scope Window Use**

Click the **Channel Measurements** tab on the Spectrum Analyzer toolstrip. In the **Channel Measurements** section, set **Type** to ACPR. In the **Adjacent Channels** section, set **Filter Shape**.

The **Channel Measurements** tab appears when you select **Spectrum** in the **Analyzer** tab.

Data Types: char | string

### **FilterCoeff — Channel filter coefficient**

0.5 (default) | real scalar in range [0, 1]

Channel filter coefficient, specified as a real scalar in the range [0, 1].

#### **Dependency**

To enable this property, set the Type property to "acpr" and the FilterShape property to either "gaussian" or "rrc".

#### **Scope Window Use**

Click the **Channel Measurements** tab on the Spectrum Analyzer toolstrip. In the **Channel Measurements** section, set **Type** to ACPR.

In the **Adjacent Channels** section, set:

- **Filter Shape** to RRC and specify **Roll-off Factor** as a real scalar in the range [0,1].
- **Filter Shape** to Gaussian and specify **BT Product** as a real scalar in the range [0,1].

The **Channel Measurements** tab appears when you select **Spectrum** in the **Analyzer** tab.

Data Types: double

#### **NumOffsets — Number of adjacent channel pairs**

2 (default) | positive integer in range [1, 12]

Number of adjacent channel pairs, specified as a positive integer in the range [1, 12].

#### **Dependency**

To enable this property, set the Type property to "acpr".

#### **Scope Window Use**

Click the **Channel Measurements** tab on the Spectrum Analyzer toolstrip. In the **Channel Measurements** section, set **Type** to ACPR. In the **Adjacent Channels** section, set **Num Pairs** to a positive integer in the range [1, 12].

The **Channel Measurements** tab appears when you select **Spectrum** in the **Analyzer** tab.

Data Types: double

#### **ACPROffsets — Frequency of adjacent channel relative to center frequency of main channel**

[2000 3500] (default) | vector

Frequency of the adjacent channel relative to the center frequency of the main channel, specified as a real vector of length equal to the number of offset pairs specified in NumOffsets.

#### **Dependency**

To enable this property, set the Type property to "acpr".

#### **Scope Window Use**

Click the **Channel Measurements** tab on the Spectrum Analyzer toolstrip. In the **Channel Measurements** section, set **Type** to ACPR. In the **Adjacent Channels** section, set **Offset (Hz)**.

The **Channel Measurements** tab appears when you select **Spectrum** in the **Analyzer** tab.

Data Types: double

#### **Enabled — Channel measurements flag**

false (default) | true

Channel measurements flag, specified as true or false.

#### **Scope Window Use**

Click the **Channel Measurements** tab on the Spectrum Analyzer toolstrip. In the **Channel Measurements** section, select **Channel Measurements**.

The **Channel Measurements** tab appears when you select **Spectrum** in the **Analyzer** tab.

Data Types: logical

## **Examples**

### Obtain Measurements Data Programmatically for spectrumAnalyzer object

Compute and display the power spectrum of a noisy sinusoidal input signal using the spectrumAnalyzer MATLAB® object. Measure the peaks, cursor placements, adjacent channel power ratio, and distortion values in the spectrum by enabling these properties:

- PeakFinder
- CursorMeasurements
- ChannelMeasurements
- DistortionMeasurements

#### Initialization

The input sine wave has two frequencies: 1000 Hz and 5000 Hz. Create two dsp.SinWave System objects to generate these two frequencies. Create a spectrumAnalyzer object to compute and display the power spectrum.

```
Fs = 44100;
Sineobject1 = dsp.SinWave(SamplesPerFrame=1024,PhaseOffset=10,...
    SampleRate=Fs,Frequency=1000);
Sineobject2 = dsp.SinWave(SamplesPerFrame=1024,...
    SampleRate=Fs,Frequency=5000);
SA = spectrumAnalyzer(SampleRate=Fs,SpectrumType="power",...
    PlotAsTwoSidedSpectrum=false,ChannelNames={'Power spectrum of the input'},...
    YLimits=[-120 40],ShowLegend=true);
```

#### Enable Measurements Data

To obtain the measurements, set the Enabled property to true.

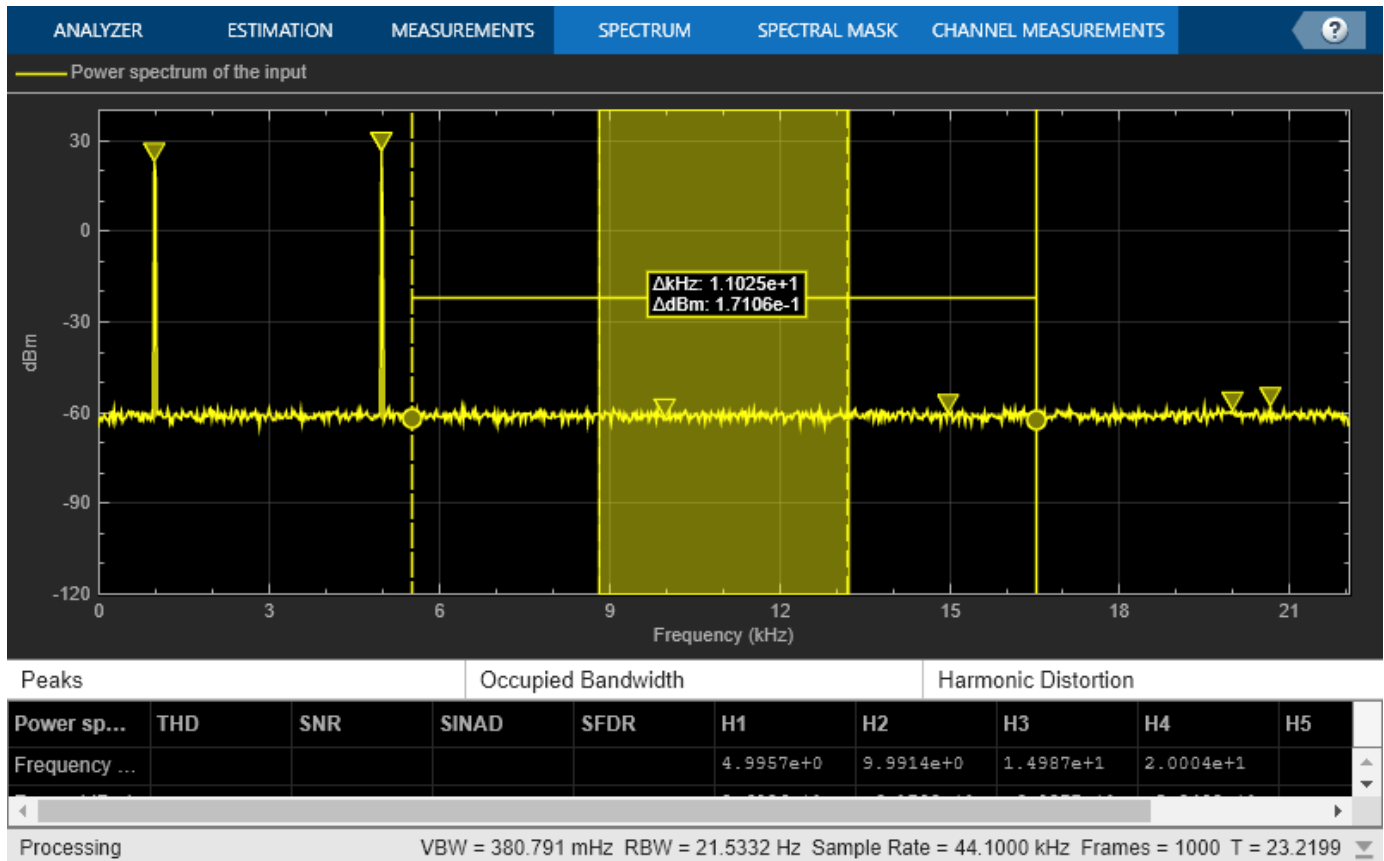
```
SA.CursorMeasurements.Enabled = true;
SA.ChannelMeasurements.Enabled = true;
SA.PeakFinder.Enabled = true;
SA.DistortionMeasurements.Enabled = true;
```

#### Use getMeasurementsData

Stream in the noisy sine wave input signal and estimate the power spectrum of the signal using the spectrumAnalyzer object. Measure the characteristics of the spectrum. Use the getMeasurementsData function to obtain these measurements programmatically. The isNewDataReady function returns true when there is new spectrum data. Store the measured data in the variable data.

```
data = [];
for Iter = 1:1000
    Sinewave1 = Sineobject1();
    Sinewave2 = Sineobject2();
    Input = Sinewave1 + Sinewave2;
    NoisyInput = Input + 0.001*randn(1024,1);
    SA(NoisyInput);
    if SA.isNewDataReady
        data = [data;getMeasurementsData(SA)];
    end
end
```





The panes at the bottom of the scope window display the measurements that you have enabled. The values in these panes match the values in the last time step of the data variable. You can access the individual fields of `data` to obtain the various measurements programmatically.

### Compare Peak Values

Use the `PeakFinder` property to obtain peak values. Verify that the peak values in the last time step of `data` match the values in the spectrum analyzer plot.

```
peakvalues = data.PeakFinder(end).Value
```

```
peakvalues = 3x1
```

```
26.3957
22.7830
-57.9977
```

```
frequencieskHz = data.PeakFinder(end).Frequency/1000
```

```
frequencieskHz = 3x1
```

```
4.9957
0.9905
```

20.6719

## **Version History**

**Introduced in R2022a**

### **See Also**

spectrumAnalyzer | SpectrumAnalyzerConfiguration

# CursorMeasurementsConfiguration

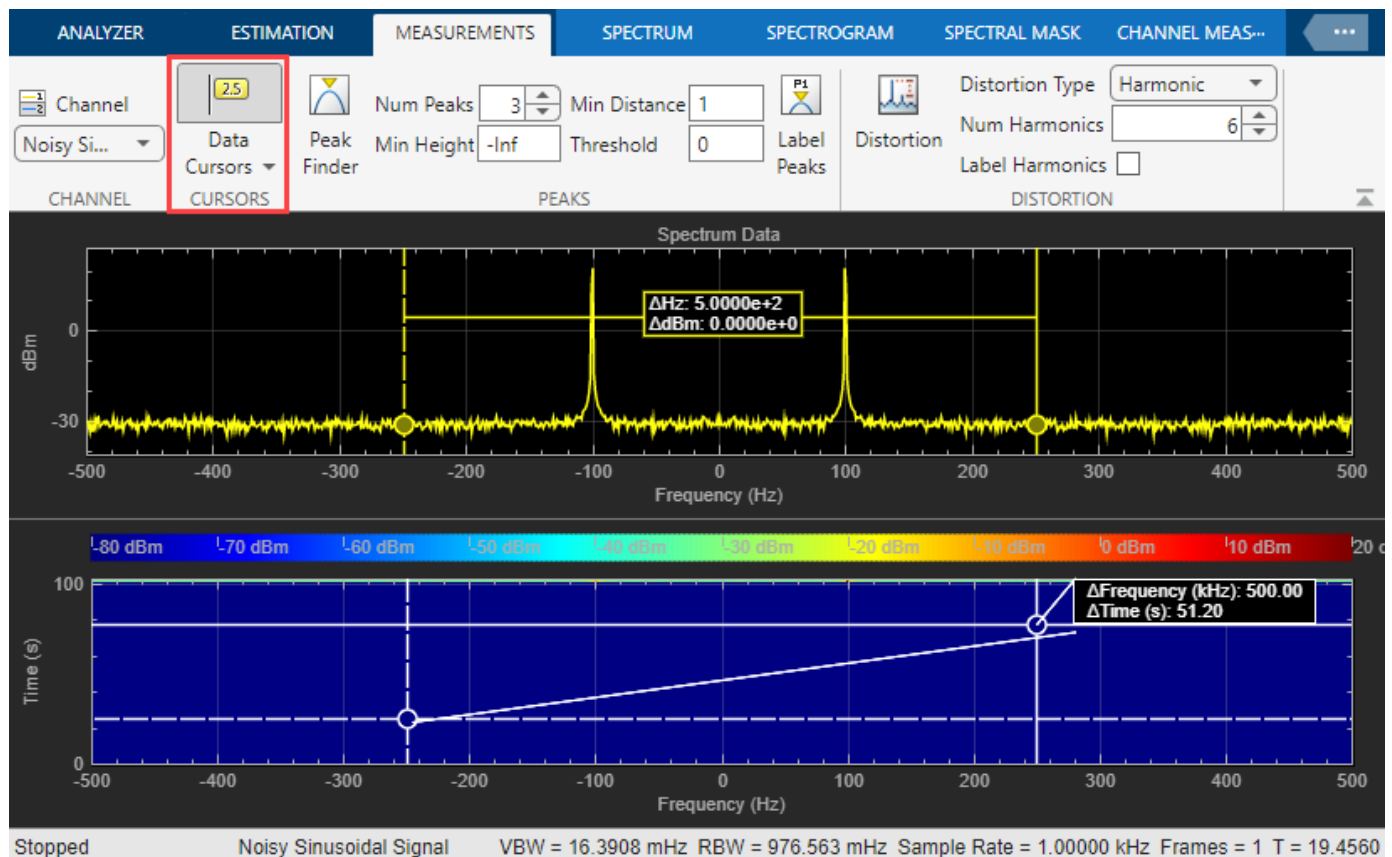
Measure signal values using vertical waveform cursors that track along the signal

## Description

Use the CursorMeasurementsConfiguration object to enable waveform cursors. You can control the cursor settings from the toolbar of the scope or from the command line.

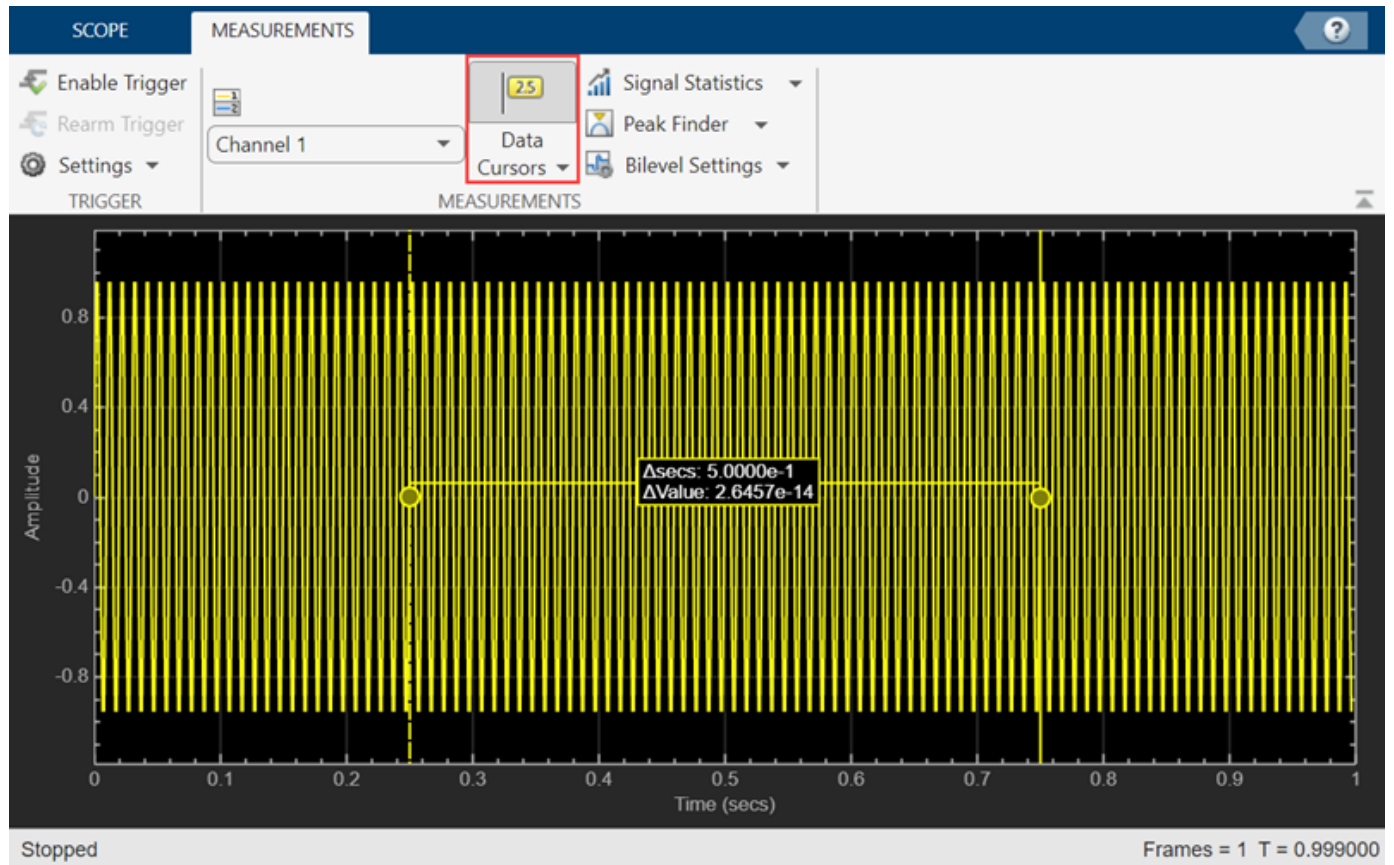
To display vertical cursors on each signal and to modify the cursor settings in the scope UI, click the **Measurements** tab and enable **Data Cursors**. Each cursor tracks a vertical line along the signal. The scope displays the difference between x- and y-values of the signal at the two cursors in the box between the cursors. The cursors appear only when the scope has at least one signal in its display. You can use the mouse to move the vertical cursors left and right.

### Spectrum Analyzer Toolstrip

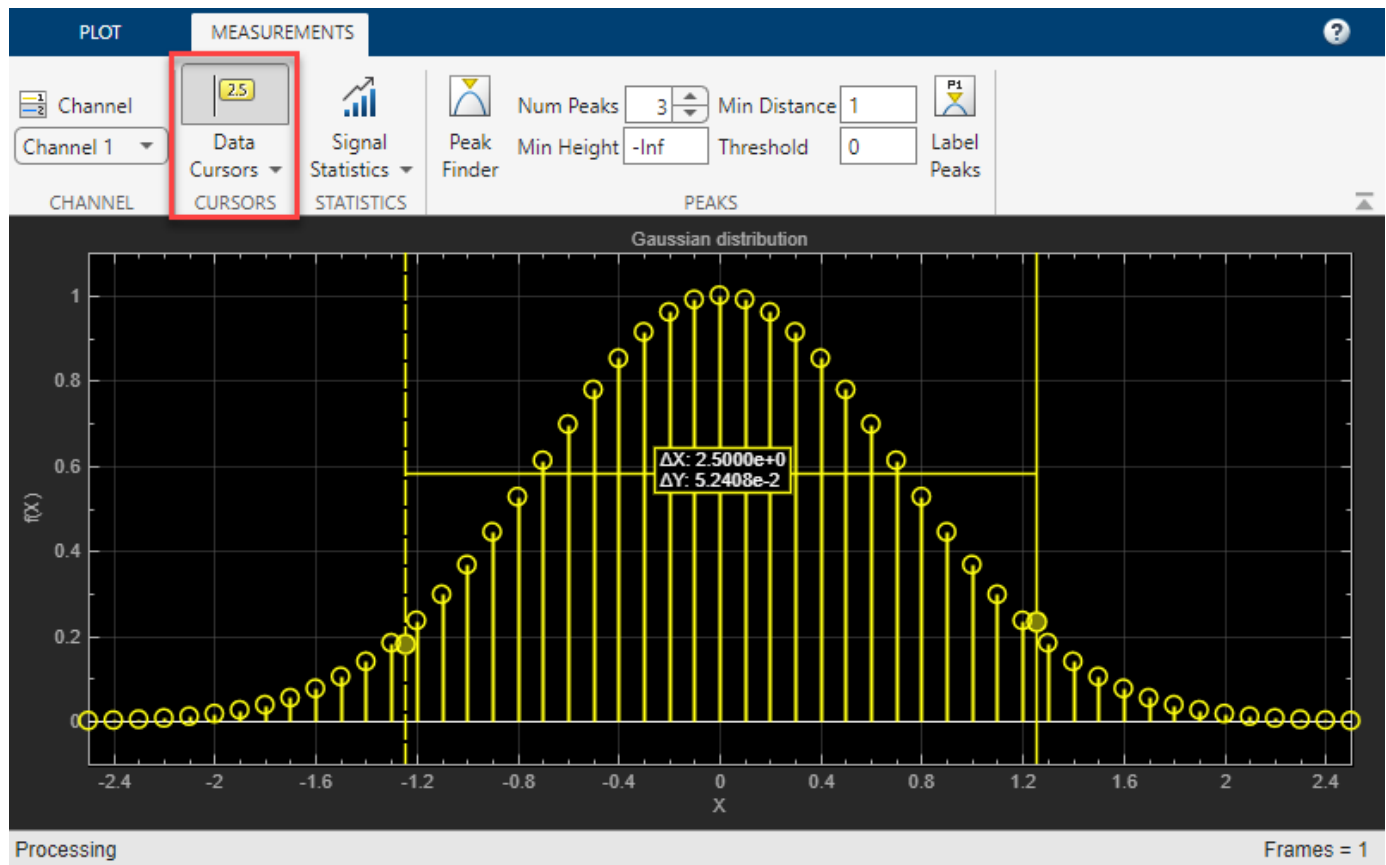


### Time Scope Toolstrip

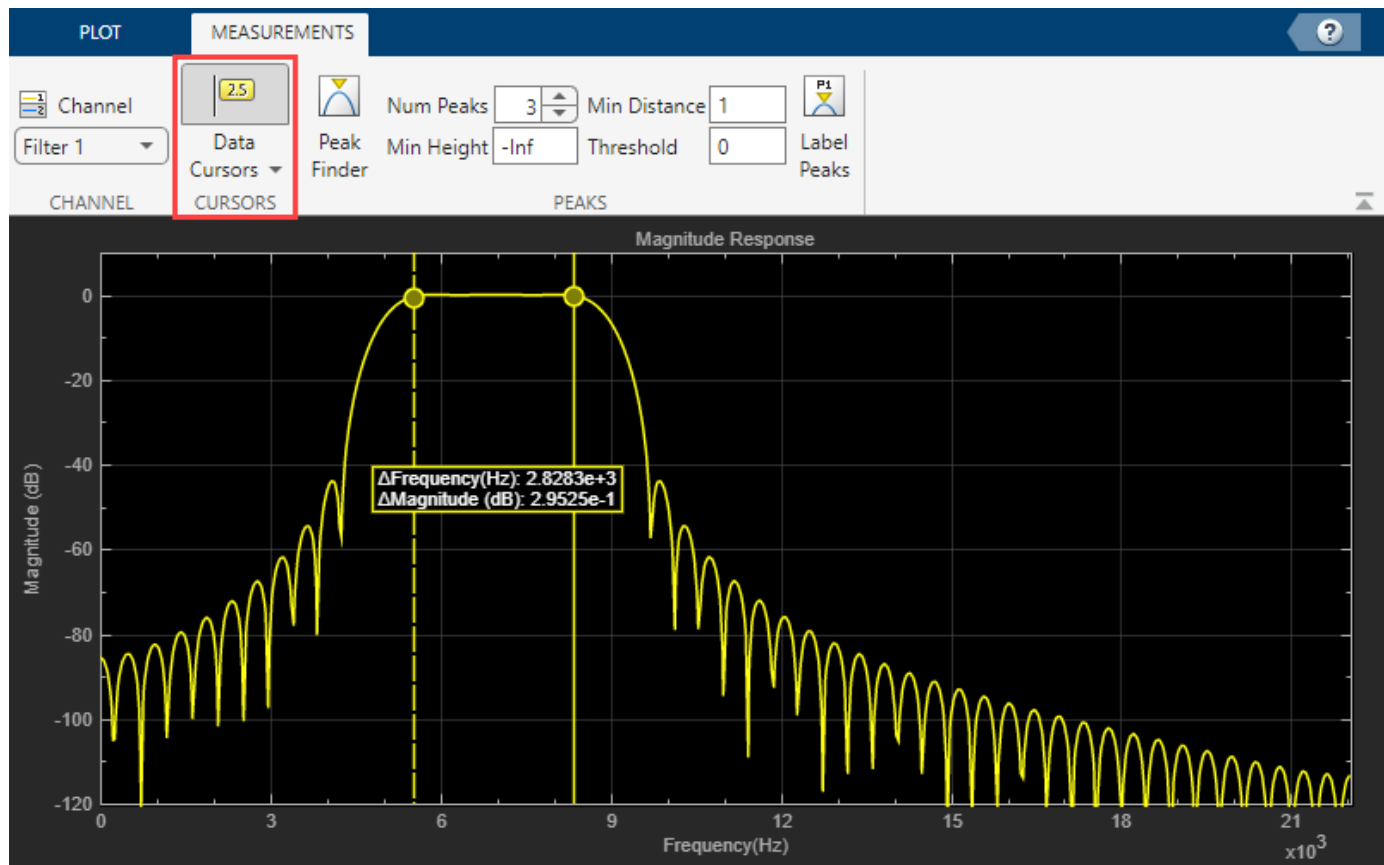
## 4 System Objects



Array Plot Toolstrip



Dynamic Filter Visualizer Toolstrip



## Creation

### Syntax

```
cursormeas = CursorMeasurementsConfiguration()
```

### Description

`cursormeas = CursorMeasurementsConfiguration()` creates a cursor measurements configuration object.

### Properties

All properties are tunable.

#### XLocation — x-coordinates of the cursors

[2 8] (default) | two-element vector

x-coordinates of the cursors, specified as a two-element vector of real numbers.

**Scope Window Use**

On the **Measurements** tab, select **Data Cursors** and then click **Data Cursors** to specify the two elements in X location properties.

Data Types: `double`

**SnapToData — Position cursors on signal data points**

`false` (default) | `true`

Position cursors on the signal data points, specified as `true` or `false`.

**Scope Window Use**

On the **Measurements** tab, select **Data Cursors** and then click **Data Cursors** to select **Snap to data**.

Data Types: `logical`

**LockSpacing — Lock spacing between cursors**

`false` (default) | `true`

Lock spacing between cursors, specified as `true` or `false`. Set this property to `true` to lock the frequency difference between the cursors.

**Scope Window Use**

On the **Measurements** tab, select **Data Cursors** and then click **Data Cursors** to select **Lock cursor spacing**.

Data Types: `logical`

**Enabled — Enable cursor measurements**

`false` (default) | `true`

Enable cursor measurements, specified as `true` or `false`. Set this property to `true` to enable cursor measurements.

**Scope Window Use**

On the **Measurements** tab, select **Data Cursors**.

Data Types: `logical`

**Examples****Configure Cursor Measurements Programmatically in Time Scope MATLAB Object**

Create a sine wave and view it in the Time Scope. Enable data cursors programmatically.

**Initialization**

Create the input sine wave using the `sin` function. Create a `timescope` MATLAB® object to display the signal. Set the `TimeSpan` property to 1 second.

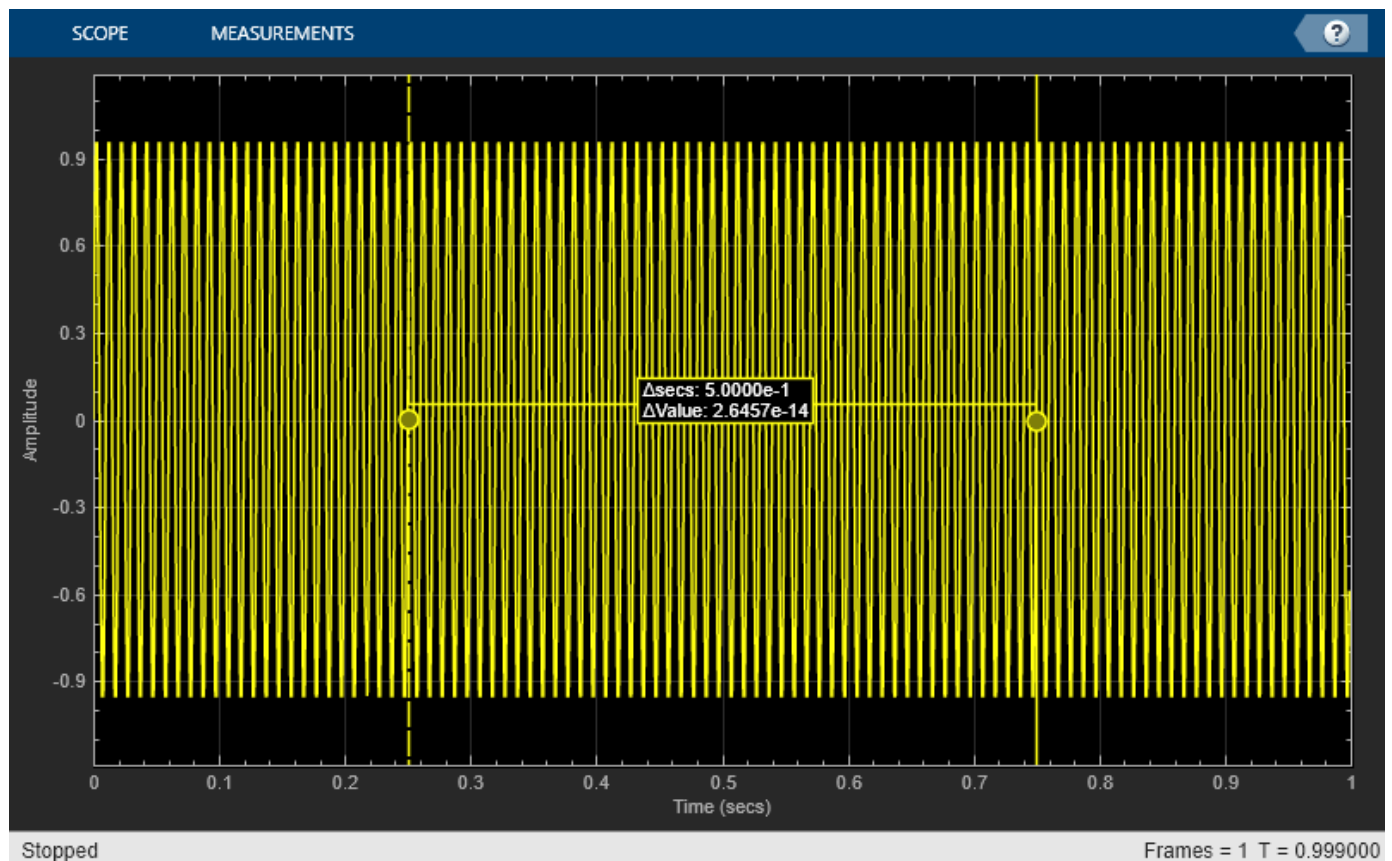
```
f = 100;
fs = 1000;
```

```
swv = sin(2.*pi.*f.*(0:1/fs:1-1/fs)).';
scope = timescope(SampleRate=fs,...
    TimeSpanSource="property",...
    TimeSpan=1);
```

### Data Cursors

Enable data cursors in the scope programmatically by setting the Enabled property of the CursorMeasurementsConfiguration object to true.

```
scope.CursorMeasurements.Enabled = true;
scope(swv);
release(scope)
```



### Obtain Measurements Data Programmatically for spectrumAnalyzer object

Compute and display the power spectrum of a noisy sinusoidal input signal using the spectrumAnalyzer MATLAB® object. Measure the peaks, cursor placements, adjacent channel power ratio, and distortion values in the spectrum by enabling these properties:

- PeakFinder
- CursorMeasurements
- ChannelMeasurements



- DistortionMeasurements

### Initialization

The input sine wave has two frequencies: 1000 Hz and 5000 Hz. Create two `dsp.SineWave` System objects to generate these two frequencies. Create a `spectrumAnalyzer` object to compute and display the power spectrum.

```
Fs = 44100;
Sineobject1 = dsp.SineWave(SamplesPerFrame=1024,PhaseOffset=10,...
    SampleRate=Fs,Frequency=1000);
Sineobject2 = dsp.SineWave(SamplesPerFrame=1024,...
    SampleRate=Fs,Frequency=5000);
SA = spectrumAnalyzer(SampleRate=Fs,SpectrumType="power",...
    PlotAsTwoSidedSpectrum=false,ChannelNames={'Power spectrum of the input'},...
    YLimits=[-120 40],ShowLegend=true);
```

### Enable Measurements Data

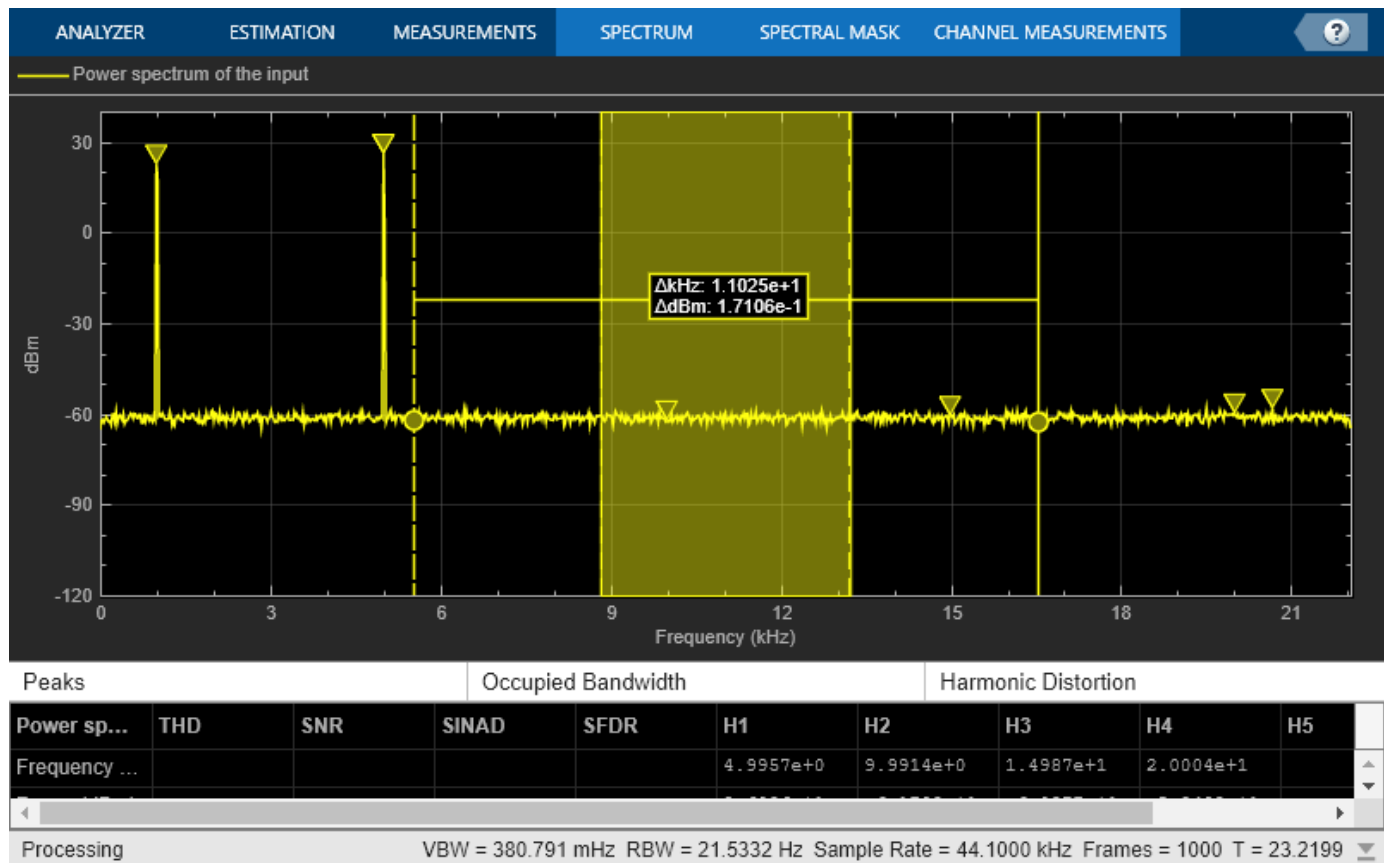
To obtain the measurements, set the `Enabled` property to `true`.

```
SA.CursorMeasurements.Enabled = true;
SA.ChannelMeasurements.Enabled = true;
SA.PeakFinder.Enabled = true;
SA.DistortionMeasurements.Enabled = true;
```

### Use getMeasurementsData

Stream in the noisy sine wave input signal and estimate the power spectrum of the signal using the `spectrumAnalyzer` object. Measure the characteristics of the spectrum. Use the `getMeasurementsData` function to obtain these measurements programmatically. The `isNewDataReady` function returns `true` when there is new spectrum data. Store the measured data in the variable `data`.

```
data = [];
for Iter = 1:1000
    Sinewave1 = Sineobject1();
    Sinewave2 = Sineobject2();
    Input = Sinewave1 + Sinewave2;
    NoisyInput = Input + 0.001*randn(1024,1);
    SA(NoisyInput);
    if SA.isNewDataReady
        data = [data;getMeasurementsData(SA)];
    end
end
```



The panes at the bottom of the scope window display the measurements that you have enabled. The values in these panes match the values in the last time step of the data variable. You can access the individual fields of data to obtain the various measurements programmatically.

### Compare Peak Values

Use the PeakFinder property to obtain peak values. Verify that the peak values in the last time step of data match the values in the spectrum analyzer plot.

```
peakvalues = data.PeakFinder(end).Value
```

```
peakvalues = 3×1
```

```
26.3957
22.7830
-57.9977
```

```
frequencieskHz = data.PeakFinder(end).Frequency/1000
```

```
frequencieskHz = 3×1
```

```
4.9957
0.9905
20.6719
```

## Obtain Measurements Data Programmatically for Spectrum Analyzer Block

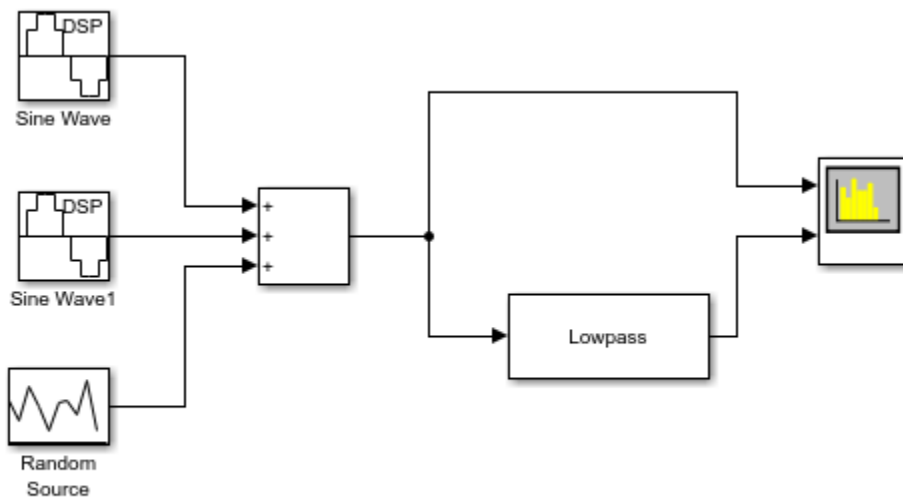
Compute and display the power spectrum of a noisy sinusoidal input signal using the Spectrum Analyzer block. Measure the cursor placements, adjacent channel power ratio, distortion, and peak values in the spectrum by enabling these block configuration properties:

- CursorMeasurements
- ChannelMeasurements
- DistortionMeasurements
- PeakFinder

## Open and Inspect the Model

Filter a streaming noisy sinusoidal input signal using a Lowpass Filter block. The input signal consists of two sinusoidal tones: 1 kHz and 15 kHz. The noise is white Gaussian noise with a mean of 0 and a variance of 0.05. The sampling frequency is 44.1 kHz. Open the model and inspect the parameter values in the blocks.

```
model = 'spectrumanalyzer_measurements.slx';
open_system(model)
```



Copyright 2018-2022 The MathWorks, Inc.

Access the configuration properties of the Spectrum Analyzer block using the `get_param` function.

```
sablock = 'spectrumanalyzer_measurements/Spectrum Analyzer';
cfg = get_param(sablock, 'ScopeConfiguration');
```

## Enable Measurements Data

To obtain the measurements, set the Enabled property to true.

```
cfg.CursorMeasurements.Enabled = true;
cfg.ChannelMeasurements.Enabled = true;
cfg.DistortionMeasurements.Enabled = true;
cfg.PeakFinder.Enabled = true;
```

## Simulate the Model

Run the model. The Spectrum Analyzer block compares the original spectrum with the filtered spectrum.

```
sim(model)
```



The panes at the bottom of the spectrum analyzer window display the measurements that you have enabled.

### Use `getMeasurementsData` function

Use the `getMeasurementsData` function to obtain the measurements programmatically.

```
data = getMeasurementsData(cfg)
```

```
data =
```

```
1x5 table
```

```
SimulationTime
```

```
9.9962
```

```
PeakFinder
```

```
1x1 struct
```

```
CursorMeasurements
```

```
1x1 struct
```

```
ChannelMeasurements
```

```
1x1 struct
```

```
DistortionMeasurements
```

```
1x1 struct
```

The values shown in the measurement panels match the values shown in `data`. You can access the individual fields of `data` to obtain the various measurements programmatically.

### Compare Peak Values

As an example, compare the peak values. Verify that the peak values obtained by `data.PeakFinder` match with the values in the spectrum analyzer window.

```
peakvalues = data.PeakFinder.Value  
frequencieskHz = data.PeakFinder.Frequency/1000
```

```
peakvalues =
```

```
26.8653  
26.2638  
-3.4329
```

```
frequencieskHz =
```

```
15.0015  
1.0049  
3.7755
```

## Version History

### Introduced in R2022a

#### R2022b: Enhancements to Data Cursor Measurements

Starting in R2022b, the `CursorMeasurementsConfiguration` object has a new `LockSpacing` property. Use this property to lock the spacing between waveform cursors in the scope window.

In the **Measurements** tab of the scope UI window, these data cursor settings are new:

- **Lock cursor spacing** -- This setting corresponds to the `LockSpacing` property in the `CursorMeasurementsConfiguration` object.
- **X location** -- These fields are enabled and correspond to the `XLocation` property in the `CursorMeasurementsConfiguration` object.

#### R2022b: Support for waveform cursors in spectrogram mode

Starting in R2022b, the Spectrum Analyzer supports data cursors even when the scope shows only the spectrogram display.

In the command line, you can edit the `CursorMeasurementsConfiguration` object when you set the `ViewType` property of the `spectrumAnalyzer` object to "spectrogram".

#### R2023a: Support for Spectrum Analyzer block

The `SpectrumAnalyzerConfiguration` object now supports the `CursorMeasurementsConfiguration` object. The `CursorMeasurementsConfiguration` object enables you to obtain and modify the cursor measurements of the Spectrum Analyzer block programmatically from the command line.

### **See Also**

`spectrumAnalyzer` | `timescope` | `dsp.ArrayPlot` | `SpectrumAnalyzerConfiguration`

### **Topics**

“Configure Time Scope MATLAB Object”

# DistortionMeasurementsConfiguration

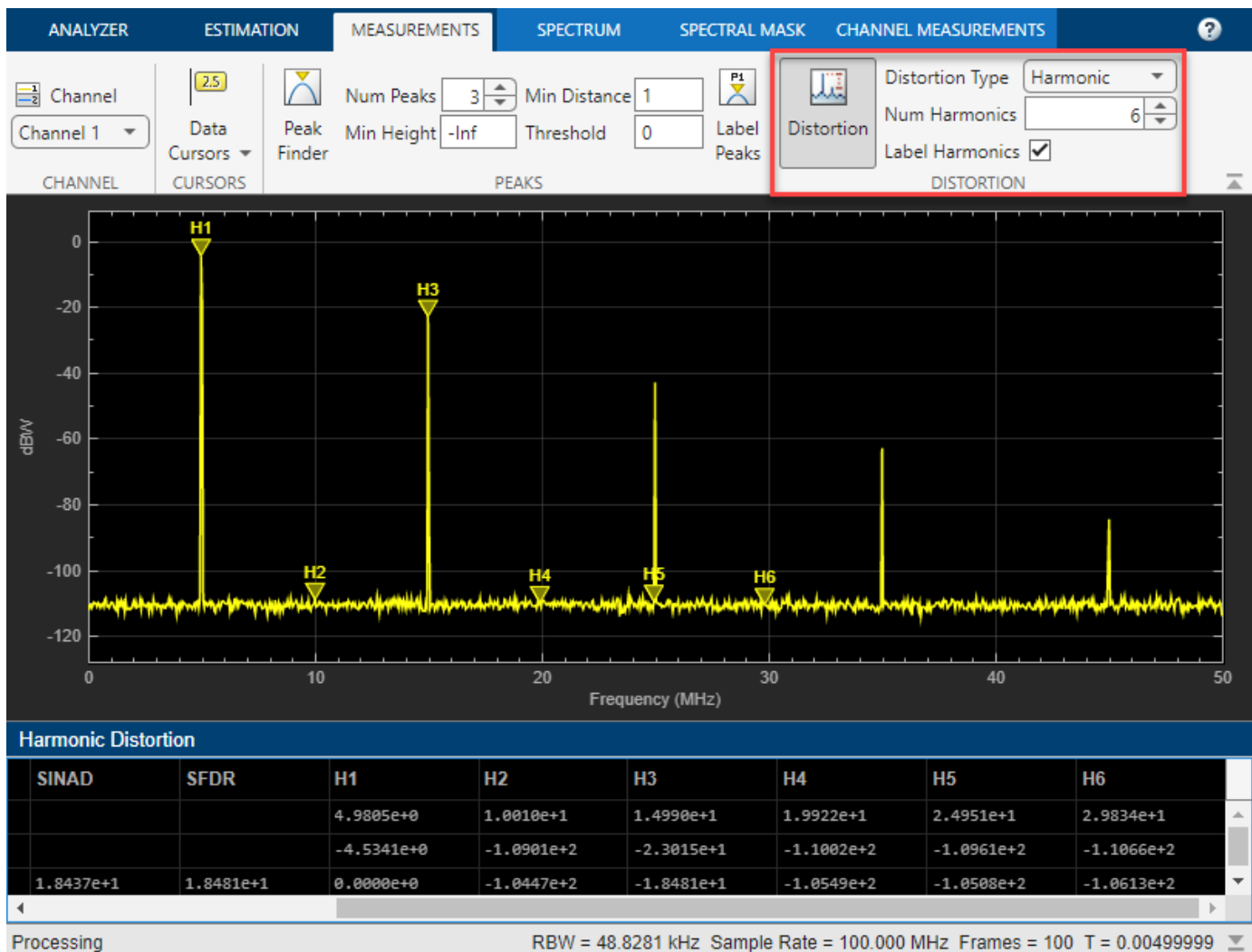
Compute and display harmonic and intermodulation distortion

## Description

Use the `DistortionMeasurementsConfiguration` object to compute and display harmonic and intermodulation distortion.

You can specify the distortion type, number of harmonics, and label the harmonics. You can control the distortion settings from the Spectrum Analyzer toolstrip or from the command line. Both the `spectrumAnalyzer` object and the `SpectrumAnalyzerConfiguration` object support the `DistortionMeasurementsConfiguration` object in the command line.

To modify the distortion settings in the Spectrum Analyzer toolstrip, click the **Measurements** tab and edit the settings in the **Distortion** section.



## Creation

### Syntax

```
distmeas = DistortionMeasurementsConfiguration()
```

### Description

`distmeas = DistortionMeasurementsConfiguration()` creates a distortion measurements configuration object `distmeas`.

### Properties

All properties are tunable.

#### Type — Type of measurement to display

"harmonic" (default) | "intermodulation"

Type of measurement data to display, specified as "harmonic" or "intermodulation".

#### Scope Window Use

Click the **Measurements** tab on the Spectrum Analyzer toolstrip. In the **Distortion** section, set **Distortion Type** to Harmonic or Intermodulation.

The **Measurements** tab appears when you select **Spectrum** in the **Analyzer** tab.

Data Types: char | string

#### NumHarmonics — Number of harmonics to measure

6 (default) | positive integer

Number of harmonics to measure, specified as a positive integer less than or equal to 99.

#### Dependency

To enable this property, set Type to "harmonic".

#### Scope Window Use

Click the **Measurements** tab on the Spectrum Analyzer toolstrip. In the **Distortion** section, set **Num Harmonics** to a positive integer less than or equal to 99.

The **Measurements** tab appears when you select **Spectrum** in the **Analyzer** tab.

Data Types: double

#### LabelValues — Label values

false (default) | true

Label harmonics (for harmonic distortion) and frequencies (for intermodulation distortion), specified as true or false.

#### Scope Window Use

Click the **Measurements** tab on the Spectrum Analyzer toolstrip. In the **Distortion** section, select **Label Harmonics**.



The **Measurements** tab appears when you select **Spectrum** in the **Analyzer** tab.

Data Types: `logical`

#### **Enabled — Enable distortion measurements**

`false` (default) | `true`

Enable distortion measurements, specified as `true` or `false`.

#### **Scope Window Use**

Click the **Measurements** tab on the Spectrum Analyzer toolstrip. In the **Distortion** section, select **Distortion**.

The **Measurements** tab appears when you select **Spectrum** in the **Analyzer** tab.

Data Types: `logical`

## **Examples**

### **Obtain Measurements Data Programmatically for spectrumAnalyzer object**

Compute and display the power spectrum of a noisy sinusoidal input signal using the `spectrumAnalyzer` MATLAB® object. Measure the peaks, cursor placements, adjacent channel power ratio, and distortion values in the spectrum by enabling these properties:

- `PeakFinder`
- `CursorMeasurements`
- `ChannelMeasurements`
- `DistortionMeasurements`

#### **Initialization**

The input sine wave has two frequencies: 1000 Hz and 5000 Hz. Create two `dsp.SineWave` System objects to generate these two frequencies. Create a `spectrumAnalyzer` object to compute and display the power spectrum.

```
Fs = 44100;
Sineobject1 = dsp.SineWave(SamplesPerFrame=1024,PhaseOffset=10,...
    SampleRate=Fs,Frequency=1000);
Sineobject2 = dsp.SineWave(SamplesPerFrame=1024,...
    SampleRate=Fs,Frequency=5000);
SA = spectrumAnalyzer(SampleRate=Fs,SpectrumType="power",...
    PlotAsTwoSidedSpectrum=false,ChannelNames={'Power spectrum of the input'},...
    YLimits=[-120 40],ShowLegend=true);
```

#### **Enable Measurements Data**

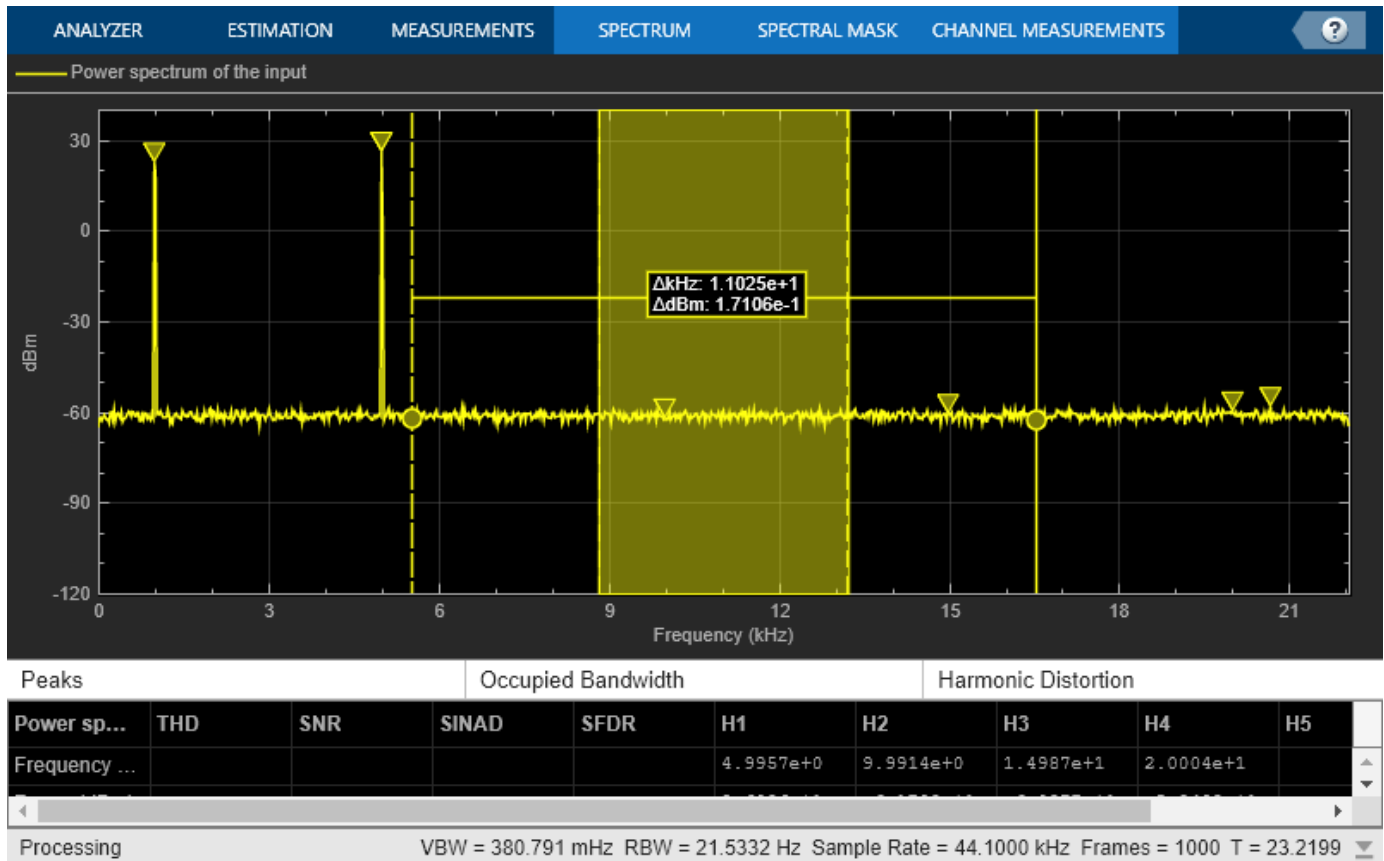
To obtain the measurements, set the `Enabled` property to `true`.

```
SA.CursorMeasurements.Enabled = true;
SA.ChannelMeasurements.Enabled = true;
SA.PeakFinder.Enabled = true;
SA.DistortionMeasurements.Enabled = true;
```

### Use getMeasurementsData

Stream in the noisy sine wave input signal and estimate the power spectrum of the signal using the spectrumAnalyzer object. Measure the characteristics of the spectrum. Use the getMeasurementsData function to obtain these measurements programmatically. The isNewDataReady function returns true when there is new spectrum data. Store the measured data in the variable data.

```
data = [];
for Iter = 1:1000
    Sinewave1 = Sineobject1();
    Sinewave2 = Sineobject2();
    Input = Sinewave1 + Sinewave2;
    NoisyInput = Input + 0.001*randn(1024,1);
    SA(NoisyInput);
    if SA.isNewDataReady
        data = [data;getMeasurementsData(SA)];
    end
end
```



The panes at the bottom of the scope window display the measurements that you have enabled. The values in these panes match the values in the last time step of the data variable. You can access the individual fields of data to obtain the various measurements programmatically.

## Compare Peak Values

Use the `PeakFinder` property to obtain peak values. Verify that the peak values in the last time step of data match the values in the spectrum analyzer plot.

```
peakvalues = data.PeakFinder(end).Value
```

```
peakvalues = 3×1
```

```
    26.3957
```

```
    22.7830
```

```
   -57.9977
```

```
frequencieskHz = data.PeakFinder(end).Frequency/1000
```

```
frequencieskHz = 3×1
```

```
    4.9957
```

```
    0.9905
```

```
   20.6719
```

## Version History

**Introduced in R2022a**

### **R2023a: Support for Spectrum Analyzer block**

The `SpectrumAnalyzerConfiguration` object now supports the `DistortionMeasurementsConfiguration` object. The `DistortionMeasurementsConfiguration` object enables you to obtain and modify the distortion measurements of the Spectrum Analyzer block programmatically from the command line. For more details, see “DistortionMeasurements” on page 4-0 .

## See Also

`spectrumAnalyzer` | `SpectrumAnalyzerConfiguration`

## PeakFinderConfiguration

Compute and display the largest calculated peak values on the scope display

### Description

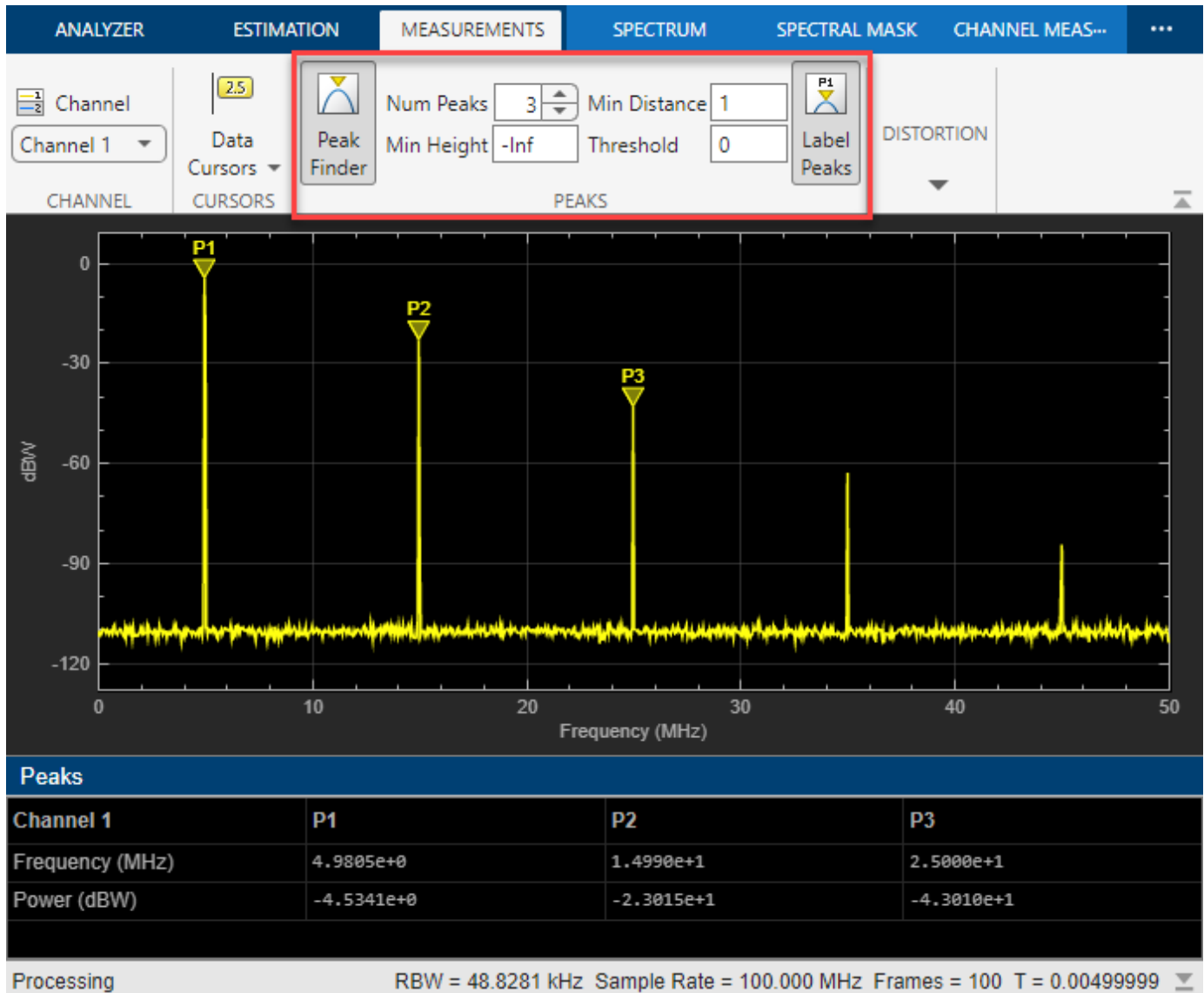
Use the `PeakFinderConfiguration` object to compute and display peaks in the scope. The scope computes and displays peaks for only the portion of the input signal that is currently on display in the scope.

You can specify the number of peaks you want the scope to display, the minimum height above which you want the scope to detect peaks, the minimum distance between peaks, and label the peaks. You can control the peak finder settings from the scope toolbar or from the command line. The algorithm defines a peak as a local maximum with lower values present on either side of the peak. It does not consider end points as peaks. For more information on the algorithm, see the `findpeaks` function.

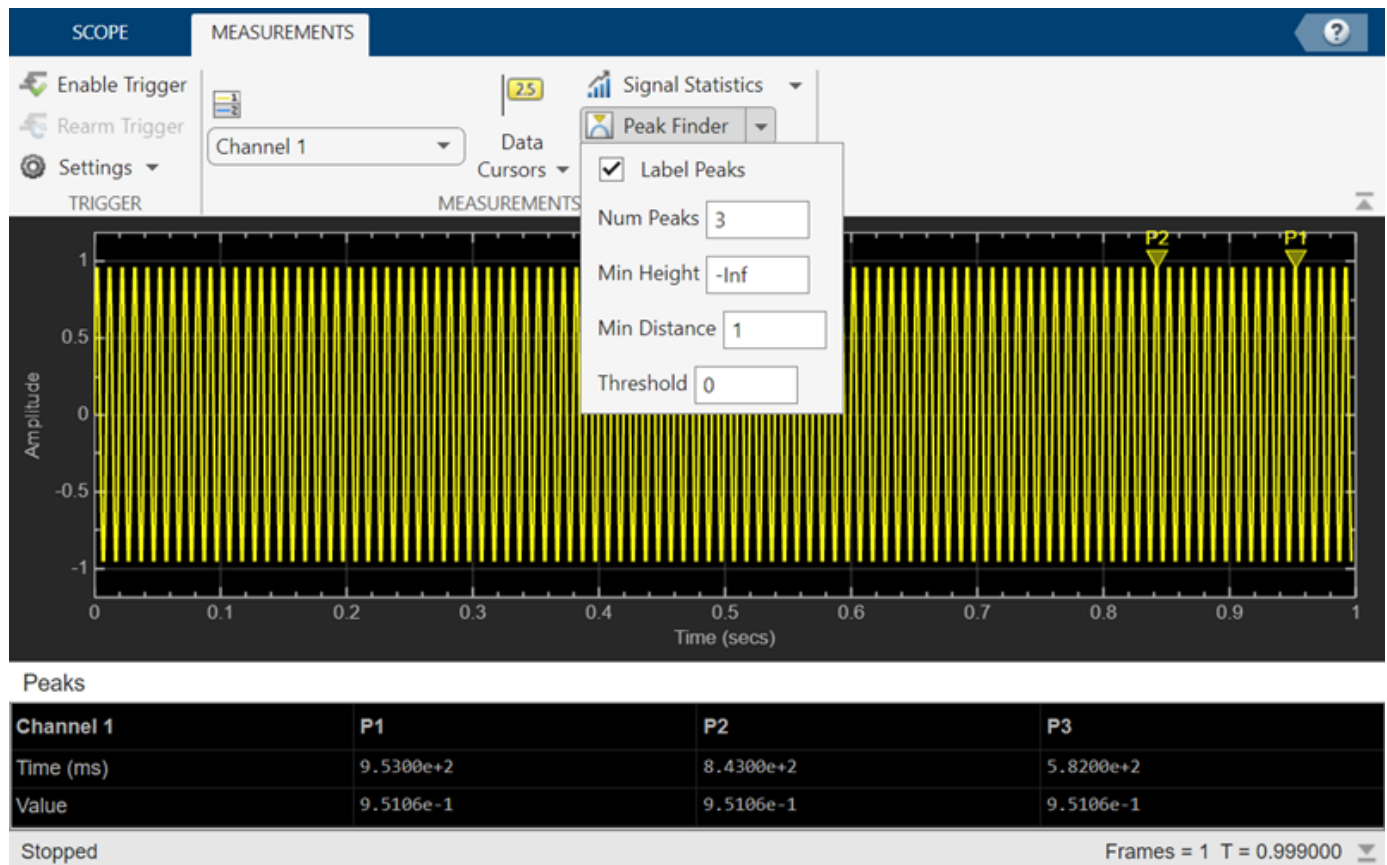
To modify the peak finder settings in the scope interface, click the **Measurements** tab and enable **Peak Finder**. Once you enable the **Peak Finder**, an arrow appears on the plot at each maxima and a Peaks panel appears at the bottom of the scope window.

The `SpectrumAnalyzerConfiguration` object supports the `PeakFinderConfiguration` object in the command line.

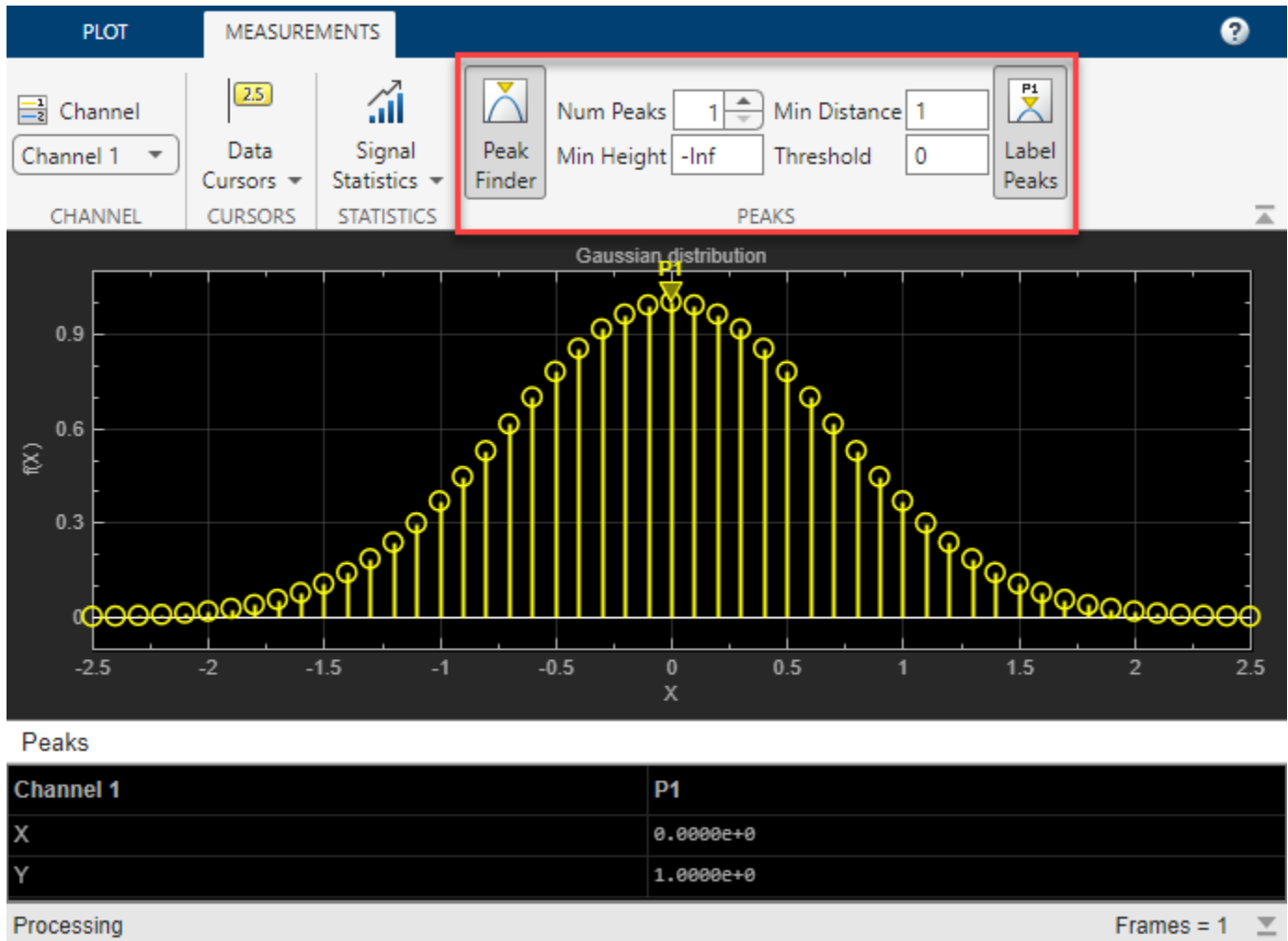
### Spectrum Analyzer Toolbar



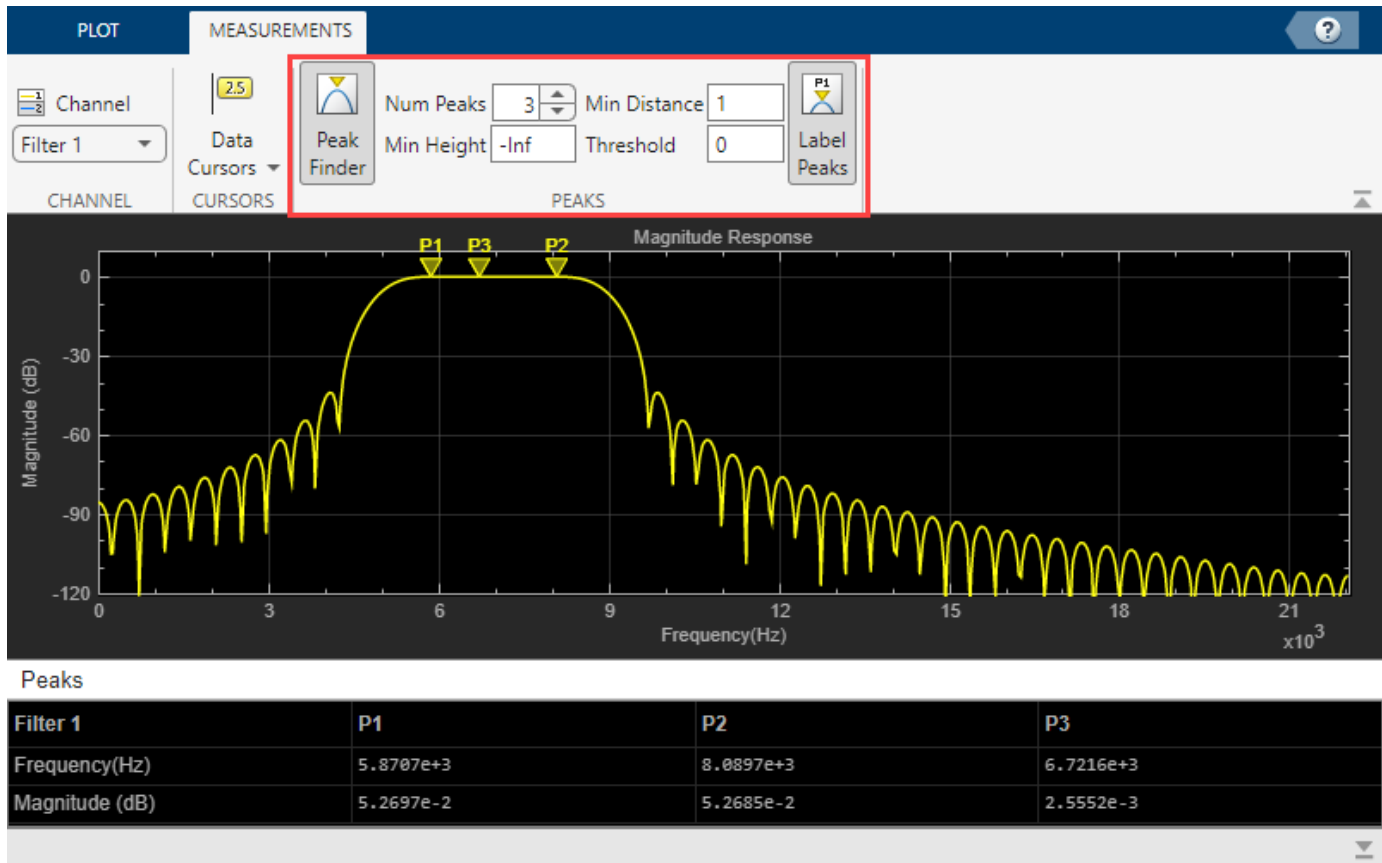
### Time Scope Toolstrip



**Array Plot Toolstrip**



### Dynamic Filter Visualizer Toolstrip



## Creation

### Syntax

```
pkfinder = PeakFinderConfiguration()
```

### Description

`pkfinder = PeakFinderConfiguration()` creates a peak finder configuration object.

### Properties

All properties are tunable.

#### MinHeight — Level above which scope detects peaks

-Inf (default) | real scalar value

Level above which the scope detects peaks, specified as a real scalar.

#### Scope Window Use

On the **Measurements** tab, select **Peak Finder**. In the peak finder settings, specify a real scalar in the **Min Height** box.



Data Types: double

**NumPeaks — Maximum number of peaks to show**

3 (default) | positive integer less than 100

Maximum number of peaks to show, specified as a positive integer less than 100.

**Scope Window Use**

On the **Measurements** tab, select **Peak Finder**. In the peak finder settings, specify a positive integer less than 100 in the **Num Peaks** box.

Data Types: double

**MinDistance — Minimum number of samples between adjacent peaks**

1 (default) | positive integer

Minimum number of samples between adjacent peaks, specified as a positive integer.

**Scope Window Use**

On the **Measurements** tab, select **Peak Finder**. In the peak finder settings, specify a positive integer in the **Min Distance** box.

Data Types: double

**Threshold — Minimum difference in height of peak and its neighboring samples**

0 (default) | nonnegative scalar

Minimum difference in the height of the peak and its neighboring samples, specified as a nonnegative scalar.

**Scope Window Use**

On the **Measurements** tab, select **Peak Finder**. In the peak finder settings, specify a nonnegative scalar in the **Threshold** box.

Data Types: double

**LabelPeaks — Label found peaks**

false (default) | true

Label found peaks, specified as true or false. The scope displays the labels (**P1**, **P2**, ...) above the arrows in the plot.

**Scope Window Use**

On the **Measurements** tab, select **Peak Finder**. In the peak finder settings, select **Label Peaks**.

Data Types: logical

**LabelFormat — Coordinates to display**

"x + y" (default) | "x" | "y"

Coordinates to display next to the calculated peak value, specified as "x", "y", or "x + y".

Data Types: char | string

**Enabled — Enable peak finder measurements**

false (default) | true

Enable peak finder measurements, specified as `true` or `false`. Set this property to `true` to enable the peak finder measurements.

### Scope Window Use

On the **Measurements** tab, select **Peak Finder**.

Data Types: `logical`

## Examples

### Enable Peak Finder Programmatically in a Time Scope Object

Create a sine wave and view it in the Time Scope. Enable the peak finder programmatically.

#### Initialization

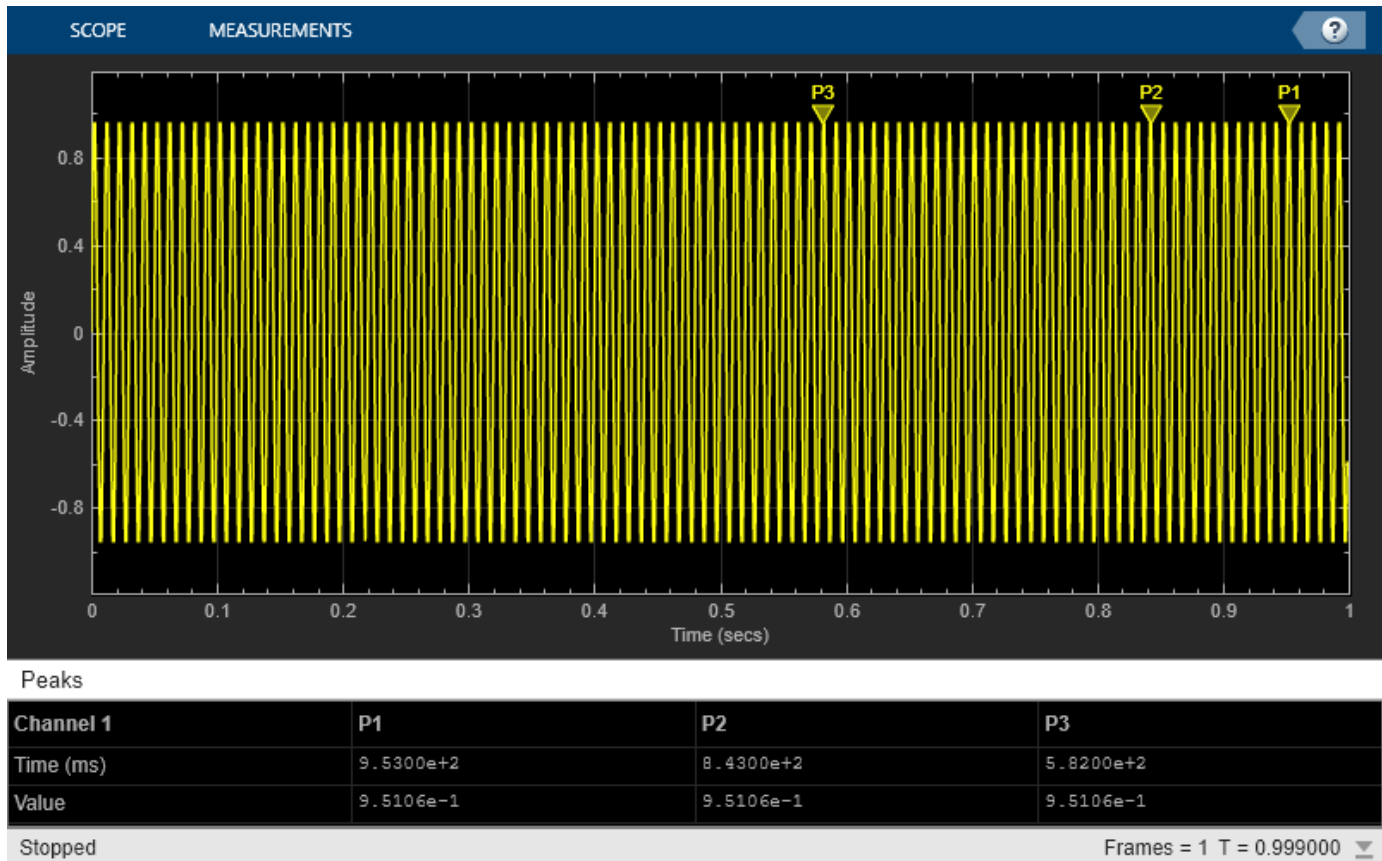
Create the input sine wave using the `sin` function. Create a `timescope` MATLAB® object to display the signal. Set the `TimeSpan` property to 1 second.

```
f = 100;  
fs = 1000;  
swv = sin(2.*pi.*f.*(0:1/fs:1-1/fs)).';  
scope = timescope(SampleRate=fs,...  
    TimeSpanSource="property", ...  
    TimeSpan=1);
```

#### Peaks

Enable the peak finder and label the peaks. Set the scope to show three peaks and label them.

```
scope.PeakFinder.Enabled = true;  
scope.PeakFinder.LabelPeaks = true;  
scope(swv)  
release(scope)
```



### Obtain Measurements Data Programmatically for spectrumAnalyzer object

Compute and display the power spectrum of a noisy sinusoidal input signal using the spectrumAnalyzer MATLAB® object. Measure the peaks, cursor placements, adjacent channel power ratio, and distortion values in the spectrum by enabling these properties:

- PeakFinder
- CursorMeasurements
- ChannelMeasurements
- DistortionMeasurements

### Initialization

The input sine wave has two frequencies: 1000 Hz and 5000 Hz. Create two dsp.SineWave System objects to generate these two frequencies. Create a spectrumAnalyzer object to compute and display the power spectrum.

```
Fs = 44100;
Sineobject1 = dsp.SineWave(SamplesPerFrame=1024,PhaseOffset=10,...
    SampleRate=Fs,Frequency=1000);
Sineobject2 = dsp.SineWave(SamplesPerFrame=1024,...
    SampleRate=Fs,Frequency=5000);
SA = spectrumAnalyzer(SampleRate=Fs,SpectrumType="power",...
```

```
PlotAsTwoSidedSpectrum=false,ChannelNames={'Power spectrum of the input'},...  
YLimits=[-120 40],ShowLegend=true);
```

### Enable Measurements Data

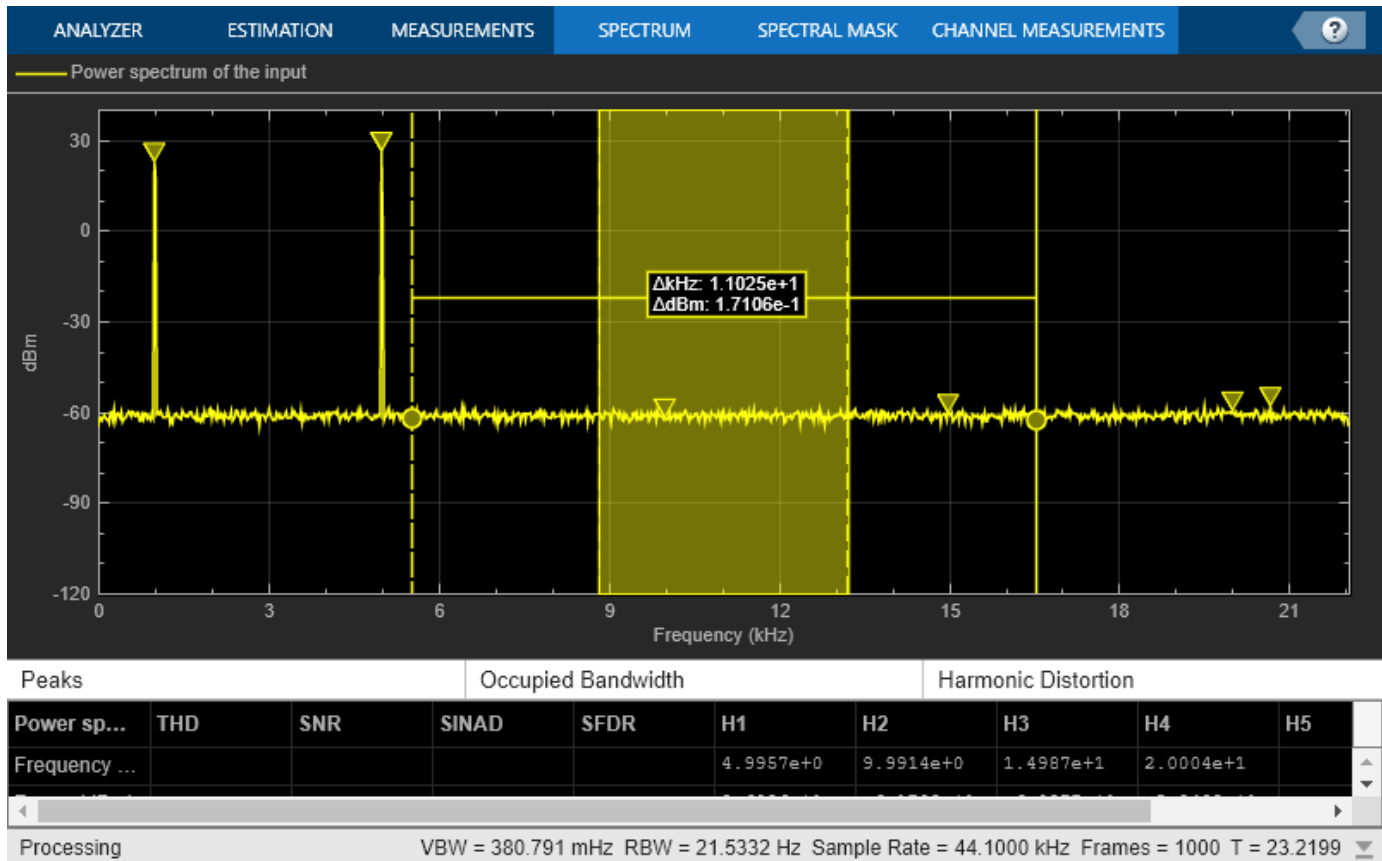
To obtain the measurements, set the Enabled property to true.

```
SA.CursorMeasurements.Enabled = true;  
SA.ChannelMeasurements.Enabled = true;  
SA.PeakFinder.Enabled = true;  
SA.DistortionMeasurements.Enabled = true;
```

### Use getMeasurementsData

Stream in the noisy sine wave input signal and estimate the power spectrum of the signal using the spectrumAnalyzer object. Measure the characteristics of the spectrum. Use the getMeasurementsData function to obtain these measurements programmatically. The isNewDataReady function returns true when there is new spectrum data. Store the measured data in the variable data.

```
data = [];  
for Iter = 1:1000  
    Sinewave1 = Sineobject1();  
    Sinewave2 = Sineobject2();  
    Input = Sinewave1 + Sinewave2;  
    NoisyInput = Input + 0.001*randn(1024,1);  
    SA(NoisyInput);  
    if SA.isNewDataReady  
        data = [data;getMeasurementsData(SA)];  
    end  
end
```



The panes at the bottom of the scope window display the measurements that you have enabled. The values in these panes match the values in the last time step of the data variable. You can access the individual fields of `data` to obtain the various measurements programmatically.

### Compare Peak Values

Use the `PeakFinder` property to obtain peak values. Verify that the peak values in the last time step of `data` match the values in the spectrum analyzer plot.

```
peakvalues = data.PeakFinder(end).Value
```

```
peakvalues = 3x1
```

```
26.3957
22.7830
-57.9977
```

```
frequencieskHz = data.PeakFinder(end).Frequency/1000
```

```
frequencieskHz = 3x1
```

```
4.9957
0.9905
```

20.6719

## Version History

Introduced in R2022a

### R2023a: Support for Spectrum Analyzer block

The `SpectrumAnalyzerConfiguration` object now supports the `PeakFinderConfiguration` object. The `PeakFinderConfiguration` object enables you to obtain and modify the peak finder measurements of the Spectrum Analyzer block programmatically from the command line.

### See Also

`spectrumAnalyzer` | `timescope` | `dsp.ArrayPlot` | `SpectrumAnalyzerConfiguration`

### Topics

“Configure Time Scope MATLAB Object”


# SpectralMaskSpecification

Display upper and lower spectral mask lines on the Spectrum Analyzer

## Description

Masks are overlaid on the spectrum. If the mask is green, the signal is passing the mask limits. If the mask is red, the signal is failing the mask limits.

You can check the status of the spectral mask from the scope toolbar, the command-line, or event listeners:

- To modify the spectral mask and see the spectral mask status, in the scope toolbar, select the spectral mask button, . In the **Spectral Mask** pane that opens, you can modify the masks and see details about what percentage of the time the mask is succeeding, which mask is failing, how many times the mask failed, and which channels are causing the failure.
- To get the current status of the spectral masks, call the function `getSpectralMaskStatus`.
- To perform an action every time the mask fails, use the `MaskTestFailed` event. To trigger a function when the mask fails, create a listener to the `MaskTestFailed` event and define a callback function to trigger. For more details about using events, see “Events”.

## Creation

### Syntax

```
mask = SpectralMaskSpecification()
mask = SpectralMaskSpecification(Name, Value)
```

### Description

`mask = SpectralMaskSpecification()` creates a spectral mask object `mask`.

`mask = SpectralMaskSpecification(Name, Value)` sets properties using one or more name-value pairs. Enclose each property name in quotes. For example, `mask = SpectralMaskSpecification("EnabledMasks", "Lower")`.

### Properties

All properties are tunable.

#### **EnabledMasks** — Spectral mask to enable

"None" (default) | "Lower" | "Upper" | "Upper and lower"

Specify which spectral masks to enable, specified as a character vector or string.

Data Types: char | string

**UpperMask — Limit for upper spectral mask**

Inf (default) | scalar | two-column matrix

Limit for upper spectral mask, specified as a scalar or two-column matrix.

If `UpperMask` is a scalar, the upper limit mask uses the power value of the scalar for all frequency values applicable to the Spectrum Analyzer.

If `UpperMask` is a matrix, the first column contains the frequency values (Hz), which correspond to the x-axis values. The second column contains the power values, which correspond to the associated y-axis values.

**Dependency**

To apply offsets to the power and frequency values, use the `ReferenceLevel` and `MaskFrequencyOffset` property values, respectively.

**LowerMask — Limit for lower spectral mask**

-Inf (default) | scalar | two-column matrix

Limit for the lower spectral mask, specified as a scalar or two-column matrix.

If `LowerMask` is a scalar, the lower limit mask uses the power value of the scalar for all frequency values applicable to the Spectrum Analyzer.

If `LowerMask` is a matrix, the first column contains the frequency values (Hz), which correspond to the x-axis values. The second column contains the power values, which correspond to the associated y-axis values.

**Dependency**

To apply offsets to the power and frequency values, use the `ReferenceLevel` and `MaskFrequencyOffset` property values, respectively.

**ReferenceLevel — Reference level for mask power values**

"Custom" (default) | "Spectrum peak"

Reference level for mask power values, specified as either "Custom" or "Spectrum peak".

When `ReferenceLevel` is "Custom", the `CustomReferenceLevel` property value is used as the reference to the power values, in dB<sub>r</sub>, in the `UpperMask` and `LowerMask` properties.

When `ReferenceLevel` is "Spectrum peak", the peak value of the current spectrum of the `SelectedChannel` is used.

**CustomReferenceLevel — Custom reference level**

0 (default) | real numeric scalar

Custom reference level, specified as a real numeric scalar, in the same units as the power units. The reference level is the value to which the power values in the `UpperMask` and `LowerMask` properties are referenced.

This property applies when `ReferenceLevel` is set to "Custom". This property uses the same units as the `PowerUnits` property of the Spectrum Analyzer.

**SelectedChannel — Input channel to use as mask level**

1 (default) | integer



Input channel with peak spectrum to use as the mask reference level, specified as an integer. This property applies when ReferenceLevel is set to "Spectrum peak".

### MaskFrequencyOffset — Frequency offset

0 (default) | finite numeric scalar

Frequency offset, specified as a finite, numeric scalar. Frequency offset is the amount of offset to apply to frequency values in the UpperMask and LowerMask properties.

## Examples

### Spectral Mask with dsp.SpectrumAnalyzer

This example shows how to create and use a spectral mask with the dsp.SpectrumAnalyzer object.

Create an upper spectral mask, set the reference level matrix, and add it to a Spectrum Analyzer object.

```
mask = SpectralMaskSpecification();
mask.EnabledMasks = 'Upper';
mask.UpperMask = [...
    0 -17; ...
    90 -17; ...
    90 30; ...
    110 30; ...
    110 -17; ...
    500 -17];

scope = dsp.SpectrumAnalyzer('PlotAsTwoSidedSpectrum',false,...
    'ShowLegend',true);
scope.SpectralMask = mask;
scope.SpectralMask

ans =
    SpectralMaskSpecification with properties:

        EnabledMasks: 'Upper'
        UpperMask: [6x2 double]
        LowerMask: -Inf
        ReferenceLevel: 'Custom'
        CustomReferenceLevel: 0
        MaskFrequencyOffset: 0

Events for class SpectralMaskSpecification: MaskTestFailed
```

Run the spectrum analyzer using a sine wave spectrum. In the Spectral Mask panel, you can see how the spectrum succeeded or failed to stay within your spectral mask.

```
sine = dsp.SineWave('Frequency',[98 100],'SampleRate',1000);
sine.SamplesPerFrame = 1024;
scope.SampleRate = sine.SampleRate;

for i=1:100
    scope(sine() + 0.05*randn(1024,2));
```

```
end
release(scope)
```

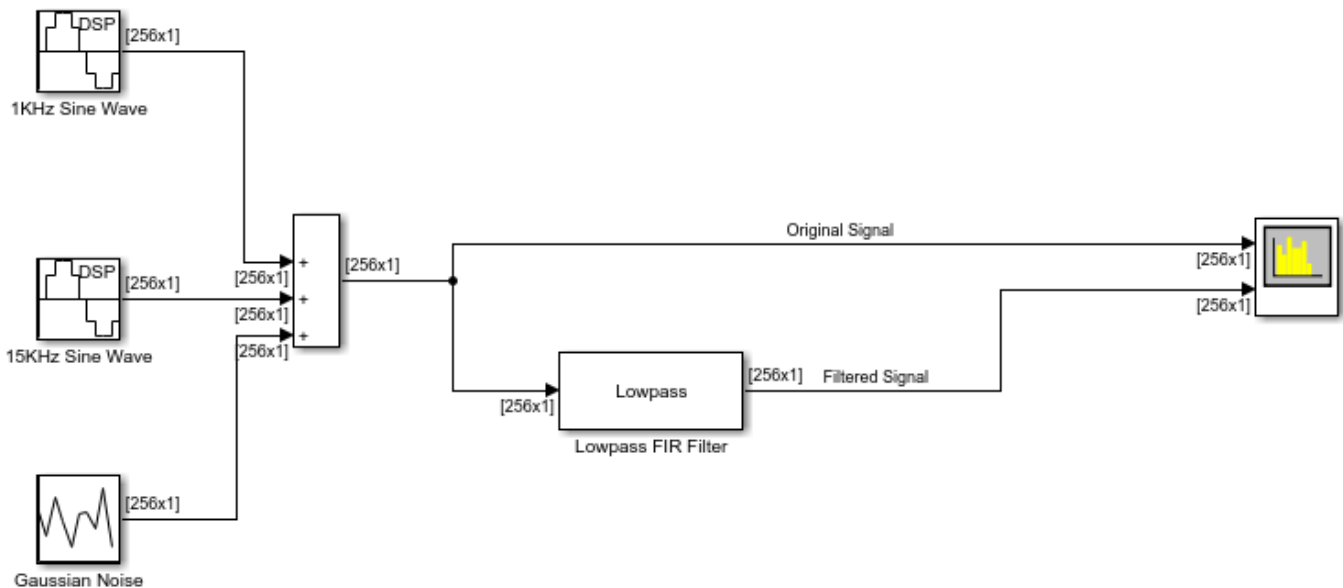
### Spectral Mask in Spectrum Analyzer Block

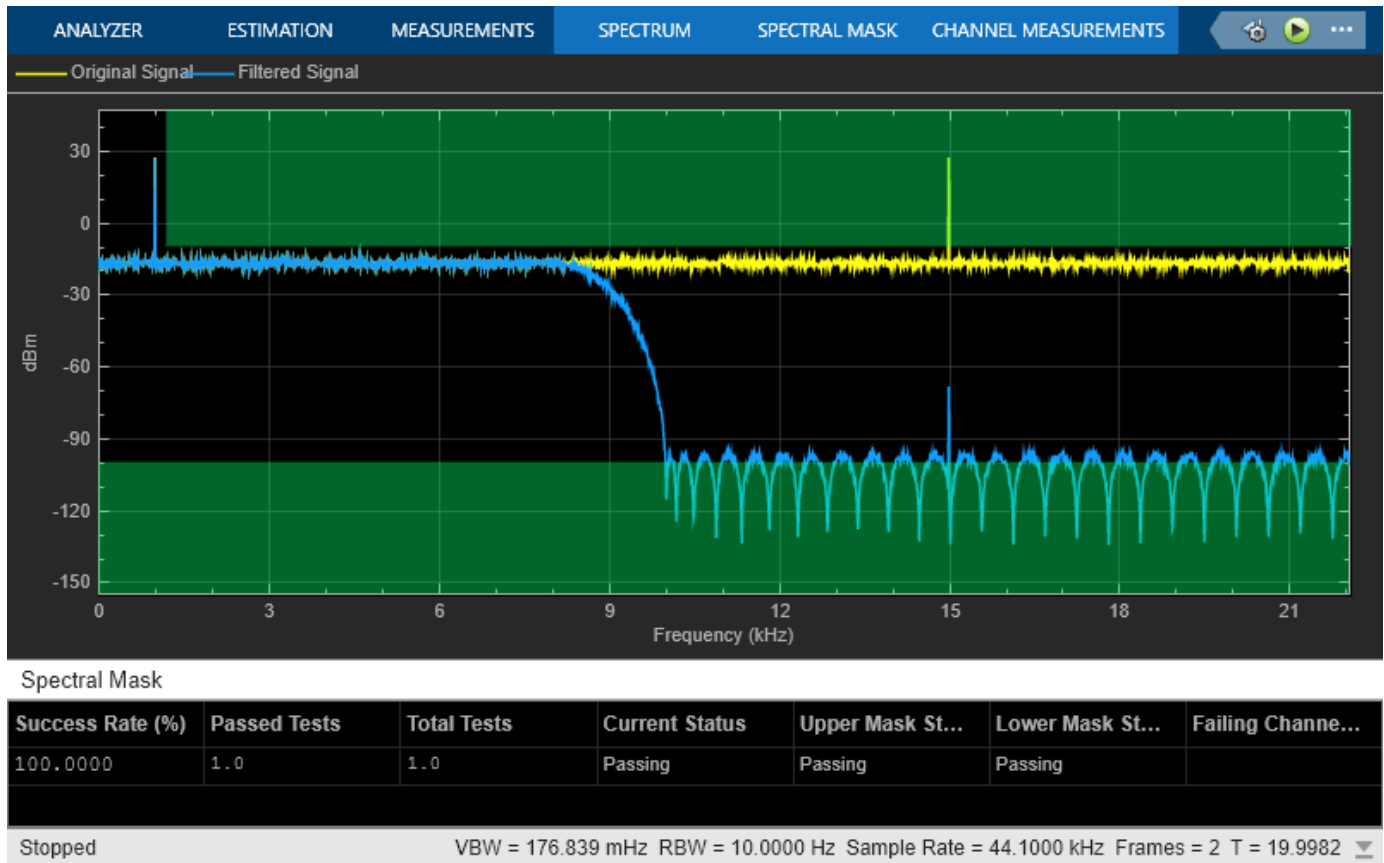
Create a new model based on the `dsp_basic_filter` template. Add a spectral mask to the Spectrum Analyzer block in the model. Run the model.

Masks are overlaid on the spectrum. If the mask is green, the signal is passing. If the mask is red, the signal is failing. The **Spectral Mask** panel shows what percentage of the time the mask is succeeding, which mask is failing, how many times the mask(s) failed, and which channels are causing the failure.

```
[~,mdl] = fileparts(tempname);
open_system(new_system(mdl,'FromTemplate','dsp_basic_filter'));
saBlock = find_system(mdl,'BlockType','SpectrumAnalyzer');

scopeConfig = get_param(saBlock{1},'ScopeConfiguration');
upperMask = [0 50; 1200 50; 1200 -10; 24000 -10];
scopeConfig.SpectralMask.UpperMask = upperMask;
scopeConfig.SpectralMask.LowerMask = -100;
scopeConfig.SpectralMask.EnabledMasks = 'upper-and-lower';
sim(mdl,StopTime='20');
```





## Version History

Introduced in R2016b


## SpectralMaskConfiguration

Display upper and lower spectral mask lines in the Spectrum Analyzer

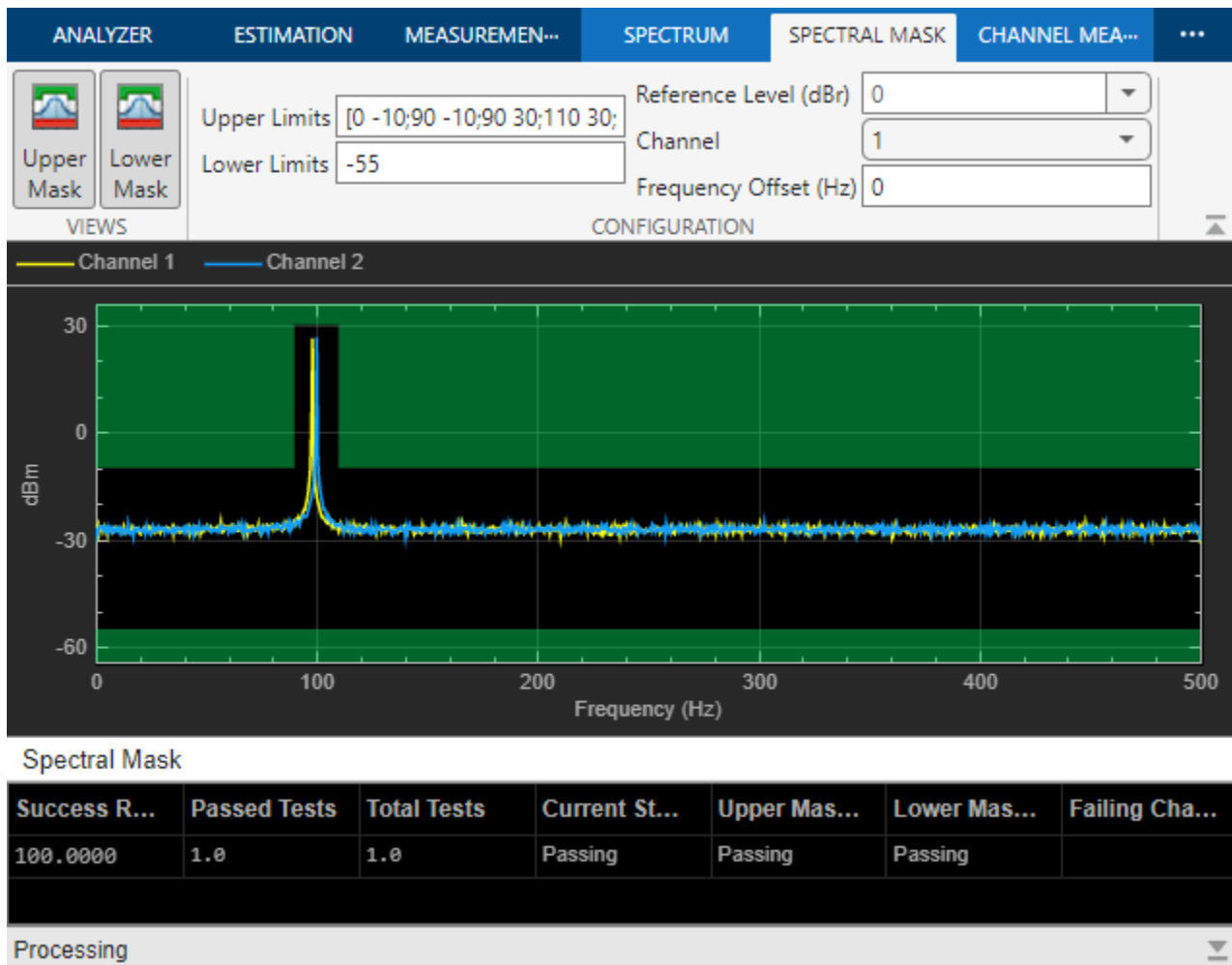
### Description

Use the `SpectralMaskConfiguration` object to overlay a spectral mask on a spectrum in the Spectrum Analyzer. If the mask is green, the signal is within the mask limits. If the mask is red, the signal fails the mask limits.

You can check the status of the spectral mask from the scope toolbar, the command line, or event listeners.

- To modify a spectral mask and see its status, in the scope toolbar, click the spectral mask button . You can modify the masks in the **Spectral Mask** tab on the Spectrum Analyzer toolstrip. Use the `getSpectralMaskStatus` function to obtain mask details, such as number of times a mask succeeded, number of times a mask failed, channels causing mask failure, and so on.
- Use the `MaskTestFailed` event to perform an action every time the mask fails. To trigger a function when the mask fails, create a listener to the `MaskTestFailed` event and define a callback function to trigger. For more details about using events, see “Events”.

Both the `spectrumAnalyzer` object and the `SpectrumAnalyzerConfiguration` object support the `SpectralMaskConfiguration` object in the command line.



## Creation

### Syntax

```
mask = SpectralMaskConfiguration()
```

### Description

mask = SpectralMaskConfiguration() creates a spectral mask object.

### Properties

All properties are tunable.

#### EnabledMasks — Spectral mask to enable

"none" (default) | "lower" | "upper" | "upper-and-lower"

Spectral mask to enable, specified as "none", "lower", "upper", or "upper-and-lower".

### Scope Window Use

Click the **Spectral Mask** tab on the Spectrum Analyzer toolstrip. In the **Views** section, select **Upper Mask**, **Lower Mask**, or both.

The **Spectral Mask** tab appears when you:

- Select **Spectrum** in the **Analyzer** tab.
- In the drop-down list under **Spectrum**, choose either Power or Power Density.

Data Types: char | string

### UpperMask — Limit for upper spectral mask

Inf (default) | scalar | two-column matrix

Limit for the upper spectral mask, specified as a scalar or two-column matrix.

If **UpperMask** is a scalar, the upper limit mask uses the same power value for all frequencies specified in the Spectrum Analyzer.

If **UpperMask** is a matrix, the first column contains the frequency values (Hz), which correspond to the x-axis values. The second column contains the power values, which correspond to the associated y-axis values.

To apply offsets to the power and frequency values, use the **ReferenceLevel** and **MaskFrequencyOffset** properties.

### Scope Window Use

Click the **Spectral Mask** tab on the Spectrum Analyzer toolstrip. In the **Configuration** section, specify **Upper Limits**.

The **Spectral Mask** tab appears when you:

- Select **Spectrum** in the **Analyzer** tab.
- In the drop-down list under **Spectrum**, choose Power or Power Density.

Data Types: double

### LowerMask — Limit for lower spectral mask

-Inf (default) | scalar | two-column matrix

Limit for the lower spectral mask, specified as a scalar or two-column matrix.

If **LowerMask** is a scalar, the lower limit mask uses the same power value for all frequencies specified in the Spectrum Analyzer.

If **LowerMask** is a matrix, the first column contains the frequency values (Hz), which correspond to the x-axis values. The second column contains the power values, which correspond to the associated y-axis values.

To apply offsets to the power and frequency values, use the **ReferenceLevel** and **MaskFrequencyOffset** property values.

**Scope Window Use**

Click the **Spectral Mask** tab on the Spectrum Analyzer toolstrip. In the **Configuration** section, specify **Lower Limits**.

The **Spectral Mask** tab appears when you:

- Select **Spectrum** in the **Analyzer** tab.
- In the drop-down list under **Spectrum**, choose Power or Power Density.

Data Types: double

**ReferenceLevel — Reference level for mask power values**

"custom" (default) | "spectrum-peak"

Reference level for mask power values, specified as either "custom" or "spectrum-peak".

When you specify ReferenceLevel as "custom", the object uses the value you specify in the CustomReferenceLevel property as the reference to the power values (in dBr) in the UpperMask and LowerMask properties.

When you specify ReferenceLevel as "spectrum-peak", the object uses the peak value of the current spectrum of the SelectedChannel as the reference power value.

**Scope Window Use**

Click the **Spectral Mask** tab on the Spectrum Analyzer toolstrip. In the **Configuration** section, set **Reference Level (dBr)** to either Spectrum peak or enter a real scalar.

The **Spectral Mask** tab appears when you:

- Select **Spectrum** in the **Analyzer** tab.
- In the drop-down list under **Spectrum**, choose Power or Power Density.

Data Types: char | string

**CustomReferenceLevel — Custom reference level**

0 (default) | real numeric scalar

Custom reference level to the power values, specified as a real numeric scalar. The reference level should have the same units as the SpectrumUnits property of the Spectrum Analyzer. The reference level is the value to which the object references the power values in the UpperMask and LowerMask properties.

**Dependency**

To enable this property, set ReferenceLevel to "custom". This property uses the same units as the SpectrumUnits property of the Spectrum Analyzer.

**Scope Window Use**

Click the **Spectral Mask** tab on the Spectrum Analyzer toolstrip. In the **Configuration** section, set **Reference Level (dBr)** to either Spectrum peak or enter a real scalar.

The **Spectral Mask** tab appears when you:

- Select **Spectrum** in the **Analyzer** tab.
- In the drop-down list under **Spectrum**, choose Power or Power Density.

Data Types: double

### **SelectedChannel** — Input channel with peak spectrum

1 (default) | integer

Input channel with peak spectrum to use as the mask reference level, specified as an integer.

#### **Dependency**

To enable this property, set `ReferenceLevel` to "spectrum-peak".

#### **Scope Window Use**

Click the **Spectral Mask** tab on the Spectrum Analyzer toolstrip. In the **Configuration** section, specify **Channel** as an integer.

To enable **Channel**, set **Reference Level (dBr)** in the **Configuration** section to Spectrum peak and display some data on the scope.

The **Spectral Mask** tab appears when you:

- Select **Spectrum** in the **Analyzer** tab.
- In the drop-down list under **Spectrum**, choose Power or Power Density.

### **MaskFrequencyOffset** — Frequency offset

0 (default) | finite numeric scalar

Frequency offset, specified as a finite numeric scalar. The object offsets the frequency values in the `UpperMask` and `LowerMask` properties by this value.

#### **Scope Window Use**

Click the **Spectral Mask** tab on the Spectrum Analyzer toolstrip. In the **Configuration** section, set **Frequency Offset (Hz)** to a real scalar.

The **Spectral Mask** tab appears when you:

- Select **Spectrum** in the **Analyzer** tab.
- In the drop-down list under **Spectrum**, choose Power or Power Density.

Data Types: double

## **Object Functions**

`getSpectralMaskStatus` Get test results of current spectral mask

## **Examples**

### **Spectral Mask with spectrumAnalyzer**

Create an upper spectral mask, set the reference level matrix, and add it to a Spectrum Analyzer object.



```

mask = SpectralMaskConfiguration();
mask.EnabledMasks = "upper";
mask.UpperMask = [...
    0 -17; ...
    90 -17; ...
    90 30; ...
    110 30; ...
    110 -17; ...
    500 -17];

scope = spectrumAnalyzer(PlotAsTwoSidedSpectrum=false,...
    ShowLegend=true);
scope.SpectralMask = mask;
scope.SpectralMask

ans =
    SpectralMaskConfiguration with properties:

        EnabledMasks: 'upper'
        UpperMask: [6x2 double]
        LowerMask: -Inf
        ReferenceLevel: 'custom'
    CustomReferenceLevel: 0
    MaskFrequencyOffset: 0

Events for class SpectralMaskConfiguration: MaskTestFailed

```

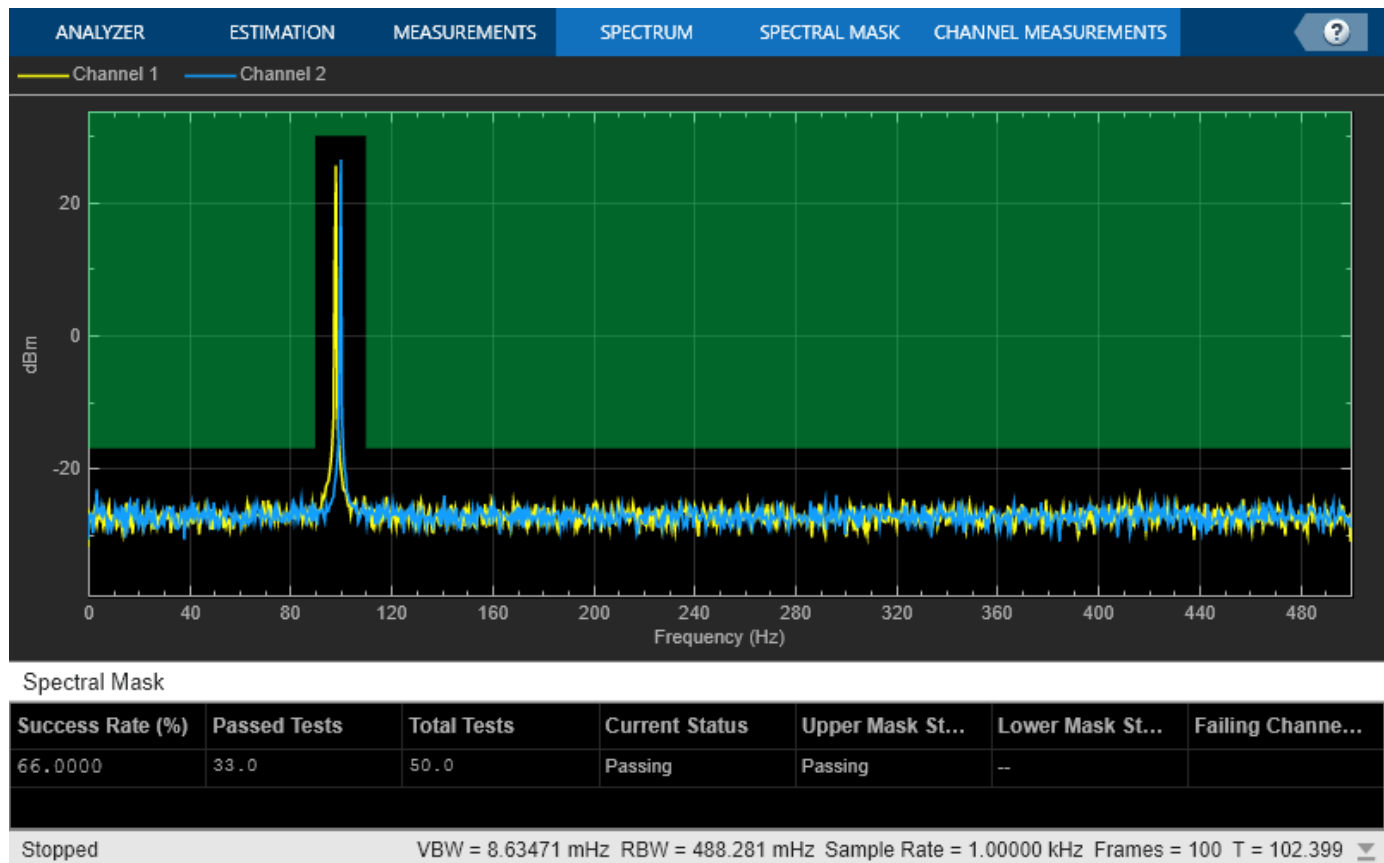
Run the spectrum analyzer using a sine wave spectrum. In the **Spectral Mask** panel at the bottom of the Spectrum Analyzer window, see how the spectrum succeeded or failed to stay within the spectral mask.

```

sine = dsp.SineWave(Frequency=[98 100],SampleRate=1000);
sine.SamplesPerFrame = 1024;
scope.SampleRate = sine.SampleRate;

for i=1:100
    scope(sine() + 0.05*randn(1024,2));
end
release(scope)

```



### Get Spectral Mask Status

Add a spectral mask to an existing spectrumAnalyzer object. Use the `getSpectralMaskStatus` function to get the spectral mask status.

```
sine = dsp.SineWave(Frequency=[98 100],SampleRate=1000);
sine.SamplesPerFrame = 1024;
scope = spectrumAnalyzer(SampleRate=sine.SampleRate,...
    PlotAsTwoSidedSpectrum=false,ShowLegend=true,YLimits=[-60 40]);
hide(scope);

scope.SpectralMask.EnabledMasks = "upper-and-lower";
upperMask = [0 -10; 90 -10; 90 30; 110 30; 110 -10; 500 -10];
set(scope.SpectralMask,UpperMask=upperMask,LowerMask=-55);

for i=1:100
    scope(sine() + 0.05*randn(1024,2));
end

res = getSpectralMaskStatus(scope)

res = struct with fields:
    IsCurrentlyPassing: 1
    NumPassedTests: 44
```

```

NumTotalTests: 50
SuccessRate: 88
FailingMasks: ''
FailingChannels: [1x0 double]
SimulationTime: 102.3990

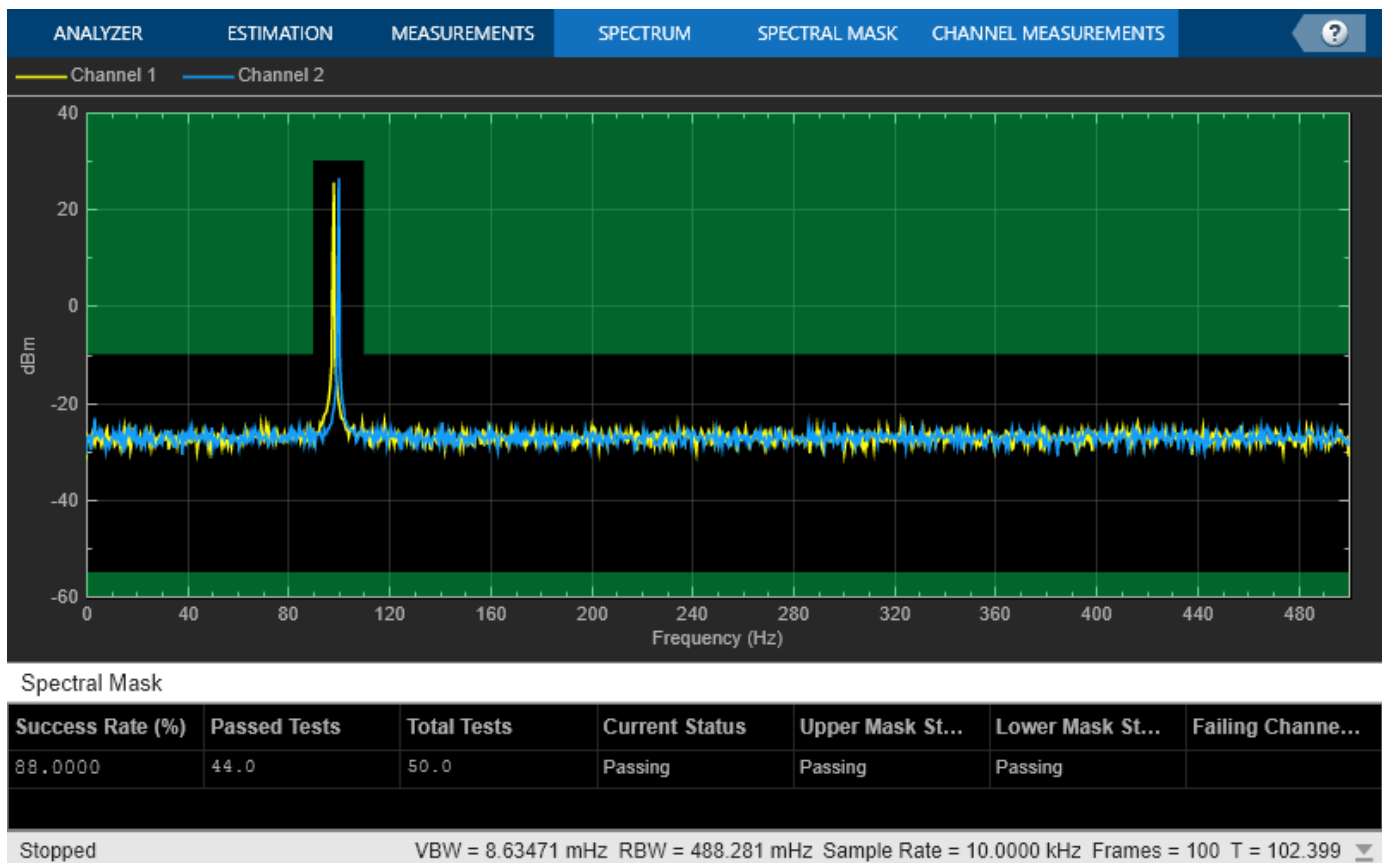
```

You can also view the status in the **Spectral Mask** tab on the Spectrum Analyzer toolstrip.

```

show(scope);
release(scope);

```



## Version History

Introduced in R2022a

## See Also

[spectrumAnalyzer](#) | [SpectrumAnalyzerConfiguration](#)

## ArrayPlotConfiguration

(Not recommended) Control Array Plot block appearance and behavior from MATLAB

---

**Note** `ArrayPlotConfiguration` is not recommended. Use `get_param` and `set_param` to modify the Array Plot block properties. For more information, see “Compatibility Considerations”.

---

### Description

Use the Array Plot Configuration object to control the appearance and behavior of an Array Plot block. Create a configuration object with `get_param`, and then change property values using the object with dot notation.

You can also modify the Array Plot block properties using `get_param` and `set_param`.

### Creation

`MyScopeConfiguration = get_param(gcbh, 'ScopeConfiguration')` creates a new Array Plot Configuration object. If you do not provide the full path to the block, you must first select the block in the model.

### Properties

#### Plot Configuration

##### **NumInputPorts** — Number of input ports

'1' (default) | character vector

Number of input ports on a scope block, specified as a character vector. The maximum number of input ports is 96.

##### **XDataMode** — Source of the x-data spacing

'Sample increment and X-offset' (default) | 'Custom'

Specify whether to use the `SampleIncrement` and `XOffset` property values to determine spacing, or specify your own custom spacing. If you specify 'Custom', you also must specify the `CustomXData` property values.

You can set this property only before simulation.

Data Types: `char` | `string`

##### **SampleIncrement** — x-axis spacing

'1' (default) | character vector

The spacing between samples along the x-axis, specified as a finite scalar in a character vector. The input signal is only y-axis data. x-axis data is set automatically based on the `XOffset` and `SampleIncrement` properties.

Example: When `XOffset` is 0 and `SampleIncrement` is 1, the x-axis values are set to 0, 1, 2, 3, 4, ... .

Example: When `XOffset` is -1 and `SampleIncrement` is 0.25, the x-axis values are set to -1, -0.75, -0.5, -0.25, 0, ... .

#### Dependency

To use this property, set `XDataMode` to `'Sample increment and X-offset'`.

#### XOffset — Display offset of x-axis

`'0'` (default) | character vector

Display offset of x-axis, specified as a numeric scalar in a character vector. x-axis data is set automatically based on both the `SampleIncrement` and `XOffset` values. The x-offset represents the first value on the x-axis.

Example: When `XOffset` is 0 and `SampleIncrement` is 1, the x-axis values are set to 0, 1, 2, 3, 4, ... .

Example: When `XOffset` is -1 and `SampleIncrement` is 0.25, the x-axis values are set to -1, -0.75, -0.5, -0.25, 0, ... .

#### Dependency

To use this property, set `XDataMode` to `'Sample increment and X-offset'`.

#### CustomXData — x-data values

`''` (default) | character vector

Specify the desired x-data values as a numeric vector in a character vector. The row or column vector must be equal to the frame length of the inputs. If you use the default (empty vector) value, the x-data is uniformly spaced and set to  $(0:L-1)$ , where  $L$  is the frame length.

Example: `scopeConfiguration.XDataMode = 'Custom';`  
`scopeConfiguration.CustomXData = 'logspace(0,log10(44100/2),1024)'`

#### Dependency

To use this property, set `XDataMode` to `'Custom'`.

#### XScale — Scale of x-axis

`"Linear"` (default) | `"Log"`

Specify whether the scale of the x-axis is `"Linear"` or `"Log"`. If `XOffset` is a negative value, you cannot set this property to `"Log"`.

#### Scope Window Use

Open the **Plot** tab, click **Settings**, and set **XScale**.

Data Types: `char` | `string`

#### YScale — Scale of y-axis

`"Linear"` (default) | `"Log"`

Specify whether the scale of the y-axis is `"Linear"` or `"Log"`.

**Scope Window Use**

Open the **Plot** tab, click **Settings**, and set **YScale**.

Data Types: char | string

**Visualization****PlotType — Control type of plot**

"Stem" (default) | "Line" | "Stairs"

Specify the type of plot to use for all the input signals displayed in the scope window:

- "Stem" - The scope displays the input signal as circles with vertical lines extending down to the x-axis at each of the sampled values.
- "Line" - The scope displays the input signal as lines connecting each of the sampled values.
- "Stairs" - The scope displays the input signal as a stair-step graph. A stair-step graph is made up of only horizontal lines and vertical lines. Each horizontal line represents the signal value for a discrete sample period and is connected to two vertical lines. Each vertical line represents a change in values occurring at a sample. Stair-step graphs are useful for drawing time history graphs of digitally sampled data.

**Scope Window Use**

Open the **Settings** and set **Plot Type**.

**AxesScaling — Axes scaling mode**

"OnceAtStop" (default) | "Auto" | "Manual" | "Updates"

Specify when the scope scales the axes. Valid values are:

- "Auto" — The scope scales the axes as needed to fit the data, both during and after simulation.
- "Manual" — The scope does not scale the axes automatically.
- "OnceAtStop" — The scope scales the axes when the simulation stops.
- "Updates" — The scope scales the axes once and only once after a specified number of updates. To specify the number of updates, use the `AxesScalingNumUpdates` property.

**AxesScalingNumUpdates — Number of updates before scaling**

"10" (default) | integer character vector | integer string scalar

Set this property to delay auto scaling the y-axis.

**Dependency**

To enable this property, set `AxesScaling` to "Updates".

**Name — Window name**

'Array Plot' (default) | character vector | string scalar

Specify the name of the scope. This name appears as the title of the scope's figure window. To specify a title of a scope plot, use the `Title` property.

Data Types: char | string

**Position — Scope window position and size in pixels**

screen center (default) | [left bottom width height]

Specify, in pixels, the size and location of the scope window as a four-element vector of the form `[left bottom width height]`. By default, the scope window appears in the center of your screen with a width of 800 pixels and height of 450 pixels. The default values for this property may change depending on your screen resolution.


### MaximizeAxes — Maximize axes control

"Auto" (default) | "On" | "Off"

Specify whether to display the scope in maximized-axes mode. In this mode, the axes are expanded to fit into the entire display. To conserve space, labels do not appear in each display. Instead, tick-mark values appear on top of the plotted data. You can select one of the following options:

- "Auto" — The axes appear maximized in all displays only if the `Title` and `YLabel` properties are empty for every display. If you enter any value in any display for either of these properties, the axes are not maximized.
- "On" — The axes appear maximized in all displays. Any values entered into the `Title` and `YLabel` properties are hidden.
- "Off" — None of the axes appear maximized.

### Scope Window Use

Hover over the array plot to see the maximize axes button .

Data Types: `char` | `string`

### Title — Display title

' ' (default) | character vector | string scalar

Specify the display title as a character vector or string.

### Scope Window Use

Open the **Plot** tab, click **Settings**, and set **Title**.

Data Types: `char` | `string`

### PlotAsMagnitudePhase — Plot signal as magnitude and phase

false (default) | true

- `true` - The scope plots the magnitude and phase of the input signal on two separate axes within the same active display.
- `false` - The scope plots the real and imaginary parts of the input signal on two separate axes within the same active display.

This property is useful for complex-valued input signals. Turning on this property affects the phase for real-valued input signals. When the amplitude of the input signal is nonnegative, the phase is 0 degrees. When the amplitude of the input signal is negative, the phase is 180 degrees.

### Scope Window Use

On the **Plot** tab, select the **Magnitude Phase** button.

### XLabel — x-axis label

"" (default) | character vector | string scalar

Specify the text for the scope to display below the x-axis.

**Scope Window Use**

Open the **Plot** tab, click **Settings**, and set **XLabel**.

Data Types: char | string

**YLabel — y-axis label**

"Amplitude" (default) | character vector | string scalar

Specify the text for the scope to display to the left of the y-axis.

**Dependencies**

This property applies only when `PlotAsMagnitudePhase` is `false`. When `PlotAsMagnitudePhase` is `true`, the two y-axis labels are read-only values "Magnitude" and "Phase", for the magnitude plot and the phase plot, respectively.

**Scope Window Use**

Open the **Plot** tab, click **Settings**, and set **YLabel**.

Data Types: char | string

**YLimits — y-axis limits**

[-10,10] (default) | [ymin, ymax]

Specify the y-axis limits as a two-element numeric vector, [ymin, ymax].

If `PlotAsMagnitudePhase` is `false`, the default is [-10,10]. If `PlotAsMagnitudePhase` is `true`, the default is [0,10].

**Dependencies**

When `PlotAsMagnitudePhase` is `true`, this property specifies the y-axis limits of only the magnitude plot. The y-axis limits of the phase plot are always [-180,180].

**Scope Window Use**

Open the **Plot** tab, click **Settings**, and set **Y-Axis Limits** as a two-element numeric vector.

**ShowGrid — Display grid**

true (default) | false

Set this property to `true` to show grid lines on the plot.

**Scope Window Use**

Open the **Plot** tab, click **Settings**, and select **Grid**.

**ShowLegend — Show legend**

false (default) | true

To show a legend with the input names, set this property to `true`.

From the legend, you can control which signals are visible. This control is equivalent to changing the visibility in the **Style** dialog box. In the scope legend, click a signal name to hide the signal in the



scope. To show the signal, click the signal name again. To show only one signal, right-click the signal name. To show all signals, press **Esc**.

---

**Note** The legend only shows the first 20 signals. Any additional signals cannot be viewed or controlled from the legend.

---

### Scope Window Use

On the **Plot** tab, click **Legend**.

Data Types: `logical`

### ChannelNames — Channel names

empty cell (default) | cell array of character vectors | array of strings

Specify the input channel names as a cell array of character vectors or an array of strings. The names appear in the legend, **Settings**, and **Measurements** panels. If you do not specify names, the channels are labeled as Channel 1, Channel 2, etc.

### Dependency

To see channel names, set `ShowLegend` to `true`.

Data Types: `char`

### OpenAtSimulationStart — Flag to open scope when simulation starts

`true` (default) | `false`

Flag to open scope when simulation starts, specified as `true` or `false`.

Set this property to `true` to open the scope when the simulation starts. Set this property to `false` to prevent the scope from opening when you simulate the model.

### Scope Window Use

This parameter is available only from the **Property Inspector**. To launch the **Property Inspector**, click the **Modeling** tab of the Simulink model window that contains the Spectrum Analyzer block. In the **Design** section, click the drop-down arrow and select **Property Inspector**. The **Property Inspector** window opens on the right. If you select the Spectrum Analyzer block, the **Property Inspector** window shows the corresponding parameters. In the parameters that show up, select or clear the **Open at simulation start** parameter.

Data Types: `logical`

## Examples

### Change the Array Plot Block Title and Scaling From the Command Line

Modify an Array Plot block with a configuration object.

Create a new Simulink model with a randomly-generated name.

```
sysname='ArrayPlotExample';
new_system(sysname);
```

Add a new Array Plot block to the model.

```
add_block('built-in/ArrayPlot',[sysname,'/ArrayPlot'])
```

Call the `get_param` function to retrieve the default Array Plot block configuration properties.

```
scopeConfig = get_param([sysname,'/ArrayPlot'],'ScopeConfiguration')
```

```
scopeConfig =
```

```
ArrayPlotConfiguration with properties:
```

```
    NumInputPorts: '1'  
      XDataMode: 'Sample increment and X-offset'  
SampleIncrement: '1'  
      XOffset: '0'  
    CustomXData: '[]'  
      XScale: 'Linear'  
      YScale: 'Linear'  
    PlotType: 'Stem'  
    AxesScaling: 'OnceAtStop'  
AxesScalingNumUpdates: '100'  
    Name: 'ArrayPlot'  
    Position: [240 262 800 500]  
MaximizeAxes: 'Auto'  
PlotAsMagnitudePhase: 0  
    Title: ''  
    XLabel: ''  
    YLabel: 'Amplitude'  
    YLimits: [-10 10]  
    ShowGrid: 1  
    ShowLegend: 0  
    ChannelNames: {''}  
OpenAtSimulationStart: 1  
    Visible: 0
```

Modify the Array Plot title and axes scaling.

```
scopeConfig.Title = 'My Array Plot';  
scopeConfig.AxesScaling = 'Manual'
```

```
scopeConfig =
```

```
ArrayPlotConfiguration with properties:
```

```
    NumInputPorts: '1'  
      XDataMode: 'Sample increment and X-offset'  
SampleIncrement: '1'  
      XOffset: '0'  
    CustomXData: '[]'  
      XScale: 'Linear'  
      YScale: 'Linear'  
    PlotType: 'Stem'  
    AxesScaling: 'OnceAtStop'  
AxesScalingNumUpdates: '100'  
    Name: 'ArrayPlot'  
    Position: [240 262 800 500]  
MaximizeAxes: 'Auto'  
PlotAsMagnitudePhase: 0  
    Title: ''  
    XLabel: ''  
    YLabel: 'Amplitude'
```

```
YLimits: [-10 10]
ShowGrid: 1
ShowLegend: 0
ChannelNames: {''}
OpenAtSimulationStart: 1
Visible: 0
```

## Version History

### Introduced in R2013a

#### **R2021a: ArrayPlotConfiguration is not recommended**

*Not recommended starting in R2021a*

The ArrayPlotConfiguration object is not recommended. To view and modify the Array Plot block properties from the command-line, use `get_param` and `set_param`. For example, to change the number of input ports, use these commands.

```
modelName='ArrayPlotExample';
new_system(modelName);
add_block('built-in/ArrayPlot',[sysname,'/ArrayPlot'])
set_param([modelName,'/ArrayPlot'],'NumInputPorts','4')
```

#### **R2022b: Channel names support array of strings**

Starting in R2022b, you can specify the ChannelNames property of the ArrayPlotConfiguration object as an array of strings.

```
scopeConfig = get_param(gcbh,'ScopeConfiguration')
scopeConfig.ChannelNames=["Input", "Lowpass Output"];
```

## See Also

Array Plot

### Topics

“Control Scope Blocks Programmatically” (Simulink)

# dsp.DynamicFilterVisualizer

Display time-varying magnitude and phase response of digital filters

## Description

The `dsp.DynamicFilterVisualizer` object displays the magnitude response and phase response of time-varying digital filters or time-varying filter coefficients. The input to this object can be a filter coefficients vector or a filter System object.

Using the dynamic filter visualizer, you can configure the plot settings, find the peak values, enable cursor measurements, and even generate script to recreate the plot with the current settings from the visualizer interface. For details, see “Configure Filter Visualizer”.

## Creation

### Syntax

```
dfv = dsp.DynamicFilterVisualizer
dfv = dsp.DynamicFilterVisualizer(nfft)
dfv = dsp.DynamicFilterVisualizer(nfft,Fs)
dfv = dsp.DynamicFilterVisualizer(nfft,Fs,range)
dfv = dsp.DynamicFilterVisualizer(Name=Value)
```

### Description

`dfv = dsp.DynamicFilterVisualizer` returns a dynamic filter visualizer object, `dfv`, that displays the magnitude response of digital filters or filter coefficients.

`dfv = dsp.DynamicFilterVisualizer(nfft)` returns a dynamic filter visualizer with the `FFTLength` property set to `nfft`.

`dfv = dsp.DynamicFilterVisualizer(nfft,Fs)` returns a dynamic filter visualizer with the `FFTLength` property set to `nfft` and the `SampleRate` property set to `Fs`.

`dfv = dsp.DynamicFilterVisualizer(nfft,Fs,range)` returns a dynamic filter visualizer with the `FFTLength` property set to `nfft`, the `SampleRate` property set to `Fs`, and the `FrequencyRange` property set to `range`.

`dfv = dsp.DynamicFilterVisualizer(Name=Value)` returns a dynamic filter visualizer with each specified property set to the specified value. You can specify name-value pair arguments in any order.

## Properties

### Frequently Used

#### **FFTLength** — FFT length

2048 (default) | positive integer

FFT length that the dynamic filter visualizer uses to compute spectral estimates, specified as a positive integer.

**Tunable:** Yes

#### Scope Window Use

In the **Configuration** section on the **Plot** tab of the Dynamic Filter Visualizer toolstrip, click **Settings**. In the Filter Visualizer Settings window that opens, under **Data and Axes**, set **FFT Length** to a positive integer.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

#### NormalizedFrequency — Flag to display normalized frequency

`false` (default) | `true`

Flag to display normalized frequency, specified as one of these values:

- `true` -- The filter visualizer displays the frequency response in normalized frequency units (0 to 1).
- `false` -- The filter visualizer displays the frequency response in Hz. You can specify the input sample rate through the `SampleRate` property.

Data Types: `logical`

#### SampleRate — Sampling rate of input

44100 (default) | positive scalar

Sampling rate of the input signal, specified as a real positive scalar in Hz.

**Tunable:** Yes

#### Dependency

To enable this property, set `NormalizedFrequency` to `false`.

#### Scope Window Use

On the **Plot** tab, in the **Configuration** section, click **Settings**. In the Filter Visualizer Settings window, under **Data and Axes**, set **Sample Rate (Hz)** to a positive scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

#### FrequencyRange — Range of frequency axis

[0 22050] (default) | two-element numeric vector

Range of the frequency axis, specified as a two-element numeric vector that is monotonically increasing and of the form [*fmin*, *fmax*].

- When you set the `NormalizedFrequency` property to `true`, *fmax* is in normalized frequency units and must be a positive scalar that is less than or equal to 1.
- When you set the `NormalizedFrequency` property to `false`, *fmax* is in Hz and must be less than or equal to  $F_s/2$ , where  $F_s$  is the value you specify in the `SampleRate` property.

**Tunable:** Yes

**Scope Window Use**

On the **Plot** tab, in the **Configuration** section, click **Settings**. In the Filter Visualizer Settings window, under **Data and Axes**, set **Frequency Range** to a two-element numeric vector.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**XScale — x-axis scale**

`'Log'` (default) | `'Linear'`

X-axis scale, specified as either `'Linear'` or `'Log'`.

**Tunable:** Yes

**Scope Window Use**

In the **Configuration** section on the **Plot** tab of the Dynamic Filter Visualizer toolstrip, click **Settings**. In the Filter Visualizer Settings window that opens, under **Data and Axes**, set **X-Scale** to `Linear` or `Log`.

**MagnitudeDisplay — y-axis units**

`'Magnitude (dB)'` (default) | `'Magnitude'` | `'Magnitude squared'`

Y-axis units, specified as one of the following:

- `'Magnitude'`
- `'Magnitude (dB)'`
- `'Magnitude squared'`

**Tunable:** Yes

**Scope Window Use**

In the **Configuration** section on the **Plot** tab of the Dynamic Filter Visualizer toolstrip, click **Settings**. In the Filter Visualizer Settings window that opens, under **Data and Axes**, set **Display Unit** to `Magnitude`, `Magnitude (dB)`, or `Magnitude Squared`.

**PlotAsMagnitudePhase — Plot magnitude and phase response of filter**

`false` (default) | `true`

- `true` - The filter visualizer plots the magnitude and phase responses of the filter on two separate axes.
- `false` - The filter visualizer plots only the magnitude response of the filter.

**Tunable:** Yes

**Scope Window Use**

In the **Configuration** section on the **Plot** tab of the Dynamic Filter Visualizer toolstrip, click the **Magnitude Phase** button.

**PlotType — Option to control type of plot**

`"Line"` (default) | `"Stairs"` | `"Stem"`

Specify the type of plot to use in the filter visualizer window as one of these:

- "Line" - The filter visualizer connects each point on the magnitude and phase response plot with a line.
- "Stairs" - The filter visualizer displays the filter response (magnitude, phase, or both) as a stair-step graph. A stair-step graph is made up of only horizontal lines and vertical lines. Each horizontal line represents the filter response over a frequency value and is connected to two vertical lines. Each vertical line represents a change in values occurring at a frequency.
- "Stem" - The filter visualizer displays the frequency response as circles with vertical lines extending down to the x-axis at each of the frequency values.

**Tunable:** Yes

#### Scope Window Use

In the **Configuration** section on the **Plot** tab of the Dynamic Filter Visualizer toolstrip, click **Settings**. In the Filter Visualizer Settings window that opens, under **Data and Axes**, set **Plot Type** to Line, Stairs, or Stem.

#### AxesScaling — Axes scaling mode





"OnceAtStop" (default) | "Auto" | "Manual" | "Updates"

Specify the scaling mode for the axes as one of these:

- "Auto" — The filter visualizer scales the axes as needed to fit the data, both during and after simulation.
- "Manual" — The filter visualizer does not scale the axes automatically.
- "OnceAtStop" — The filter visualizer scales the axes when the simulation stops.
- "Updates" — The filter visualizer scales the axes limits once after a set number of visual updates. The number of updates is determined by the value of the AxesScalingNumUpdates property.

**Tunable:** Yes

#### Scope Window Use

Hover over the filter visualizer to see the maximize , pan , zoom , and autoscale  buttons. You can also zoom and pan using your mouse.

Data Types: char | string

#### AxesScalingNumUpdates — Number of updates before scaling

100 (default) | real positive integer

Specify the number of updates before scaling as a real, positive scalar integer.

**Tunable:** Yes

#### Dependency

To enable this property, set AxesScaling to "Updates".

Data Types: double

#### Measurements

##### MeasurementChannel — Channel for which to obtain measurements

1 (default) | positive integer

Channel for which to obtain measurements, specified as a positive integer in the range [1  $N$ ], where  $N$  is the number of input channels.

**Tunable:** Yes

### Scope Window Use

Click the **Measurements** tab on the Dynamic Filter Visualizer toolstrip. In the **Channel** section, select a **Channel**.

Data Types: double

### CursorMeasurements — Cursor measurements

CursorMeasurementsConfiguration object

Cursor measurements to display waveform cursors, specified as a CursorMeasurementsConfiguration object.

All CursorMeasurementsConfiguration properties are tunable.

**Tunable:** Yes

### Scope Window Use

Click the **Measurements** tab on the Dynamic Filter Visualizer toolstrip and modify the cursor measurements in the **Cursors** section.

### PeakFinder — Peak finder measurements

PeakFinderConfiguration object

Peak finder measurements to compute and display the largest calculated peak values, specified as a PeakFinderConfiguration object.

All PeakFinderConfiguration properties are tunable.

**Tunable:** Yes

### Scope Window Use

Click the **Measurements** tab on the Dynamic Filter Visualizer toolstrip and modify the peak finder measurements in the **Peaks** section.

### Visualization

#### Name — Caption to display on Dynamic Filter Visualizer window

'Dynamic Filter Visualizer' (default) | character vector | string scalar

Caption to display on the Dynamic Filter Visualizer window, specified as a character vector or a string scalar.

Example: 'Dynamic Filter Visualizer'

Example: "Dynamic Filter Visualizer"

**Tunable:** Yes

#### Position — Scope window position in pixels

[left bottom 800 500] (default) | four-element double vector



Scope window position in pixels, specified as a four-element double vector of the form [left bottom width height]. The default value of this property is dependent on the screen resolution, and is such that the window is positioned in the center of the screen, with a width and height of 800 and 500 pixels, respectively.

**Tunable:** Yes

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### MaximizeAxes — Maximize axes control

"Auto" (default) | "On" | "Off"

Specify whether to display the filter visualizer in maximized-axes mode. In this mode, the axes are expanded to fit into the entire display. To conserve space, labels do not appear in each display. Instead, tick-mark values appear on top of the plotted data. You can select one of the following options:

- "Auto" — The axes appear maximized in all displays only if the `Title` and `YLabel` properties are empty for every display. If you enter any value in any display for either of these properties, the axes are not maximized.
- "On" — The axes appear maximized in all displays. Any values entered into the `Title` and `YLabel` properties are hidden.
- "Off" — None of the axes appear maximized.

**Tunable:** Yes

### Scope Window Use

Hover over the Dynamic Filter Visualizer window to see the maximize axes button



Data Types: `char` | `string`

### Title — Display title

' ' (default) | character vector | string scalar

Display title, specified as a character vector or a string scalar.

Example: 'Magnitude Response'

Example: "Magnitude Response"

**Tunable:** Yes

### Scope Window Use

In the **Configuration** section on the **Plot** tab of the Dynamic Filter Visualizer toolstrip, click **Settings**. In the Filter Visualizer Settings window that opens, under **Display and Labels**, set **Title** to a character vector or a string scalar.

### YLimits — y-axis limits

[-25 25] (default) | two-element row vector

Y-axis limits, specified as a two-element numeric row vector with the second element greater than the first element and of the form [*ymin*, *ymax*].

**Tunable:** Yes

**Scope Window Use**

In the **Configuration** section on the **Plot** tab of the Dynamic Filter Visualizer toolbar, click **Settings**. In the Filter Visualizer Settings window that opens, under **Data and Axes**, set **Y-Axis Limits** to a two-element numeric vector.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**ShowLegend — Show or hide legend**

`false` (default) | `true`

When this property is set to `false`, no legend is displayed. When this property is set to `true`, a legend with automatic string labels for each input filter is displayed.

**Tunable:** Yes

**Scope Window Use**

In the **Configuration** section on the **Plot** tab of the Dynamic Filter Visualizer toolbar, click **Legend**.

Data Types: `logical`

**FilterNames — Names for input filters**

`{ '' }` (default) | cell array of character vectors | array of strings

Names to label the input filters in the legend, specified as a cell array of character vectors or an array of strings. The default is an empty cell array. When this property is set to an empty cell array, the filters are named by default names, such as `Filter 1`, `Filter 2`, and so on.

**Tunable:** Yes

**Scope Window Use**

In the **Configuration** section on the **Plot** tab of the Dynamic Filter Visualizer toolbar, click **Legend**. In the legend that appears on the plot, click the filter name.

**ShowGrid — Display grid**

`true` (default) | `false`

Set this property to `true` to show grid lines on the plot.

**Scope Window Use**

In the **Configuration** section on the **Plot** tab of the Dynamic Filter Visualizer toolbar, click **Settings**. In the Filter Visualizer Settings window that opens, under **Data and Axes**, select **Grid**.

**UpperMask — Upper limit mask**

`Inf` (default) | two-column matrix

Upper limit spectral mask, specified as a two-column matrix. The first column represents the frequency values (Hz), and the second column represents the magnitude spectrum of the upper limit mask.

**Tunable:** Yes

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**LowerMask — Lower limit mask**

-Inf (default) | two-column matrix

Lower limit spectral mask, specified as a two-column matrix. The first column represents the frequency values (Hz), and the second column represents the magnitude spectrum of the lower limit mask.

**Tunable:** Yes

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**Usage****Syntax**

```
dfv(filt)
dfv(B,A)
```

**Description**

`dfv(filt)` displays the time-varying magnitude response of the object filter, `filt`, in the Dynamic Filter Visualizer figure, as long as `filt` has a valid `freqz()` implementation.

`dfv(B,A)` displays the magnitude response for the digital filters with numerator and denominator polynomial coefficients stored in  $B_1$  and  $A_1$ ,  $B_2$  and  $A_2$ , ..., and  $B_N$  and  $A_N$ , respectively.

**Input Arguments****filt — Input filter**

filter System object

Input filter System object with a valid `freqz()` implementation.

**B — Numerator polynomial coefficients**

row vector

Numerator polynomial coefficients, specified as a row vector.

Data Types: single | double

**A — Denominator polynomial coefficients**

scalar | row vector

Denominator polynomial coefficients, specified as a:

- scalar -- The filter is an FIR filter.
- row vector -- The filter is an IIR filter.

Data Types: single | double

## Object Functions

### Specific to `dsp.DynamicFilterVisualizer`

`step` Display time-varying magnitude response

### Specific to Scopes

`show` Display scope window  
`hide` Hide scope window  
`isVisible` Determine visibility of scope  
`generateScript` Generate MATLAB script to create scope with current settings

## Examples

### Plot Time-Varying Magnitude and Phase Response of FIR Filter

Design an FIR filter with a time-varying magnitude and phase response. Plot this varying response on a dynamic filter visualizer in normalized frequency units.

Create a `dsp.DynamicFilterVisualizer` object. Set the `PlotAsMagnitudePhase` and the `NormalizedFrequency` properties to true.

```
dfv = dsp.DynamicFilterVisualizer(PlotAsMagnitudePhase=1,...  
    NormalizedFrequency=true,ShowLegend=true,...  
    Title='Magnitude and Phase Response',...  
    FilterNames="FIR Filter")
```

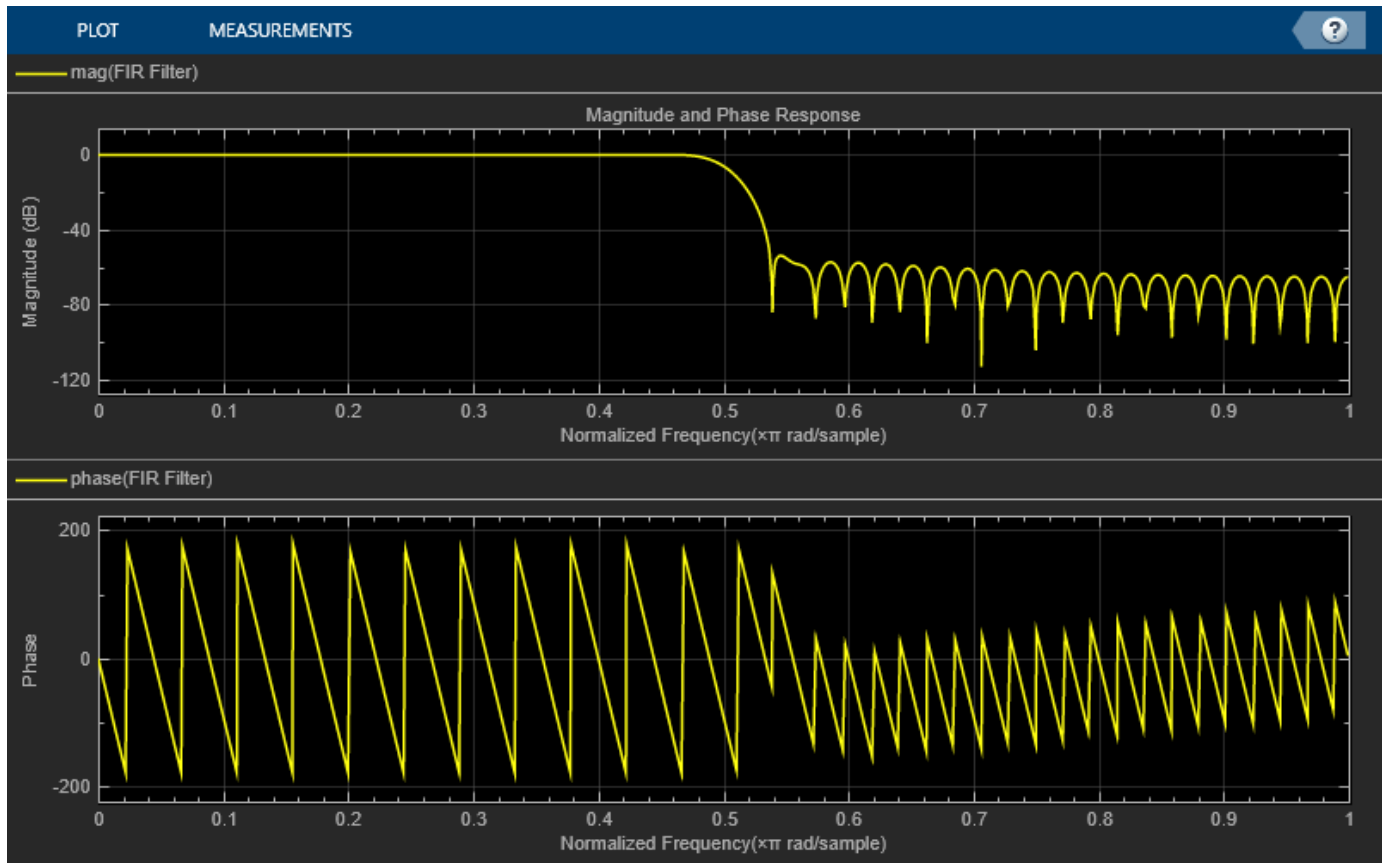
```
dfv =  
    dsp.DynamicFilterVisualizer with properties:
```

```
        FFTLength: 2048  
    NormalizedFrequency: 1  
        FrequencyRange: [0 1]  
            XScale: 'Linear'  
    MagnitudeDisplay: 'Magnitude (dB)'  
    PlotAsMagnitudePhase: 1  
            PlotType: 'Line'  
        AxesScaling: 'Auto'
```

Show all properties

Vary the cutoff frequency of the FIR filter  $k$  from 0.1 to 0.5 in increments of 0.001. View the varying magnitude and phase response using the dynamic filter visualizer.

```
for k = 0.1:0.001:0.5  
    b = fir1(90,k);  
    dfv(b,1);  
end
```



### Plot Time-Varying Magnitude Response of Variable Bandwidth FIR Filter

Visualize the varying magnitude response of the variable bandwidth FIR filter using the dynamic filter visualizer.

Create a `dsp.DynamicFilterVisualizer` object.

```
dfv = dsp.DynamicFilterVisualizer(YLimits=[-160 10],...
    FilterNames="Variable Bandwidth FIR Filter")
```

```
dfv =
    dsp.DynamicFilterVisualizer with properties:
```

```

        FFTLength: 2048
    NormalizedFrequency: 0
        SampleRate: 44100
    FrequencyRange: [0 22050]
        XScale: 'Linear'
    MagnitudeDisplay: 'Magnitude (dB)'
    PlotAsMagnitudePhase: 0
        PlotType: 'Line'
    AxesScaling: 'Manual'
```

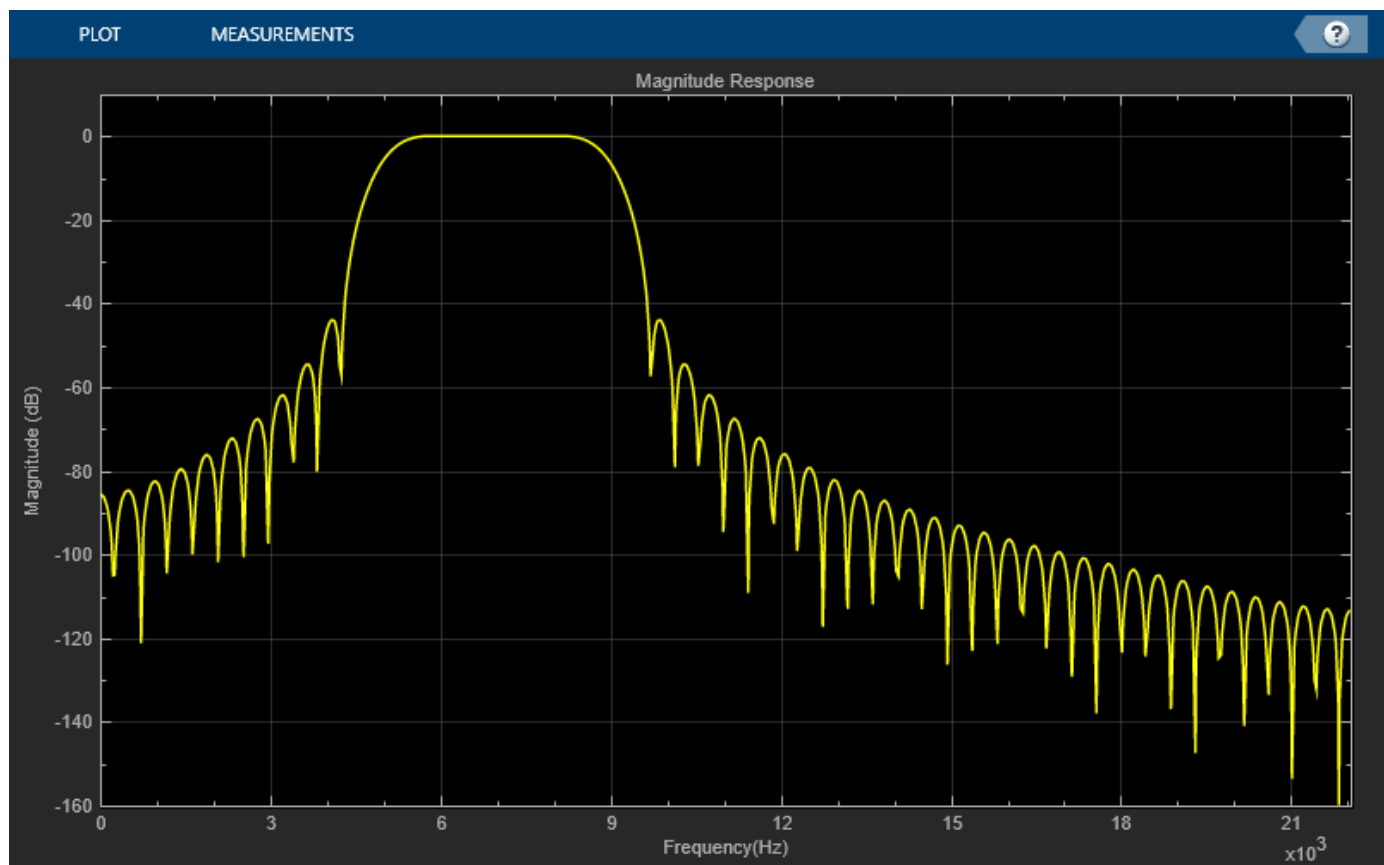
Show all properties

Design a bandpass variable bandwidth FIR filter with a center frequency of 5 kHz and a bandwidth of 4 kHz.

```
Fs = 44100;
vbw = dsp.VariableBandwidthFIRFilter(FilterType='Bandpass',...
    FilterOrder=100,...
    SampleRate=Fs,...
    CenterFrequency=5e3,...
    Bandwidth=4e3);
```

Vary the center frequency of the filter. Visualize the varying magnitude response of the filter using the `dsp.DynamicFilterVisualizer` object.

```
for idx = 1:100
    dfv(vbw);
    vbw.CenterFrequency = vbw.CenterFrequency + 20;
end
```



## Version History

Introduced in R2018b

**R2022b:** Filter names support array of strings

Starting in R2022b, you can specify the `FilterNames` property of the `dsp.DynamicFilterVisualizer` object as an array of strings.

```
dfv = dsp.DynamicFilterVisualizer(FilterNames=["Filter 1", "Filter 2"]);
```

### **R2022b: Enhancements to the dsp.DynamicFilterVisualizer object and the Dynamic Filter Visualizer UI**

The `dsp.DynamicFilterVisualizer` object has the following new properties starting R2022b:

- `PlotAsMagnitudePhase` -- Split the magnitude and phase of the input signal and plot them on two separate axes within the same window
- `PlotType` -- Plot type
- `AxesScaling` -- Axes scaling mode
- `AxesScalingNumUpdates` -- Number of updates before scaling
- `MaximizeAxes` -- Maximize axes control
- `ShowGrid` -- Grid visibility

The **Dynamic Filter Visualizer** app that you launch using the object has these enhancements in R2022b:

- **Magnitude Phase** button -- This button corresponds to the `PlotAsMagnitudePhase` property in the object.
- **Generate Script** button -- Generate a script to re-create your dynamic filter visualizer plot with the same settings. When you click this button, an editor window opens with the code that you can use to re-create your `dsp.DynamicFilterVisualizer` object.

### **R2022b: Configure dsp.DynamicFilterVisualizer measurements programmatically**

You can now configure the cursor and the peak finder measurements in the `dsp.DynamicFilterVisualizer` object programmatically using these properties:

- `MeasurementChannel`
- `CursorMeasurements`
- `PeakFinder`

### **R2023a: Normalized frequency in dsp.DynamicFilterVisualizer**

When you set the `NormalizedFrequency` property to `true`, the dynamic filter visualizer shows the frequency response of the filter in normalized frequency units (0 to 1).

## **See Also**

### **Functions**

`show` | `hide` | `step` | `isVisible` | `generateScript`

### **Objects**

`timescope` | `dsp.ArrayPlot` | `dsp.LogicAnalyzer` | `spectrumAnalyzer`

**Blocks**

Filter Visualizer

**Topics**

“Configure Filter Visualizer”

“System Identification Using RLS Adaptive Filtering”



# dsp.FourthOrderSectionFilter

Implement cascade of fourth-order section filter

## Description

The `dsp.FourthOrderSectionFilter` implements a cascade of fourth-order section filters.

## Creation

### Syntax

```
fos = dsp.FourthOrderSectionFilter
fos = dsp.FourthOrderSectionFilter(num,den)
fos = dsp.FourthOrderSectionFilter(Name,Value)
```

### Description

`fos = dsp.FourthOrderSectionFilter` returns a `FourthOrderSectionFilter` object, `fos`, that implements a cascade of fourth order filter sections.

`fos = dsp.FourthOrderSectionFilter(num,den)` returns a `FourthOrderSectionFilter` object with the `Numerator` property set to `num` and the `Denominator` property set to `den`.

`fos = dsp.FourthOrderSectionFilter(Name,Value)` returns a `FourthOrderSectionFilter` object with each specified property name set to the specified value. You can specify additional name-value pair arguments in any order.

Example: `fos = dsp.FourthOrderSectionFilter('Numerator',num,'Denominator',den)`

## Properties

### **num** — Numerator coefficients of filter

[1 0.1 0.2 0.3 0.4] (default) | row vector | matrix

Numerator coefficients of the filter, specified as an  $L$ -by-5 matrix, where  $L$  is the number of filter sections. The size of this property cannot change when the object is locked. However, the values can be modified.

Data Types: `single` | `double`

Complex Number Support: Yes

### **den** — Denominator coefficients of filter

[1 0.1 0.2 0.3 0.4] (default) | row vector | matrix

Denominator coefficients of the filter, specified as an  $L$ -by-5 matrix or an  $L$ -by-4 matrix, where  $L$  is the number of filter sections. The leading denominator coefficients are assumed to be 1 always. If the denominator is of size  $L$ -by-4, one(s) are appended to make the size  $L$ -by-5. If the denominator is of size  $L$ -by-5, the first column values are ignored and appended with 1s. The size of this property cannot change when the object is locked. However, the values can be modified.

Data Types: `single` | `double`  
Complex Number Support: Yes

### Usage

### Syntax

$y = \text{fos}(x)$

### Description

$y = \text{fos}(x)$  filters the input signal using the specified fourth-order section filter to produce the filtered output,  $y$ .

### Input Arguments

#### **x** — Input signal

vector | matrix

Input signal, specified as a vector or a matrix.

The input can be a variable-sized signal, that is, the frame size of each channel (number of rows) can change even after the object is locked. However, the number of channels (number of columns) cannot change.

Data Types: `single` | `double`

### Output Arguments

#### **y** — Filtered output

vector | matrix

Filtered output, returned as a vector or a matrix. The output has the same size, data type, and complexity as the input signal.

Data Types: `single` | `double`

### Object Functions

#### Specific to `dsp.FourthOrderSectionFilter`

|                       |  |
|-----------------------|--|
| <code>fvtool</code>   | Visualize frequency response of DSP filters                  |
| <code>freqz</code>    | Frequency response of discrete-time filter System object     |
| <code>impz</code>     | Impulse response of discrete-time filter System object       |
| <code>info</code>     | Information about filter System object                       |
| <code>coeffs</code>   | Returns the filter System object coefficients in a structure |
| <code>cost</code>     | Estimate cost of implementing filter System object           |
| <code>grpdelay</code> | Group delay response of discrete-time filter System object   |

#### Common to All Objects

`step` Run System object algorithm

|          |  |
|----------|--|
| release  | Release resources and allow changes to System object property values and input characteristics |
| reset    | Reset internal states of System object   |
| clone    | Create duplicate System object   |
| isLocked | Determine if System object is in use   |

## Examples

### Filter Noisy Signal Using Fourth-Order Section (FOS) Filter

Filter a noisy sinusoidal signal using the `dsp.FourthOrderSectionFilter` object. Visualize the original and filtered signals using a spectrum analyzer.

#### Input Signal

The input signal is the sum of two sine waves with frequencies 100 Hz and 350 Hz. Add zero-mean white Gaussian noise with a standard deviation of  $1e-4$  to the sum of sine waves. The sample rate is 1000 Hz.

```
frameSize = 1024;
fs = 1000;
Sine1 = dsp.SineWave(5,100,'SamplesPerFrame',1024,'SampleRate',fs);
Sine2 = dsp.SineWave(2,350,pi/2,'SamplesPerFrame',1024,...
    'SampleRate',fs);
x = Sine1()+Sine2()+1e-4.*randn(Sine1.SamplesPerFrame,1);
```

#### Fourth-Order Section (FOS) Filter Coefficients

The numerator and denominator coefficients for the FOS filter are obtained using `designParamEq` which is part of Audio Toolbox:

```
%N = [2,4];
%gain = [5,10];
%centerFreq = [0.025,0.75];
%bandwidth = [0.025,0.35];
%mode = 'fos';
%[num,den] = designParamEQ(N,gain,centerFreq,bandwidth,mode,'Orientation','row');

num = [1.0223    -1.9368    0.9205         0         0
       1.5171    2.3980    1.4317    0.6416    0.2752];

den = [-1.9368    0.9428         0         0
       2.0136    1.9224    1.0260    0.3016];
```

#### Initialize Filter and Spectrum Analyzer

Construct the FOS IIR filter using the `num` and `den` coefficients. Construct a spectrum analyzer to visualize the original sinusoidal signal and the filtered signal.

```
fos = dsp.FourthOrderSectionFilter('Numerator',num,...
    'Denominator',den);

scope = spectrumAnalyzer(...
    'SampleRate',fs,...
    'PlotAsTwoSidedSpectrum',false,...
```

```

'FrequencyScale','linear',...
'Method','welch',...
'Title','Original and Filtered Signals',...
>ShowLegend',true,...
'ChannelNames',{'Original Signal','Filtered Signal'});

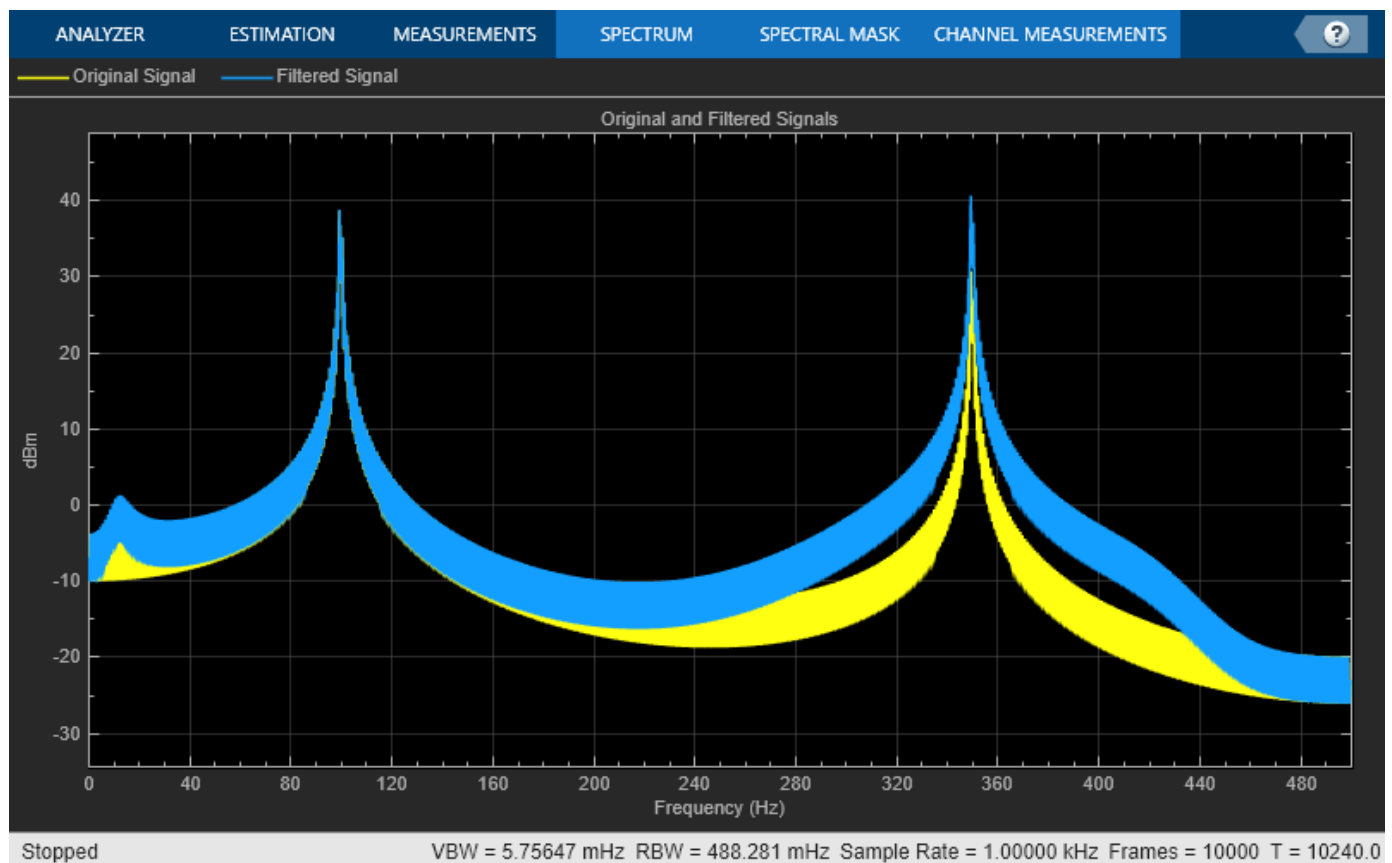
```

Filter the input signal, and visualize the original and filtered spectrums.

```

for i = 1:10000
    y = fos(x);
    scope([x,y]);
end
release(scope);

```



### Lowpass Fourth-Order Section (FOS) Filter

Design a lowpass fourth-order section (FOS) filter using the `fdesign` function. Using this filter, filter a noisy sinusoidal signal with two tones, one at 3 kHz, and the other at 12 kHz.

Design a fifth-order filter using the elliptic method in the `'df2tsos'` structure. Use L-infinity norm scaling in the frequency domain. Specify the passband frequency to be  $0.15\pi$  rad/sample and the stopband frequency to be  $0.25\pi$  rad/sample. Specify 1 dB of allowable passband ripple and a stopband attenuation of 60 dB.

```
Fp = 0.15;  
Fst = 0.25;  
Ap = 1;  
Ast = 60;
```

The filter coefficients are scaled using an `fdopts.sosscaling` object. The scaling object is defined to have no numerator constraints, and the `ScaleValueConstraint` is set to `'unit'`, specifying the scaling to be unity scaling.

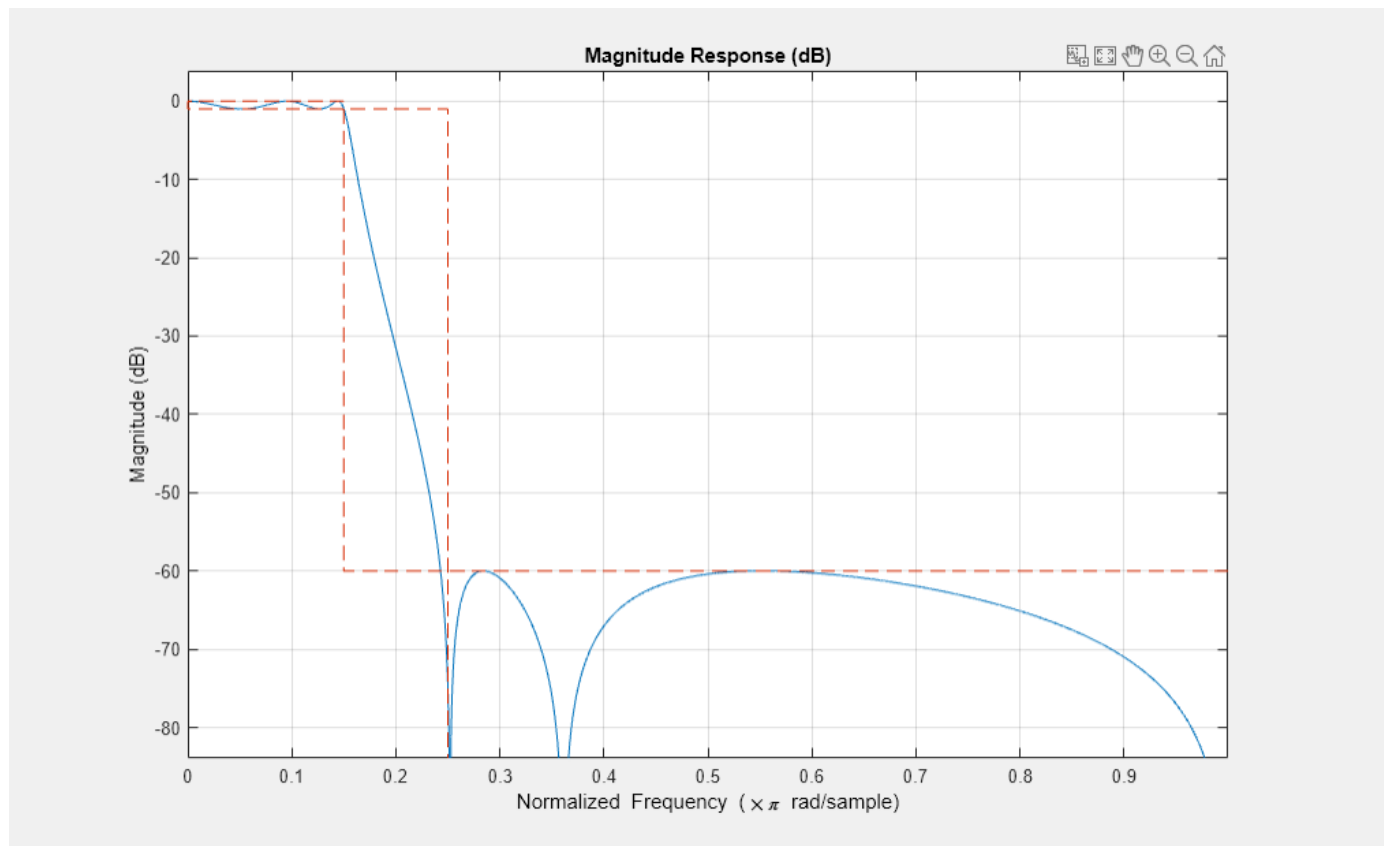
```
fdo = fdopts.sosscaling;  
fdo.NumeratorConstraint='none';  
fdo.ScaleValueConstraint='unit';  
  
f = fdesign.lowpass('Fp,Fst,Ap,Ast',Fp,Fst,Ap,Ast);  
hFilter = design(f,'ellip','SystemObject',true,...  
    'FilterStructure','df2tsos','SOSScaleNorm','Linf',...  
    'SOSScaleOpts',fdo)
```

```
hFilter =  
    dsp.SOSFilter with properties:  
  
        Structure: 'Direct form II transposed'  
    CoefficientSource: 'Property'  
        Numerator: [3x3 double]  
        Denominator: [3x3 double]  
    HasScaleValues: true  
        ScaleValues: [1 1 1 1.0000]
```

Show all properties

Visualize the lowpass frequency response of the designed filter using `fvtool`.

```
fvtool(hFilter)
```



Convert the lowpass filter to a bandpass filter using the `iirlp2bp` function.

```
[num,den] = iirlp2bp(hFilter.Numerator,hFilter.Denominator,Fp,[0.25,0.75])
```

```
num = 3x5
```

```
    0.2456         0   -0.2456         0         0
    0.4175   -0.0000   -0.4206   -0.0000    0.4175
    0.4281   -0.0000   -0.6433   -0.0000    0.4281
```

```
den = 3x5
```

```
    1.0000   -0.0000    0.5088         0         0
    1.0000   -0.0000    0.0060   -0.0000    0.8657
    1.0000   -0.0000    0.5160   -0.0000    0.5283
```

Create a fourth-order section filter using these numerator and denominator coefficients.

```
fos = dsp.FourthOrderSectionFilter(num,den)
```

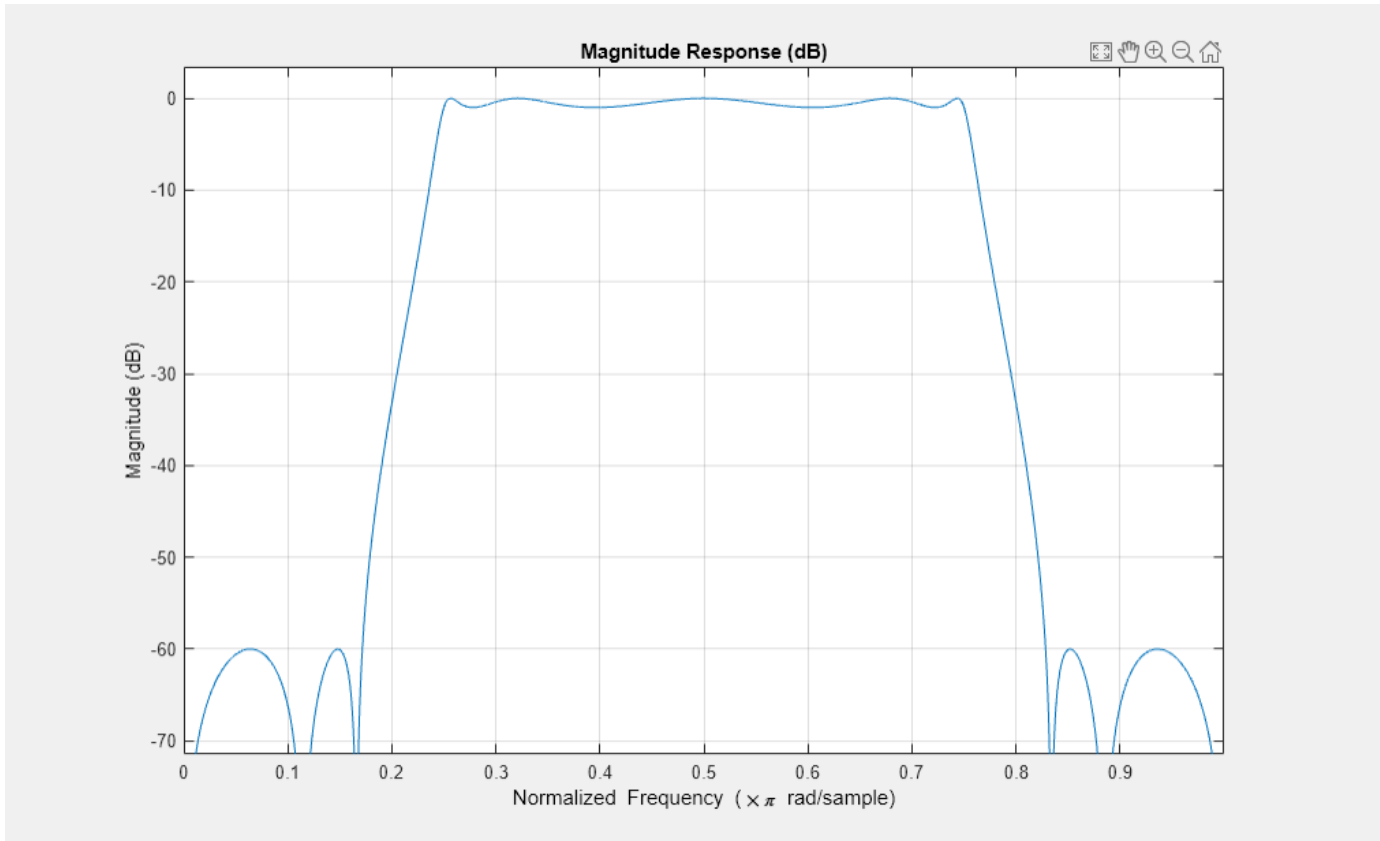
```
fos =
```

```
FourthOrderSectionFilter with properties:
```

```
    Numerator: [3x5 double]
    Denominator: [3x5 double]
```

Visualize the frequency response of the fourth-order section filter using `fvtool`.

```
fvtool(fos);
```



Filter a noisy input signal with the fourth-order section filter. Visualize the spectra of the original signal and the filtered signal using the spectrum analyzer.

The input is a sum of two sine waves with frequencies 3 kHz and 12 kHz, respectively. The input sample rate is 44.1 kHz, and the frame size is set to 1024 samples.

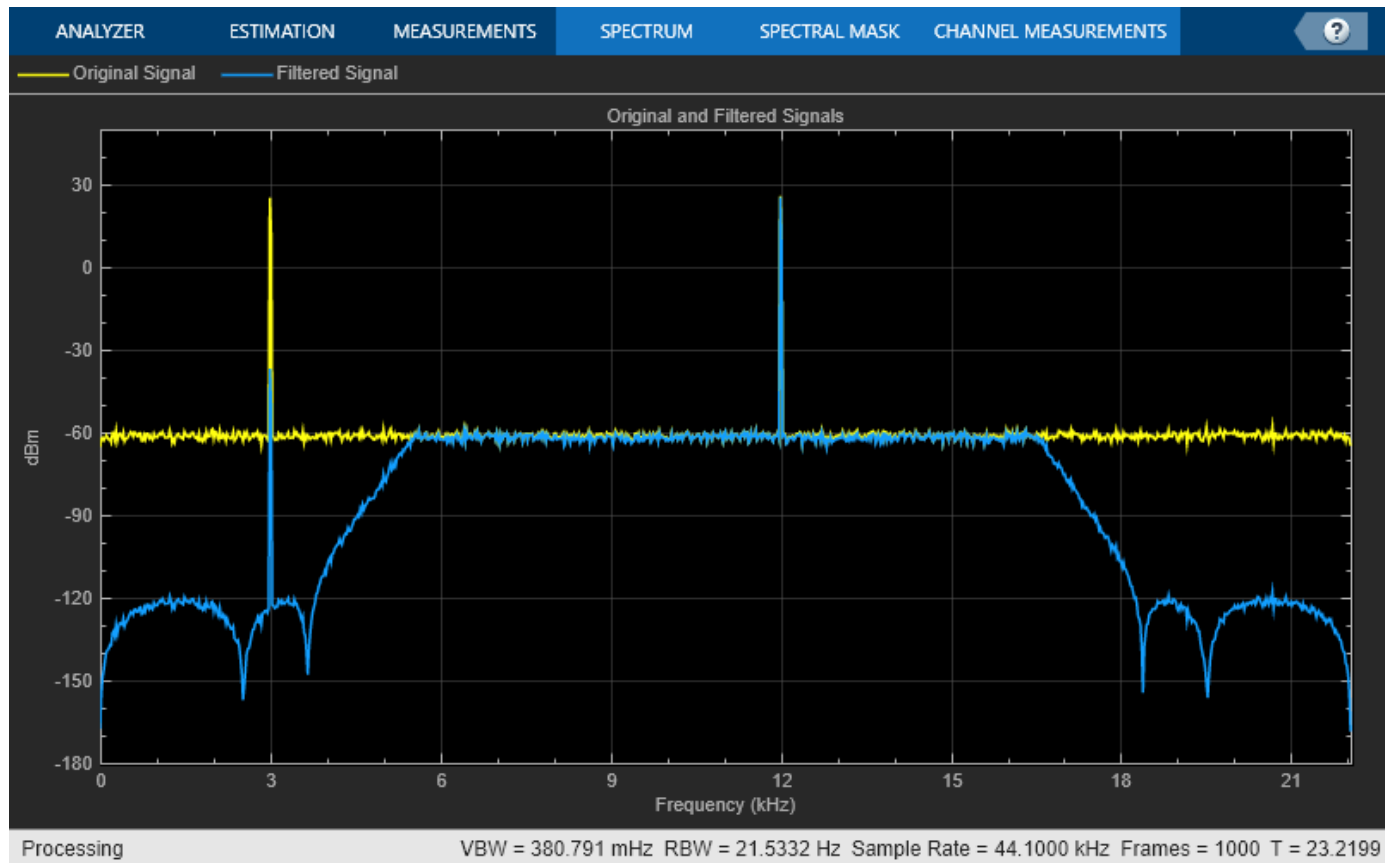
```
fs = 44100;
FrameLength = 1024;
```

```
SINE1 = dsp.SineWave('SamplesPerFrame',FrameLength,'SampleRate',fs,'Frequency',3000);
SINE2 = dsp.SineWave('SamplesPerFrame',FrameLength,'SampleRate',fs,'Frequency',12000);
```

Initialize a spectrum analyzer to visualize the signal spectra.

```
scope = spectrumAnalyzer(...
    'SampleRate',fs,...
    'PlotAsTwoSidedSpectrum',false,...
    'Title','Original and Filtered Signals',...
    'ShowLegend',true,...
    'YLimits',[-180 50],...
    'ChannelNames',{'Original Signal','Filtered Signal'});
for index = 1:1000
    x = SINE1() + SINE2() + 0.001*randn(FrameLength,1);
    y = fos(x);
```

```
scope([x,y]);  
end
```



## Version History

Introduced in R2019a

## See Also

### Objects

`dsp.SOSFilter`

### Blocks

Fourth-Order Section Filter



# dsp.ISTFT

Inverse short-time FFT

## Description

The `dsp.ISTFT` object computes the inverse short-time Fourier transform (ISTFT) of the frequency-domain input signal and returns the time-domain output. The object accepts frames of Fourier-transformed data, converts these frames into the time domain using the IFFT operation, and performs overlap-add to reconstruct the data. The output of the object is the reconstructed signal normalized by a factor that depends on the hop length and `sum(window)`. For more details, see “Algorithms” on page 4-1483.

## Creation

### Syntax

```
istf = dsp.ISTFT
istf = dsp.ISTFT(window)
istf = dsp.ISTFT(window,overlap)
istf = dsp.ISTFT(window,overlap,isconjsym)
istf = dsp.ISTFT(window,overlap,isconjsym,woa)
istf = dsp.ISTFT(Name,Value)
```

### Description

`istf = dsp.ISTFT` returns an object, `istf`, that implements inverse short-time FFT. The object processes the data independently across each input channel over time.

`istf = dsp.ISTFT(window)` returns an inverse short-time FFT object with the `Window` property set to `window`.

`istf = dsp.ISTFT(window,overlap)` returns an inverse short-time FFT object with the `Window` property set to `window` and the `OverlapLength` property set to `overlap`.

`istf = dsp.ISTFT(window,overlap,isconjsym)` returns an inverse short-time FFT object with the `Window` property set to `window`, `OverlapLength` property set to `overlap`, and the `ConjugateSymmetricInput` property set to `isconjsym`.

`istf = dsp.ISTFT(window,overlap,isconjsym,woa)` returns an inverse short-time FFT object with the `Window` property set to `window`, with the `OverlapLength` property set to `overlap`, the `ConjugateSymmetricInput` property set to `isconjsym`, and the `WeightedOverlapAdd` property set to `woa`.

`istf = dsp.ISTFT(Name,Value)` returns an inverse short-time FFT object with each specified property name set to the specified value. You can specify additional name-value pair arguments in any order.

## Properties

### Window — Synthesis window

`sqrt(hann(512, 'periodic'))` (default) | vector

Synthesis window, specified as a vector of real elements.

**Tunable:** Yes

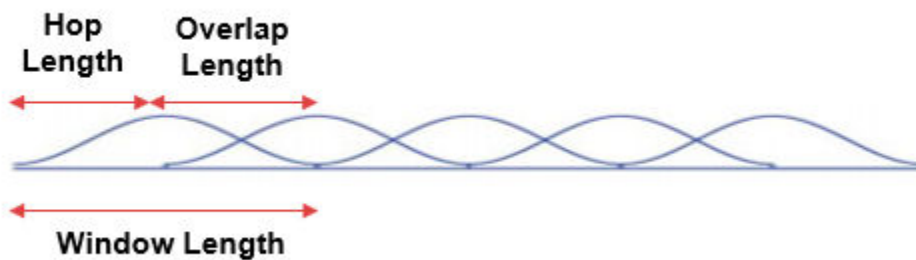
Data Types: `single` | `double`

### OverlapLength — Overlap length

256 (default) | positive integer

Number of samples by which consecutive windows overlap, specified as a positive integer. The windows overlap to reduce the artifacts at the data boundaries.

Hop length is the difference between the window length and the overlap length.



Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### ConjugateSymmetricInput — Input is conjugate symmetric

`true` (default) | `false`

Set this property to `true` if the input is conjugate symmetric, which yields real-valued outputs. The FFT of a real-valued signal is conjugate symmetric, and setting this property to `true` optimizes the IFFT computation method. Setting this property to `false` for conjugate symmetric inputs results in complex output values with small imaginary parts. Setting this property to `true` for non-conjugate symmetric inputs results in invalid outputs.

Data Types: `logical`

### WeightedOverlapAdd — Apply weighted overlap-add

`true` (default) | `false`

Set this property to `true` to apply weighted overlap-add. In weighted overlap-add, the IFFT output is multiplied by the window before overlap-add. Set this property to `false` to skip multiplication by the window.

Data Types: `logical`

### FrequencyRange — Frequency range

`'twosided'` (default) | `'onesided'`

Specify the frequency range as 'onesided' or 'twosided'. If you set the FrequencyRange property to:

- 'twosided' -- The inverse short-time FFT is computed for a two-sided short-time FFT. The FFT length used is equal to the input frame length.
- 'onesided' -- The one-sided inverse short-time FFT is computed for a one-sided short-time FFT. If the input frame length is odd, the FFT length used is  $(\text{frame length} - 1) \times 2$ . If the input frame length is even, the FFT length used is  $(\text{frame length} \times 2) - 1$ .

## Usage

### Syntax

```
y = istft(x)
```

### Description

`y = istft(x)` applies inverse short-time FFT on the input `x`, and returns the time-domain output `y`.

### Input Arguments

#### **x** — Input signal

vector | matrix

Frequency-domain input signal, specified as a vector or a matrix. If the input is a matrix, the object treats each column as an independent channel. The FFT length is equal to the number of rows of `x`. The FFT length, hence the number of input rows must be greater than or equal to the window length.

Data Types: `single` | `double`

### Output Arguments

#### **y** — ISTFT output

vector | matrix

Inverse short-time FFT output, returned as a vector or a matrix. The output frame length (number of rows in `y`) is equal to  $WL - OL$ , where  $WL$  is the window length and  $OL$  is the overlap length.

The output is complex with small imaginary parts when the input `x` is conjugate symmetric and the `ConjugateSymmetricInput` property is set to `false`. The data type of the output matches the data type of the input signal.

Data Types: `single` | `double`

## Object Functions

|                       |  |
|-----------------------|--|
| <code>step</code>     | Run System object algorithm  |
| <code>release</code>  | Release resources and allow changes to System object property values and input characteristics |
| <code>reset</code>    | Reset internal states of System object   |
| <code>clone</code>    | Create duplicate System object   |
| <code>isLocked</code> | Determine if System object is in use   |

## Examples

### Short-Time Spectral Attenuation

Short-time spectral attenuation is achieved by applying a time-varying attenuation to the short-time spectrum of a noisy signal. The gain of the attenuation is determined by the estimate of the noise power in each subband of the spectrum. This gain, when applied to the noisy spectrum, attenuates the subbands with higher noise power and lifts the subbands with lesser noise power.

Here are the steps involved in performing the short-time spectral attenuation:

- 1 Analyze the noisy input signal by computing the short-time Fourier transform (STFT).
- 2 Multiply each subband of the transformed signal with a real positive gain less than 1.
- 3 Synthesize the denoised subbands by taking the inverse short-time Fourier transform (ISTFT). The reconstructed signal is the denoised input signal.

Use the `dsp.STFT` and `dsp.ISTFT` objects to compute the short-time and the inverse short-time Fourier transforms, respectively.

### Noisy Input Signal

The input is an audio signal sampled at the 22,050 Hz. The `dsp.AudioFileReader` object reads this signal in frames of 512 samples. The audio signal is corrupted by white Gaussian noise that has a standard deviation of 0.05. Use the `audioDeviceWriter` object to play the noisy audio signal to your computer's audio device.

```
FrameLength = 512;
afr = dsp.AudioFileReader('speech_dft.wav',...
    'SamplesPerFrame',FrameLength);
adw = audioDeviceWriter('SampleRate',afr.SampleRate);

noiseStd = 0.05;
while ~isDone(afr)
    cleanAudio = afr();
    noisyAudio = cleanAudio + noiseStd * randn(FrameLength,1);
    adw(noisyAudio);
end
reset(afr)
```

### Initialize Short-Time and Inverse Short-Time Fourier Transform Objects

Initialize the `dsp.STFT` and `dsp.ISTFT` objects. Set the window length equal to the input frame length and the hop length to 16. The overlap length is the difference between the window length and the hop length,  $OL = WL - HL$ . Set the FFT length to 1024.

```
WindowLength = FrameLength;
HopLength = 16;
numHopsPerFrame = FrameLength / 16;
FFTLength = 1024;
```

The window used to compute the STFT and ISTFT is a periodic hamming window with length 512. The `ConjugateSymmetricInput` flag of the `istf` object is set to `true`, indicating that the output of the `istf` object is a conjugate-symmetric signal.

```
win = hamming(WindowLength, 'periodic');
stf = dsp.STFT(win, WindowLength-HopLength, FFTLength);
istf = dsp.ISTFT(win, WindowLength-HopLength, 1, 0);
```

### Gain Estimator

The next step is to define the gain estimator parameters. This gain is applied to the noisy spectrum to attenuate the subbands with higher noise power and lifter the subbands with lesser noise power.

```
dec = 16;
alpha = 15;
stftNorm = (sum(win.*win) / dec).^2;
```

### Spectral Attenuation

Feed the audio signal to `stf` one hop-length at a time. Apply the estimated gain to the transformed signal. Reconstruct the denoised version of the original speech signal by performing an inverse Fourier transform on the individual frequency bands. Play the denoised audio signal to the computer's audio device.

```
while ~isDone(afr)
    cleanAudio = afr();
    noisyAudio = cleanAudio + noiseStd * randn(FrameLength,1);
    y = zeros(FrameLength,1); % y holds the denoised audio frame

    % Feed audio to stft one hop-length at a time
    for index = 1:numHopsPerFrame
        X = stf(noisyAudio((index-1)*HopLength+1:index*HopLength));
        % Gain estimator
        Z = abs(X).^2 / (noiseStd^2 * alpha) / stftNorm;
        Z(Z<=1) = 1;
        Z = 1 - 1./Z;
        Z = sign(Z) .* sqrt(abs(Z));
        X = X .* Z;
        % Convert back to time-domain
        y((index-1)*HopLength+1:index*HopLength) = istf(X);
    end
    % Listen to denoised audio:
    adw(y);
end
```

### Perfect Reconstruction

Perfect reconstruction is when the output of `dsp. ISTFT` matches the input to `dsp. STFT`. Perfect reconstruction is obtained if the analysis window,  $g(n)$ , obeys the constant overlap-add (COLA) property at hop-size  $R$ .

$$\sum_{m=-\infty}^{\infty} g(n - mR) = 1, \forall n \in \mathbb{Z} \quad (g \in \text{COLA}(R))$$

A signal is perfectly reconstructed if the output of the `dsp. ISTFT` object matches the input to the `dsp. STFT` object.

### iscola Function

The `iscola` function checks that the specified window and overlap satisfy the COLA constraint to ensure that the inverse short-time Fourier transform (ISTFT) results in perfect reconstruction for

non-modified spectra. The function returns a logical `true` if the combination of input parameters is COLA-compliant and a logical `false` if not. The `method` argument of the function is set to `'ola'` or `'wola'` depending on whether the inversion method uses weighted overlap-add (WOLA).

Check if `hann()` window of length 120 samples and an overlap length of 60 samples is COLA compliant.

```
winLen = 120;
overlapLen = 60;
win = hann(winLen,'periodic');
tf = iscola(win,overlapLen,'ola')
```

```
tf = logical
    1
```

### Initialization

Initialize the `dsp.STFT` and `dsp.ISTFT` System objects with this hann window that is COLA compliant. Set the FFT length to equal the window length.

```
frameLen = winLen-overlapLen;
stf = dsp.STFT('Window',win,'OverlapLength',overlapLen,'FFTLength',winLen);
istf = dsp.ISTFT('Window',win,'OverlapLength',overlapLen,'WeightedOverlapAdd',0);
```

### Reconstruct Data

Compute the STFT of a random signal. Set the length of the input signal to equal the hop length (window length - overlap length). Since the window is COLA compliant, the ISTFT of this non-modified spectra perfectly reconstructs the original time-domain signal.

To confirm, compare the input,  $x$  to the reconstructed output,  $y$ . Due to the latency introduced by the objects, the reconstructed output is shifted in time compared to the input. Therefore, to compare, take the norm of the difference between the reconstructed output,  $y$  and the previous input,  $x_{prev}$ . The norm is very small, indicating that the output signal is a perfectly reconstructed version of the input signal.

```
n = zeros(1,100);
xprev = 0;
for i = 1:100
    x = randn(frameLen,1);
    X = stf(x);
    y = istf(X);
    n(1,i) = norm(y-xprev);
    xprev = x;
end
max(abs(n))
```

```
ans = 1.8077e-13
```

### ISTFT with Weighted Overlap-Add (WOLA)

In WOLA, a second window called the synthesis window,  $f(n)$ , is applied after the IFFT operation and before overlap-add. The synthesis and analysis windows are typically identical and are usually obtained by taking the square root of windows satisfying COLA (thereby ensuring perfect reconstruction).

## iscola Function

Check if `sqrt(hann())` window of length 120 samples and an overlap length of 60 samples is WOLA compliant. Set the method argument of the `iscola` function to 'wola'. The output of the `iscola` function is 1 indicating that this window is WOLA compliant.

```
winWOLA = sqrt(hann(winLen,'periodic'));
tfWOLA = iscola(winWOLA,overlapLen,'wola')

tfWOLA = logical
         1
```

## Reconstruct Data with WOLA

Release the `dsp.STFT` and `dsp.ISTFT` System objects and set the window to `sqrt(hann(winLen,'periodic'))` window. To use weighted overlap-add on the ISTFT side, set the 'WeightedOverlapAdd' to true.

```
release(stf);
release(istf);
stf.Window = winWOLA;
istf.Window = winWOLA;
istf.WeightedOverlapAdd = true;

n = zeros(1,100);
xprev = 0;
for i = 1:100
    x = randn(frameLen,1);
    X = stf(x);
    y = istf(X);
    n(1,i) = norm(y-xprev);
    xprev = x;
end
max(abs(n))

ans = 3.7930e-15
```

The norm of the difference between the input signal and the reconstructed signal is very small indicating that the signal has been reconstructed perfectly.

## More About

### Inverse Short-Time Fourier Transform (ISTFT)

The inverse short-time Fourier transform of a discrete frequency-domain signal is computed by taking the IFFT of the input frequency subbands, overlap-adding the inverted signals, and normalizing the output to reconstruct the data.

The object accepts frames of Fourier-transformed data. These segments are converted into the time-domain using the IFFT operation. The inverted segments are overlapped so that the artifacts at the boundary are reduced. To reconstruct the data, the overlapped signals are added and normalized by a factor that is a ratio of the hop length and `sum(window)`.

The ISTFT is given by

$$\begin{aligned}
 y(n) &= \frac{1}{2\pi} \int_{-\pi}^{\pi} \sum_{m=-\infty}^{\infty} Y_m(\omega) e^{j\omega n} d\omega \\
 &= \sum_{m=-\infty}^{\infty} \frac{1}{2\pi} \int_{-\pi}^{\pi} Y_m(\omega) e^{j\omega n} d\omega \\
 &= \sum_{m=-\infty}^{\infty} y_m(n)
 \end{aligned}$$

where,

- $y(n)$  -- Reconstructed signal at time  $n$ .
- $Y_m(\omega)$  -- Frequency-domain input.

### ISTFT with Weighted Overlap-Add (WOLA)

In WOLA, a second window (usually called the synthesis window) is applied after the IFFT operation and before overlap-add. WOLA is used to suppress discontinuities at frame boundaries caused by nonlinear processing of the STFT.

The analysis window (on the STFT side) and the synthesis window (on the ISTFT side) are typically identical, and are usually obtained by taking the square root of windows satisfying the constant overlap-add (COLA) property, thereby ensuring perfect reconstruction. For details on the COLA property, see the More About on page 4-1494 section in dsp . STFT page.

The inverse FFT of frequency-domain input  $Y_m(\omega)$  produces the output  $y_m(n)$ , where  $n = 0$  to  $N - 1$ ,

and is given by  $y_m(n) = \frac{1}{2\pi} \int_{-\pi}^{\pi} Y_m(\omega) e^{j\omega n} d\omega$ .

The synthesis window,  $f(n)$ , applied to  $y_m(n)$  yields the weighted output frame:

$$y_m^f(n) = y_m(n) \cdot f(n), \quad \forall n = 0, \dots, N - 1.$$

Translate the  $m^{\text{th}}$  output frame to time  $mR$ :

$$y_m^f(n) = y_m^f(n - mR).$$

Add the translated signal to the accumulated output signal,  $y(n)$ :

$$y(n) = y(n - 1) + y_m^f(n).$$

To obtain perfect reconstruction in the absence of spectral modifications, then the following condition must be true:

$$\begin{aligned}
 y(n) &= \sum_{m=-\infty}^{\infty} x(n)g(n - mR)f(n - mR) \\
 &= x(n) \sum_{m=-\infty}^{\infty} g(n - mR)f(n - mR), \\
 &= x(n),
 \end{aligned}$$



which is true if and only if

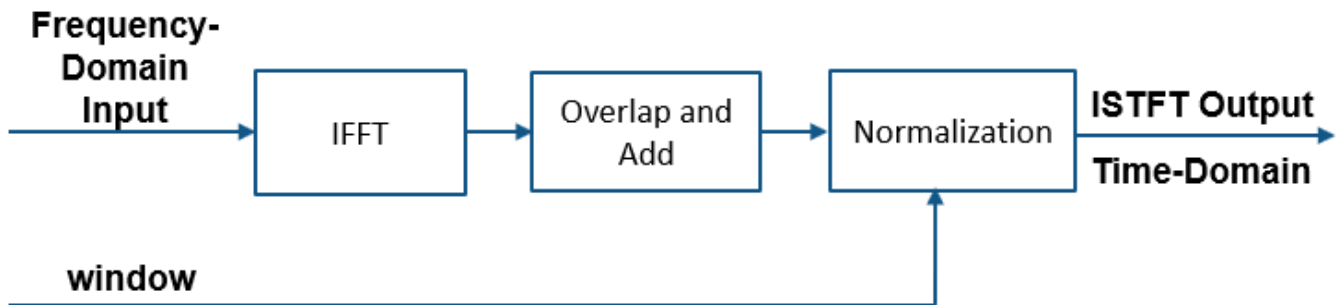
$$\sum_m g(n - mR)f(n - mR) = 1, \quad \forall n \in \mathbb{Z}.$$

where,

- $g(n)$  -- Analysis window on the STFT side.
- $f(n)$  -- Synthesis window on the ISTFT side.

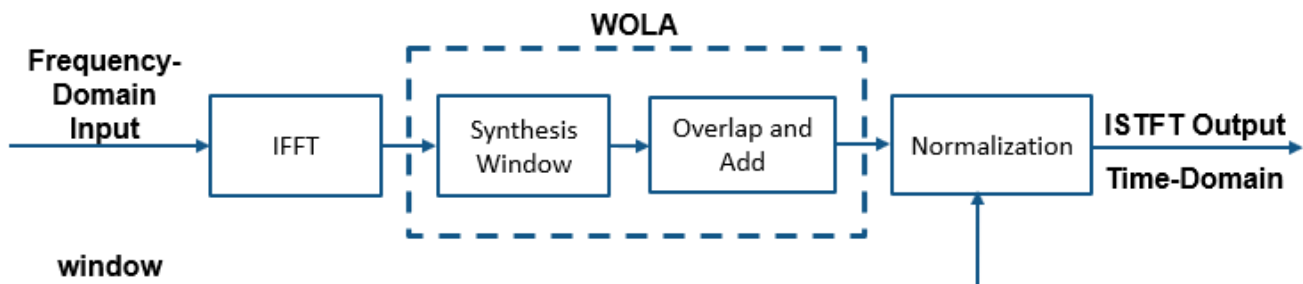
## Algorithms

Here is a sketch of how the algorithm is implemented without weighted overlap-add (WOLA):



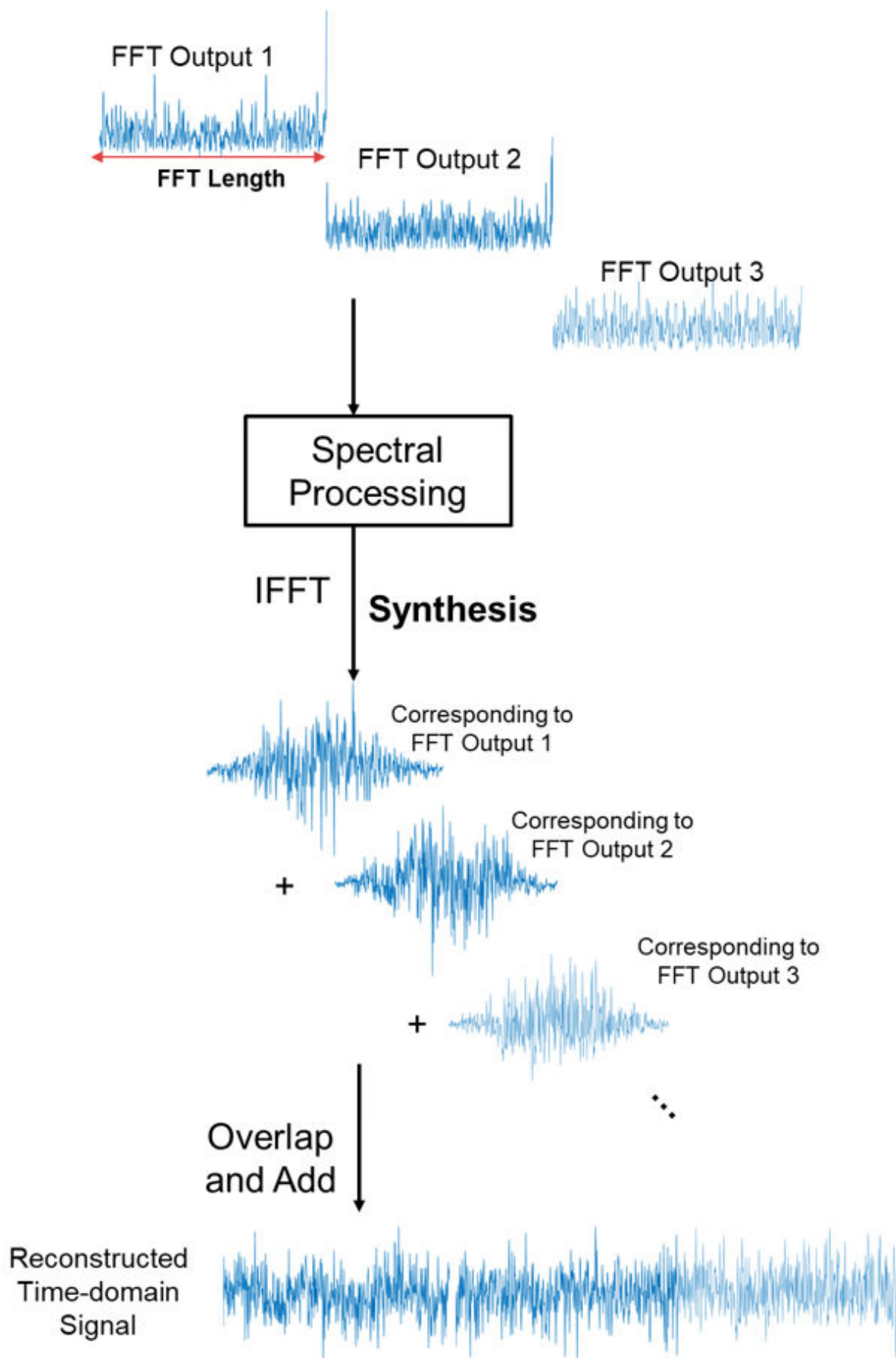
The frequency-domain input is inverted using IFFT, and then overlap-add is performed. Note that each run of the algorithm generates  $R$  new output time-domain samples, where  $R$  is the hop length. The hop length is defined as  $WL - OL$ , where  $WL$  is the window length and  $OL$  is the overlap length. The normalization stage multiplies the output by  $R/\text{sum}(win)$ , where  $win$  is the window vector specified in the Window property.

Here is a sketch of how the algorithm is implemented with Weighted Overlap-Add (WOLA):



In WOLA, a second window (usually called the synthesis window) is applied after the IFFT operation and before overlap-add. WOLA is used to suppress discontinuities at frame boundaries caused by nonlinear processing of the STFT. For more details, see More About on page 4-1481.

Here is an illustration of how the input frequency subbands look when inverted with IFFT and overlap-added together to reconstruct a time-domain signal.



The analysis window (on the STFT side) and the synthesis window (on the ISTFT side) are typically identical. To ensure perfect reconstruction, the windows are usually obtained by taking the square root of windows satisfying the constant overlap-add (COLA) property. For details on the COLA property and how perfect reconstruction is defined, see the More About on page 4-1494 in `dsp.STFT` page.

## Version History

Introduced in R2019a

## References

- [1] Allen, J.B., and L. R. Rabiner. "A Unified Approach to Short-Time Fourier Analysis and Synthesis," *Proceedings of the IEEE*, Vol. 65, pp. 1558-1564, Nov. 1977.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

When the FFT length, which is determined by the number of rows in the input signal, is not a power of two, the executable generated from this object relies on prebuilt dynamic library files (.dll files) included with MATLAB. Use the `packNGo` function to package the code generated from this object and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see "How To Run a Generated Executable Outside MATLAB".

This limitation does not apply when the FFT length is a power of two.

## See Also

### Objects

`dsp.STFT`

### Blocks

Inverse Short-Time FFT

# dsp.STFT

Short-time FFT

## Description

The `dsp.STFT` object computes the short-time Fourier transform (STFT) of the time-domain input signal. The object accepts frames of time-domain data, buffers them to the desired window length and overlap length, multiplies the samples by the window, and then performs FFT on the buffered windows. For more details, see “Algorithms” on page 4-1495.

Use the STFT to analyze the frequency content of a signal that varies with time.

## Creation

### Syntax

```
stf = dsp.STFT
stf = dsp.STFT(window)
stf = dsp.STFT(window,overlap)
stf = dsp.STFT(window,overlap,nfft)
stf = dsp.STFT(Name,Value)
```

### Description

`stf = dsp.STFT` returns an object, `stf`, that implements the short-time FFT. The object processes the data independently across each input channel over time.

`stf = dsp.STFT(window)` returns a short-time FFT object with the `Window` property set to `window`.

`stf = dsp.STFT(window,overlap)` returns a short-time FFT object with the `Window` property set to `window` and the `OverlapLength` property set to `overlap`.

`stf = dsp.STFT(window,overlap,nfft)` returns a short-time FFT object with the `Window` property set to `window`, the `OverlapLength` property set to `overlap`, and the `FFTLength` property set to `nfft`.

`stf = dsp.STFT(Name,Value)` returns a short-time FFT object with each specified property name set to the specified value. You can specify additional name-value pair arguments in any order.

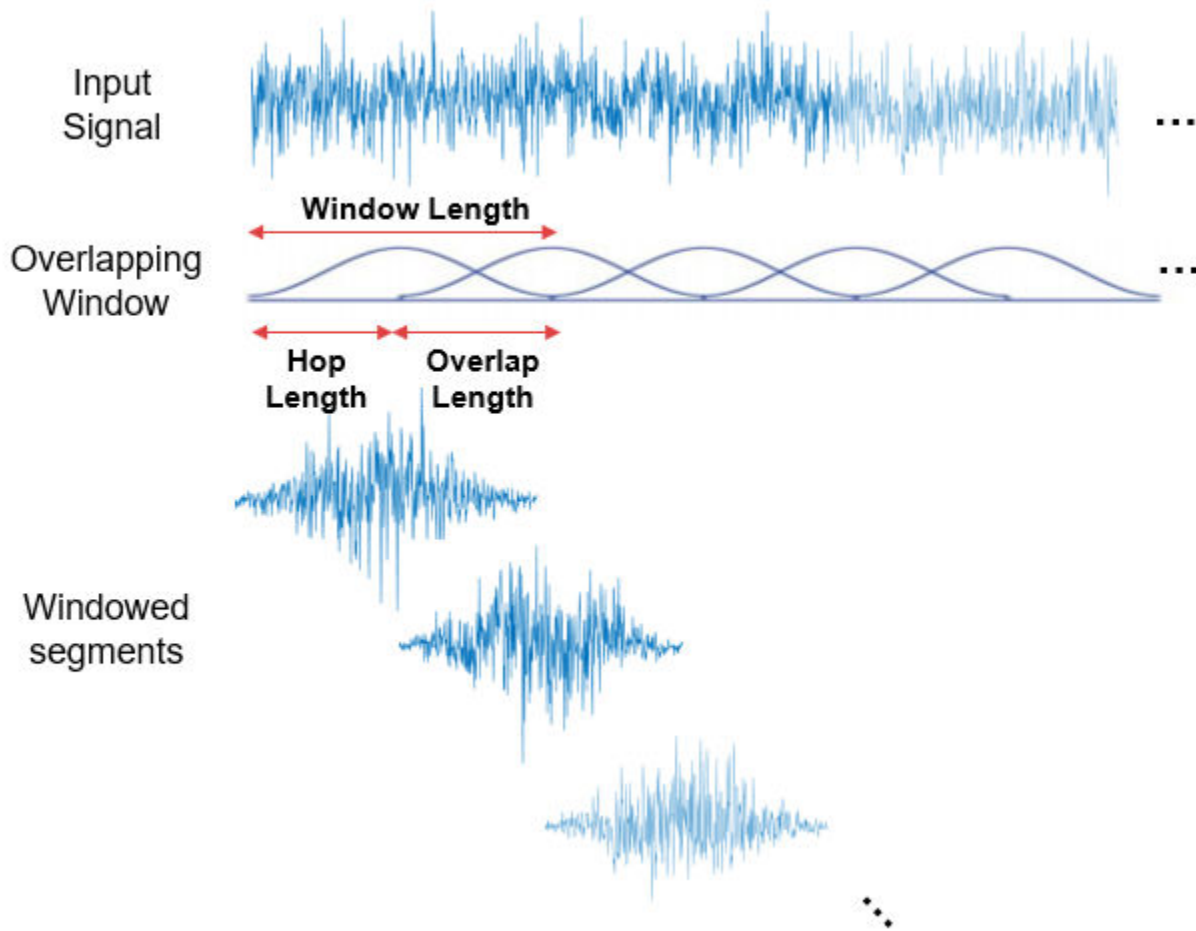
## Properties

### Window — Analysis window

`sqrt(hann(512,'periodic'))` (default) | vector

Analysis window, specified as a vector of real elements.

The object buffers the input into overlapping window segments using the specified window length and overlap length, and then multiplies each overlapped segment by the window.



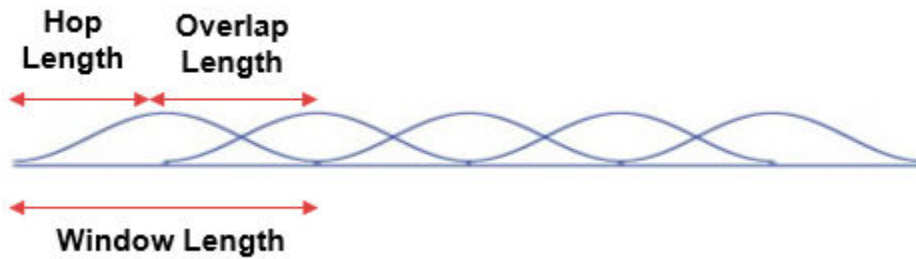
**Tunable:** Yes

Data Types: single | double

**OverlapLength — Overlap length**

256 (default) | positive integer

Number of samples by which consecutive windows overlap, specified as a positive integer. The window overlap reduces the artifacts at the data boundaries.



Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### FFTLength — FFT length

512 (default) | positive integer

FFT length, specified as a positive integer. This property determines the length of the STFT output (number of rows). The FFT length must be greater than or equal to the window length.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### FrequencyRange — Frequency range

'twosided' (default) | 'onesided'

Frequency range over which the short-time FFT is computed, specified as:

- 'twosided' --The short-time FFT is computed for complex or real inputs signals. The length of the short-time FFT is equal to the value you specify in the `FFTLength` property.
- 'onesided' -- The one-sided short-time FFT is computed for real input signals only. When the FFT length is even, the short-time FFT length is  $\text{FFTLength}/2+1$ . If FFT length is odd, the length of the short-time FFT is equal to  $(\text{FFTLength}+1)/2$ .

## Usage

### Syntax

```
y = stf(x)
```

### Description

`y = stf(x)` applies short-time FFT on the input `x` and returns the frequency-domain output `y`.

### Input Arguments

#### **x** — Input signal

vector | matrix

Time-domain input signal, specified as a vector or a matrix. If the input is a matrix, the object treats each column as an independent channel. The frame size (number of rows in `x`) must be equal to or less than the hop length (window length – overlap length).

The input can be a variable-sized signal. That is, the frame size of the signal can change in between calls to the object algorithm without calling the `release` function. The number of channels must remain the same.

If the `FrequencyRange` property is set to `'onesided'`, the input must be real. If the `FrequencyRange` property is set to `'twosided'`, the input can be real or complex.

Data Types: `single` | `double`

### Output Arguments

#### **y** — STFT output

vector | matrix

Short-time FFT output, returned as a vector or a matrix.

If there are enough samples (equal to hop length) to form an STFT output, `y` is an `FFTLength`-by-`N` matrix, where `N` is the number of input channels. If there are not enough samples to form an STFT output, `y` is empty.

The data type of the output matches that of the input signal.

Data Types: `single` | `double`

### Object Functions

|                                 |  |
|---------------------------------|--|
| <code>step</code>               | Run System object algorithm  |
| <code>release</code>            | Release resources and allow changes to System object property values and input characteristics |
| <code>reset</code>              | Reset internal states of System object   |
| <code>clone</code>              | Create duplicate System object   |
| <code>isLocked</code>           | Determine if System object is in use   |
| <code>getFrequencyVector</code> | Get the vector of frequencies at which the short-time FFT is computed                          |

### Examples

#### Short-Time Spectral Attenuation

Short-time spectral attenuation is achieved by applying a time-varying attenuation to the short-time spectrum of a noisy signal. The gain of the attenuation is determined by the estimate of the noise power in each subband of the spectrum. This gain, when applied to the noisy spectrum, attenuates the subbands with higher noise power and lifts the subbands with lesser noise power.

Here are the steps involved in performing the short-time spectral attenuation:

- 1 Analyze the noisy input signal by computing the short-time Fourier transform (STFT).
- 2 Multiply each subband of the transformed signal with a real positive gain less than 1.
- 3 Synthesize the denoised subbands by taking the inverse short-time Fourier transform (ISTFT). The reconstructed signal is the denoised input signal.

Use the `dsp.STFT` and `dsp.ISTFT` objects to compute the short-time and the inverse short-time Fourier transforms, respectively.

#### Noisy Input Signal

The input is an audio signal sampled at the 22,050 Hz. The `dsp.AudioFileReader` object reads this signal in frames of 512 samples. The audio signal is corrupted by white Gaussian noise that has a



standard deviation of 0.05. Use the `audioDeviceWriter` object to play the noisy audio signal to your computer's audio device.

```
FrameLength = 512;
afr = dsp.AudioFileReader('speech_dft.wav',...
    'SamplesPerFrame',FrameLength);
adw = audioDeviceWriter('SampleRate',afr.SampleRate);

noiseStd = 0.05;
while ~isDone(afr)
    cleanAudio = afr();
    noisyAudio = cleanAudio + noiseStd * randn(FrameLength,1);
    adw(noisyAudio);
end
reset(afr)
```

### Initialize Short-Time and Inverse Short-Time Fourier Transform Objects

Initialize the `dsp.STFT` and `dsp.ISTFT` objects. Set the window length equal to the input frame length and the hop length to 16. The overlap length is the difference between the window length and the hop length,  $OL = WL - HL$ . Set the FFT length to 1024.

```
WindowLength = FrameLength;
HopLength = 16;
numHopsPerFrame = FrameLength / 16;
FFTLength = 1024;
```

The window used to compute the STFT and ISTFT is a periodic hamming window with length 512. The `ConjugateSymmetricInput` flag of the `istft` object is set to `true`, indicating that the output of the `istft` object is a conjugate-symmetric signal.

```
win = hamming(WindowLength,'periodic');
stf = dsp.STFT(win,WindowLength-HopLength,FFTLength);
istf = dsp.ISTFT(win,WindowLength-HopLength,1,0);
```

### Gain Estimator

The next step is to define the gain estimator parameters. This gain is applied to the noisy spectrum to attenuate the subbands with higher noise power and lifter the subbands with lesser noise power.

```
dec = 16;
alpha = 15;
stftNorm = (sum(win.*win) / dec).^2;
```

### Spectral Attenuation

Feed the audio signal to `stf` one hop-length at a time. Apply the estimated gain to the transformed signal. Reconstruct the denoised version of the original speech signal by performing an inverse Fourier transform on the individual frequency bands. Play the denoised audio signal to the computer's audio device.

```
while ~isDone(afr)
    cleanAudio = afr();
    noisyAudio = cleanAudio + noiseStd * randn(FrameLength,1);
    y = zeros(FrameLength,1); % y holds the denoised audio frame

    % Feed audio to stft one hop-length at a time
    for index = 1:numHopsPerFrame
```

```

X = stf(noisyAudio((index-1)*HopLength+1:index*HopLength));
% Gain estimator
Z = abs(X).^2 / (noiseStd^2 * alpha) / stftNorm;
Z(Z<=1) = 1;
Z = 1 - 1./Z;
Z = sign(Z) .* sqrt(abs(Z));
X = X .* Z;
% Convert back to time-domain
y((index-1)*HopLength+1:index*HopLength) = istf(X);
end
% Listen to denoised audio:
adw(y);
end

```

### Perfect Reconstruction

Perfect reconstruction is when the output of `dsp.ISTFT` matches the input to `dsp.STFT`. Perfect reconstruction is obtained if the analysis window,  $g(n)$ , obeys the constant overlap-add (COLA) property at hop-size  $R$ .

$$\sum_{m=-\infty}^{\infty} g(n - mR) = 1, \quad \forall n \in \mathbb{Z} \quad (g \in \text{COLA}(R))$$

A signal is perfectly reconstructed if the output of the `dsp.ISTFT` object matches the input to the `dsp.STFT` object.

### iscola Function

The `iscola` function checks that the specified window and overlap satisfy the COLA constraint to ensure that the inverse short-time Fourier transform (ISTFT) results in perfect reconstruction for non-modified spectra. The function returns a logical `true` if the combination of input parameters is COLA-compliant and a logical `false` if not. The `method` argument of the function is set to `'ola'` or `'wola'` depending on whether the inversion method uses weighted overlap-add (WOLA).

Check if `hann()` window of length 120 samples and an overlap length of 60 samples is COLA compliant.

```

winLen = 120;
overlapLen = 60;
win = hann(winLen,'periodic');
tf = iscola(win,overlapLen,'ola')

```

```

tf = logical
     1

```

### Initialization

Initialize the `dsp.STFT` and `dsp.ISTFT` System objects with this hann window that is COLA compliant. Set the FFT length to equal the window length.

```

frameLen = winLen-overlapLen;
stf = dsp.STFT('Window',win,'OverlapLength',overlapLen,'FFTLength',winLen);
istf = dsp.ISTFT('Window',win,'OverlapLength',overlapLen,'WeightedOverlapAdd',0);

```

## Reconstruct Data

Compute the STFT of a random signal. Set the length of the input signal to equal the hop length (window length - overlap length). Since the window is COLA compliant, the ISTFT of this non-modified spectra perfectly reconstructs the original time-domain signal.

To confirm, compare the input,  $x$  to the reconstructed output,  $y$ . Due to the latency introduced by the objects, the reconstructed output is shifted in time compared to the input. Therefore, to compare, take the norm of the difference between the reconstructed output,  $y$  and the previous input,  $x_{prev}$ . The norm is very small, indicating that the output signal is a perfectly reconstructed version of the input signal.

```
n = zeros(1,100);
xprev = 0;
for i = 1:100
    x = randn(frameLen,1);
    X = stf(x);
    y = istf(X);
    n(1,i) = norm(y-xprev);
    xprev = x;
end
max(abs(n))
```

```
ans = 1.8077e-13
```

## ISTFT with Weighted Overlap-Add (WOLA)

In WOLA, a second window called the synthesis window,  $f(n)$ , is applied after the IFFT operation and before overlap-add. The synthesis and analysis windows are typically identical and are usually obtained by taking the square root of windows satisfying COLA (thereby ensuring perfect reconstruction).

### iscola Function

Check if `sqrt(hann())` window of length 120 samples and an overlap length of 60 samples is WOLA compliant. Set the method argument of the `iscola` function to `'wola'`. The output of the `iscola` function is 1 indicating that this window is WOLA compliant.

```
winWOLA = sqrt(hann(winLen,'periodic'));
tfWOLA = iscola(winWOLA,overlapLen,'wola')
```

```
tfWOLA = logical
        1
```

## Reconstruct Data with WOLA

Release the `dsp.STFT` and `dsp.ISTFT` System objects and set the window to `sqrt(hann(winLen,'periodic'))` window. To use weighted overlap-add on the ISTFT side, set the `'WeightedOverlapAdd'` to `true`.

```
release(stf);
release(istf);
stf.Window = winWOLA;
istf.Window = winWOLA;
istf.WeightedOverlapAdd = true;
```

```
n = zeros(1,100);
xprev = 0;
for i = 1:100
    x = randn(frameLen,1);
    X = stf(x);
    y = istf(X);
    n(1,i) = norm(y-xprev);
    xprev = x;
end
max(abs(n))

ans = 3.7930e-15
```

The norm of the difference between the input signal and the reconstructed signal is very small indicating that the signal has been reconstructed perfectly.

## More About

### Short-Time Fourier Transform

The short-time Fourier transform of a discrete time-domain signal is computed by taking the Fourier transform of short windowed segments of the time-domain data.

The discrete-time domain signal to be transformed is broken up into short segments. These segments usually overlap each other so that the artifacts at the boundaries are reduced. Each segment is Fourier transformed, and the complex result is added to a matrix, which records the magnitude and phase for each point in time and frequency.

The STFT is given by

$$X_m(\omega) = \sum_{n=-\infty}^{\infty} x(n)g(n-mR)e^{j\omega n}$$

where,

- $x(n)$  -- Input signal at time  $n$ .
- $g(n)$  -- Length  $M$  window function.
- $X_m(\omega)$  -- DTFT of windowed data centered about time  $mR$ .
- $R$  -- Hop size, in samples, between successive DTFTs.

If the window  $g(n)$  has the constant overlap-add (COLA) property at hop-size  $R$ , that is, if

$$\sum_{m=-\infty}^{\infty} g(n-mR) = 1, \forall n \in Z \quad (g \in COLA(R)),$$

then the sum of the successive DTFTs over time equals the DTFT of the whole signal  $X(\omega)$ :

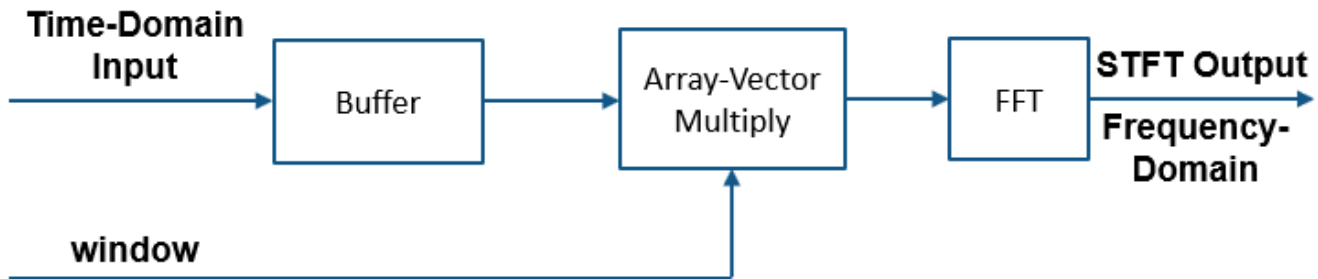
$$\begin{aligned}
\sum_{m=-\infty}^{\infty} X_m(\omega) &\triangleq \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} x(n)g(n-mR)e^{j\omega n}, \\
&= \sum_{n=-\infty}^{\infty} x(n)e^{j\omega n} \cdot \sum_{m=-\infty}^{\infty} g(n-mR), \\
&= \sum_{n=-\infty}^{\infty} x(n)e^{j\omega n} \cdot 1, \\
&\triangleq \text{DTFT}_{\omega}(x) = X(\omega).
\end{aligned}$$

Taking the inverse short-time Fourier transform of this DTFT reconstructs the original time-domain signal.

The magnitude squared of the STFT yields the spectrogram representation of the Power Spectral Density of the function.

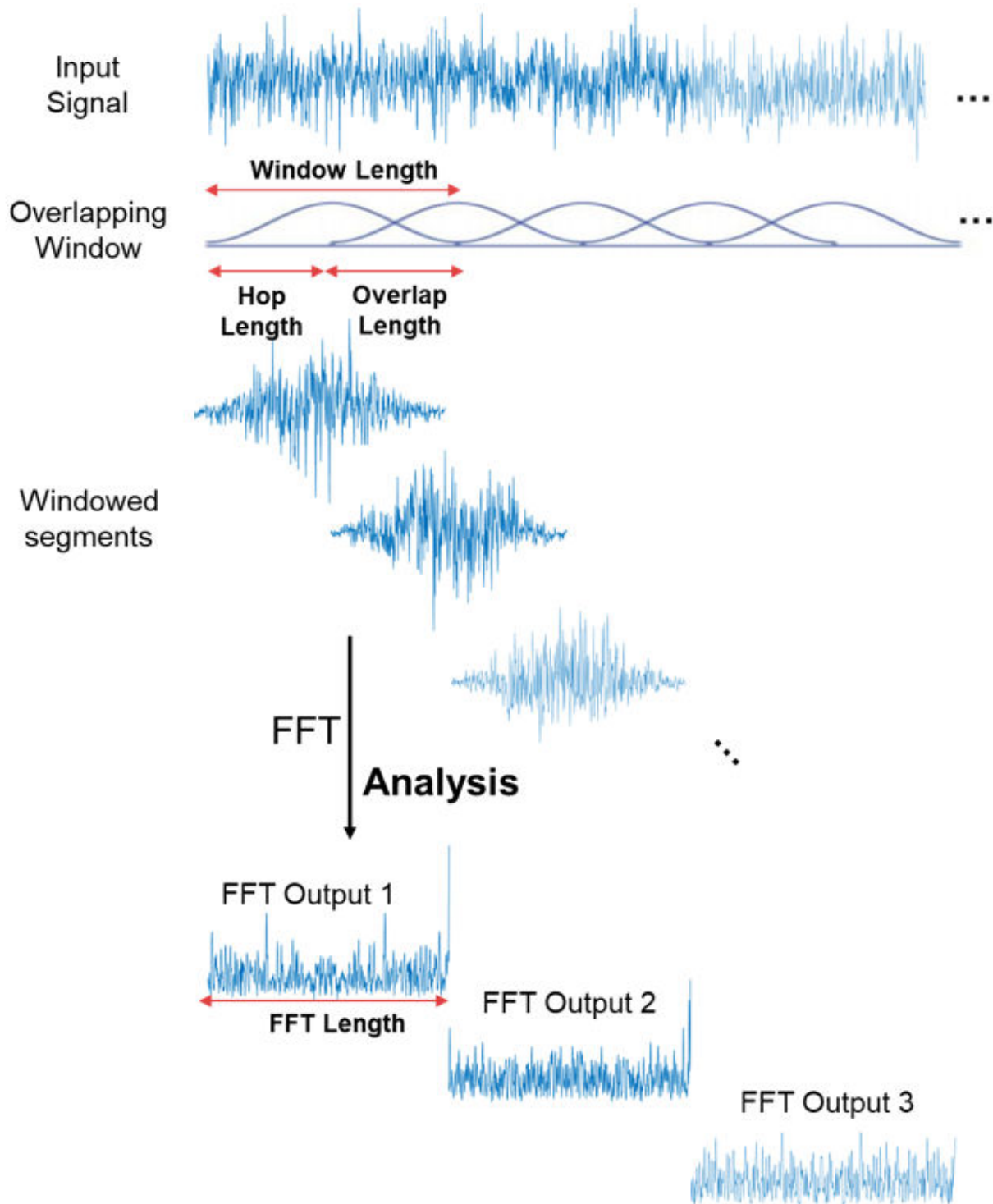
## Algorithms

Here is a sketch of how the algorithm is implemented:



The time-domain input signal is buffered based on a user-specified window length ( $WL$ ) and overlap length ( $OL$ ). The hop size,  $R$ , is defined as  $R = WL - OL$ . Buffered windows are multiplied by a user-specified window of length  $WL$ . The STFT output is the FFT of this product. The number of time-domain samples required to form a new FFT output is  $R$ .

Here is an illustration of how a random signal looks like in the original time-domain, after multiplying with the overlapping windows, and after applying FFT on the multiplied windows:



|

## Version History

Introduced in R2019a

## References

- [1] Allen, J.B., and L. R. Rabiner. "A Unified Approach to Short-Time Fourier Analysis and Synthesis," *Proceedings of the IEEE*, Vol. 65, pp. 1558-1564, Nov. 1977.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

### Objects

dsp.ISTFT

### Blocks

Short-Time FFT





# Functions

---

# addCursor

**Package:** dsp

Add cursor to Logic Analyzer

## Syntax

```
cursorTag = addCursor(scope)
cursorTag = addCursor(scope,Name,Value)
```

## Description

`cursorTag = addCursor(scope)` adds in a cursor to the display. A tag value is returned, which can be used to modify and delete the cursor.

`cursorTag = addCursor(scope,Name,Value)` sets properties using one or more name-value pairs. Enclose each property name in single quotes.

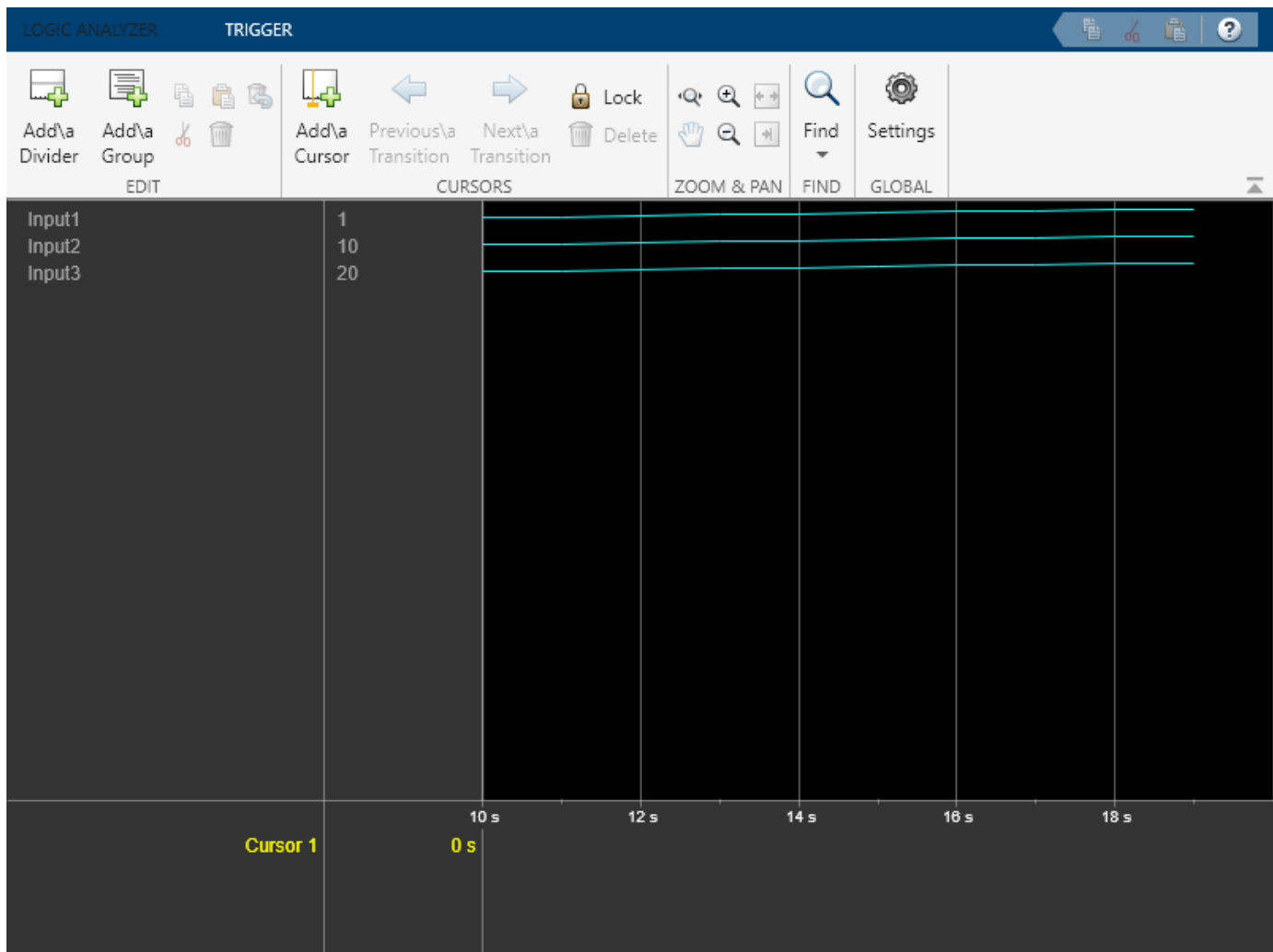
## Examples

### Modify Logic Analyzer Cursors Programmatically

This example shows how to use functions to create, manipulate, and delete cursors in a `dsp.LogicAnalyzer` object.

### Create Logic Analyzer and Signals

```
scope = dsp.LogicAnalyzer('NumInputPorts',3);
for ii = 1:20
    scope(ii,10*ii,20*ii);
end
```



### Add Cursor

Add a cursor at 15 seconds and show the cursor information.

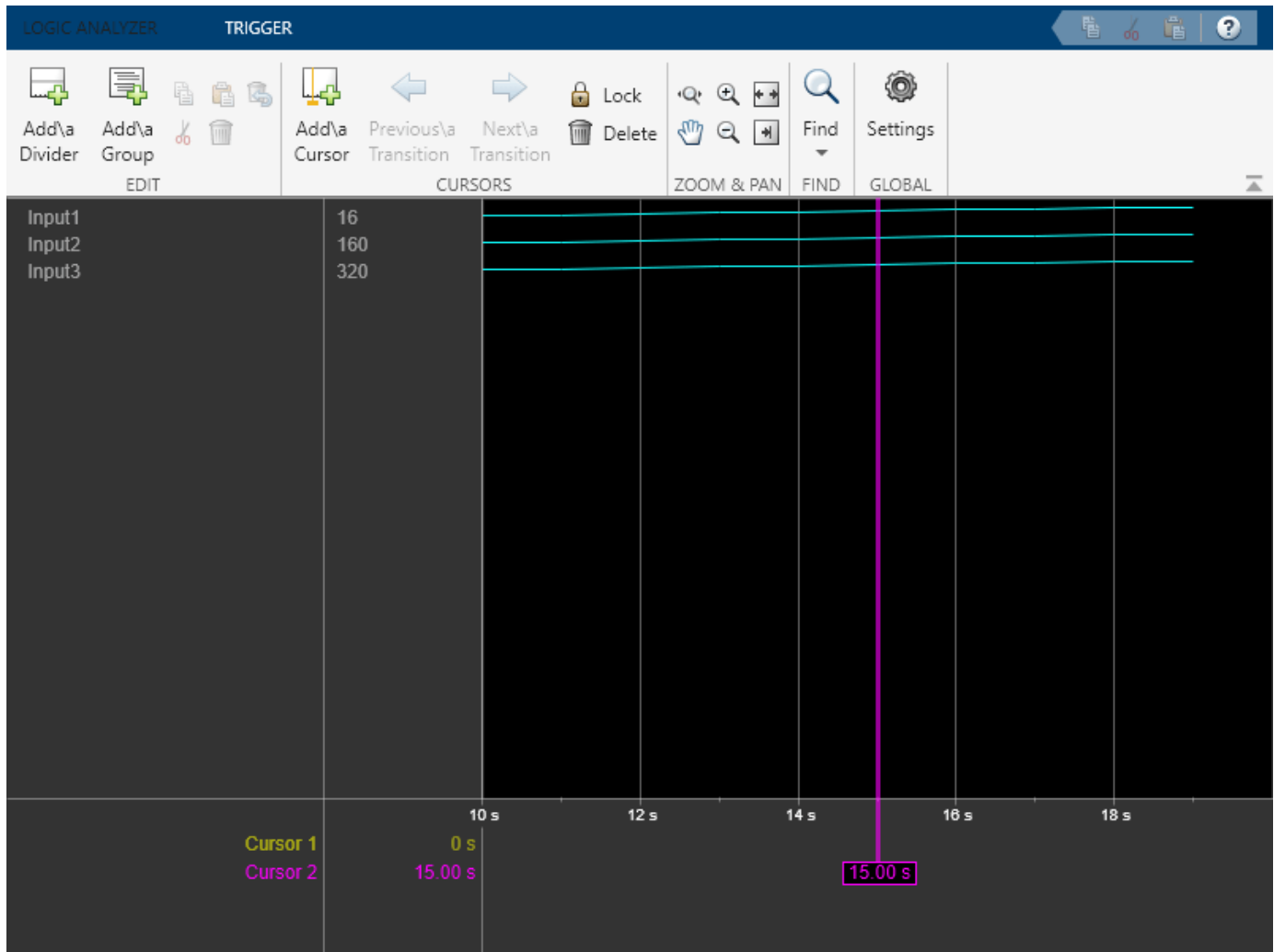
```
cursor = addCursor(scope, 'Location', 15, 'Color', 'Cyan');
getCursorInfo(scope, cursor)
```

```
ans = struct with fields:
    Location: 15
    Color: [0 1 1]
    Locked: 0
    Tag: 'C2'
```

### Modify Cursor

Change the cursor color to magenta.

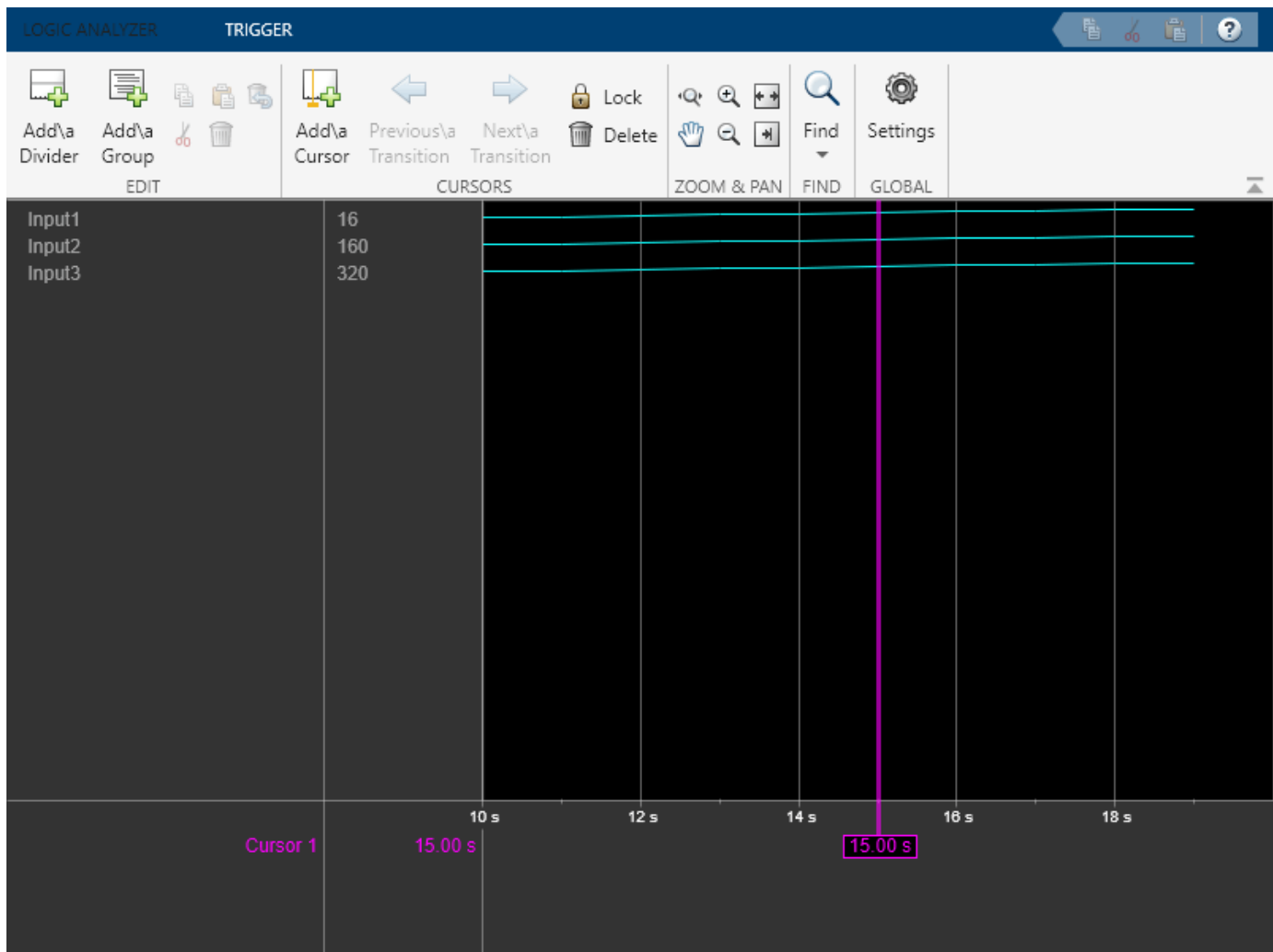
```
hide(scope)
modifyCursor(scope, cursor, 'Color', 'Magenta')
show(scope)
```



### Remove Cursor

Delete the yellow cursor at 0 seconds.

```
hide(scope)
tags = getCursorTags(scope);
deleteCursor(scope, tags{1});
show(scope)
```



## Input Arguments

**scope** – The Logic Analyzer object to which you want to add a cursor

`dsp.LogicAnalyzer` object

Example: `addCursor(scope)` adds a cursor with the default characteristics.

### Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

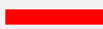






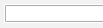
*Before R2021a, use commas to separate each name and value, and enclose `Name` in quotes.*

Example: `'Location',2,'Color','Blue'` specifies that a cursor should be moved to the 2-second mark and colored blue.

### Color – Color of the cursor

'Yellow' (default) | character vector | three element numeric vector | string scalar

Color of the cursor, specified as a [R G B] number value, color name, or color short name:

| Color Name | Short Name | RGB Triplet | Appearance  |
|------------|------------|-------------|---|
| "red"      | "r"        | [1 0 0]     |  |
| "green"    | "g"        | [0 1 0]     |  |
| "blue"     | "b"        | [0 0 1]     |  |
| "cyan"     | "c"        | [0 1 1]     |  |
| "magenta"  | "m"        | [1 0 1]     |  |
| "yellow"   | "y"        | [1 1 0]     |  |
| "black"    | "k"        | [0 0 0]     |  |
| "white"    | "w"        | [1 1 1]     |  |

Example: 'Color', 'blue'

Example: 'Color', [0,0,1]

Data Types: char | string | double | single | uint8 | uint16 | uint32 | uint64 | int8 | int16 | int32 | int64

### Location — Location of the cursor

0 (default) | numeric scalar

Specify as a numeric scalar value, in seconds, the cursor location.

Example: 'Location', 1

Data Types: double

### Locked — Locked status of the cursor

false (default) | true

Locked status of the cursor, specified as false or true.

- `true` — the cursor location cannot be changed. Logic Analyzer denotes the locked cursor by assigning a default color of gray. This color cannot be changed.
- `false` — the cursor location can be changed. Logic Analyzer denotes the unlocked cursor by assigning a default color of yellow.

Example: 'Locked', true

## Version History

Introduced in R2013a

### See Also

dsp.LogicAnalyzer | deleteCursor | getCursorTags | getCursorInfo | modifyCursor

# addDivider

**Package:** dsp

Add divider to Logic Analyzer

## Syntax

```
dividerTag = addDivider(scope)
dividerTag = addDivider(scope,Name,Value)
```

## Description

`dividerTag = addDivider(scope)` adds a divider to the display. A tag value is returned, which can be used to modify and delete the divider.

`dividerTag = addDivider(scope,Name,Value)` sets properties using one or more name-value pairs. Enclose each property name in single quotes.

## Examples

### Manipulate Logic Analyzer Programatically

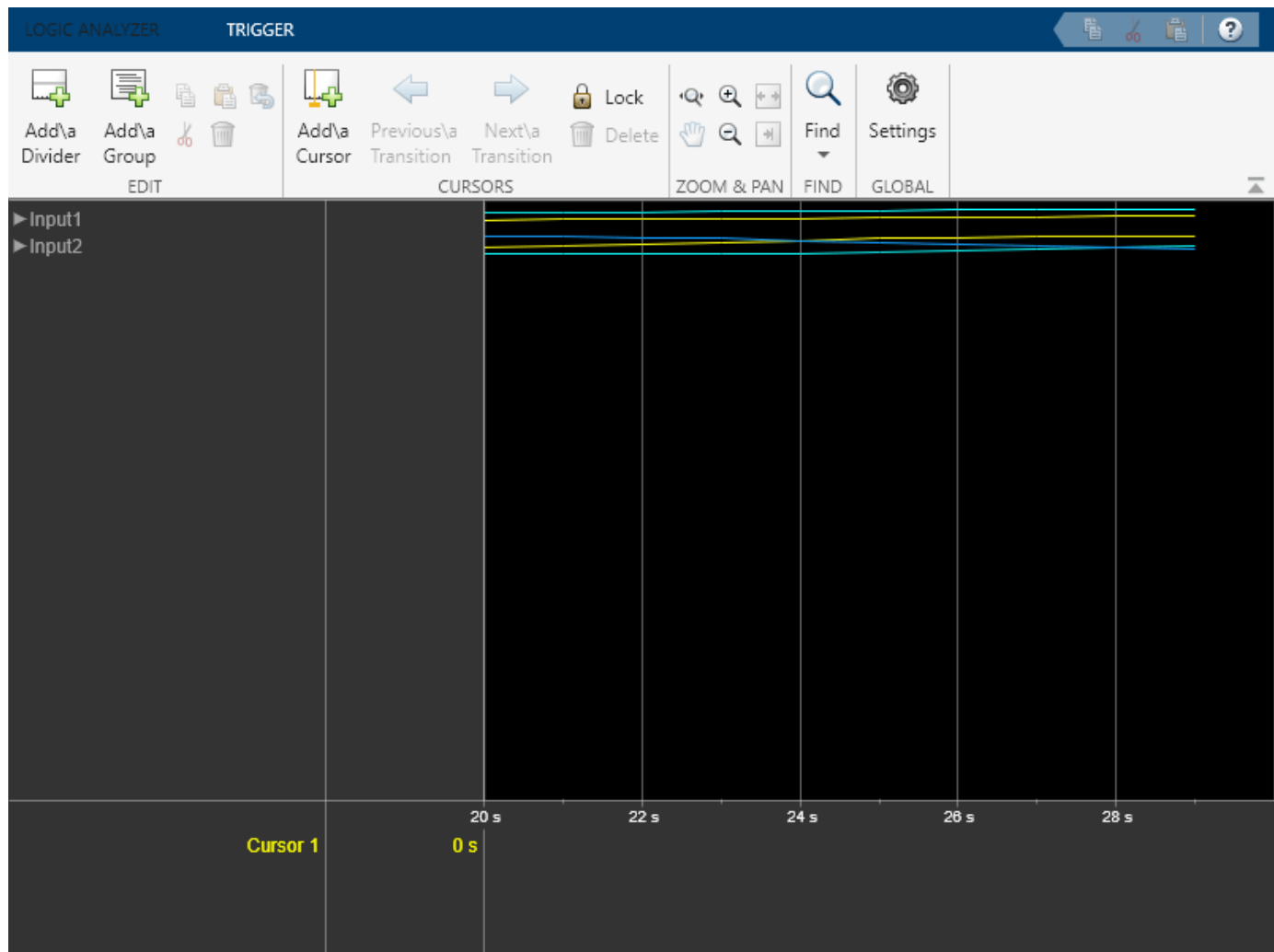
Use functions to construct and manipulate a `dsp.LogicAnalyzer` System object.

#### Display Waves on Logic Analyzer scope.

```
scope = dsp.LogicAnalyzer('NumInputPorts',2);

stop = 30;
for count = 1:stop
    sinValVec = sin(count/stop*2*pi);
    cosValVec = cos(count/stop*2*pi);
    cosValVecOffset = cos((count+10)/stop*2*pi);

    scope([count (count-(stop/2))],[sinValVec cosValVec cosValVecOffset])
end
```



### Reorganize Display

```
hide(scope)
digitalDividerTag = addDivider(scope, 'Name', 'Digital', 'Height', 20);
analogDividerTag = addDivider(scope, 'Name', 'Analog', 'Height', 40);

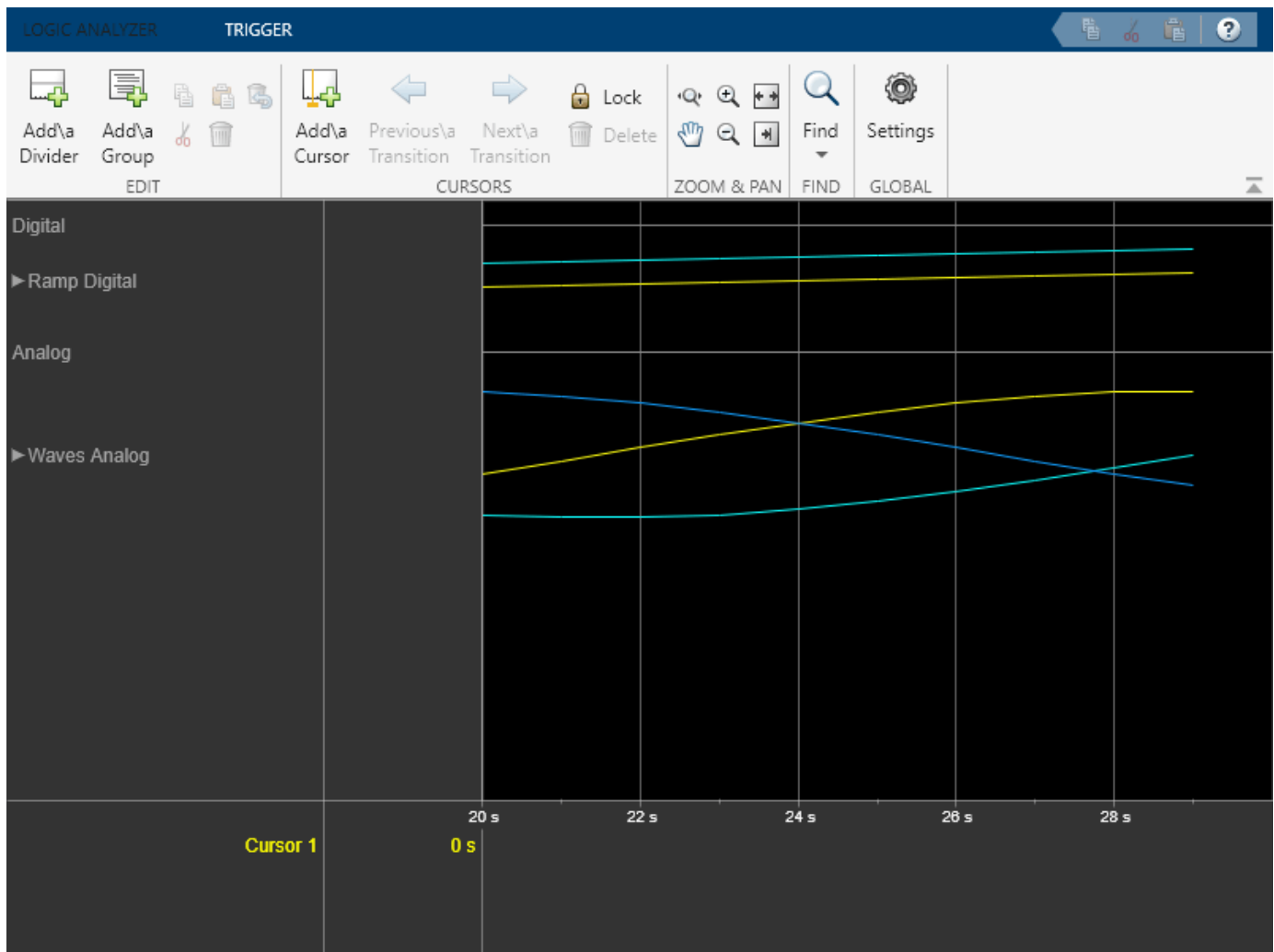
tags = getDisplayChannelTags(scope);

modifyDisplayChannel(scope, tags{1}, 'InputChannel', 1, ...
    'Name', 'Ramp Digital', 'Height', 40);
modifyDisplayChannel(scope, tags{2}, 'InputChannel', 2, ...
    'Name', 'Waves Analog', 'Format', 'Analog', 'Height', 80);

moveDisplayChannel(scope, digitalDividerTag, 'DisplayChannel', 1)
moveDisplayChannel(scope, tags{2}, 'DisplayChannel', length(tags))

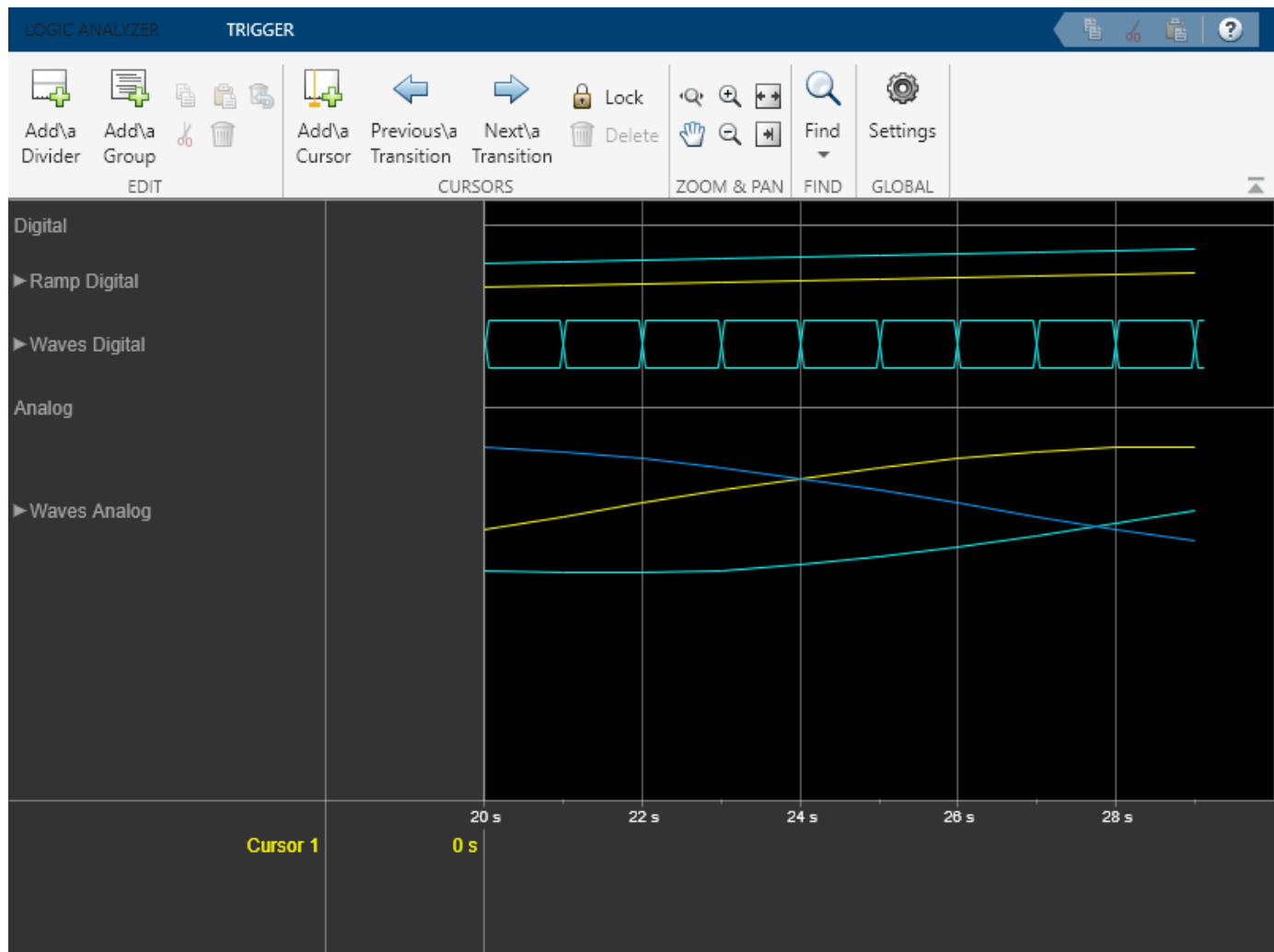
show(scope)
```





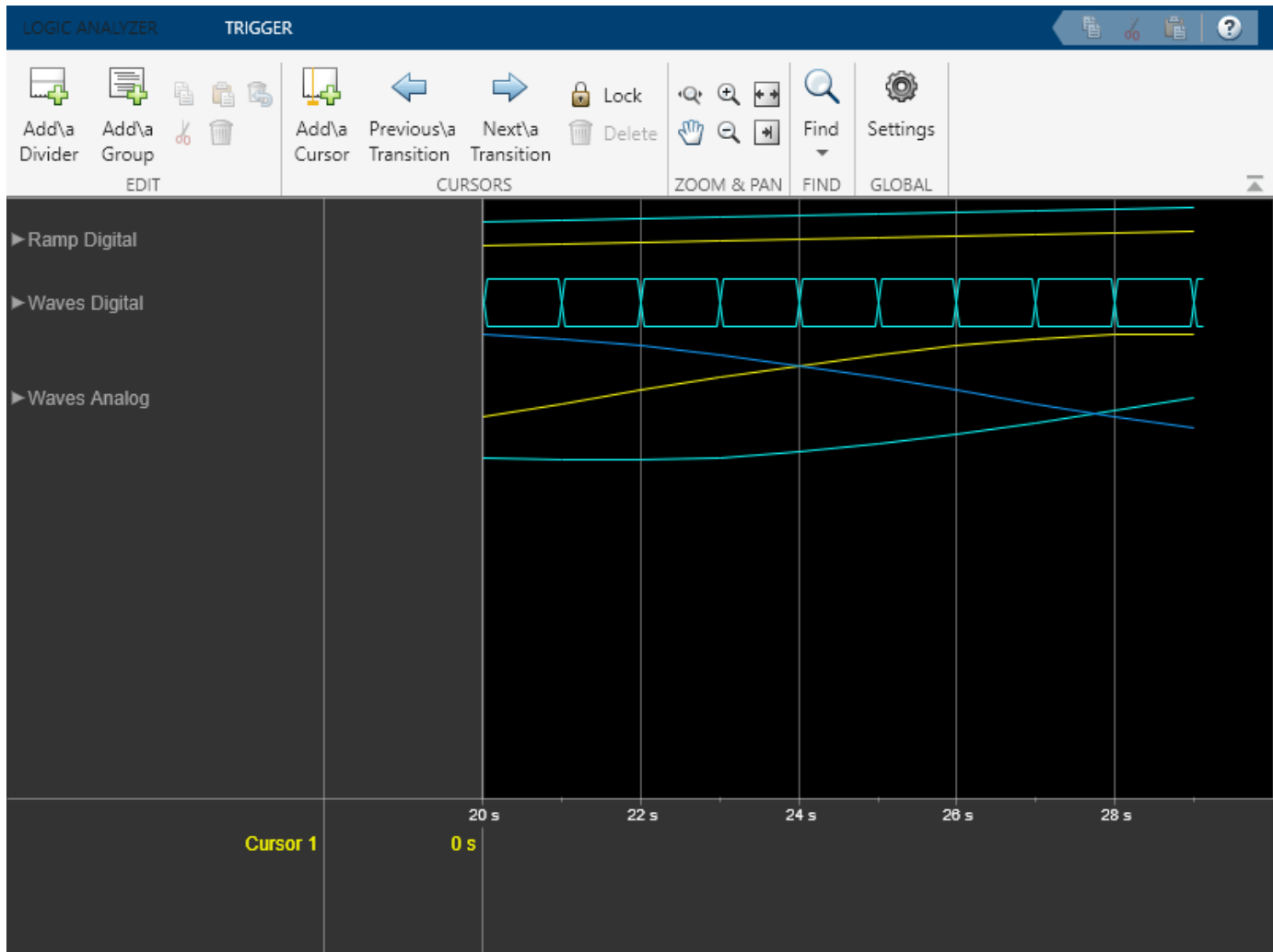
### Duplicate Wave and Check Information

```
hide(scope)
addWave(scope, 'InputChannel', 2, 'Name', 'Waves Digital', 'Format', 'Digital', ...
'Height', 30, 'DisplayChannel', 3);
show(scope)
```



### Remove Dividers

```
hide(scope)
deleteDisplayChannel(scope,digitalDividerTag)
deleteDisplayChannel(scope,analogDividerTag)
show(scope)
```



### Clear variables

```
clear analogDividerTag cosValVec cosValVecOffset count digitalDividerTag duplicateWave scope sin
```

## Input Arguments

### scope — Logic Analyzer object

dsp.LogicAnalyzer object

dsp.LogicAnalyzer object to which you want to add a divider.

Example: `addDivider(scope)` adds a divider with the default characteristics.

### Divider Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: 'DisplayChannel',2,'Name','MyDivider' specifies that a divider should be added to display channel 2 and named "MyDivider".

**DisplayChannel — Channel on the display that shows this divider**

NumInputPorts (default) | scalar numeric value in the range (1,NumInputPorts)

Specify as a scalar numeric value the display channel that shows this divider. By default, the divider is added to the end of the display.

Example: 'DisplayChannel',2

Data Types: double | single | uint8 | uint16 | uint32 | uint64 | int8 | int16 | int32 | int64

**Height — Height of the divider**

0 (default) | scalar integer

Specify, in pixels, the height of the divider as a scalar integer in the range 8-200. If you choose 0, the value of the DisplayChannelHeight property in the Logic Analyzer is used.

Example: 'Height',2

Data Types: double

**Name — The name or label for the divider**

' ' (default) | character vector | string scalar

Specify the name that you would like to set for the new divider.

Example: 'Name','MyDivider'

Data Types: char | string

**Output Arguments****dividerTag — tag for new divider**

random character vector

A tag for the newly added divider. Use the tag name to modify and delete the divider.

**Version History**

Introduced in R2013a

**See Also**

dsp.LogicAnalyzer | addCursor | addWave

# addWave

**Package:** dsp

Add wave to Logic Analyzer

## Syntax

```
waveTag = addWave(scope)
waveTag = addWave(scope,Name,Value)
```

## Description

`waveTag = addWave(scope)` adds a wave to the display. A tag value is returned, which can be used to modify and delete the wave.

`waveTag = addWave(scope,Name,Value)` sets properties using one or more name-value pairs. Enclose each property name in single quotes.

## Examples

### Manipulate Logic Analyzer Programatically

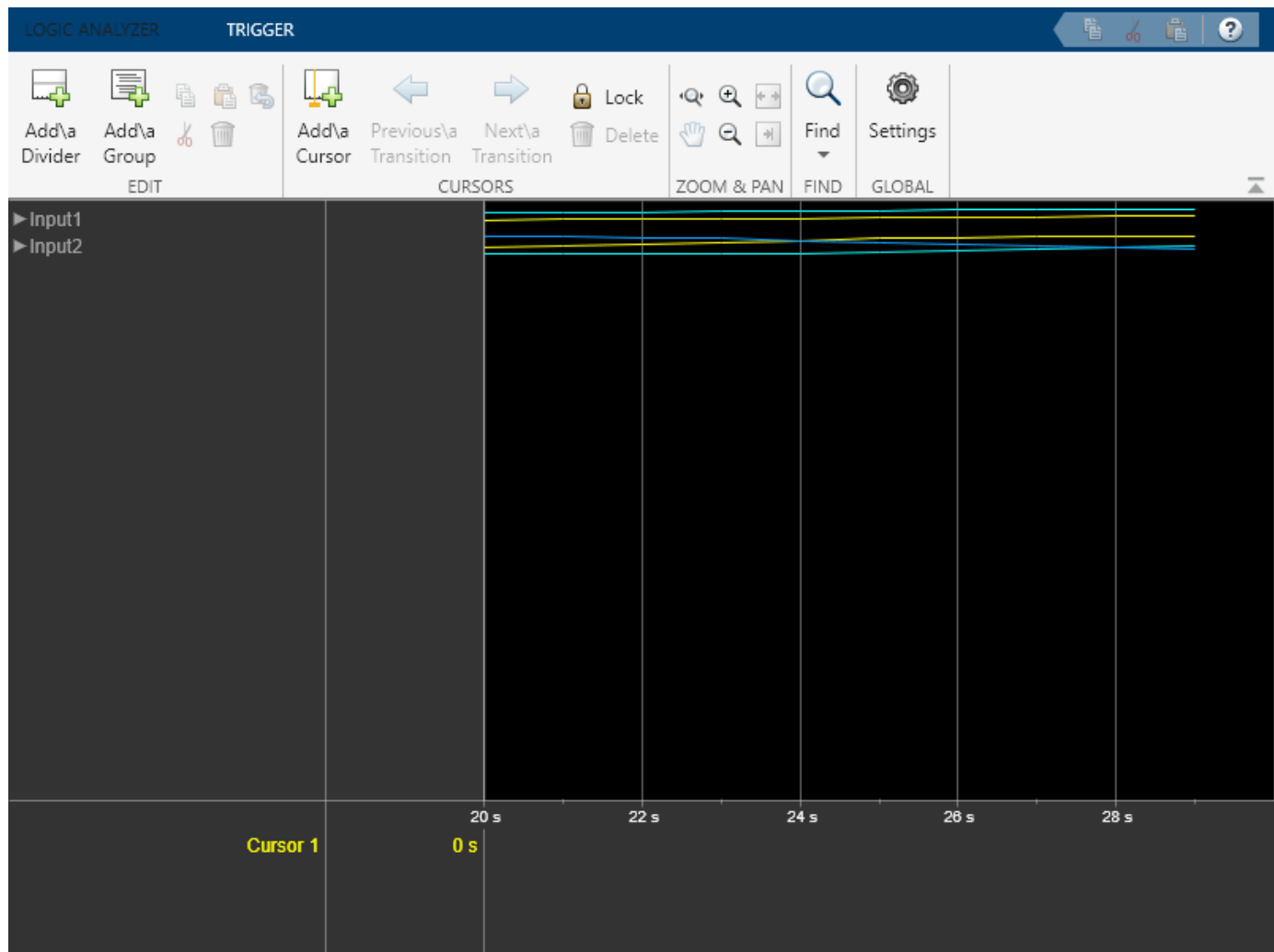
Use functions to construct and manipulate a `dsp.LogicAnalyzer` System object.

### Display Waves on Logic Analyzer scope.

```
scope = dsp.LogicAnalyzer('NumInputPorts',2);

stop = 30;
for count = 1:stop
    sinValVec = sin(count/stop*2*pi);
    cosValVec = cos(count/stop*2*pi);
    cosValVecOffset = cos((count+10)/stop*2*pi);

    scope([count (count-(stop/2))],[sinValVec cosValVec cosValVecOffset])
end
```



### Reorganize Display

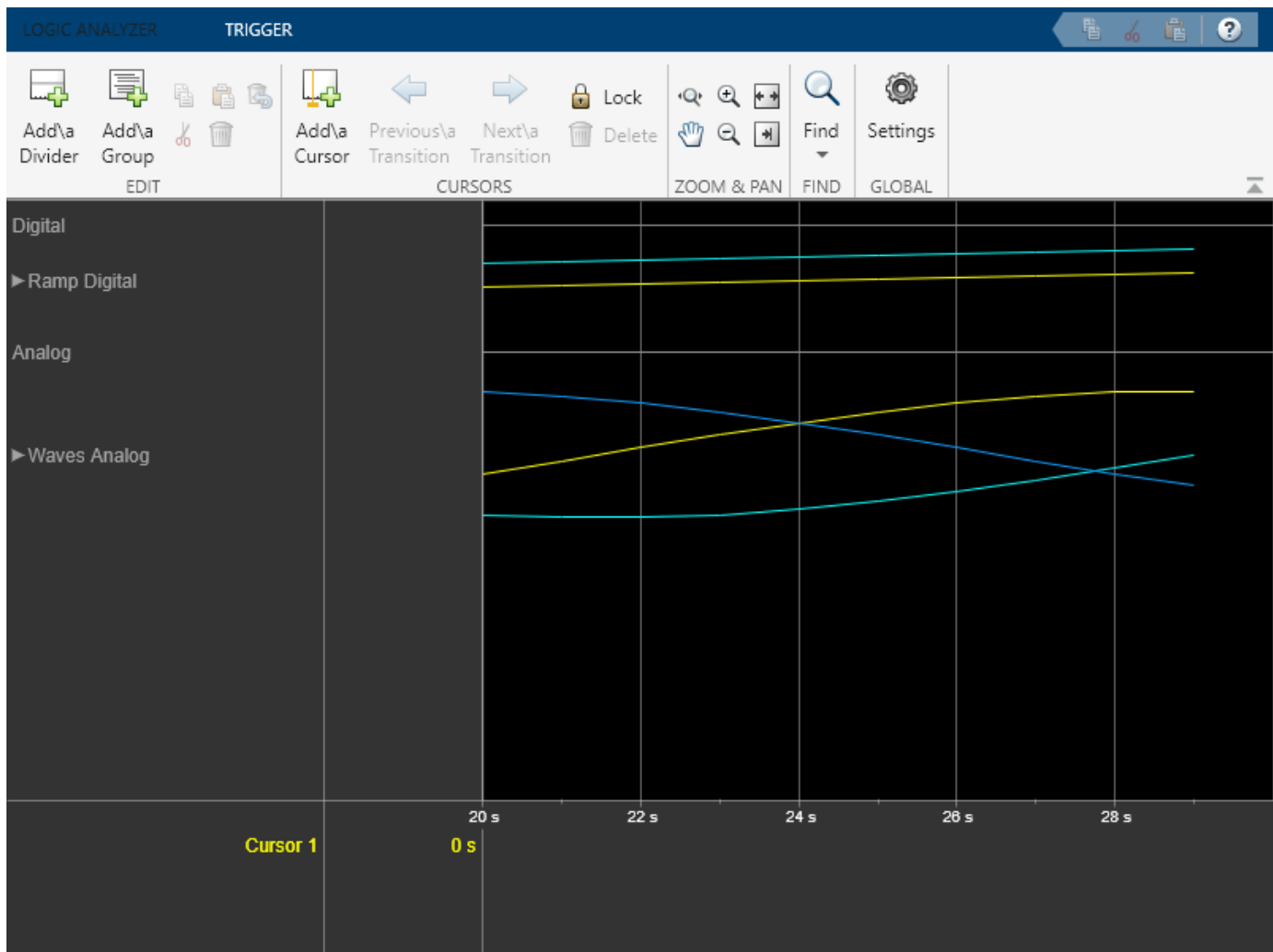
```
hide(scope)
digitalDividerTag = addDivider(scope, 'Name', 'Digital', 'Height', 20);
analogDividerTag = addDivider(scope, 'Name', 'Analog', 'Height', 40);

tags = getDisplayChannelTags(scope);

modifyDisplayChannel(scope, tags{1}, 'InputChannel', 1, ...
    'Name', 'Ramp Digital', 'Height', 40);
modifyDisplayChannel(scope, tags{2}, 'InputChannel', 2, ...
    'Name', 'Waves Analog', 'Format', 'Analog', 'Height', 80);

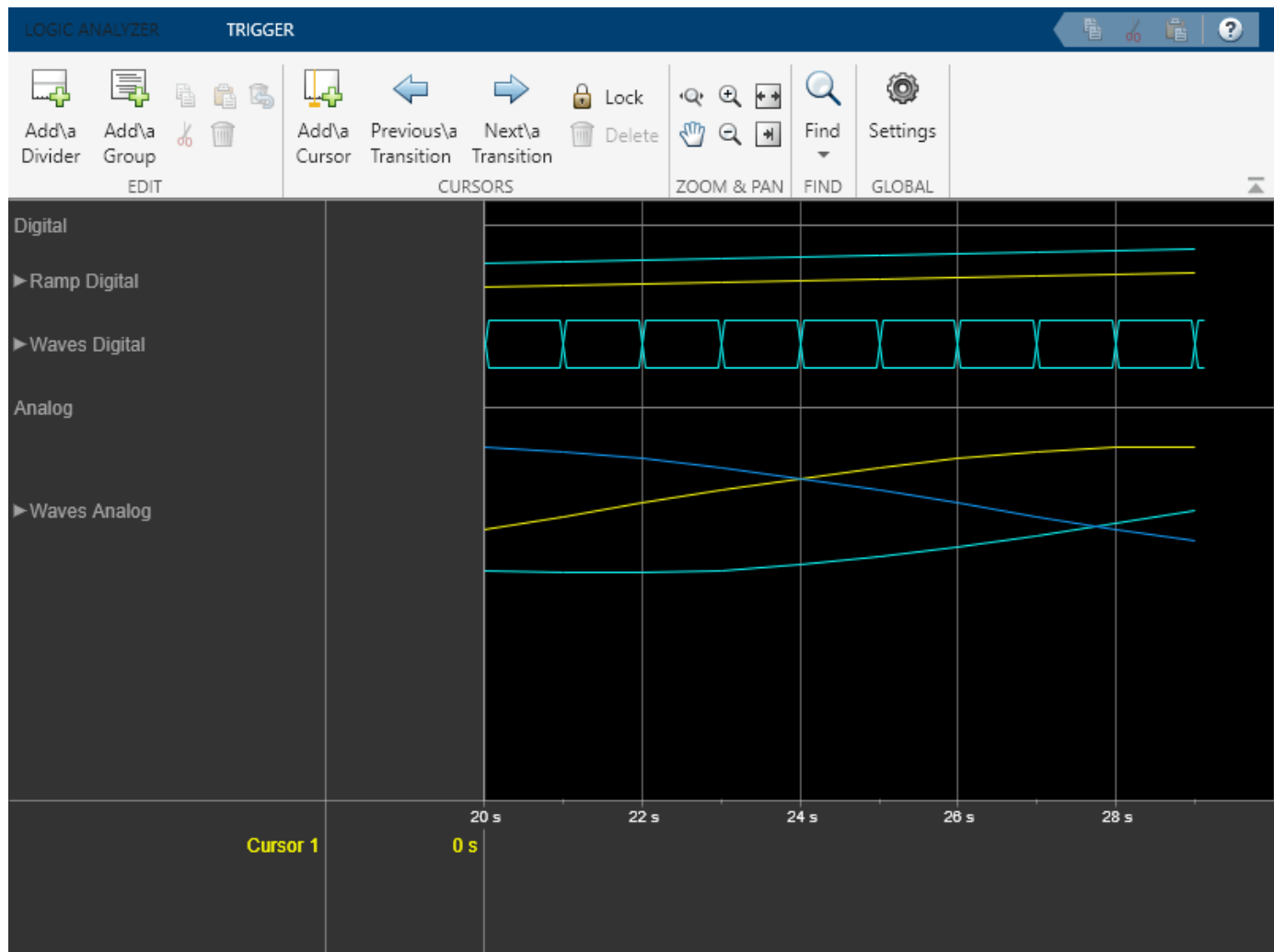
moveDisplayChannel(scope, digitalDividerTag, 'DisplayChannel', 1)
moveDisplayChannel(scope, tags{2}, 'DisplayChannel', length(tags))

show(scope)
```



### Duplicate Wave and Check Information

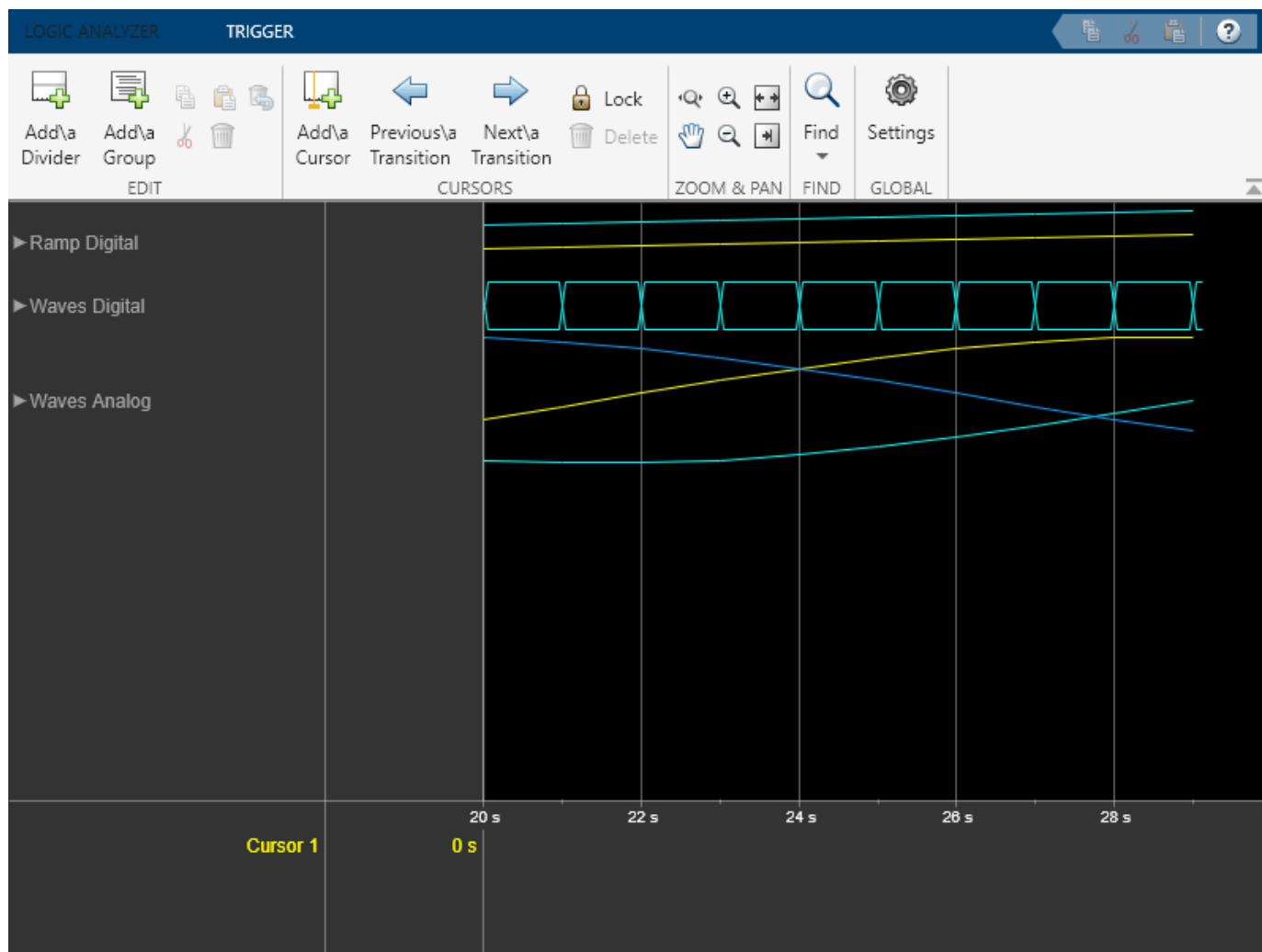
```
hide(scope)
addWave(scope, 'InputChannel', 2, 'Name', 'Waves Digital', 'Format', 'Digital', ...
'Height', 30, 'DisplayChannel', 3);
show(scope)
```



### Remove Dividers

```
hide(scope)
deleteDisplayChannel(scope,digitalDividerTag)
deleteDisplayChannel(scope,analogDividerTag)
show(scope)
```





### Clear variables

```
clear analogDividerTag cosValVec cosValVecOffset count digitalDividerTag duplicateWave scope sin
```

## Input Arguments

**scope** — The Logic Analyzer object to which you want to add a wave

`dsp.LogicAnalyzer` object

Example: 'addWave(scope)' adds a wave with the default characteristics.

### Wave Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: 'InputChannel',2,'Color','Blue' specifies that a wave should be added to input channel 1 and colored blue.

**Color — Color of the wave**

'Default' (default) | character vector | three element numeric vector | string scalar

Color of the wave, specified as an [R G B] value or one of the following:

- 'Black'
- 'Blue'
- 'Cyan'
- 'Default'
- 'Green'
- 'Magenta'
- 'Red'
- 'White'
- 'Yellow'

When you choose 'Default', the value of the `DisplayChannelColor` property in the Logic Analyzer is used.

Example: 'Color', 'Blue'

Example: 'Color', [0,0,1]

Data Types: char | string | double | single | uint8 | uint16 | uint32 | uint64 | int8 | int16 | int32 | int64

**DisplayChannel — Channel on the display that shows this wave**

NumInputPorts (default) | scalar numeric value in the range (1,NumInputPorts)

Specify as a scalar numeric value the display channel that shows this wave. By default, the wave is added to the end of the display.

Example: 'DisplayChannel', 2

Data Types: double | single | uint8 | uint16 | uint32 | uint64 | int8 | int16 | int32 | int64

**FontSize — Font size for values in the wave**

0 (default) | scalar nonnegative integer

Specify as a scalar nonnegative integer the font size in points. When you choose 0, the value of the `DisplayChannelFontSize` property in the Logic Analyzer is used.

Example: 'FontSize', 8

Data Types: double

**Format — Display format for the wave**

'Default' (default) | 'Analog' | 'Digital'

When you choose 'Default', the value of the `DisplayChannelFormat` property in the Logic Analyzer is used.

Example: 'Format', 'Digital'

Data Types: char | string

**Height — Height of the wave**

0 (default) | scalar integer

Specify as a scalar integer the height of the wave in the display in units of 16 pixels. When you choose 0, the value of the `DisplayChannelHeight` property in the Logic Analyzer is used.

Example: `'Height',2`

Data Types: double

**InputChannel — Input channel that corresponds to this wave**

1 (default) | scalar integer in the range (1,NumInputPorts)

This property specifies the input channel whose data is used for this wave. By default, it will connect the first input to this wave.

Example: `'InputChannel',2`

Data Types: double | single | uint8 | uint16 | uint32 | uint64 | int8 | int16 | int32 | int64

**Name — Name or label for the wave**

' ' (default) | character vector | string scalar

Specify the name that you would like to set for the new wave.

Example: `'Name','MyWave'`

Data Types: char | string

**Radix — Radix for the wave**

'Default' (default) | 'Binary' | 'Hexadecimal' | 'Octal' | 'Signed decimal' | 'Unsigned decimal'

When the input signals are of class double, single, or logical, you should not set this property. When you choose 'Default', the value of the `DisplayChannelRadix` property in the Logic Analyzer is used.

Data Types: char | string

**Output Arguments****waveTag — tag for new wave**

random character vector

A tag for the newly added wave. Use the tag name to modify and delete the wave.

**Version History**

Introduced in R2013a

**See Also**

`dsp.LogicAnalyzer` | `addCursor` | `addDivider` | `deleteDisplayChannel` | `modifyDisplayChannel` | `moveDisplayChannel`

## allpass2wdf

Allpass to Wave Digital Filter coefficient transformation

### Syntax

```
w = allpass2wdf(a)
W = allpass2wdf(A)
```

### Description

`w = allpass2wdf(a)` accepts a vector of real allpass polynomial filter coefficients `a`, and returns the transformed coefficient `w`. `w` can be used with allpass filter objects such as `dsp.AllpassFilter`, and `dsp.CoupledAllpassFilter`, with `Structure` set to 'Wave Digital Filter'.

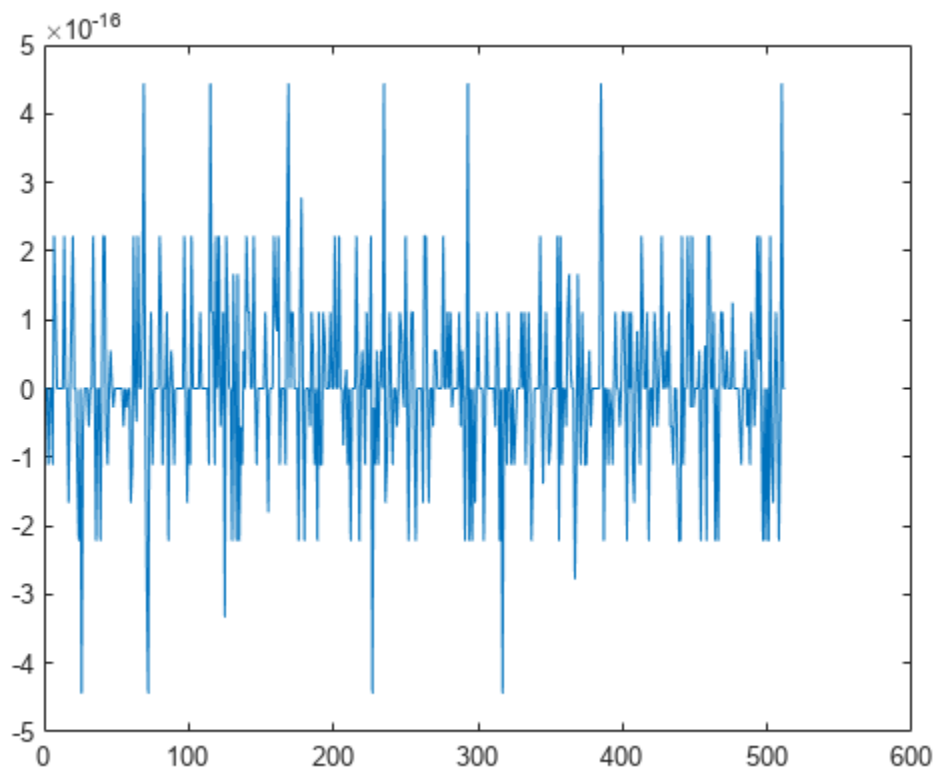
`W = allpass2wdf(A)` accepts the cell array of allpass polynomial coefficient vectors `A`. Each cell of `A` holds the coefficients of a section of a cascade allpass filter. `W` is also a cell array, and each cell of `W` contains the transformed version of the coefficients in the corresponding cell of `A`. `W` can be used with allpass filter objects such as `dsp.AllpassFilter` and `dsp.CoupledAllpassFilter`, with `structure` set to 'Wave Digital Filter'.

### Examples

#### Convert Allpass coefficients to Wave Digital Filter Coefficients

Create a second order allpass filter with coefficients `a = [0 0.5]`. Convert these coefficients into wave digital filter form using `allpass2wdf`. Assign the transformed coefficients to an allpass filter using the wave digital filter structure. Pass a random input to both these filters and compare the outputs.

```
a = [0 0.5];
allpass = dsp.AllpassFilter('AllpassCoefficients', a);
w = allpass2wdf(a);
allpasswdf = dsp.AllpassFilter('Structure', 'Wave Digital Filter', ...
    'WDFCoefficients', w);
in = randn(512, 1);
outputAllpass = allpass(in);
outputAllpasswdf = allpasswdf(in);
plot(outputAllpass-outputAllpasswdf)
```



The difference between the two outputs is very small.

## Input Arguments

### **a** — allpass filter coefficients

vector of real numbers

Numeric vector of allpass filter coefficients, specified as real numbers. **a** can have length only equal to 1, 2, and 4. When the length is 4, the first and third components must both be zero. **a** can be a row or a column vector.

Example: 0.7

Data Types: double | single

### **A** — allpass filter coefficients

vector of cells

Cascade of allpass filter coefficients, specified as a cell vector. Every cell of **A** must contain a real vector of length 1, 2, or 4. When the length is 4, the first and third components must both be zero. **A** can be a row or column vector of cells.

Example: {0.7, [0.1, 0.2]}

Data Types: double | single

## Output Arguments

### **w** — transformed version of the coefficients **a**

vector of real numbers

Numeric vector of transformed coefficients, determined as a real number, to use with single-section allpass filter objects having `Structure` set to 'Wave Digital Filter'. `w` is always returned as a numeric row vector.

Example: `0.7`

Data Types: `double` | `single`

### **W** — transformed version of the coefficients cell array **A**

vector of cell

Cascade of transformed allpass filter coefficients, determined as a cell array, to use with multi-section allpass filter objects having `Structure` set to 'Wave Digital Filter'. `W` is always returned as a column of cells.

Example: `{0.7; [0.2, -0.0833]}`

Data Types: `double` | `single`

## Algorithms

In the more general case, the input coefficients `A` define a cascade or multisection allpass filter. `allpass2wdf` applies separately to each section of the same transformation used in the single-section case. In the single-section case, the numeric coefficients vector `a` contains a standard polynomial representation of an allpass filter of order 1, 2, or 4. For example, in the first order case,

$$a = [a_1]$$

represents the first order transfer function:

$$H_1(z) = \frac{z^{-1} + a_1}{1 + a_1 z^{-1}}$$

and in the second order case,

$$a = [a_1, a_2]$$

represents the second order transfer function:

$$H_2(z) = \frac{z^{-2} + a_1 z^{-1} + a_2}{1 + a_1 z^{-1} + a_2 z^{-2}}$$

.

The allpass transfer functions  $H_1$  and  $H_2$  can also have the following alternative representations, using decoupled coefficients in vector  $w_1$  or  $w_2$  respectively.

$$\tilde{H}_1(z) = \frac{z^{-1} + w_1}{1 + w_1 z^{-1}}$$

$$\tilde{H}_2(z) = \frac{z^{-2} + w_2(1 + w_1)z^{-1} + w_1}{1 + w_2(1 + w_1)z^{-1} + w_1z^{-2}}$$

For allpass coefficients,  $w$  is often used to derive adaptor multipliers for Wave Digital Filter structures, and it is required by a number of allpass based filters in DSP System Toolbox when Structure is set to 'Wave Digital Filter' (e.g. `dsp.AllpassFilter`, and `dsp.CoupledAllpassFilter`).

For a given vector of section coefficients  $a$ , `allpass2wdf` computes the corresponding vector  $w$  such that

when  $i = 1, 2$  or  $4$

$$\tilde{H}_i(z) = H_i(z)$$

This results in using the following formulas:

for order 1:

$$w_1 = a_1$$

for order 2:

$$w_1 = a_2$$

$$w_2 = \frac{a_1}{1 + a_2}$$

for order 4:

$$w_1 = a_4$$

$$w_3 = \frac{a_2}{1 + a_4}$$

$$w_2 = w_4 = 0$$

## Version History

Introduced in R2014a

## References

- [1] M. Lutovac, D. Tomic, B. Evans, *Filter Design for Signal Processing using MATLAB and Mathematica*. Prentice Hall, 2001.

## See Also

### Functions

`wdf2allpass` | `tf2ca` | `tf2latc`

### Objects

`dsp.AllpassFilter` | `dsp.CoupledAllpassFilter`

## allpassbpc2bpc

Allpass filter for complex bandpass transformation

### Syntax

```
[AllpassNum,AllpassDen] = allpassbpc2bpc(Wo,Wt)
```

### Description

[AllpassNum,AllpassDen] = allpassbpc2bpc(Wo,Wt) returns the numerator, AllpassNum, and the denominator, AllpassDen, of the first-order allpass mapping filter for performing a complex bandpass to complex bandpass frequency transformation. For more information, see “Complex Bandpass to Complex Bandpass Frequency Transformation” on page 5-26.

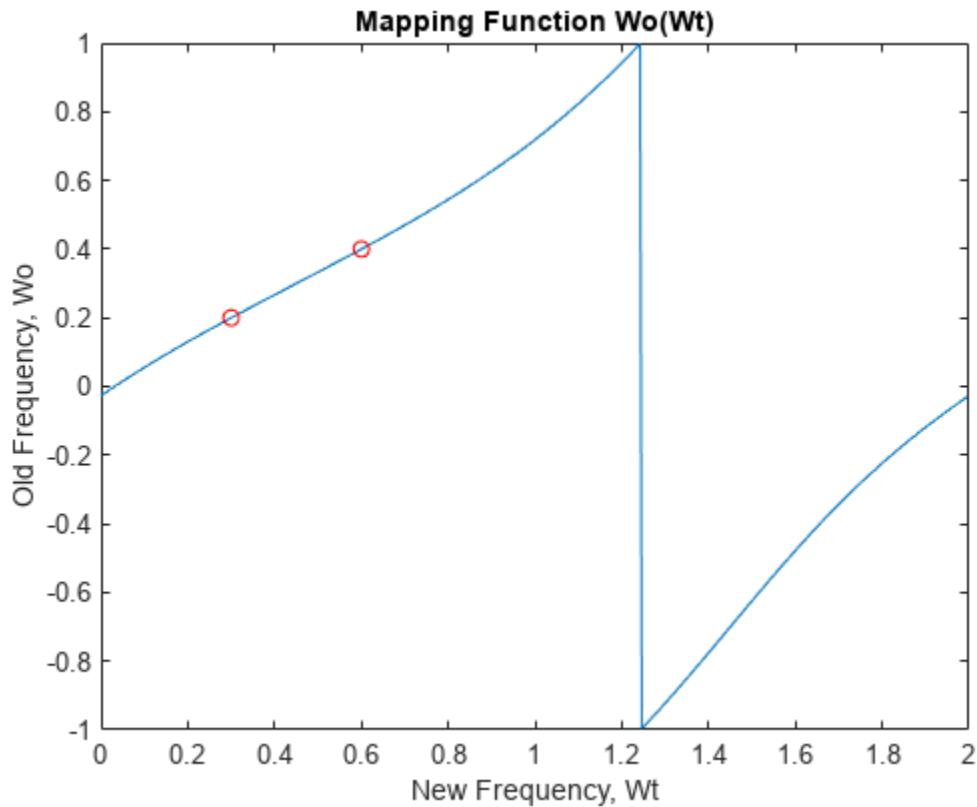
### Examples

#### Design of the allpass mapping filter

This example shows how to design allpass mapping filter, changing the complex bandpass filter with the band edges at  $W_{o1} = 0.2$  and  $W_{o2} = 0.4$  to the new band edges of  $W_{t1} = 0.3$  and  $W_{t2} = 0.6$ . Find the frequency response of the allpass mapping filter:

```
Wo = [0.2, 0.4]; Wt = [0.3, 0.6];  
[AllpassNum, AllpassDen] = allpassbpc2bpc(Wo, Wt);  
[ha, f] = freqz(AllpassNum, AllpassDen, 'whole');  
plot(f/pi, -angle(ha)/pi, Wt, Wo, 'ro')  
title('Mapping Function Wo(Wt)');  
xlabel('New Frequency, Wt'); ylabel('Old Frequency, Wo');
```





## Input Arguments

### **$W_o$ — Frequency values to be transformed from prototype filter**

real vector with values in the range  $(-1, 1)$

Frequency values to be transformed from the prototype filter, specified as a real vector with values in the range  $(-1, 1)$ .

Data Types: single | double

### **$W_t$ — Desired frequency locations in target filter**

real vector with values in the range  $(-1, 1)$

Desired frequency locations in the transformed target filter, specified as a real vector with values in the range  $(-1, 1)$ .

Data Types: single | double

## Output Arguments

### **AllpassNum — Numerator of mapping filter**

complex-valued vector

Numerator of the mapping filter, returned as a complex-valued vector.

Data Types: double  
Complex Number Support: Yes

### **AllpassDen — Denominator of mapping filter**

complex-valued vector

Denominator of the mapping filter, returned as a complex-valued vector.

Data Types: double  
Complex Number Support: Yes

## **More About**

### **Complex Bandpass to Complex Bandpass Frequency Transformation**

Complex bandpass to complex bandpass frequency transformation effectively places two features of an original filter, located at frequencies  $W_{o1}$  and  $W_{o2}$ , at the required target frequency locations  $W_{t1}$  and  $W_{t2}$ . It is assumed that  $W_{t2}$  is greater than  $W_{t1}$ . In most of the cases the features selected for the transformation are the band edges of the filter passbands. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Frequencies must be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter,  $F_1$  and  $F_2$ , with  $F_1$  preceding  $F_2$ . Feature  $F_1$  will still precede  $F_2$  after the transformation. However, the distance between  $F_1$  and  $F_2$  will not be the same before and after the transformation.

This transformation can also be used for transforming other types of filters; e.g., complex notch filters or resonators can be repositioned at two distinct desired frequencies at any place around the unit circle. This is very attractive for adaptive systems.

## **Version History**

**Introduced in R2011a**

### **See Also**

iirbpc2bpc | zpkbpc2bpc

## allpasslp2bp

Allpass filter for lowpass to bandpass transformation

### Syntax

```
[AllpassNum,AllpassDen] = allpasslp2bp(Wo,Wt)
```

### Description

[AllpassNum,AllpassDen] = allpasslp2bp(Wo,Wt) returns the numerator, AllpassNum, and the denominator, AllpassDen, of the second-order allpass mapping filter for performing a real lowpass to real bandpass frequency transformation. For more information, see “Real Lowpass to Real Bandpass Frequency Transformation” on page 5-29.

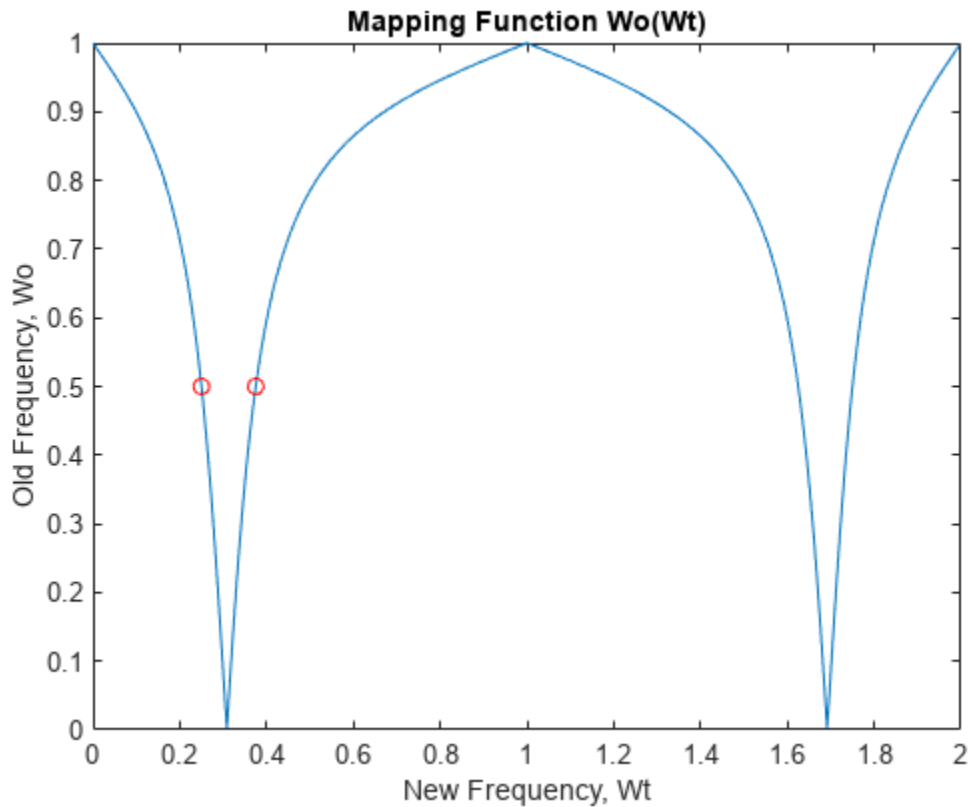
### Examples

#### Design Allpass Mapping Filter from Lowpass Filter

Design the allpass mapping filter changing the lowpass filter with cutoff frequency  $W_o$  at 0.5 to the real-valued bandpass filter with cutoff frequencies  $W_{t1}$  and  $W_{t2}$  at 0.25 and 0.375, respectively.

Compute the frequency response and plot the phase response normalized to  $\pi$ , which is in effect the mapping function  $W_o(W_t)$ . Please note that the transformation works in the same way for both positive and negative frequencies.

```
Wo = 0.5;
Wt = [0.25 0.375];
[AllpassNum, AllpassDen] = allpasslp2bp(Wo,Wt);
[h,f] = freqz(AllpassNum,AllpassDen,'whole');
plot(f/pi,abs(angle(h))/pi,Wt,Wo,'ro');
title('Mapping Function Wo(Wt)');
xlabel('New Frequency, Wt');
ylabel('Old Frequency, Wo');
```



## Input Arguments

### **Wo** — Frequency value to be transformed from prototype filter

real scalar in the range  $(0, 1)$

Frequency value to be transformed from the prototype filter, specified as a real scalar in the range  $(0, 1)$ .

Data Types: single | double

### **Wt** — Desired frequency locations in target filter

real vector with values in the range  $(0, 1)$

Desired frequency locations in the transformed target filter, specified as a real vector with values in the range  $(0, 1)$ .

Data Types: single | double

## Output Arguments

### **AllpassNum** — Numerator of mapping filter

real-valued vector

Numerator of the mapping filter, returned as a real-valued vector.

Data Types: double

### AllpassDen — Denominator of mapping filter

real-valued vector

Denominator of the mapping filter, returned as a real-valued vector.

Data Types: double

## More About

### Real Lowpass to Real Bandpass Frequency Transformation

Real lowpass to real bandpass frequency transformation effectively places one feature of an original filter, located at frequency  $-W_o$ , at the required target frequency location,  $W_{t1}$ , and the second feature, originally at  $+W_o$ , at the new location,  $W_{t2}$ . It is assumed that  $W_{t2}$  is greater than  $W_{t1}$ . This transformation implements the DC mobility, which means that the Nyquist feature stays at Nyquist, but the DC feature moves to a location dependent on the selection of  $W_t$ .

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter,  $F_1$  and  $F_2$ , with  $F_1$  preceding  $F_2$ . Feature  $F_1$  will still precede  $F_2$  after the transformation. However, the distance between  $F_1$  and  $F_2$  will not be the same before and after the transformation.

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

Choice of the feature subject to the lowpass to bandpass transformation is not restricted only to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature, for example, the stopband edge, the DC, the deep minimum in the stopband, or other features.

Lowpass to bandpass transformation can also be used for transforming other types of filters, for example, real notch filters or resonators can be doubled and repositioned at two distinct desired frequencies.

## Version History

Introduced in R2011a

## References

- [1] Nowrouzian, B., and A.G. Constantinides. "Prototype Reference Transfer Function Parameters in the Discrete-Time Frequency Transformations." In *Proceedings of the 33rd Midwest Symposium on Circuits and Systems*, 1078-82. Calgary, Alta., Canada: IEEE, 1991. <https://doi.org/10.1109/MWSCAS.1990.140912>.
- [2] Nowrouzian, B., and L.T. Bruton. "Closed-Form Solutions for Discrete-Time Elliptic Transfer Functions." In [1992] *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, 784-87. Washington, DC, USA: IEEE, 1992. <https://doi.org/10.1109/MWSCAS.1992.271206>.
- [3] Constantinides, A.G. "Spectral transformations for digital filters." *Proceedings of the IEEE*, vol. 117, no. 8: 1585-1590. August 1970.

**See Also**

`iirlp2bp` | `zpklp2bp`

## allpasslp2bpc

Allpass filter for lowpass to complex bandpass transformation

### Syntax

```
[AllpassNum,AllpassDen] = allpasslp2bpc(Wo,Wt)
```

### Description

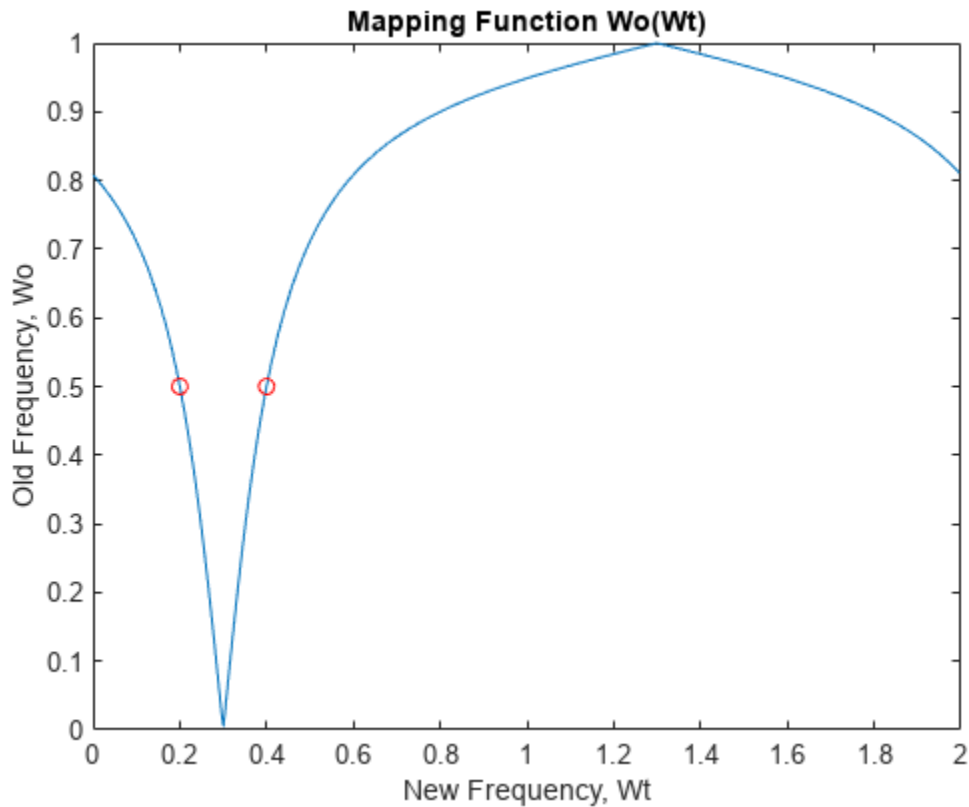
[AllpassNum,AllpassDen] = allpasslp2bpc(Wo,Wt) returns the numerator, AllpassNum, and the denominator, AllpassDen, of the first-order allpass mapping filter for performing a real lowpass to complex bandpass frequency transformation. For more information, see “Real Lowpass to Complex Bandpass Frequency Transformation” on page 5-33.

### Examples

#### Design Allpass Matching Filter

Design the allpass mapping filter changing the real lowpass filter with the cutoff frequency  $W_o$  at 0.5 into a complex bandpass filter with band edges  $W_{t1}$  and  $W_{t2}$  precisely defined at 0.2 and 0.4, respectively. Calculate the frequency response of the mapping filter in the full range.

```
Wo = 0.5;
Wt = [0.2 0.4];
[AllpassNum,AllpassDen] = allpasslp2bpc(Wo,Wt);
[h,f] = freqz(AllpassNum,AllpassDen,'whole');
plot(f/pi,abs(angle(h))/pi,Wt,Wo,'ro');
title('Mapping Function Wo(Wt)');
xlabel('New Frequency, Wt');
ylabel('Old Frequency, Wo');
```



## Input Arguments

### Wo — Frequency value to be transformed from prototype filter

real scalar in the range  $(0, 1)$

Frequency value to be transformed from the prototype filter, specified as a real scalar in the range  $(0, 1)$  with 1 corresponding to half the sample rate.

Data Types: single | double

### Wt — Desired frequency locations in target filter

real vector with values in the range  $(-1, 1)$

Desired frequency locations in the transformed target filter, specified as a real vector with values in the range  $(-1, 1)$  with 1 corresponding to half the sample rate.

Data Types: single | double

## Output Arguments

### AllpassNum — Numerator of mapping filter

complex-valued vector

Numerator of the mapping filter, returned as a complex-valued vector.



Data Types: double

### AllpassDen — Denominator of mapping filter

complex-valued vector

Denominator of the mapping filter, returned as a complex-valued vector.

Data Types: double

## More About

### Real Lowpass to Complex Bandpass Frequency Transformation

Real lowpass to complex bandpass frequency transformation effectively places one feature of the original filter, located at frequency  $-W_o$ , at the required target frequency location,  $W_{t1}$ , and the second feature, originally at  $+W_o$ , at the new location,  $W_{t2}$ . It is assumed that  $W_{t2}$  is greater than  $W_{t1}$ .

Relative positions of other features of the original filter do not change in the target filter. This means that it is possible to select two features of an original filter,  $F_1$  and  $F_2$ , with  $F_1$  preceding  $F_2$ . Feature  $F_1$  will still precede  $F_2$  after the transformation. However, the distance between  $F_1$  and  $F_2$  will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to bandpass transformation is not restricted only to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature for example, the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Lowpass to bandpass transformation can also be used for transforming other types of filters, for example, real notch filters or resonators can be doubled and positioned at two distinct desired frequencies at any place around the unit circle forming a pair of complex notches/resonators. This transformation can be used for designing bandpass filters for radio receivers from the high-quality prototype lowpass filter.

## Version History

Introduced in R2011a

### See Also

iirlp2bpc | zpklp2bpc

## allpasslp2bs

Allpass filter for lowpass to bandstop transformation

### Syntax

```
[AllpassNum,AllpassDen] = allpasslp2bs(Wo,Wt)
```

### Description

`[AllpassNum,AllpassDen] = allpasslp2bs(Wo,Wt)` returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the second-order allpass mapping filter for performing a real lowpass to real bandstop frequency transformation. This transformation effectively places one feature of an original filter, located at frequency  $-W_o$ , at the required target frequency location,  $W_{t1}$ , and the second feature, originally at  $+W_o$ , at the new location,  $W_{t2}$ . It is assumed that  $W_{t2}$  is greater than  $W_{t1}$ . This transformation implements the "Nyquist Mobility," which means that the DC feature stays at DC, but the Nyquist feature moves to a location dependent on the selection of  $W_o$  and  $W_t$ .

Relative positions of other features of an original filter change in the target filter. This means that it is possible to select two features of an original filter,  $F_1$  and  $F_2$ , with  $F_1$  preceding  $F_2$ . After the transformation feature  $F_2$  will precede  $F_1$  in the target filter. However, the distance between  $F_1$  and  $F_2$  will not be the same before and after the transformation.

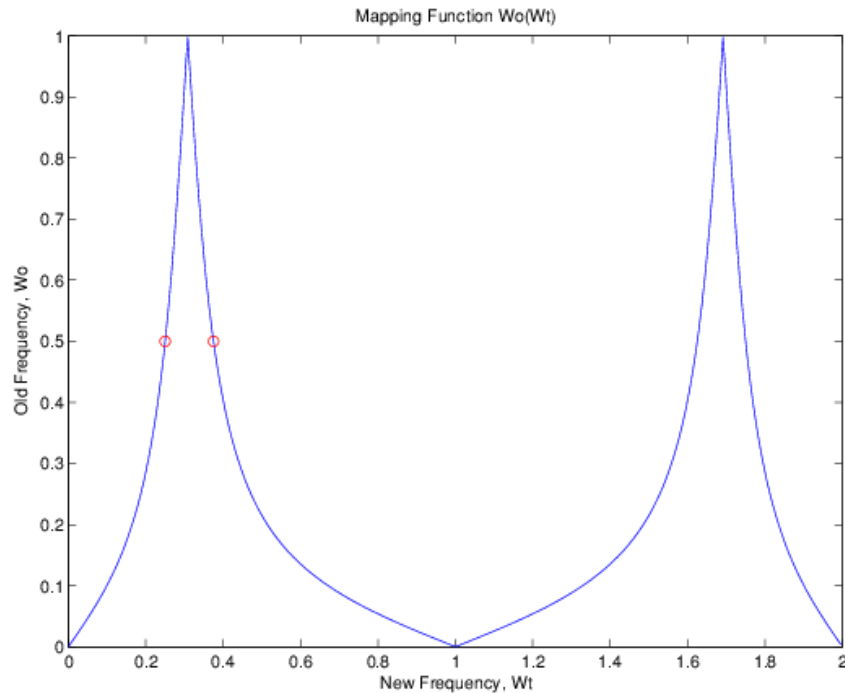
Choice of the feature subject to the lowpass to bandstop transformation is not restricted only to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

### Examples

Design the allpass filter changing the lowpass filter with cutoff frequency at  $W_o=0.5$  to the real bandstop filter with cutoff frequencies at  $W_{t1}=0.25$  and  $W_{t2}=0.375$ :

```
Wo = 0.5; Wt = [0.25, 0.375];
[AllpassNum, AllpassDen] = allpasslp2bs(Wo, Wt);
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
plot(f/pi, abs(angle(h))/pi, Wt, Wo, 'ro');
title('Mapping Function Wo(Wt)');
xlabel('New Frequency, Wt'); ylabel('Old Frequency, Wo');
```

In the figure, you find the mapping filter function as determined by the example. Note the response is normalized to  $\pi$ :



## Arguments

| Variable          | Description  |
|-------------------|--|
| $W_o$             | Frequency value to be transformed from the prototype filter  |
| $W_t$             | Desired frequency locations in the transformed target filter |
| <i>AllpassNum</i> | Numerator of the mapping filter                              |
| <i>AllpassDen</i> | Denominator of the mapping filter                            |

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

## Version History

Introduced in R2011a

## References

Constantinides, A.G., "Spectral transformations for digital filters," *IEEE Proceedings*, vol. 117, no. 8, pp. 1585-1590, August 1970.

Nowrouzian, B. and A.G. Constantinides, "Prototype reference transfer function parameters in the discrete-time frequency transformations," *Proceedings 33rd Midwest Symposium on Circuits and Systems*, Calgary, Canada, vol. 2, pp. 1078-1082, August 1990.

Nowrouzian, B. and L.T. Bruton, "Closed-form solutions for discrete-time elliptic transfer functions," *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, vol. 2, pp. 784-787, 1992.

Constantinides, A.G., "Design of bandpass digital filters," *IEEE Proceedings*, vol. 1, pp. 1129-1231, June 1969.

**See Also**

iirlp2bs | zpklp2bs

## allpasslp2bsc

Allpass filter for lowpass to complex bandstop transformation

### Syntax

```
[AllpassNum,AllpassDen] = allpasslp2bsc(Wo,Wt)
```

### Description

[AllpassNum,AllpassDen] = allpasslp2bsc(Wo,Wt) returns the numerator, AllpassNum, and the denominator, AllpassDen, of the first-order allpass mapping filter for performing a real lowpass to complex bandstop frequency transformation. This transformation effectively places one feature of an original filter, located at frequency  $-W_o$ , at the required target frequency location,  $W_{t1}$ , and the second feature, originally at  $+W_o$ , at the new location,  $W_{t2}$ . It is assumed that  $W_{t2}$  is greater than  $W_{t1}$ . Additionally the transformation swaps passbands with stopbands in the target filter.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter,  $F_1$  and  $F_2$ , with  $F_1$  preceding  $F_2$ . Feature  $F_1$  will still precede  $F_2$  after the transformation. However, the distance between  $F_1$  and  $F_2$  will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to bandstop transformation is not restricted only to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Lowpass to bandpass transformation can also be used for transforming other types of filters; e.g., real notch filters or resonators can be doubled and positioned at two distinct desired frequencies at any place around the unit circle forming a pair of complex notches/resonators. This transformation can be used for designing bandstop filters for band attenuation or frequency equalizers, from the high-quality prototype lowpass filter.

### Examples

Design the allpass filter changing the real lowpass filter with the cutoff frequency of  $W_o=0.5$  into a complex bandstop filter with band edges of  $W_{t1}=0.2$  and  $W_{t2}=0.4$  precisely defined:

```
Wo = 0.5; Wt = [0.2,0.4];
[AllpassNum, AllpassDen] = allpasslp2bsc(Wo, Wt);
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
```

### Arguments

| Variable  | Description   |
|-----------|---|
| <i>Wo</i> | Frequency value to be transformed from the prototype filter. It should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate. |

| <b>Variable</b>   | <b>Description</b>  |
|-------------------|---|
| <i>Wt</i>         | Desired frequency locations in the transformed target filter. They should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate. |
| <i>AllpassNum</i> | Numerator of the mapping filter   |
| <i>AllpassDen</i> | Denominator of the mapping filter   |

## **Version History**

**Introduced in R2011a**

### **See Also**

`iirlp2bsc` | `zpklp2bsc`

## allpasslp2hp

Allpass filter for lowpass to highpass transformation

### Syntax

```
[AllpassNum,AllpassDen] = allpasslp2hp(Wo,Wt)
```

### Description

`[AllpassNum,AllpassDen] = allpasslp2hp(Wo,Wt)` returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the first-order allpass mapping filter for performing a real lowpass to real highpass frequency transformation. This transformation effectively places one feature of an original filter, located originally at frequency,  $W_o$ , at the required target frequency location,  $W_t$ , at the same time rotating the whole frequency response by half of the sampling frequency. Result is that the DC and Nyquist features swap places.

Relative positions of other features of an original filter change in the target filter. This means that it is possible to select two features of an original filter,  $F_1$  and  $F_2$ , with  $F_1$  preceding  $F_2$ . After the transformation feature  $F_2$  will precede  $F_1$  in the target filter. However, the distance between  $F_1$  and  $F_2$  will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to highpass transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband.

Lowpass to highpass transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can change their position in a simple way by using the lowpass to highpass transformation.

### Examples

Design the allpass filter changing the lowpass filter to the highpass filter with its cutoff frequency moved from  $W_o = 0.5$  to  $W_t = 0.25$ .

Plot the phase response normalized to  $\pi$ , which is in effect the mapping function  $W_o(W_t)$ . Please note that the transformation works in the same way for both positive and negative frequencies:

```
Wo = 0.5; Wt = 0.25;
[AllpassNum, AllpassDen] = allpasslp2hp(Wo, Wt);
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
plot(f/pi, abs(angle(h))/pi, Wt, Wo, 'ro');
title('Mapping Function Wo(Wt)');
xlabel('New Frequency, Wt');
ylabel('Old Frequency, Wo');
```

## Arguments

| Variable     | Description   |
|--------------|---|
| $W_o$        | Frequency value to be transformed from the prototype filter |
| $W_t$        | Desired frequency location in the transformed target filter |
| $AllpassNum$ | Numerator of the mapping filter                             |
| $AllpassDen$ | Denominator of the mapping filter                           |

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

## Version History

Introduced in R2011a

## References

Constantinides, A.G., "Spectral transformations for digital filters," *IEE Proceedings*, vol. 117, no. 8, pp. 1585-1590, August 1970.

Nowrouzian, B. and A.G. Constantinides, "Prototype reference transfer function parameters in the discrete-time frequency transformations," *Proceedings 33rd Midwest Symposium on Circuits and Systems*, Calgary, Canada, vol. 2, pp. 1078-1082, August 1990.

Nowrouzian, B. and L.T. Bruton, "Closed-form solutions for discrete-time elliptic transfer functions," *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, vol. 2, pp. 784-787, 1992.

Constantinides, A.G., "Frequency transformations for digital filters," *Electronics Letters*, vol. 3, no. 11, pp. 487-489, November 1967.

## See Also

`iirlp2hp` | `zpklp2hp`



## allpasslp2lp

Allpass filter for lowpass to lowpass transformation

### Syntax

```
[AllpassNum,AllpassDen] = allpasslp2lp(Wo,Wt)
```

### Description

`[AllpassNum,AllpassDen] = allpasslp2lp(Wo,Wt)` returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the first-order allpass mapping filter for performing a real lowpass to real lowpass frequency transformation. This transformation effectively places one feature of an original filter, located originally at frequency  $W_o$ , at the required target frequency location,  $W_t$ .

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter,  $F_1$  and  $F_2$ , with  $F_1$  preceding  $F_2$ . Feature  $F_1$  will still precede  $F_2$  after the transformation. However, the distance between  $F_1$  and  $F_2$  will not be the same before and after the transformation.

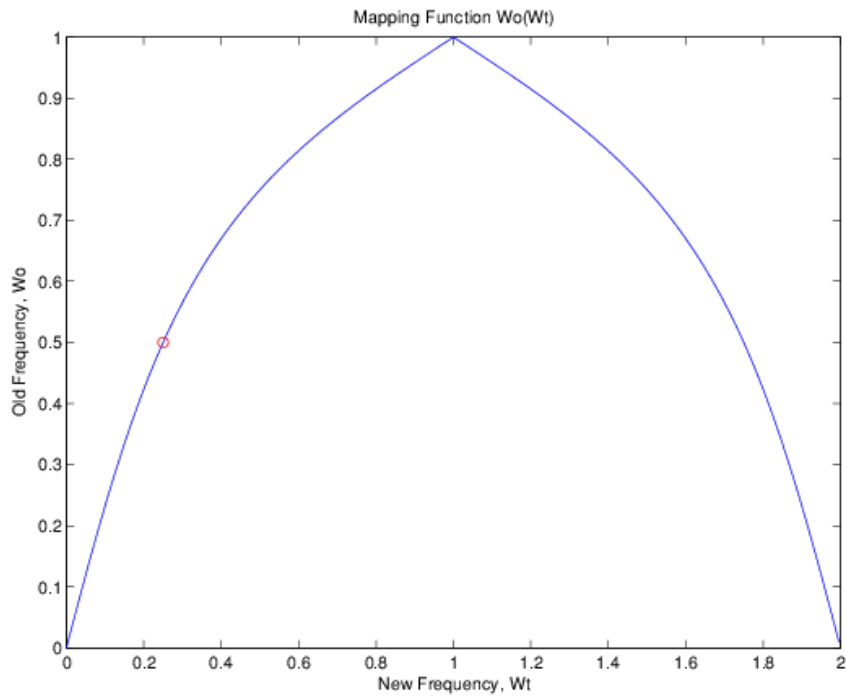
Choice of the feature subject to the lowpass to lowpass transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband and so on.

Lowpass to lowpass transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can change their position in a simple way by applying the lowpass to lowpass transformation.

### Examples

Design the allpass filter changing the lowpass filter cutoff frequency originally at  $W_o=0.5$  to  $W_t=0.25$ . Plot the phase response normalized to  $\pi$ , which is in effect the mapping function  $W_o(W_t)$ . Please note that the transformation works in the same way for both positive and negative frequencies:

```
Wo = 0.5; Wt = 0.25;
[AllpassNum, AllpassDen] = allpasslp2lp(Wo, Wt);
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
plot(f/pi, abs(angle(h))/pi, Wt, Wo, 'ro');
title('Mapping Function Wo(Wt)');
xlabel('New Frequency, Wt'); ylabel('Old Frequency, Wo');
```



As shown in the figure, `allpasslp2lp` generates a mapping function that converts your prototype lowpass filter to a target lowpass filter with different passband specifications.

## Arguments

| Variable                | Description   |
|-------------------------|---|
| $W_o$                   | Frequency value to be transformed from the prototype filter |
| $W_t$                   | Desired frequency location in the transformed target filter |
| <code>AllpassNum</code> | Numerator of the mapping filter                             |
| <code>AllpassDen</code> | Denominator of the mapping filter                           |

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

## Version History

Introduced in R2011a

## References

Constantinides, A.G., "Spectral transformations for digital filters," *IEEE Proceedings*, vol. 117, no. 8, pp. 1585-1590, August 1970.

Nowrouzian, B. and A.G. Constantinides, "Prototype reference transfer function parameters in the discrete-time frequency transformations," *Proceedings 33rd Midwest Symposium on Circuits and Systems*, Calgary, Canada, vol. 2, pp. 1078-1082, August 1990.

Nowrouzian, B. and L.T. Bruton, "Closed-form solutions for discrete-time elliptic transfer functions," *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, vol. 2, pp. 784-787, 1992.

Constantinides, A.G., "Frequency transformations for digital filters," *Electronics Letters*, vol. 3, no. 11, pp. 487-489, November 1967.

**See Also**

iirlp2lp | zpklp2lp

## allpasslp2mb

Allpass filter for lowpass to M-band transformation

### Syntax

```
[AllpassNum,AllpassDen] = allpasslp2mb(Wo,Wt)
[AllpassNum,AllpassDen] = allpasslp2mb(Wo,Wt,Pass)
```

### Description

`[AllpassNum,AllpassDen] = allpasslp2mb(Wo,Wt)` returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the  $M$ th-order allpass mapping filter for performing a real lowpass to real multipassband frequency transformation. Parameter  $M$  is the number of times an original feature is replicated in the target filter. This transformation effectively places one feature of an original filter, located at frequency  $W_o$ , at the required target frequency locations,  $W_{t1}, \dots, W_{tM}$ . By default the DC feature is kept at its original location.

`[AllpassNum,AllpassDen] = allpasslp2mb(Wo,Wt,Pass)` allows you to specify an additional parameter, `Pass`, which chooses between using the "DC Mobility" and the "Nyquist Mobility." In the first case the Nyquist feature stays at its original location and the DC feature is free to move. In the second case the DC feature is kept at an original frequency and the Nyquist feature is movable.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter,  $F_1$  and  $F_2$ , with  $F_1$  preceding  $F_2$ . Feature  $F_1$  will still precede  $F_2$  after the transformation. However, the distance between  $F_1$  and  $F_2$  will not be the same before and after the transformation.

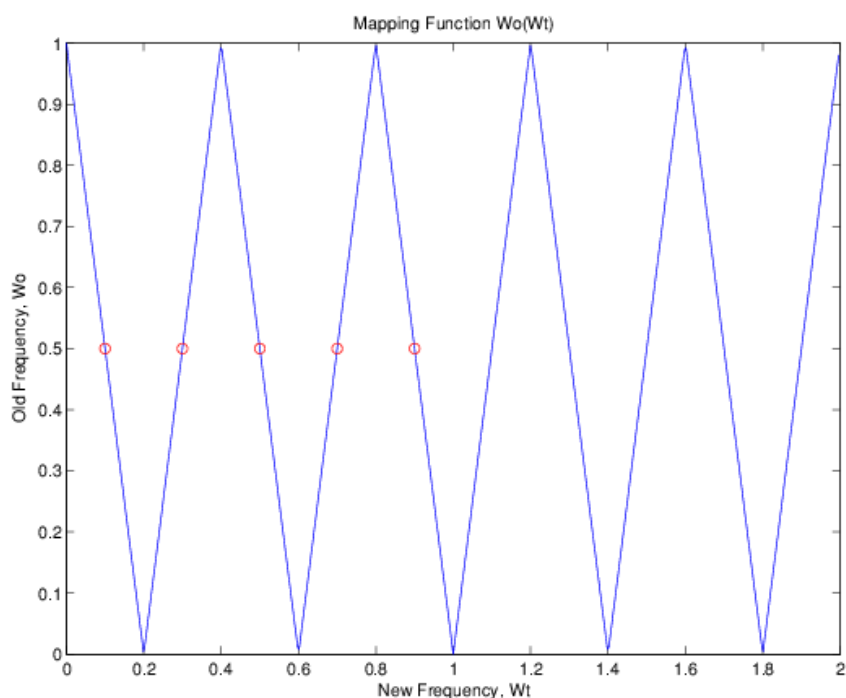
Choice of the feature subject to this transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can be easily replicated at a number of required frequency locations without redesigning them. A good application would be an adaptive tone cancellation circuit reacting to the changing number and location of tones.

### Examples

Design the allpass filter changing the real lowpass filter with the cutoff frequency of  $W_o=0.5$  into a real multiband filter with band edges of  $W_t=[1:2:9]/10$  precisely defined. Plot the phase response normalized to  $\pi$ , which is in effect the mapping function  $W_o(W_t)$ . Please note that the transformation works in the same way for both positive and negative frequencies:

```
Wo = 0.5; Wt = [1:2:9]/10;
[AllpassNum, AllpassDen] = allpasslp2mb(Wo, Wt);
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
plot(f/pi, abs(angle(h))/pi, Wt, Wo, 'ro');
title('Mapping Function Wo(Wt)');
xlabel('New Frequency, Wt'); ylabel('Old Frequency, Wo');
```



## Arguments

| Variable          | Description   |
|-------------------|---|
| $W_o$             | Frequency value to be transformed from the prototype filter                           |
| $W_t$             | Desired frequency locations in the transformed target filter                          |
| <i>Pass</i>       | Choice ( ' pass ' / ' stop ' ) of passband/stopband at DC, ' pass ' being the default |
| <i>AllpassNum</i> | Numerator of the mapping filter   |
| <i>AllpassDen</i> | Denominator of the mapping filter   |

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

## Version History

Introduced in R2011a

## References

Franchitti, J.C., "All-pass filter interpolation and frequency transformation problems," *MSc Thesis*, Dept. of Electrical and Computer Engineering, University of Colorado, 1985.

Feyh, G., J.C. Franchitti and C.T. Mullis, "All-pass filter interpolation and frequency transformation problem," *Proceedings 20th Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, California, pp. 164-168, November 1986.

Mullis, C.T. and R.A. Roberts, *Digital Signal Processing*, section 6.7, Reading, Massachusetts, Addison-Wesley, 1987.

Feyh, G., W.B. Jones and C.T. Mullis, *An extension of the Schur Algorithm for frequency transformations*, *Linear Circuits, Systems and Signal Processing: Theory and Application*, C. J. Byrnes et al Eds, Amsterdam: Elsevier, 1988.

**See Also**

iirlp2mb | zpklp2mb

# allpasslp2mbc

Allpass filter for lowpass to complex M-band transformation

## Syntax

```
[AllpassNum,AllpassDen] = allpasslp2mbc(Wo,Wt)
```

## Description

`[AllpassNum,AllpassDen] = allpasslp2mbc(Wo,Wt)` returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the  $M$ th-order allpass mapping filter for performing a real lowpass to complex multipassband frequency transformation. Parameter  $M$  is the number of times an original feature is replicated in the target filter. This transformation effectively places one feature of an original filter, located at frequency  $W_o$ , at the required target frequency locations,  $W_{t1}, \dots, W_{tM}$ .

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter,  $F_1$  and  $F_2$ , with  $F_1$  preceding  $F_2$ . Feature  $F_1$  will still precede  $F_2$  after the transformation. However, the distance between  $F_1$  and  $F_2$  will not be the same before and after the transformation.

Choice of the feature subject to this transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can be easily replicated at a number of required frequency locations without the need to design them again. A good application would be an adaptive tone cancellation circuit reacting to the changing number and location of tones.

## Examples

Design the allpass filter changing the real lowpass filter with the cutoff frequency of  $W_o=0.5$  into a complex multiband filter with band edges of  $W_t=[-3+1:2:9]/10$  precisely defined:

```
Wo = 0.5; Wt = [-3+1:2:9]/10;
[AllpassNum, AllpassDen] = allpasslp2mbc(Wo, Wt);
```

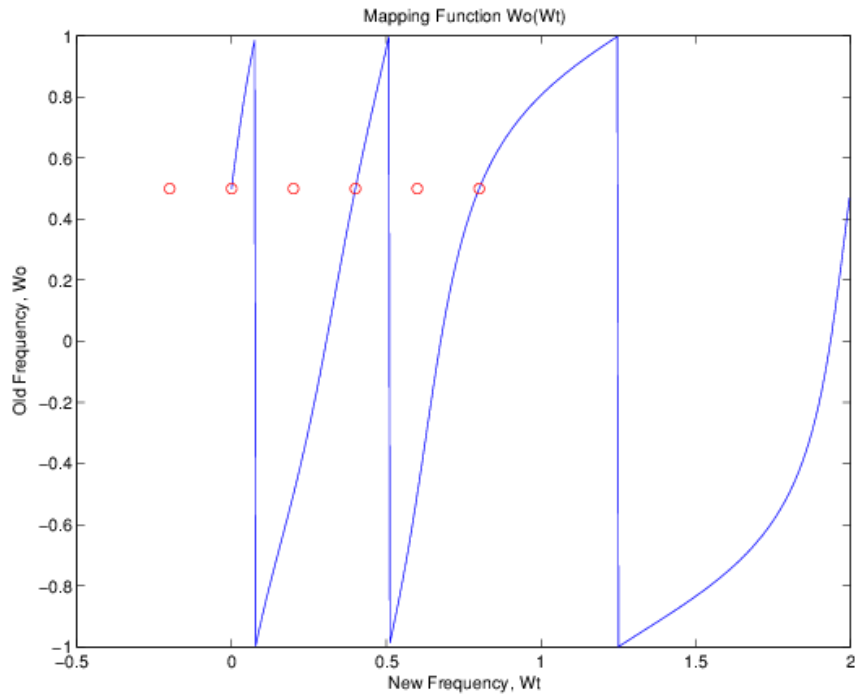
Calculate the frequency response of the mapping filter in the full range:

```
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
```

Plot the phase response normalized to  $\pi$ , which is in effect the mapping function  $W_o(W_t)$ . Please note that the transformation works in the same way for both positive and negative frequencies:

```
plot(f/pi, angle(h)/pi, Wt, Wo, 'ro');
title('Mapping Function Wo(Wt)');
xlabel('New Frequency, Wt'); ylabel('Old Frequency, Wo');
```

In this example, the resulting mapping function converts real filters to multiband complex filters.



## Arguments

| Variable          | Description   |
|-------------------|---|
| $w_0$             | Frequency value to be transformed from the prototype filter. It should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.     |
| $w_t$             | Desired frequency locations in the transformed target filter. They should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate. |
| <i>AllpassNum</i> | Numerator of the mapping filter   |
| <i>AllpassDen</i> | Denominator of the mapping filter   |

## Version History

Introduced in R2011a

## See Also

`iirlp2mbc` | `zpklp2mbc`



## allpasslp2xc

Allpass filter for lowpass to complex N-point transformation

### Syntax

```
[AllpassNum,AllpassDen] = allpasslp2xc(Wo,Wt)
```

### Description

`[AllpassNum,AllpassDen] = allpasslp2xc(Wo,Wt)` returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the Nth-order allpass mapping filter, where N is the allpass filter order, for performing a real lowpass to complex multipoint frequency transformation. Parameter N also specifies the number of replicas of the prototype filter created around the unit circle after the transformation. This transformation effectively places N features of the, original filter located at frequencies  $W_{o1}, \dots, W_{oN}$ , at the required target frequency locations,  $W_{t1}, \dots, W_{tM}$ .

Relative positions of other features of an original filter are the same in the target filter for the Nyquist mobility and are reversed for the DC mobility. For the Nyquist mobility this means that it is possible to select two features of an original filter,  $F_1$  and  $F_2$ , with  $F_1$  preceding  $F_2$ . Feature  $F_1$  will still precede  $F_2$  after the transformation. However, the distance between  $F_1$  and  $F_2$  will not be the same before and after the transformation. For DC mobility feature  $F_2$  will precede  $F_1$  after the transformation.

Choice of the feature subject to this transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones. The only condition is that the features must be selected in such a way that when creating N bands around the unit circle, there will be no band overlap.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can be easily replicated at a number of required frequency locations. A good application would be an adaptive tone cancellation circuit reacting to the changing number and location of tones.

### Examples

Design the allpass filter moving four features of an original complex filter given in  $W_o$  to the new independent frequency locations  $W_t$ . Please note that the transformation creates N replicas of an original filter around the unit circle, where N is the order of the allpass mapping filter:

```
Wo = [-0.2, 0.3, -0.7, 0.4]; Wt = [0.3, 0.5, 0.7, 0.9];
[AllpassNum, AllpassDen] = allpasslp2xc(Wo, Wt);
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
```

### Arguments

| Variable | Description  |
|----------|--|
| $W_o$    | Frequency values to be transformed from the prototype filter |

| <b>Variable</b>   | <b>Description</b>   |
|-------------------|--|
| <i>Wt</i>         | Desired frequency locations in the transformed target filter |
| <i>AllpassNum</i> | Numerator of the mapping filter                              |
| <i>AllpassDen</i> | Denominator of the mapping filter                            |

Frequencies must be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

## **Version History**

**Introduced in R2011a**

### **See Also**

`iirlp2xc` | `zpklp2xc`

## allpasslp2xn

Allpass filter for lowpass to N-point transformation

### Syntax

```
[AllpassNum,AllpassDen] = allpasslp2xn(Wo,Wt)
[AllpassNum,AllpassDen] = allpasslp2xn(Wo,Wt,Pass)
```

### Description

[AllpassNum,AllpassDen] = allpasslp2xn(Wo,Wt) returns the numerator, AllpassNum, and the denominator, AllpassDen, of the Nth-order allpass mapping filter, where N is the allpass filter order, for performing a real lowpass to real multipoint frequency transformation. Parameter N also specifies the number of replicas of the prototype filter created around the unit circle after the transformation. This transformation effectively places N features of an original filter, located at frequencies  $W_{o1}, \dots, W_{oN}$ , at the required target frequency locations,  $W_{t1}, \dots, W_{tM}$ . By default the DC feature is kept at its original location.

[AllpassNum,AllpassDen] = allpasslp2xn(Wo,Wt,Pass) allows you to specify an additional parameter, Pass, which chooses between using the “DC Mobility” and the “Nyquist Mobility.” In the first case the Nyquist feature stays at its original location and the DC feature is free to move. In the second case the DC feature is kept at an original frequency and the Nyquist feature is movable.

Relative positions of other features of an original filter are the same in the target filter for the Nyquist mobility and are reversed for the DC mobility. For the Nyquist mobility this means that it is possible to select two features of an original filter,  $F_1$  and  $F_2$ , with  $F_1$  preceding  $F_2$ . Feature  $F_1$  will still precede  $F_2$  after the transformation. However, the distance between  $F_1$  and  $F_2$  will not be the same before and after the transformation. For DC mobility feature  $F_2$  will precede  $F_1$  after the transformation.

Choice of the feature subject to this transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones. The only condition is that the features must be selected in such a way that when creating N bands around the unit circle, there will be no band overlap.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can be easily replicated at a number of required frequency locations without the need of designing them again. A good application would be an adaptive tone cancellation circuit reacting to the changing number and location of tones.

### Arguments

| Variable | Description  |
|----------|--|
| $W_o$    | Frequency values to be transformed from the prototype filter |
| $W_t$    | Desired frequency locations in the transformed target filter |

| <b>Variable</b>   | <b>Description</b>   |
|-------------------|--|
| <i>Pass</i>       | Choice ( 'pass ' / 'stop ' ) of passband/stopband at DC, 'pass ' being the default |
| <i>AllpassNum</i> | Numerator of the mapping filter  |
| <i>AllpassDen</i> | Denominator of the mapping filter  |

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

## **Version History**

**Introduced in R2011a**

## **References**

Cain, G.D., A. Krukowski and I. Kale, "High Order Transformations for Flexible IIR Filter Design," *VII European Signal Processing Conference (EUSIPCO'94)*, vol. 3, pp. 1582-1585, Edinburgh, United Kingdom, September 1994.

Krukowski, A., G.D. Cain and I. Kale, "Custom designed high-order frequency transformations for IIR filters," *38th Midwest Symposium on Circuits and Systems (MWSCAS'95)*, Rio de Janeiro, Brazil, August 1995.

## **See Also**

`iirlp2xn` | `zpklp2xn`

# allpassrateup

Allpass filter for integer upsample transformation

## Syntax

```
[AllpassNum,AllpassDen] = allpassrateup(N)
```

## Description

`[AllpassNum,AllpassDen] = allpassrateup(N)` returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the  $N$ th-order allpass mapping filter for performing the rateup frequency transformation, which creates  $N$  equal replicas of the prototype filter frequency response.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter,  $F_1$  and  $F_2$ , with  $F_1$  preceding  $F_2$ . Feature  $F_1$  will still precede  $F_2$  after the transformation. However, the distance between  $F_1$  and  $F_2$  will not be the same before and after the transformation.

## Examples

Design the allpass filter creating the effect of upsampling the digital filter four times:

Choose any feature from an original filter, say at  $W_o=0.2$ :

```
N = 4;
Wo = 0.2; Wt = Wo/N + 2*[0:N-1]/N;
[AllpassNum, AllpassDen] = allpassrateup(N);
```

Calculate the frequency response of the mapping filter in the full range:

```
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
```

## Arguments

| Variable          | Description                                    |
|-------------------|--|
| $N$               | Frequency replication ratio (upsampling ratio) |
| <i>AllpassNum</i> | Numerator of the mapping filter                |
| <i>AllpassDen</i> | Denominator of the mapping filter              |

## Version History

Introduced in R2011a

## See Also

iirateup | zpkrateup

## allpassshift

Allpass filter for real shift transformation

### Syntax

```
[AllpassNum,AllpassDen] = allpassshift(Wo,Wt)
```

### Description

`[AllpassNum,AllpassDen] = allpassshift(Wo,Wt)` returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the second-order allpass mapping filter for performing a real frequency shift transformation. This transformation places one selected feature of an original filter, located at frequency  $W_o$ , at the required target frequency location,  $W_t$ . This transformation implements the “DC mobility,” which means that the Nyquist feature stays at Nyquist, but the DC feature moves to a location dependent on the selection of  $W_o$  and  $W_t$ .

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter,  $F_1$  and  $F_2$ , with  $F_1$  preceding  $F_2$ . Feature  $F_1$  will still precede  $F_2$  after the transformation. However, the distance between  $F_1$  and  $F_2$  will not be the same before and after the transformation.

Choice of the feature subject to the real shift transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can be moved to a different frequency by applying a shift transformation. In such a way you can avoid designing the filter from the beginning.

### Examples

Design the allpass filter precisely shifting one feature of the lowpass filter originally at  $W_o=0.5$  to the new frequencies of  $W_t=0.25$ :

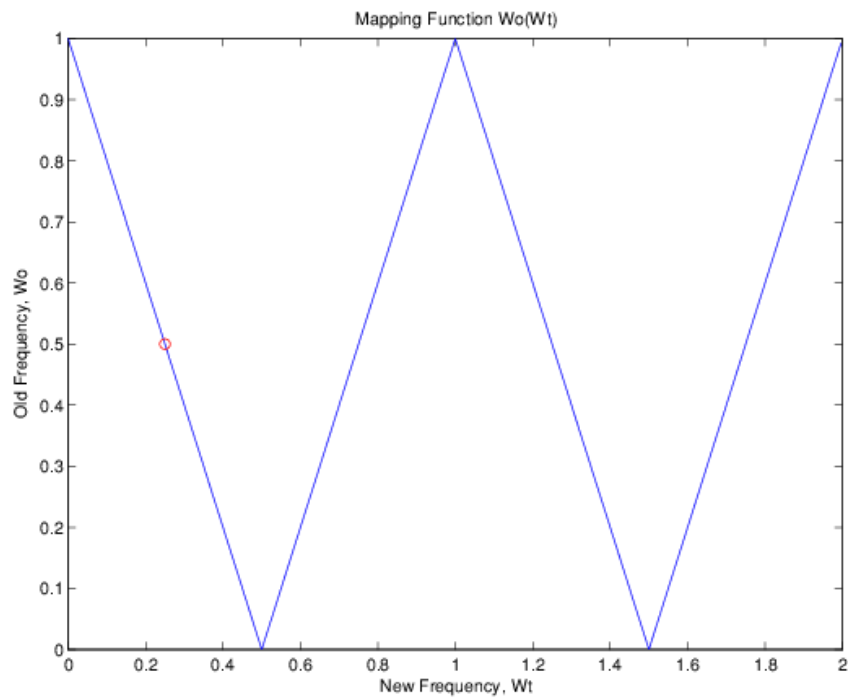
```
Wo = 0.5; Wt = 0.25;
[AllpassNum, AllpassDen] = allpassshift(Wo, Wt);
```

Calculate the frequency response of the mapping filter in the full range:

```
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
```

Plot the phase response normalized to  $\pi$ , which is in effect the mapping function  $W_o(W_t)$ . Please note that the transformation works in the same way for both positive and negative frequencies:

```
plot(f/pi, abs(angle(h))/pi, Wt, Wo, 'ro');
title('Mapping Function Wo(Wt)');
xlabel('New Frequency, Wt'); ylabel('Old Frequency, Wo');
```



## Arguments

| Variable          | Description   |
|-------------------|---|
| $W_o$             | Frequency value to be transformed from the prototype filter |
| $W_t$             | Desired frequency location in the transformed target filter |
| <i>AllpassNum</i> | Numerator of the mapping filter                             |
| <i>AllpassDen</i> | Denominator of the mapping filter                           |

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

## Version History

Introduced in R2011a

## See Also

iirshift | zpkshift

## allpassshiftc

Allpass filter for complex shift transformation

### Syntax

```
[AllpassNum,AllpassDen] = allpassshiftc(Wo,Wt)
[AllpassNum,AllpassDen] = allpassshiftc(0,0.5)
[AllpassNum,AllpassDen] = allpassshiftc(0,-0.5)
```

### Description

[AllpassNum,AllpassDen] = allpassshiftc(Wo,Wt) returns the numerator, AllpassNum, and denominator, AllpassDen, vectors of the allpass mapping filter for performing a complex frequency shift of the frequency response of the digital filter by an arbitrary amount.

[AllpassNum,AllpassDen] = allpassshiftc(0,0.5) calculates the allpass filter for doing the Hilbert transformation, a 90 degree counterclockwise rotation of an original filter in the frequency domain.

[AllpassNum,AllpassDen] = allpassshiftc(0,-0.5) calculates the allpass filter for doing an inverse Hilbert transformation, i.e. a 90 degree clockwise rotation of an original filter in the frequency domain.

### Examples

Design the allpass filter precisely rotating the whole filter by the amount defined by the location of the selected feature from an original filter,  $W_o=0.5$ , and its required position in the target filter,  $W_t=0.25$ :

```
Wo = 0.5; Wt = 0.25;
[AllpassNum, AllpassDen] = allpassshiftc(Wo, Wt);
```

Calculate the frequency response of the mapping filter in the full range:

```
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
```

### Arguments

| Variable          | Description   |
|-------------------|---|
| <i>Wo</i>         | Frequency value to be transformed from the prototype filter |
| <i>Wt</i>         | Desired frequency location in the transformed target filter |
| <i>AllpassNum</i> | Numerator of the mapping filter                             |
| <i>AllpassDen</i> | Denominator of the mapping filter                           |

Frequencies must be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.



## Version History

Introduced in R2011a

## References

Oppenheim, A.V., R.W. Schafer and J.R. Buck, *Discrete-Time Signal Processing*, Prentice-Hall International Inc., 1989.

Dutta-Roy, S.C. and B. Kumar, "On Digital Differentiators, Hilbert Transformers, and Half-band Low-pass Filters," *IEEE Transactions on Education*, vol. 32, pp. 314-318, August 1989.

## See Also

iirshiftc | zpkshiftc

## autoscale

Automatic dynamic range scaling

### Syntax

```
autoscale(hd,x)  
hnew = autoscale(hd,x)
```

### Description

`autoscale(hd,x)` provides dynamic range scaling for each node of the filter `hd`. This method runs signal `x` through `hd` in floating-point to simulate filtering. `autoscale` uses the maximum and minimum data obtained from that simulation at each filter node to set fraction lengths to cover the simulation full range and maximize the precision. Word lengths are not changed during autoscaling.

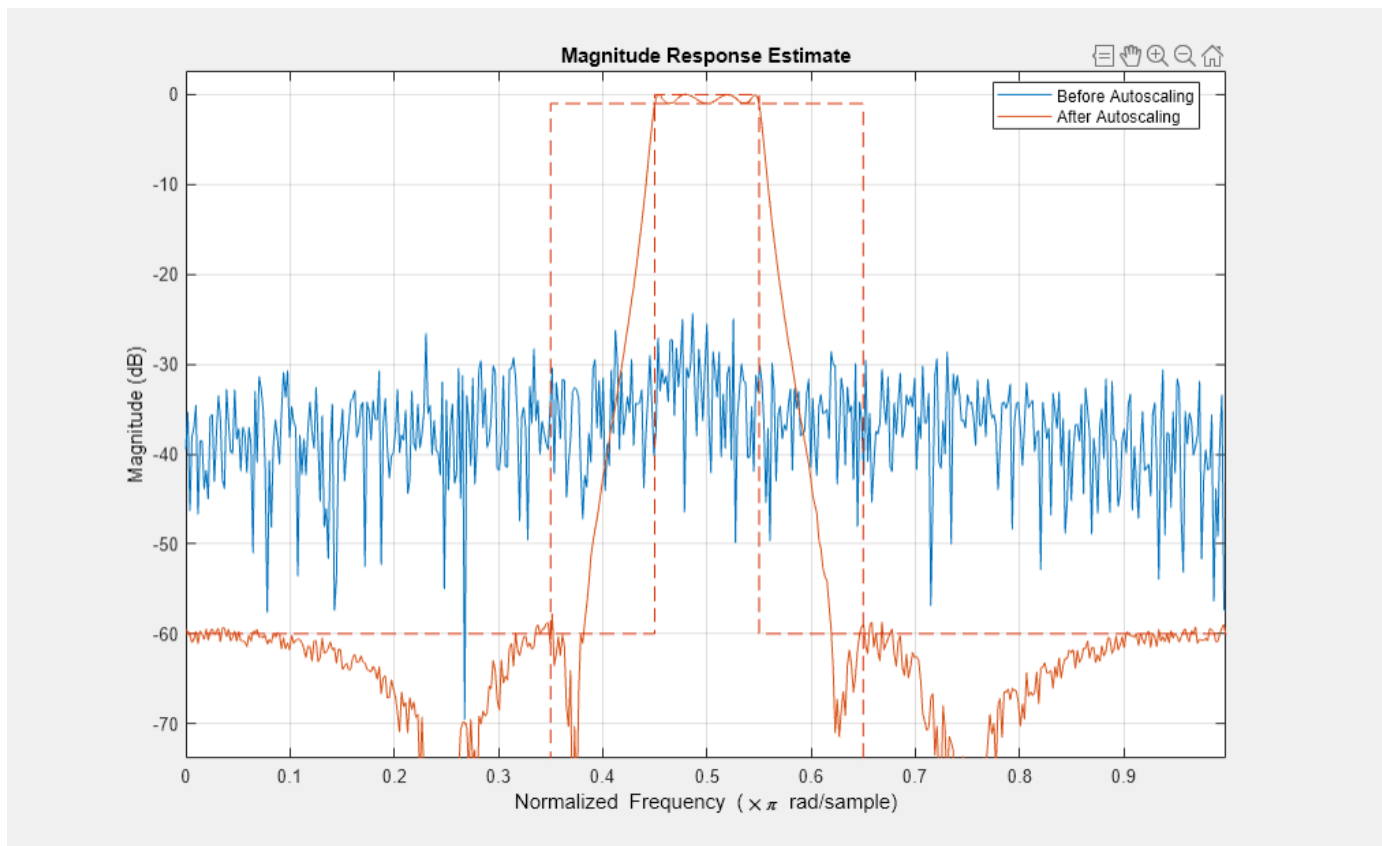
`hnew = autoscale(hd,x)` If you request an output, `autoscale` returns a new filter with the scaled fraction lengths. The original filter is not changed.

For an introductory example of the automatic scale process, see “Floating-Point to Fixed-Point Conversion of IIR Filters”.

### Examples

#### Dynamic Range Scaling in a Lattice ARMA Filter

```
hd = design(fdesign.bandpass,'ellip');  
hd = convert(hd,'latticearma');  
hd.arithmetic = 'fixed';  
rng(4); x = rand(100,10); % Training input data.  
hd(2) = autoscale(hd,x);  
hfvt = fvtool(hd,'Analysis','magestimate','ShowReference','off');  
legend(hfvt, 'Before Autoscaling', 'After Autoscaling')
```



## Version History

Introduced in R2011a

## bandedgeFrequencies

**Package:** dsp

Compute the bandedge frequencies

### Syntax

```
w = bandedgeFrequencies(obj)
f = bandedgeFrequencies(obj, Fs)
```

### Description

`w = bandedgeFrequencies(obj)` returns a vector of normalized frequencies, `w`, containing the bandedge frequencies between adjacent bandpass filters in the `dsp.Channelizer` System object.

`f = bandedgeFrequencies(obj, Fs)` returns a vector of bandedge frequencies in Hz, using the sample rate `Fs`.

### Examples

#### Bandedge Frequencies of Channelizer

Compute the normalized bandedge frequencies of the bandpass filters in a channelizer using the `bandedgeFrequencies` function.

```
channelizer = dsp.Channelizer;
w = bandedgeFrequencies(channelizer)
```

```
w = 1×8
```

```
    -2.7489    -1.9635    -1.1781    -0.3927     0.3927     1.1781     1.9635     2.7489
```

To compute the frequencies in Hz, pass a sampling frequency, `Fs`.

$$f = (\omega/2\pi) \times Fs$$

where:

- $f$  is frequency in Hz.
- $\omega$  is frequency in radians.

Here, the sampling frequency is 44,100 Hz.

```
f = bandedgeFrequencies(channelizer, 44100)
```

```
f = 1×8
    104 ×
```

```
    -1.9294    -1.3781    -0.8269    -0.2756     0.2756     0.8269     1.3781     1.9294
```

## Input Arguments

### **obj** — Input filter System object

`dsp.Channelizer`

Input filter, specified as a `dsp.Channelizer` System object.

### **Fs** — Sample rate

positive scalar

Sample rate used to compute the bandedge frequencies in Hz, specified as a positive scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

## Output Arguments

### **w** — Normalized bandedge frequencies

row vector

Normalized frequencies containing the bandedge frequencies between adjacent bandpass filters in the channelizer, returned as a row vector. The length of the vector equals the number of frequency bands.

Data Types: `double`

### **f** — Bandedge frequencies in Hz

row vector

Bandedge frequencies in Hz, returned as a row vector. The length of the vector equals the number of frequency bands. To return the frequencies in Hz, input the sample rate,  $F_s$ . Frequency in Hz is given by  $[w/(2\pi)] \times F_s$ , where  $w$  is the normalized frequency in rad/sample, and  $F_s$  is the sampling rate in Hz.

Data Types: `double`

## Version History

Introduced in R2017b

## See Also

### Functions

`polyphase` | `tf` | `fvtool` | `freqz` | `coeffs` | `centerFrequencies` | `getFilters`

### Objects

`dsp.Channelizer`

## block

Generate block from a digital filter

### Syntax

```
block(hd)
block(hd,'propertyname1',propertyvalue1,'propertyname2',propertyvalue2,...)
```

### Description

`block(hd)` generates a DSP System Toolbox block equivalent to the digital filter, `hd`.

`block(hd,'propertyname1',propertyvalue1,'propertyname2',propertyvalue2,...)` generates a DSP System Toolbox block using the options specified in the property name/property value pairs. The valid properties and their values are

| Property Name  | Property Values                                       | Description and Values  |
|----------------|---|---|
| Destination    | 'current' (default), 'new', or <i>Subsystemname</i> . | Determine which Simulink model gets the block. Enter 'current', 'new', or specify the name of an existing subsystem with <i>subsystemname</i> . 'current' adds the block to your current Simulink model. Specifying 'new' opens a new model and adds the block. If you provide the name of a subsystem in <i>subsystemname</i> , <code>block</code> adds the new block to your specified subsystem. |
| Blockname      | 'filter' (default)                                    | Specify the name of the generated block. The name appears below the block in the model. When you do not specify a block name, the default is <code>filter</code> .  |
| OverwriteBlock | 'off' (default), or 'on'.                             | Tell <code>block</code> whether to overwrite an existing block of the same name, or create a new block. 'off' is the default setting— <code>block</code> does not overwrite existing blocks with matching names. Switching from 'off' to 'on' directs <code>block</code> to overwrite existing blocks.  |
| MapStates      | 'off' (default), or 'on'.                             | Specify whether to apply the current filter states to the new block. This lets you save states from a filter object you may have used or configured in a specific way. The default setting of 'off' means the states are not transferred to the block. Choosing 'on' preserves the current filter states in the block.  |

| Property Name    | Property Values   | Description and Values   |
|------------------|---|--|
| Link2Obj         | 'off' (default), or 'on'.   | Specify how to set the <b>Coefficient source</b> in the block mask. The default setting is 'off' and the <b>Coefficient source</b> is set to <b>Dialog parameters</b> . Setting Link2Obj to 'on' sets the <b>Coefficient source</b> to <b>Discrete-time filter object (DFILT)</b> . The Link2Obj and MapCoeffstoPorts cannot be simultaneously 'on'. |
| MapCoeffstoPorts | 'off' (default) or 'on'   | Specify whether to map the coefficients of the filter to the ports of the block. The Link2Obj and MapCoeffstoPorts cannot be simultaneously 'on'.  |
| CoeffNames       | { 'Num' } (default FIR)<br>{ 'Num', 'Den' } (default direct form IIR)<br>{ 'Num', 'Den', 'g' } (default IIR SOS),<br>{ 'K' } (default form lattice) | Specify the coefficient variable names as a cell array of character vectors.<br><br>MapCoeffstoPorts must be set to 'on' for this property to apply.   |
| InputProcessing  | 'columns as channels' (default), 'elements as channels'   | Specify sample-based (elements as channels) or frame-based (columns as channels) processing.   |
| RateOption       | 'enforce single rate' (default) or 'allow multirate'  | Specify how the block adjusts the rate at the output to accommodate the reduced number of samples. This parameter applies only when InputProcessing is 'columns as channels'.  |

### Using block to Realize a Fixed-Point Digital Filter

When the source filter `hd` is fixed-point, the input word and fraction lengths for the block are derived from the block input signal. The realization process issues a warning and ignores the input word and input fraction lengths that are part of the source filter object, choosing to inherit the settings from the input data. Other fixed-point properties map directly to settings for word and fraction length in the realized block.

## Examples

### Create a Block from a Lowpass Filter

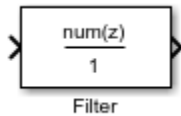
Create a lowpass filter specification object. Specify the passband frequency to be  $0.15\pi$  rad/sample and the stopband frequency to be  $0.25\pi$  rad/sample. Specify 1 dB of allowable passband ripple and a stopband attenuation of 60 dB.

In the first example, use `block` with the default syntax, letting the function determine the block name and configuration.

```
d = fdesign.lowpass('Fp,Fst,Ap,Ast',0.15,0.25,1,60);
hd = design(d);
```

Now use the default syntax to create a block.

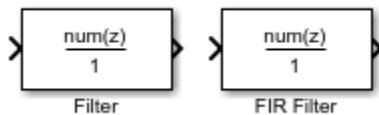
```
block(hd);
```



In this second example, define the block name to meet your needs by using the property name/property value pair input arguments.

```
block(hd, 'blockname', 'FIR Filter');
```

The figure shows the blocks in a Simulink model. When you try these examples, you see that the second block writes over the first block location. You can avoid this by moving the first block before you generate the second, always naming your block with the blockname property, or setting the Destination property to new which puts the filter block in a new Simulink model.



## Version History

Introduced in R2011a

### See Also

realizemdl

### Topics

“Realize Filters as Simulink Subsystem Blocks”



# butter

Butterworth IIR digital filter design

## Syntax

```
butterFilter = butter(designSpecs,SystemObject=true)
butterFilter = butter(designSpecs,designoption=value,... SystemObject=true)
butterFilter = design(designSpecs,'butter',SystemObject=true)
```

## Description

`butterFilter = butter(designSpecs,SystemObject=true)` designs a butterworth IIR digital filter using specifications in the object `designSpecs`.

Depending on the filter specification object `designSpecs`, the `butter` design method might not be valid. Use `designmethods` with the filter specification object to determine if a butterworth IIR digital filter design is possible.

```
designmethods(designSpecs,SystemObject=true)
```

`butterFilter = butter(designSpecs,designoption=value,... SystemObject=true)` returns a butterworth IIR filter with one or more specified designed options and the corresponding values.

To view a list of available design options, run the `designoptions` function on the specification object. The function also lists the default design options the filter uses.

```
designoptions(designSpecs,'butter')
```

`butterFilter = design(designSpecs,'butter',SystemObject=true)` is an alternative syntax for designing the butterworth IIR digital filter.

For complete help about using the `butter` design method for a specification object, `designSpecs`, enter the following at the MATLAB command prompt.

```
help(designSpecs,'butter')
```

## Examples

### Design Butterworth Filter

Design a butterworth filter with lowpass and highpass frequency responses. The filter design procedure is:

- 1 Specify the filter design specifications using a `fdesign` function.
- 2 Pick a design method provided by the `designmethods` function.
- 3 To determine the available design options to choose from, use the `designoptions` function.
- 4 Design the filter using the `design` function.

## Lowpass Filter

Construct a default lowpass filter design specification object using `fdesign.lowpass`.

```
designSpecs = fdesign.lowpass

designSpecs =
    lowpass with properties:

        Response: 'Lowpass'
        Specification: 'Fp,Fst,Ap,Ast'
        Description: {4x1 cell}
        NormalizedFrequency: 1
            Fpass: 0.4500
            Fstop: 0.5500
            Apass: 1
            Astop: 60
```

Determine the available design methods using the `designmethods` function. To design a butterworth filter, pick `butter`.

```
designmethods(designSpecs,SystemObject=true)

Design Methods that support System objects for class fdesign.lowpass (Fp,Fst,Ap,Ast):

butter
cheby1
cheby2
ellip
equiripple
ifir
kaiserwin
multistage
```

While designing the filter, you can specify additional design options. View a list of the options using the `designoptions` function. This function also shows the default design options the filter uses.

```
designoptions(designSpecs,'butter',SystemObject=true)

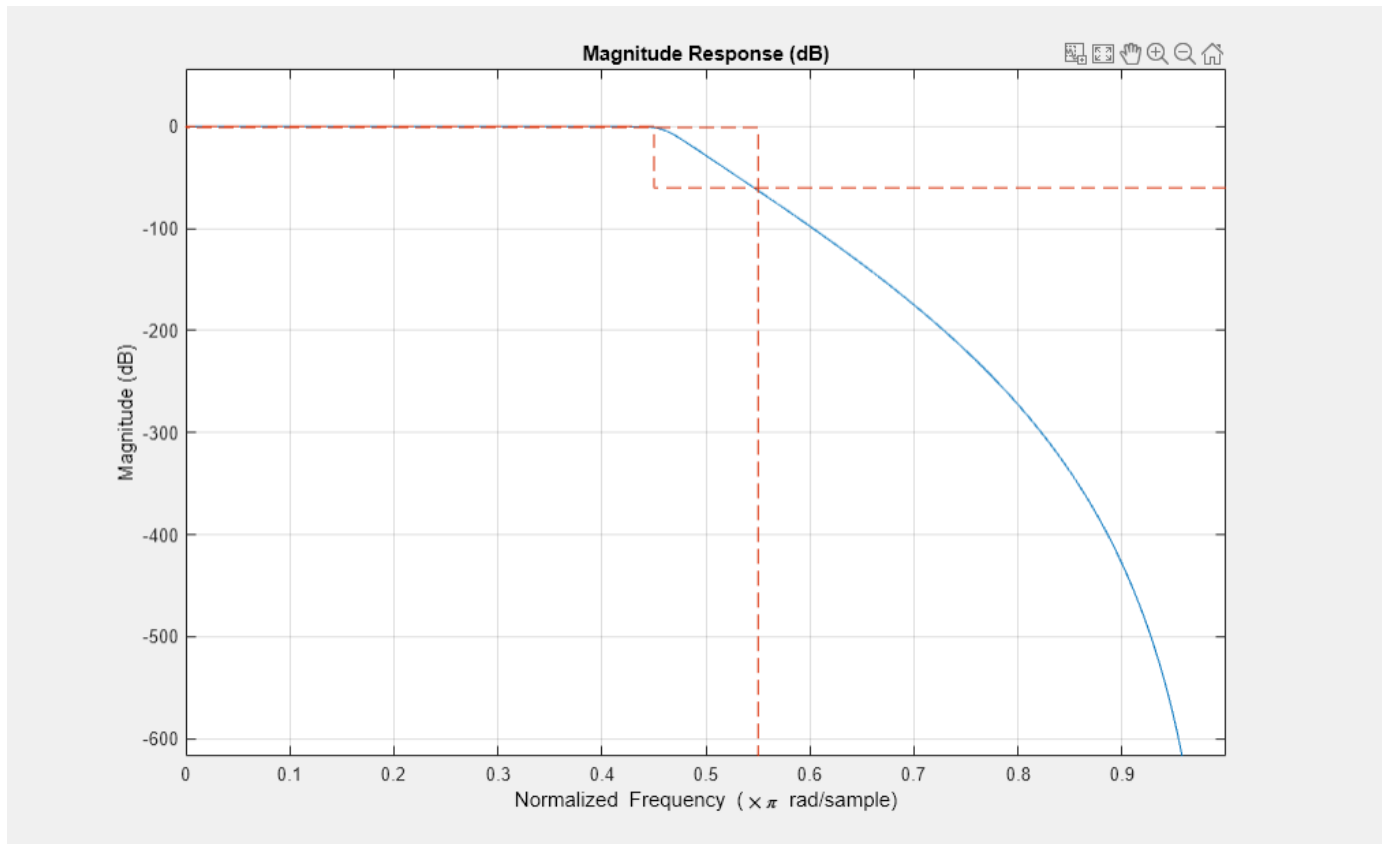
ans = struct with fields:
    FilterStructure: {'df1sos' 'df2sos' 'df1tsos' 'df2tsos' 'cascadeallpass' 'cascade...
        SOSScaleNorm: 'ustring'
        SOSScaleOpts: 'fdopts.sosscaling'
        MatchExactly: {'passband' 'stopband'}
    DefaultFilterStructure: 'df2sos'
    DefaultMatchExactly: 'stopband'
    DefaultSOSScaleNorm: ''
    DefaultSOSScaleOpts: [1x1 fdopts.sosscaling]
```

Use the `design` function to design the filter. Pass `'butter'` and the specifications given by variable `designSpecs`, as input arguments. Specify the `'matchexactly'` design option to `'passband'`.

```
lpFilter = design(designSpecs,'butter',matchexactly='passband',SystemObject=true);
```

Visualize the frequency response of the designed filter.

```
fvtool(lpFilter)
```



## Highpass Filter

Construct a highpass filter design specification object using `fdesign.highpass`. Specify the order to be 7 and the 3 dB frequency to be  $0.6\pi$  radians/sample.

```
designSpecs = fdesign.highpass('N,F3dB',7,.6);
```

Determine the available design methods. To design a butterworth filter, pick `butter`.

```
designmethods(designSpecs,SystemObject=true)
```

Design Methods that support System objects for class `fdesign.highpass(N,F3dB)`:

```
butter
maxflat
```

While designing the filter, you can specify additional design options. View a list of the options using the `designoptions` function. This function also shows the default design options the filter uses.

```
designoptions(designSpecs,'butter',SystemObject=true)
```

```
ans = struct with fields:
    FilterStructure: {'df1sos' 'df2sos' 'df1tsos' 'df2tsos' 'cascadeallpass' 'cascad
    SOSScaleNorm: 'ustring'
    SOSScaleOpts: 'fdopts.sosscaling'
```

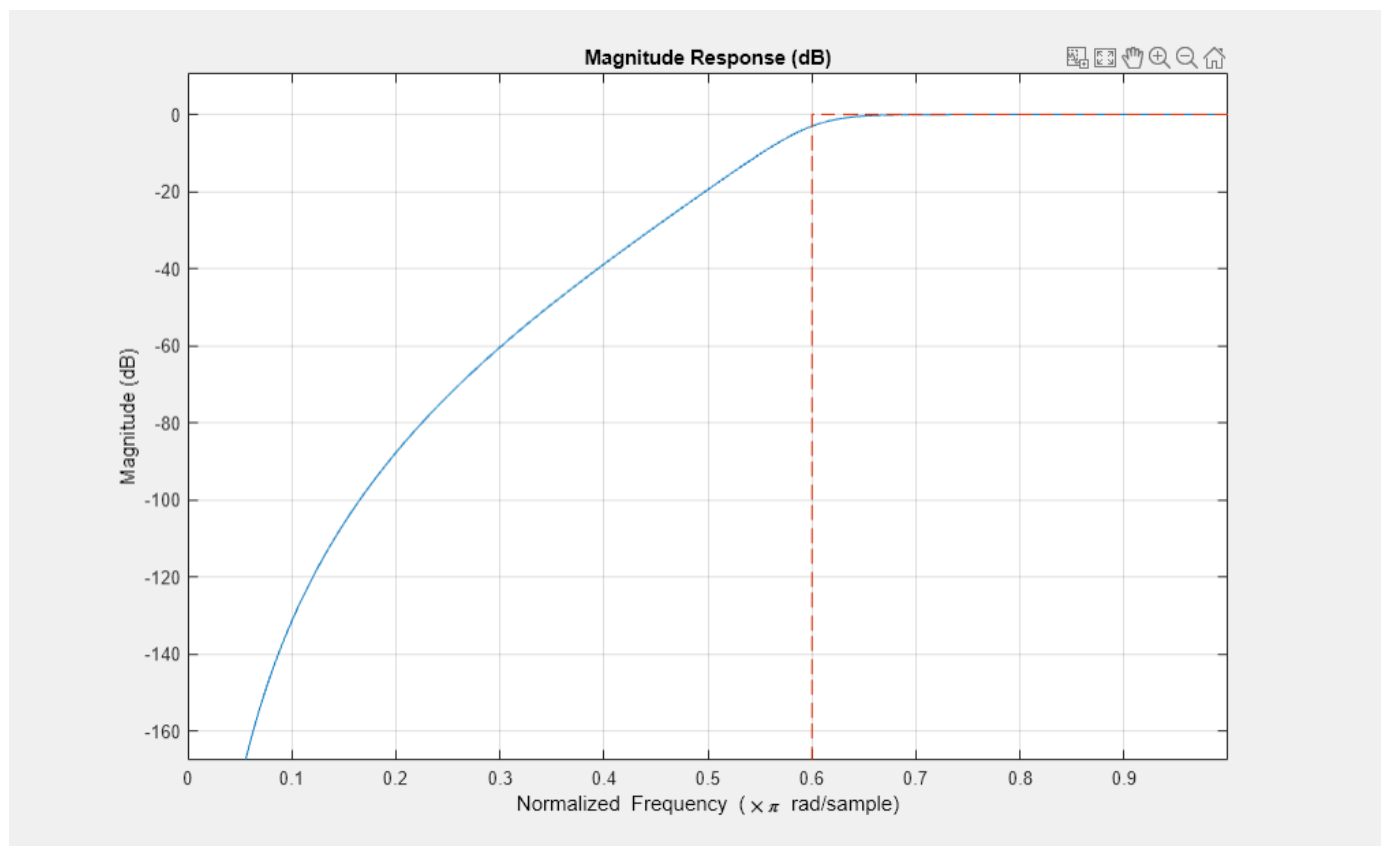
```
DefaultFilterStructure: 'df2sos'  
DefaultSOSScaleNorm: ''  
DefaultSOSScaleOpts: [1x1 fdopts.sosscaling]
```

To design the butterworth filter, use the design function and specify 'butter' as an input. Set 'FilterStructure' to 'cascadeallpass'.

```
hpFilter = design(designSpecs, 'butter', FilterStructure='cascadeallpass', SystemObject=true);
```

Visualize the highpass frequency response.

```
fvtool(hpFilter)
```



## Input Arguments

### **designSpecs** — Filter specification object

object

Filter specification object, specified as one of the following:

- `fdesign.bandpass`
- `fdesign.bandstop`
- `fdesign.comb`
- `fdesign.decimator`

- `fdesign.halfband`
- `fdesign.highpass`
- `fdesign.interpolator`
- `fdesign.lowpass`
- `fdesign.notch`
- `fdesign.nyquist`
- `fdesign.peak`

## Output Arguments

### **butterFilter** — Butterworth IIR filter

System object

Butterworth IIR filter, returned as a filter System object. The System object and the values of its properties depend on the input `designSpecs` object and the other design options specified to the function.

## Version History

### Introduced in R2011a

### **R2022b: Designs `dsp.SOSFilter` object instead of `dsp.BiquadFilter` object**

*Behavior changed in R2022b*

The `butter` design method now generates a `dsp.SOSFilter` object instead of the `dsp.BiquadFilter` object.

Initialize the default `fdesign.lowpass` filter specification object. Use the design function with the `SystemObject=true` flag to design a `dsp.SOSFilter` filter object.

```
designSpecs = fdesign.lowpass;
filtObj = design(designSpecs, 'butter', SystemObject=true)
```

```
filtObj =
```

```
  dsp.SOSFilter with properties:
```

```
      Structure: 'Direct form II'
CoefficientSource: 'Property'
      Numerator: [13×3 double]
      Denominator: [13×3 double]
HasScaleValues: true
      ScaleValues: [0.4151 0.3718 0.3374 0.3099 0.2878 0.2701 0.2558 ... ]
```

```
Show all properties
```

To design a `dsp.BiquadFilter` object instead of the `dsp.SOSFilter` object, you can use the `UseLegacyBiquadFilter` flag and set it to `true`.

```
filtObj = design(designSpecs, 'butter', SystemObject=true, ...
UseLegacyBiquadFilter=true)
```

```
filtObj =
```

```
  dsp.BiquadFilter with properties:
```

```
        Structure: 'Direct form II'  
    SOSMatrixSource: 'Property'  
        SOSMatrix: [13×6 double]  
        ScaleValues: [14×1 double]  
    InitialConditions: 0  
    OptimizeUnityScaleValues: true
```

Show all properties

## See Also

### Functions

design | designmethods | designoptions | fdesign | cheby1 | cheby2 | ellip | equiripple | kaiserwin

## ca2tf

Convert coupled allpass filter to transfer function form

### Syntax

```
[b,a] = ca2tf(d1,d2)
[b,a] = ca2tf(d1,d2,beta)
[b,a,bp] = ca2tf(d1,d2)
[b,a,bp] = ca2tf(d1,d2,beta)
```

### Description

`[b,a] = ca2tf(d1,d2)` returns the vector of coefficients of **b** and **a**. **b** and **a** corresponds to the numerator and the denominator of the transfer function  $H(z)$ , respectively, where **d1** and **d2** are real vectors corresponding to the denominators of the allpass filters  $H1(z)$  and  $H2(z)$ .

$$H(z) = B(z)/A(z) = \frac{1}{2}[H1(z) + H2(z)]$$

`[b,a] = ca2tf(d1,d2,beta)` returns the vector of coefficients **b** and the vector of coefficients **a** corresponding to the numerator and the denominator of the transfer function  $H(z)$ , respectively, where **d1** and **d2** are complex vectors and **beta** is a complex scalar.

$$H(z) = B(z)/A(z) = \frac{1}{2}[-(\bar{\beta}) \cdot H1(z) + \beta \cdot H2(z)]$$

`[b,a,bp] = ca2tf(d1,d2)` also returns the vector **bp** of real coefficients corresponding to the numerator of the power-complementary filter  $G(z)$ , where **d1** and **d2** are real vectors.

$$G(z) = Bp(z)/A(z) = \frac{1}{2}[H1(z) - H2(z)]$$

`[b,a,bp] = ca2tf(d1,d2,beta)` also returns the vector of coefficients **bp** of real or complex coefficients that correspond to the numerator of the power-complementary filter  $G(z)$ , where **d1** and **d2** are complex vectors and **beta** is a complex scalar.

$$G(z) = Bp(z)/A(z) = \frac{1}{2j}[-(\bar{\beta}) \cdot H1(z) + \beta \cdot H2(z)]$$

### Examples

#### Convert Coupled Allpass Filter to Transfer Function Form

Create a coupled allpass filter.

```
[b,a] = cheby1(10,.5,.4);
```

The `tf2ca` function returns the denominators of the coupled allpass filter.

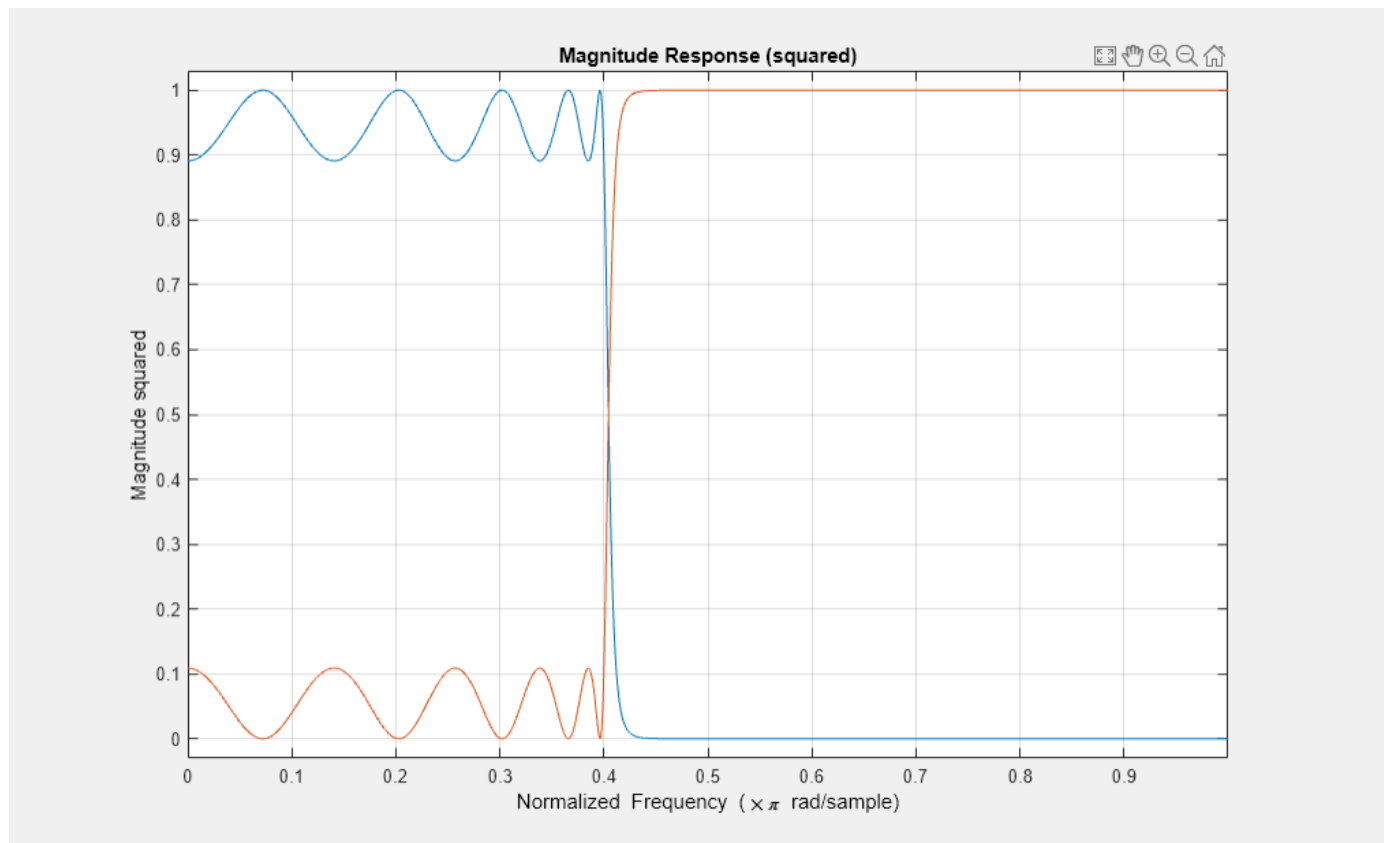
```
[d1,d2,beta] = tf2ca(b,a);
```

Reconstruct the original and the power-complementary filters using the `ca2tf` function.

```
[num,den,numpc] = ca2tf(d1,d2,beta);
```

Plot the magnitude response of the original filter and the power-complementary one.

```
fvtool(num,den,numpc,den,'Analysis','magnitude','MagnitudeDisplay',...
        'Magnitude Squared')
```



## Input Arguments

### **d1 — Denominator of allpass filter $H1(z)$**

vector

Denominator of the allpass filter  $H1(z)$ , specified as a real- or complex-valued vector.

Data Types: `single` | `double`

### **d2 — Denominator of allpass filter $H2(z)$**

vector

Denominator of the allpass filter  $H2(z)$ , specified as a real- or complex-valued vector.

Data Types: `single` | `double`

### **beta — Complex scalar of magnitude 1**

complex scalar



Complex scalar of magnitude 1.

Data Types: `single` | `double`

## Output Arguments

### **b** — Numerator coefficients of transfer function

vector

Numerator coefficients of the transfer function  $H(z)$ , returned as a real- or complex-valued vector.

Data Types: `double`

### **a** — Denominator coefficients of transfer function

vector

Denominator coefficients of the transfer function  $H(z)$ , returned as a real- or complex-valued vector.

Data Types: `double`

### **bp** — Numerator coefficients of power-complementary filter $G(z)$

vector

Numerator coefficients of the power-complementary filter  $G(z)$ , returned as a real- or complex-valued vector.

Data Types: `double`

## Version History

Introduced in R2011a

## References

- [1] Vaidyanathan, P. P., and Sanjit K. Mitra. *Robust Digital Filter Structures: A Direct Approach*. IEEE Circuits and Systems Magazine 19, no. 1 (2019): 14-32. <https://doi.org/10.1109/MCAS.2018.2889204>.
- [2] Vaidyanathan, P. P. *Multirate Systems and Filter Banks*. Prentice-Hall Signal Processing Series. Englewood Cliffs, N.J: Prentice Hall, 1993.

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

All inputs must be constant. Expressions or variables are allowed if their values do not change.

## See Also

`cl2tf` | `iirpowcomp` | `tf2ca` | `tf2cl`

## cascade

Cascade of filter system objects

### Syntax

```
FC = cascade(obj1,obj2,...objn)
```

### Description

`FC = cascade(obj1,obj2,...objn)` returns an object, `FC`, of type `dsp.FilterCascade`. `FC` is a cascaded version of the input System objects `obj1`, `obj2`, ..., `objn`. You can input multiple System objects to the function. The input System objects must be supported by the cascade method. For the list of supported System objects, see “Input Arguments” on page 5-75.

### Examples

#### Design Two-Stage Decimator

Design a two-stage decimator by cascading `dsp.CICDecimator` and `dsp.CICCompensationDecimator` System objects.

#### Construct the objects

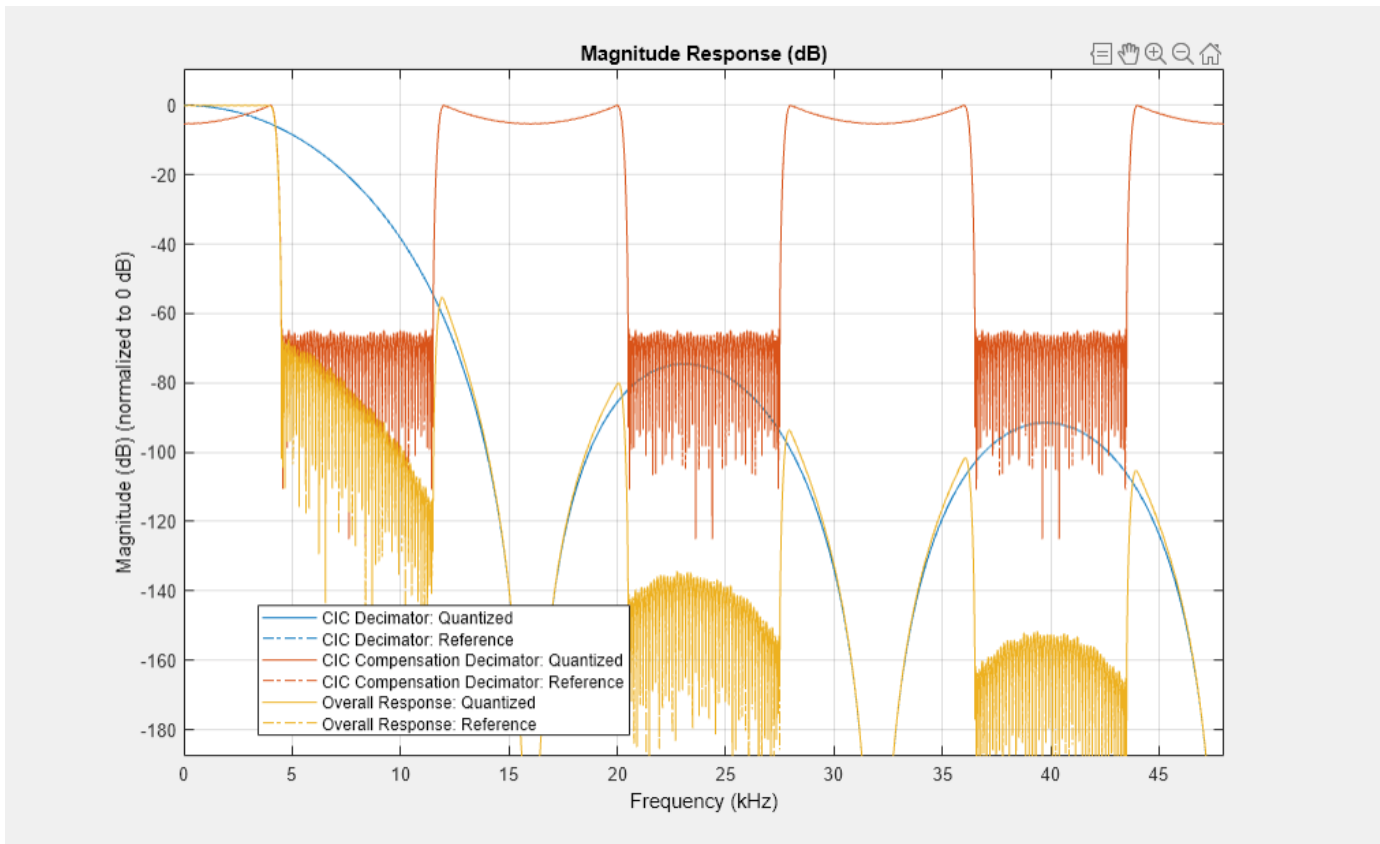
```
CICDecim = dsp.CICDecimator('DecimationFactor', 6, ...
                           'NumSections', 6);
fs = 16e3;    % Sampling frequency of input of compensation decimator
fPass = 4e3;  % Passband frequency
fStop = 4.5e3; % Stopband frequency
CICCompDecim = dsp.CICCompensationDecimator(CICDecim, ...
                                             'DecimationFactor', 2, ...
                                             'PassbandFrequency', fPass, ...
                                             'StopbandFrequency', fStop, ...
                                             'SampleRate', fs);
```

#### Create a cascade of the two objects using the cascade method

```
FC = cascade(CICDecim, CICCompDecim);
```

#### Visualize the frequency response of the cascade

```
f = fvtool(CICDecim, CICCompDecim, FC, 'Fs', [fs*6, fs, fs*6], ...
           'Arithmetic', 'fixed');
set(f, 'NormalizeMagnitudeto1', 'on');
legend(f, 'CIC Decimator', 'CIC Compensation Decimator', ...
       'Overall Response');
```



## Input Arguments

### obj1 — Filter to be cascaded

filter System object

obj1, obj2, . . . objn are filters to be cascaded. To see the list of System objects you can pass to the cascade method, type

```
dsp.FilterCascade.helpSupportedSystemObjects
```

in the MATLAB command prompt.

## Output Arguments

### FC — cascaded filter

dsp.FilterCascade System object

Cascaded filter, returned as a System object of type `dsp.FilterCascade`. For information on the properties of the filter in each stage, type `info(FC)` in the MATLAB command prompt.

## Version History

Introduced in R2016a

**See Also**

`dsp.FilterCascade`

# centerFrequencies

**Package:** dsp

Compute center frequencies

## Syntax

```
w = centerFrequencies(obj)
f = centerFrequencies(obj, Fs)
```

## Description

`w = centerFrequencies(obj)` returns a vector of normalized frequencies,  $w$ , containing the center frequencies of all the bandpass filters in the `dsp.Channelizer` System object.

`f = centerFrequencies(obj, Fs)` returns a vector of center frequencies in Hz, using the sample rate  $F_s$ .

## Examples

### Center Frequencies of Channelizer

Compute the normalized center frequencies of the bandpass filters in a channelizer using the `centerFrequencies` function.

```
channelizer = dsp.Channelizer;
w = centerFrequencies(channelizer)
```

$w = 1 \times 8$

```
-3.1416  -2.3562  -1.5708  -0.7854      0   0.7854   1.5708   2.3562
```

To compute the frequencies in Hz, pass a sampling frequency. Frequency in Hz,  $f$ , equals  $f = (\omega/2\pi) \times F_s$ , where  $w$  is frequency in radians, and  $F_s$  is the sampling rate.

```
f = centerFrequencies(channelizer, 44100)
```

$f = 1 \times 8$   
 $10^4 \times$

```
-2.2050  -1.6538  -1.1025  -0.5513      0   0.5513   1.1025   1.6538
```

## Input Arguments

**obj** — Input filter System object

`dsp.Channelizer`

Input filter, specified as a `dsp.Channelizer` System object.

**Fs — Sample rate**

positive scalar

Sample rate used to compute the center frequencies in Hz, specified as a positive scalar.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**Output Arguments****w — Normalized center frequencies**

row vector

Normalized frequencies containing the center frequencies of all the bandpass filters in the channelizer, returned as a row vector. The length of the vector equals the number of frequency bands.

Data Types: double

**f — Center frequencies in Hz**

row vector

Center frequencies in Hz, returned as a row vector. The length of the vector equals the number of frequency bands. To return the frequencies in Hz, input the sample rate,  $F_s$ . Frequency in Hz is given by  $[w/(2\pi)] \times F_s$ , where  $w$  is the normalized frequency in rad/sample, and  $F_s$  is the sampling rate in Hz.

Data Types: double

**Version History****Introduced in R2017b****See Also****Functions**

polyphase | tf | fvtool | freqz | coeffs | bandedgeFrequencies | getFilters

**Objects**

dsp.Channelizer

# cheby1

Chebyshev Type I filter using specification object

## Syntax

```
chebOneFilter = cheby1(designSpecs,SystemObject=true)
chebOneFilter = cheby1(designSpecs,designoption=value,... SystemObject=true)
chebOneFilter = design(designSpecs,'cheby1',SystemObject=true)
```

## Description

`chebOneFilter = cheby1(designSpecs,SystemObject=true)` designs a type I Chebyshev IIR digital filter using specifications in the `designSpecs` object.

For help about using the `cheby1` design method for a filter specification object, enter the following at the MATLAB command prompt.

```
help(designSpecs,'cheby1')
```

You cannot use the `cheby1` design method for certain filter specification objects. Use the `designmethods` function with the filter specification object to determine if the `cheby1` design method is valid for your filter specifications.

```
designmethods(designSpecs,SystemObject=true)
```

`cheby1` returns filters that use second-order sections (SOS). SOS filters are particularly well-suited for most fixed-point applications.

`chebOneFilter = cheby1(designSpecs,designoption=value,... SystemObject=true)` returns a type I Chebyshev IIR digital filter with the specified design options. You can specify one or more design options and their corresponding values.

To view a list of available design options, run the `designoptions` function on the filter specification object. The function also lists the default design options that the filter uses.

```
designoptions(designSpecs,'cheby1')
```

`chebOneFilter = design(designSpecs,'cheby1',SystemObject=true)` is an alternative syntax for designing a type I Chebyshev IIR digital filter.

## Examples

### Design Chebyshev Type I Filter

Design a type 1 Chebyshev IIR filter with lowpass and highpass frequency responses. The filter design procedure is:

- 1 Specify the filter design specifications using a `fdesign` function.
- 2 Pick a design method provided by the `designmethods` function.

- 3 To determine the available design options, use the `designoptions` function.
- 4 Design the filter using the `design` function.

### Lowpass Filter

Create a default lowpass filter specification object using the `fdesign.lowpass` function.

```
designSpecs = fdesign.lowpass;
```

Determine the available design methods using the `designmethods` function. To design a type 1 Chebyshev filter, pick `cheby1`.

```
designmethods(designSpecs, SystemObject=true)
```

```
Design Methods that support System objects for class fdesign.lowpass (Fp,Fst,Ap,Ast):
```

```
butter
cheby1
cheby2
ellip
equiripple
ifir
kaiserwin
multistage
```

While designing the filter, you can specify additional design options. View the options using the `designoptions` function. This function also shows the default design options the filter uses.

```
designoptions(designSpecs, 'cheby1', SystemObject=true)
```

```
ans = struct with fields:
    FilterStructure: {'df1sos' 'df2sos' 'df1tsos' 'df2tsos' 'cascadeallpass' 'cascad
    SOSScaleNorm: 'ustring'
    SOSScaleOpts: 'fdopts.sosscaling'
    MatchExactly: {'passband' 'stopband'}
    DefaultFilterStructure: 'df2sos'
    DefaultMatchExactly: 'passband'
    DefaultSOSScaleNorm: ''
    DefaultSOSScaleOpts: [1x1 fdopts.sosscaling]
```

Use the `design` function to design the filter. Pass `'cheby1'` and the `designSpecs` object containing the filter design specifications as input arguments. Set the filter structure to `'df1sos'` to design a filter with a direct form I SOS structure.

```
LowpassCheb1 = design(designSpecs, 'cheby1', FilterStructure='df1sos', ...
    SystemObject=true)
```

```
LowpassCheb1 =
    dsp.SOSFilter with properties:
```

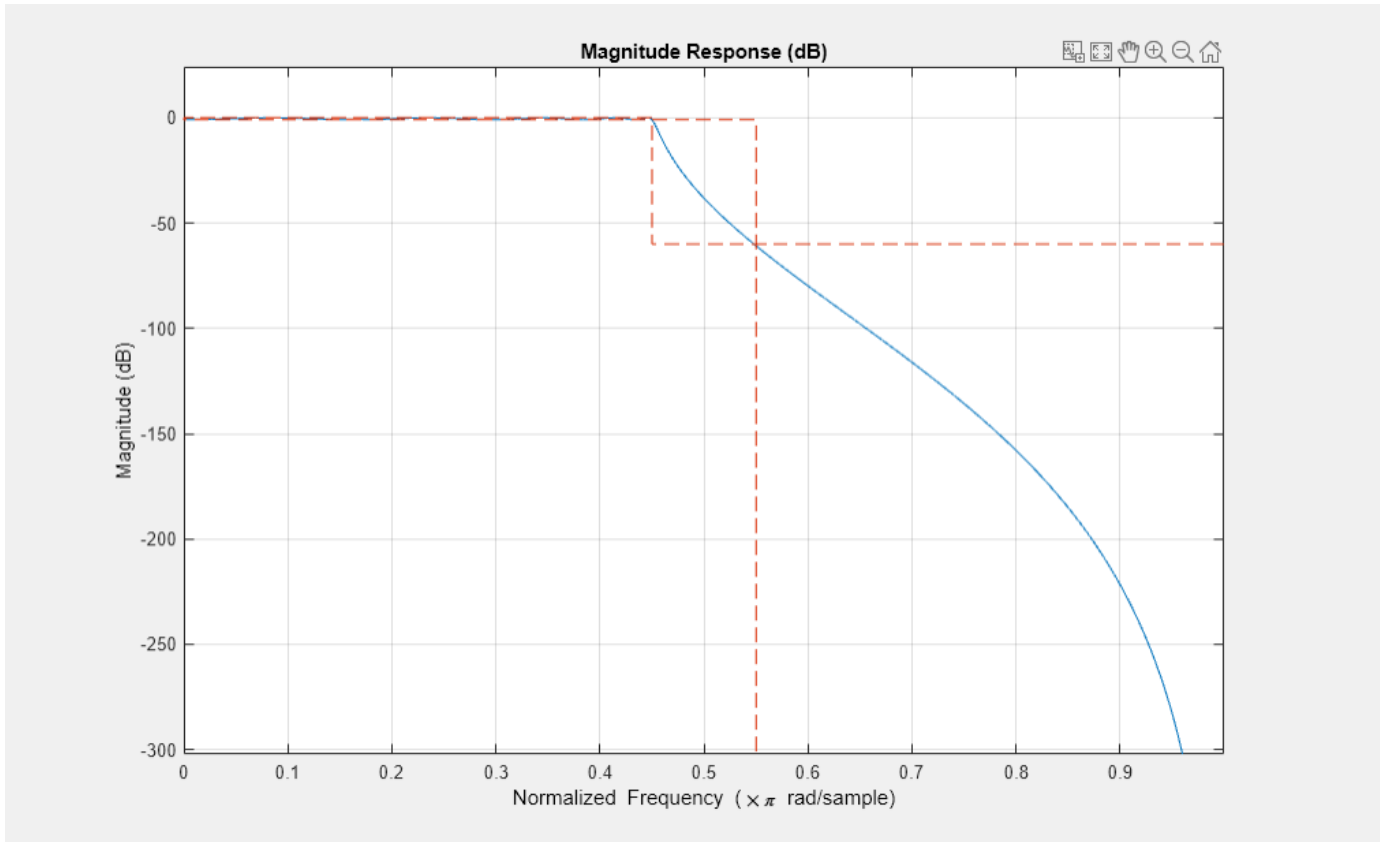
```
    Structure: 'Direct form I'
    CoefficientSource: 'Property'
    Numerator: [5x3 double]
    Denominator: [5x3 double]
    HasScaleValues: true
    ScaleValues: [0.4117 0.3484 0.2445 0.1195 0.0258 0.8913]
```



Show all properties

Use `fvtool` to view the frequency response of the designed filter.

```
fvtool(LowpassCheb1)
```



## Highpass Filter

Create a highpass filter design specification object using the `fdesign.highpass` function. Specify the filter order, passband edge frequency, and the passband ripple.

```
designSpecs = fdesign.highpass('n,fp,ap',7,20,.4,50);
```

Determine the available design methods.

```
designmethods(designSpecs,SystemObject=true)
```

Design Methods that support System objects for class `fdesign.highpass(N,Fp,Ap)`:

```
cheby1
```

While designing the filter, you can specify additional design options. View a list of the options using the `designoptions` function. This function also shows the default design options the filter uses.

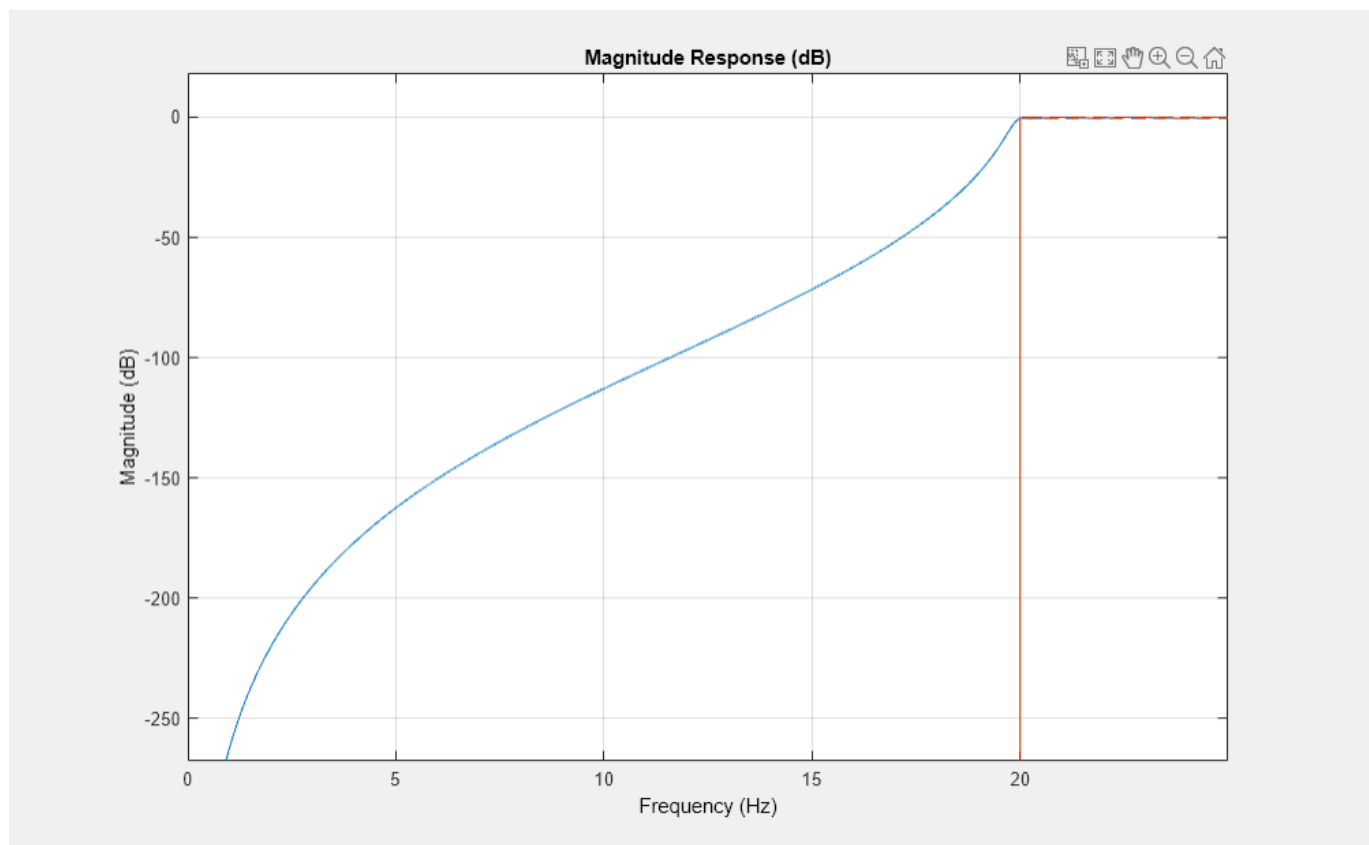
```
HighpassCheb1 = design(designSpecs,'cheby1',SystemObject=true)
```

```
HighpassCheb1 =  
  dsp.SOSFilter with properties:  
  
    Structure: 'Direct form II'  
    CoefficientSource: 'Property'  
    Numerator: [4x3 double]  
    Denominator: [4x3 double]  
    HasScaleValues: true  
    ScaleValues: [0.0943 0.0612 0.0233 0.0814 1]
```

Show all properties

Use `fvtool` to visualize the highpass frequency response.

```
fvtool(HighpassCheb1)
```



## Input Arguments

### **designSpecs** — Filter specification object

object

Filter specification object, specified as one of the following:

- `fdesign.bandpass`
- `fdesign.bandstop`

- `fdesign.highpass`
- `fdesign.lowpass`

## Output Arguments

### **chebOneFilter** — Type I Chebyshev IIR digital filter

System object

Type I Chebyshev IIR digital filter, returned as a System object. The System object and the values of its properties depend on the input `designSpecs` object and the other design options that you specify as inputs to the function.

## Version History

Introduced in R2011a

### **R2022b: Designs `dsp.SOSFilter` object instead of `dsp.BiquadFilter` object**

*Behavior changed in R2022b*

The `cheby1` design method now generates a `dsp.SOSFilter` object instead of the `dsp.BiquadFilter` object.

Initialize the default `fdesign.lowpass` filter specification object. Use the design function with the `SystemObject=true` flag to design a `dsp.SOSFilter` filter object.

```
designSpecs = fdesign.lowpass;
filtObj = design(designSpecs, 'cheby1', SystemObject=true)

filtObj =

    dsp.SOSFilter with properties:

        Structure: 'Direct form II'
    CoefficientSource: 'Property'
        Numerator: [5×3 double]
        Denominator: [5×3 double]
    HasScaleValues: true
        ScaleValues: [0.4117 0.3484 0.2445 0.1195 0.0258 0.8913]
```

Show all properties

To design a `dsp.BiquadFilter` object instead of the `dsp.SOSFilter` object, you can use the `UseLegacyBiquadFilter` flag and set it to `true`.

```
filtObj = design(designSpecs, 'cheby1', SystemObject=true, ...
    UseLegacyBiquadFilter=true)

filtObj =

    dsp.BiquadFilter with properties:

        Structure: 'Direct form II'
    SOSMatrixSource: 'Property'
        SOSMatrix: [5×6 double]
        ScaleValues: [6×1 double]
```

```
InitialConditions: 0  
OptimizeUnityScaleValues: true
```

Show all properties

### **See Also**

[butter](#) | [design](#) | [designmethods](#) | [designoptions](#) | [fdesign](#) | [cheby2](#) | [ellip](#) | [equiripple](#) | [kaiserwin](#)

## cheby2

Chebyshev Type II filter using specification object

### Syntax

```
chebTwoFilter = cheby2(designSpecs,SystemObject=true)
chebTwoFilter = cheby2(designSpecs,designoption=value,... SystemObject=true)
chebTwoFilter = design(designSpecs,'cheby2',SystemObject=true)
```

### Description

`chebTwoFilter = cheby2(designSpecs,SystemObject=true)` designs a type II Chebyshev IIR digital filter using specifications in the object `designSpecs`.

Depending on the filter specification object `designSpecs`, the `cheby2` design method might not be valid. Use `designmethods` with the filter specification object to determine if a Chebyshev type II filter design is possible.

```
designmethods(designSpecs,SystemObject=true)
```

`chebTwoFilter = cheby2(designSpecs,designoption=value,... SystemObject=true)` returns a type II Chebyshev IIR digital filter with one or more specified designed options and the corresponding values.

To view a list of available design options, run the `designoptions` function on the specification object. The function also lists the default design options the filter uses.

```
designoptions(designSpecs,'cheby2')
```

`chebTwoFilter = design(designSpecs,'cheby2',SystemObject=true)` is an alternative syntax for designing the type II Chebyshev IIR digital filter.

For complete help about using the `cheby2` design method for a specification object, `designSpecs`, enter the following at the MATLAB command prompt.

```
help(designSpecs,'cheby2')
```

### Examples

#### Design a Chebyshev Type II Filter

Design a type II Chebyshev IIR filter with lowpass and highpass frequency responses. The filter design procedure is:

- 1 Specify the filter design specifications using a `fdesign` function.
- 2 Pick a design method provided by the `designmethods` function.
- 3 To determine the available design options to choose from, use the `designoptions` function.
- 4 Design the filter using the `design` function.

## Lowpass Filter

Construct a default lowpass filter specification object using `fdesign.lowpass`.

```
designSpecs = fdesign.lowpass

designSpecs =
    lowpass with properties:

        Response: 'Lowpass'
        Specification: 'Fp,Fst,Ap,Ast'
        Description: {4x1 cell}
        NormalizedFrequency: 1
            Fpass: 0.4500
            Fstop: 0.5500
            Apass: 1
            Astop: 60
```

Determine the available design methods using the `designmethods` function. To design a type II Chebyshev filter, pick `cheby2`.

```
designmethods(designSpecs,SystemObject=true)

Design Methods that support System objects for class fdesign.lowpass (Fp,Fst,Ap,Ast):
```

```
butter
cheby1
cheby2
ellip
equiripple
ifir
kaiserwin
multistage
```

While designing the filter, you can specify additional design options. View a list of the options using the `designoptions` function. This function also shows the default design options the filter uses.

```
designoptions(designSpecs,'cheby2',SystemObject=true)

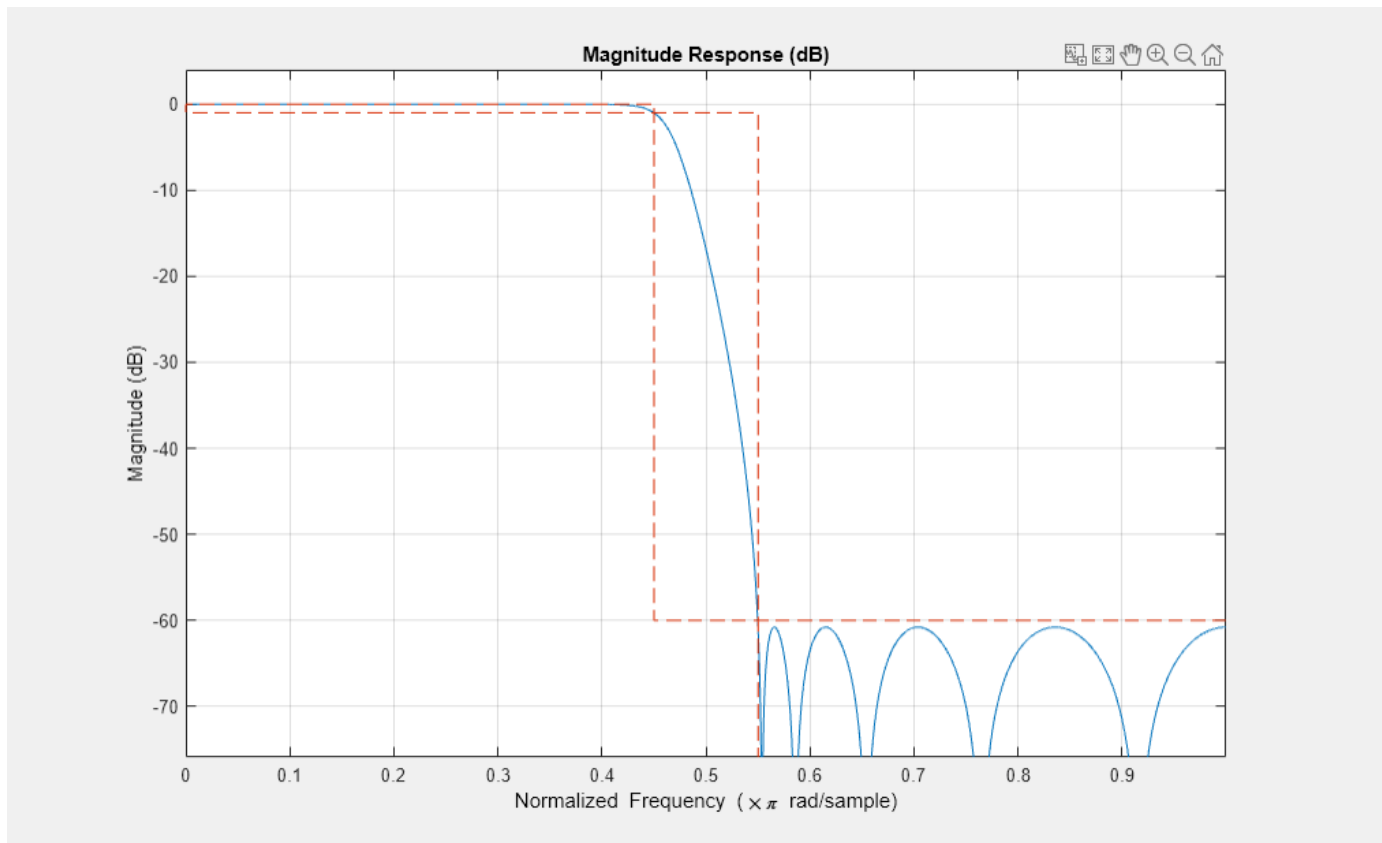
ans = struct with fields:
    FilterStructure: {'df1sos' 'df2sos' 'df1tsos' 'df2tsos' 'cascadeallpass' 'cascade...
    SOSScaleNorm: 'ustring'
    SOSScaleOpts: 'fdopts.sosscaling'
    MatchExactly: {'passband' 'stopband'}
    DefaultFilterStructure: 'df2sos'
    DefaultMatchExactly: 'stopband'
    DefaultSOSScaleNorm: ''
    DefaultSOSScaleOpts: [1x1 fdopts.sosscaling]
```

Use the `design` function to design the filter. Pass `'cheby2'` and the specifications given by the variable `designSpecs`, as input arguments. Specify the `'matchexactly'` design option to `'passband'` to ensure the performance of the filter in the passband.

```
LowpassCheb2 = design(designSpecs,'cheby2',matchexactly='passband',...
    SystemObject=true);
```

Use `fvtool` to view the frequency response of the designed filter.

```
fvtool(LowpassCheb2)
```



## Highpass Filter

Construct a highpass filter specification object using `fdesign.highpass`. Specify the filter order, stopband edge frequency, and the stopband attenuation to get the filter exactly as required.

```
designSpecs = fdesign.highpass('n,fst,ast',5,20,55,50);
```

Determine the available design methods. To design a type II Chebyshev IIR filter, pick `cheby2`.

```
designmethods(designSpecs,SystemObject=true)
```

Design Methods that support System objects for class `fdesign.highpass(N,Fst,Ast)`:

```
cheby2
```

While designing the filter, you can specify additional design options. View a list of the options using the `designoptions` function. This function also shows the default design options the filter uses.

```
HighpassCheb2 = design(designSpecs,'cheby2',SystemObject=true)
```

```
HighpassCheb2 =  
  dsp.SOSFilter with properties:
```

```

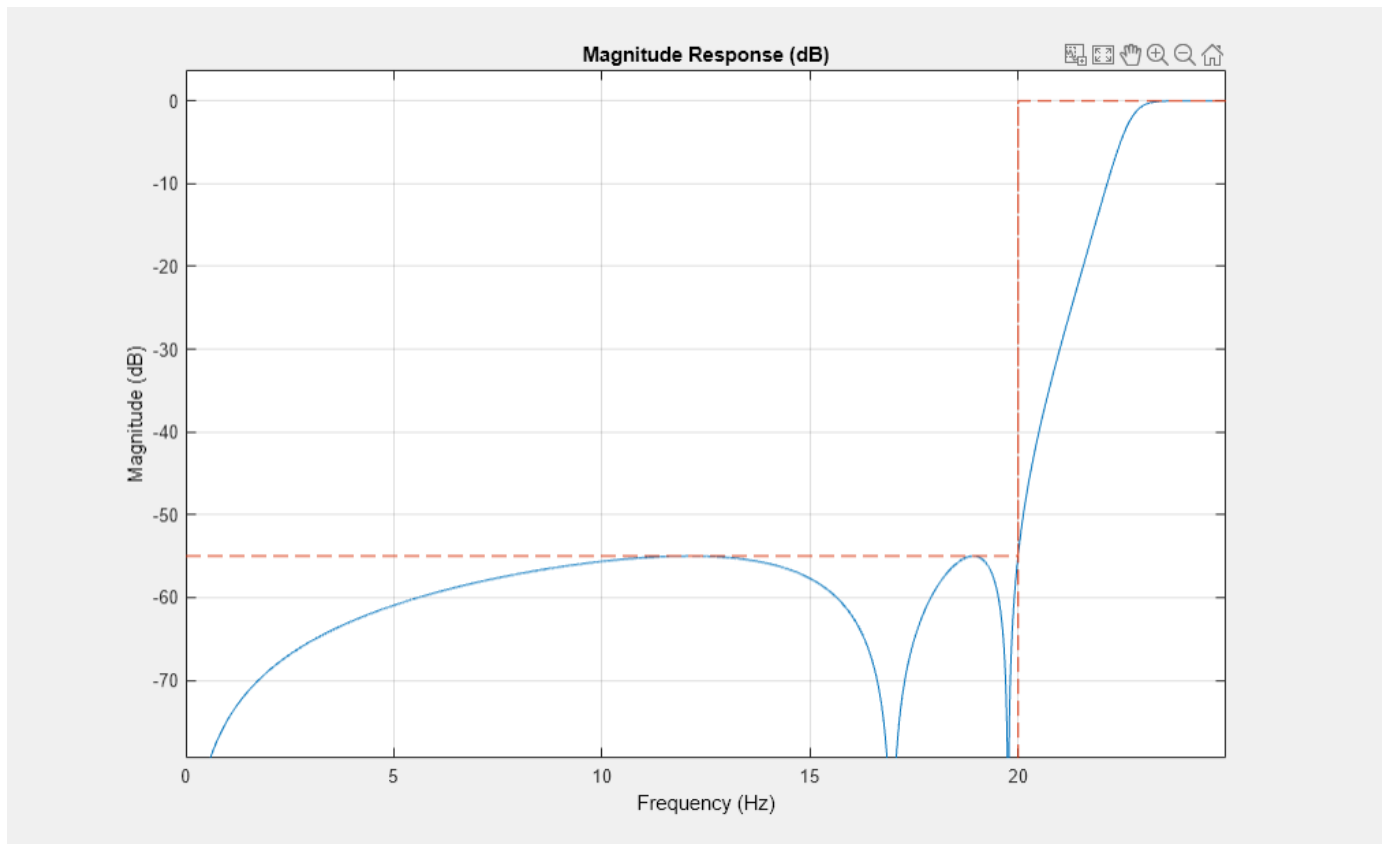
Structure: 'Direct form II'
CoefficientSource: 'Property'
Numerator: [3x3 double]
Denominator: [3x3 double]
HasScaleValues: true
ScaleValues: [0.1995 0.0880 0.1450 1]

```

Show all properties

Use `fvtool` to visualize the highpass frequency response.

```
fvtool(HighpassCheb2)
```



By design, `cheby2` returns filters that use second-order sections (SOS). For many applications, and for most fixed-point applications, SOS filters are particularly well-suited.

## Input Arguments

**designSpecs** — Filter specification object

object

Filter specification object, specified as one of the following:

- `fdesign.bandpass`
- `fdesign.bandstop`



- `fdesign.highpass`
- `fdesign.lowpass`

## Output Arguments

### **chebTwoFilter** — Type II Chebyshev IIR digital filter

System object

Type II Chebyshev IIR digital filter, returned as a filter System object. The System object and the values of its properties depend on the input `designSpecs` object and the other design options specified to the function.

## Version History

Introduced in R2011a

### **R2022b: Designs `dsp.SOSFilter` object instead of `dsp.BiquadFilter` object**

*Behavior changed in R2022b*

The `cheby2` design method now generates a `dsp.SOSFilter` object instead of the `dsp.BiquadFilter` object.

Initialize the default `fdesign.lowpass` filter specification object. Use the design function with the `SystemObject=true` flag to design a `dsp.SOSFilter` filter object.

```
designSpecs = fdesign.lowpass;
filtObj = design(designSpecs, 'cheby2', SystemObject=true)

filtObj =

    dsp.SOSFilter with properties:

        Structure: 'Direct form II'
    CoefficientSource: 'Property'
        Numerator: [5×3 double]
        Denominator: [5×3 double]
    HasScaleValues: true
        ScaleValues: [0.7003 0.5771 0.4734 0.3925 0.3466 1]

    Show all properties
```

To design a `dsp.BiquadFilter` object instead of the `dsp.SOSFilter` object, you can use the `UseLegacyBiquadFilter` flag and set it to `true`.

```
filtObj = design(designSpecs, 'cheby2', SystemObject=true, ...
    UseLegacyBiquadFilter=true)

filtObj =

    dsp.BiquadFilter with properties:

        Structure: 'Direct form II'
    SOSMatrixSource: 'Property'
        SOSMatrix: [5×6 double]
        ScaleValues: [6×1 double]
```

```
InitialConditions: 0  
OptimizeUnityScaleValues: true
```

Show all properties

### **See Also**

butter | cheby1 | design | designmethods | designoptions | fdesign | ellip | equiripple | kaiserwin

## cl2tf

Convert coupled allpass lattice to transfer function form

### Syntax

```
[b,a] = cl2tf(k1,k2)
[b,a] = cl2tf(k1,k2,beta)
[b,a,bp] = cl2tf(k1,k2)
[b,a,bp] = cl2tf(k1,k2,beta)
```

### Description

`[b,a] = cl2tf(k1,k2)` returns vectors of coefficients `b` and `a` when `k1` and `k2` are real vectors. `b` is the vector of coefficients corresponding to the numerator of the transfer function  $H(z)$ . `a` is the vector of coefficients corresponding to the denominator of the transfer function  $H(z)$ . `k1` and `k2` are real vectors corresponding to denominators of the allpass filters  $H1(z)$  and  $H2(z)$ . This is provided in the transfer function:

$$H(z) = B(z)/A(z) = \frac{1}{2}[H1(z) + H2(z)]$$

`[b,a] = cl2tf(k1,k2,beta)` returns the vectors of coefficients `b` and `a` corresponding to the numerator and denominator, respectively, of the transfer function  $H(z)$ , where `k1`, `k2`, and `beta` are complex vectors.

$$H(z) = B(z)/A(z) = \frac{1}{2}[-(\bar{\beta}) \cdot H1(z) + \beta \cdot H2(z)]$$

`[b,a,bp] = cl2tf(k1,k2)` also returns the vector `bp` of real coefficients corresponding to the numerator of the power-complementary filter  $G(z)$ , where `k1` and `k2` are real vectors.

$$G(z) = Bp(z)/A(z) = \frac{1}{2}[H1(z) - H2(z)]$$

`[b,a,bp] = cl2tf(k1,k2,beta)` also returns the vector of coefficients `bp` of possibly complex coefficients corresponding to the numerator of the power complementary filter  $G(z)$ , where `k1`, `k2`, and `beta` are complex.

$$G(z) = Bp(z)/A(z) = \frac{1}{2j}[-(\bar{\beta}) \cdot H1(z) + \beta \cdot H2(z)]$$

### Examples

#### Convert Coupled Allpass Filter Lattice to Transfer Function

Compute the reflection coefficient using `tf2cl` function.

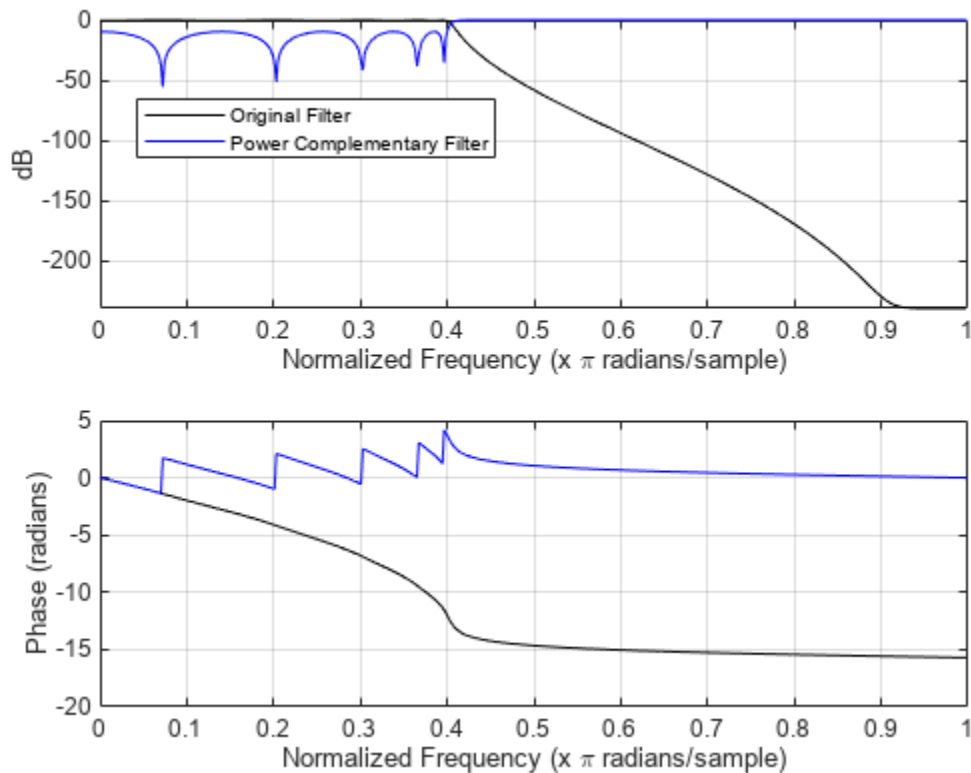
```
[b,a] = cheby1(10,.5,.4);
[k1,k2,beta] = tf2cl(b,a);
```

Construct the original and the power-complementary filters.

```
[num,den,numpc] = cl2tf(k1,k2,beta);
[h,w] = freqz(num,den);
hpc = freqz(numpc,den);
```

Plot the frequency response.

```
subplot(211)
plot(w./pi,20*log10(abs(h)), 'k')
hold on;
grid on;
plot(w./pi,20*log10(abs(hpc)), 'b')
xlabel('Normalized Frequency (x \pi radians/sample)');
ylabel('dB');
legend('Original Filter', 'Power Complementary Filter', ...
'Location', 'best');
subplot(212)
plot(w./pi,unwrap(angle(h)), 'k')
hold on;
grid on;
plot(w./pi,unwrap(angle(hpc)), 'b')
xlabel('Normalized Frequency (x \pi radians/sample)');
ylabel('Phase (radians)');
```



## Input Arguments

### **k1 — Reflection coefficients corresponding to allpass lattice structure of H1(z)**

vector

Reflection coefficients corresponding to allpass lattice structure of  $H1(z)$ , specified as a real- or complex-valued vector.

Data Types: `single` | `double`

### **k2 — Reflection coefficients corresponding to allpass lattice structure of H2(z)**

vector

Reflection coefficients corresponding to allpass lattice structure of  $H2(z)$ , specified as a real- or complex-valued vector.

Data Types: `single` | `double`

### **beta — Complex scalar of magnitude 1**

complex scalar

Complex scalar of magnitude 1.

Data Types: `single` | `double`

## Output Arguments

### **b — Numerator coefficients of transfer function**

vector

Coefficients of the numerator of the transfer function  $H(z)$ , returned as a real- or complex-valued vector.

Data Types: `double`

### **a — Denominator coefficients of transfer function**

vector

Coefficients of the denominator of the transfer function  $H(z)$ , returned as a real- or complex-valued vector.

Data Types: `double`

### **bp — Numerator coefficients of power-complementary filter G(z)**

vector

Coefficients of the numerator of the power-complementary filter  $G(z)$ , returned as a real- or complex-valued vector.

Data Types: `double`

## Version History

Introduced in R2011a

## References

- [1] Mitra, Sanjit Kumar, and James F. Kaiser, eds. *Handbook for Digital Signal Processing*. New York: Wiley, 1993.
- [2] Vaidyanathan, P. P. *Multirate Systems and Filter Banks*. Prentice-Hall Signal Processing Series. Englewood Cliffs, N.J: Prentice Hall, 1993. `CloseDeleteEdit`

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

All inputs must be constant. Expressions or variables are allowed if their values do not change.

### See Also

`tf2cl` | `tf2ca` | `ca2tf` | `tf2latc` | `latc2tf` | `iirpowcomp`

# coeffs

**Package:** dsp

Returns the filter System object coefficients in a structure

## Syntax

```
s = coeffs(sysobj)
s = coeffs(sysobj, 'Arithmetic', arithType)
```

## Description

`s = coeffs(sysobj)` returns the coefficients of filter System object, `sysobj`, in the structure `s`.

`s = coeffs(sysobj, 'Arithmetic', arithType)` returns filter coefficients for the filter System object `sysobj` with the arithmetic specified in `arithType`.

## Examples

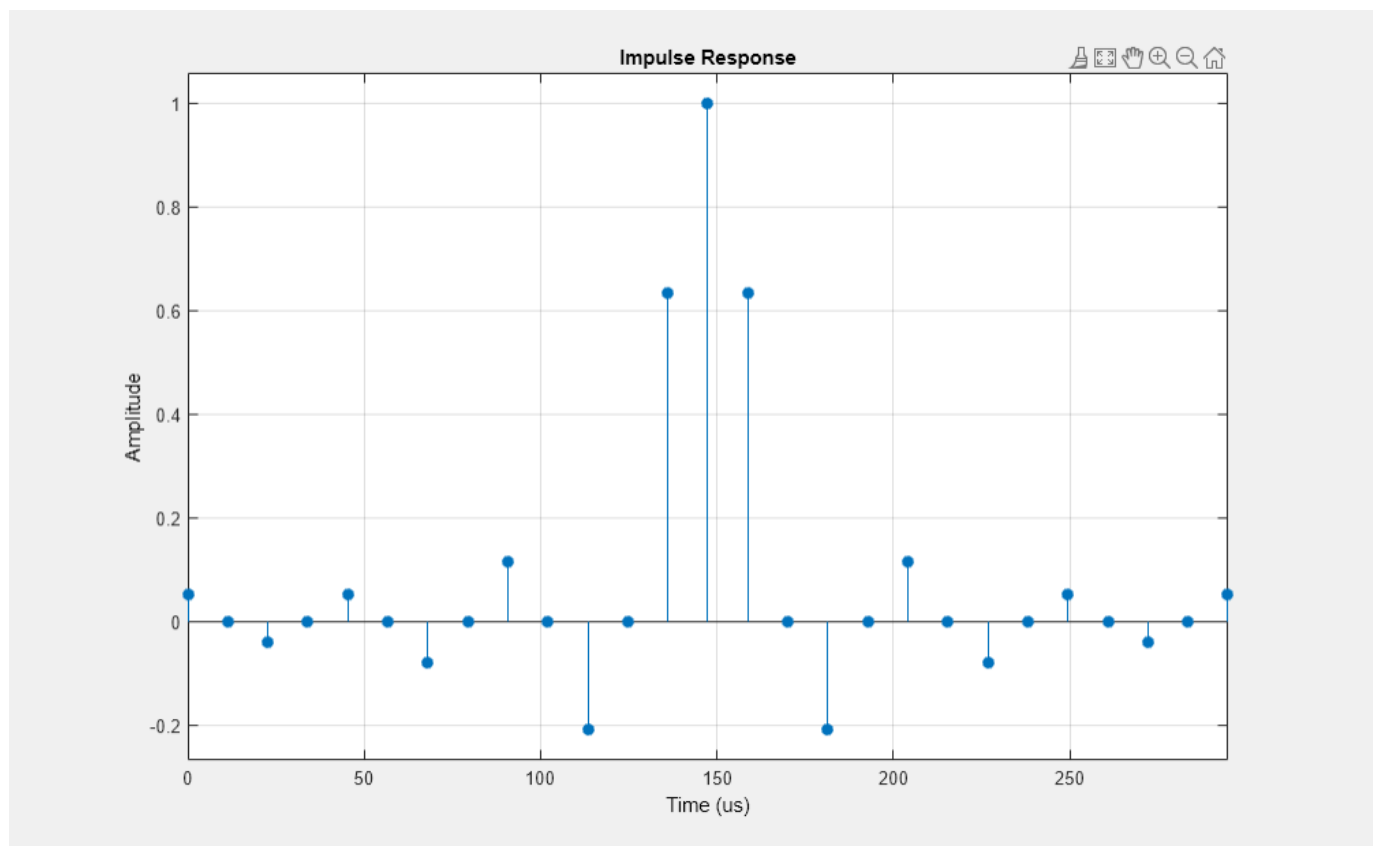
### Coefficients of an FIR Halfband Interpolator

```
FIRHalfbandInterp = dsp.FIRHalfbandInterpolator('Specification', ...
    'Filter order and transition width', 'FilterOrder', 26);
C = coeffs(FIRHalfbandInterp);
C.Numerator
```

```
ans = 1×27
```

```
    0.0525         0   -0.0379         0    0.0537         0   -0.0771         0    0.1172
```

```
% Impulse response of the filter
fvtool(FIRHalfbandInterp, 'impulse')
```



## Input Arguments

### **sysobj** — Input filter

filter System object

Input filter, specified as as one of the following filter System objects:

- `dsp.AllpassFilter`
- `dsp.AllpoleFilter`
- `dsp.BiquadFilter`
- `dsp.CICCompensationDecimator`
- `dsp.CICCompensationInterpolator`
- `dsp.CoupledAllpassFilter`
- `dsp.Differentiator`
- `dsp.FilterCascade`
- `dsp.FIRDecimator`
- `dsp.FIRFilter`
- `dsp.FIRHalfbandDecimator`
- `dsp.FIRHalfbandInterpolator`



- `dsp.FIRInterpolator`
- `dsp.FIRRateConverter`
- `dsp.FourthOrderSectionFilter`
- `dsp.HighpassFilter`
- `dsp.IIRFilter`
- `dsp.IIRHalfbandDecimator`
- `dsp.IIRHalfbandInterpolator`
- `dsp.LowpassFilter`
- `dsp.NotchPeakFilter`
- `dsp.SOSFilter`
- `dsp.VariableBandwidthFIRFilter`
- `dsp.VariableBandwidthIIRFilter`

### **arithType — Arithmetic type**

'double' (default) | 'single' | 'Fixed'

Arithmetic used in the filter analysis, specified as 'double', 'single', or 'Fixed'. When the arithmetic input is not specified and the filter System object is unlocked, the analysis tool assumes a double-precision filter. When the arithmetic input is not specified and the System object is locked, the function performs the analysis based on the data type of the locked input.

The 'Fixed' value applies to filter System objects with fixed-point properties only.

When the 'Arithmetic' input argument is specified as 'Fixed' and the filter object has the data type of the coefficients set to 'Same word length as input', the arithmetic analysis depends on whether the System object is unlocked or locked.

- unlocked -- The analysis object function cannot determine the coefficients data type. The function assumes that the coefficients data type is signed, has a 16-bit word length, and is auto scaled. The function performs fixed-point analysis based on this assumption.
- locked -- When the input data type is 'double' or 'single', the analysis object function cannot determine the coefficients data type. The function assumes that the data type of the coefficients is signed, has a 16-bit word length, and is auto scaled. The function performs fixed-point analysis based on this assumption.

To check if the System object is locked or unlocked, use the `isLocked` function.

When the arithmetic input is specified as 'Fixed' and the filter object has the data type of the coefficients set to a custom numeric type, the object function performs fixed-point analysis based on the custom numeric data type.

## **Output Arguments**

### **s — Structure**

struct

Structure with a single field, `Numerator`, containing filter coefficients.

## **Version History**

**Introduced in R2011a**

### **See Also**

#### **Topics**

"Analysis Methods for Filter System Objects" on page 3-2

# coeffs

**Package:** dsp

Coefficients of prototype lowpass filter

## Syntax

```
c = coeffs(obj)
```

## Description

`c = coeffs(obj)` returns the coefficients of the prototype lowpass filter in `dsp.Channelizer` and `dsp.ChannelSynthesizer` System objects.

## Examples

### Coefficients Of Channelizer

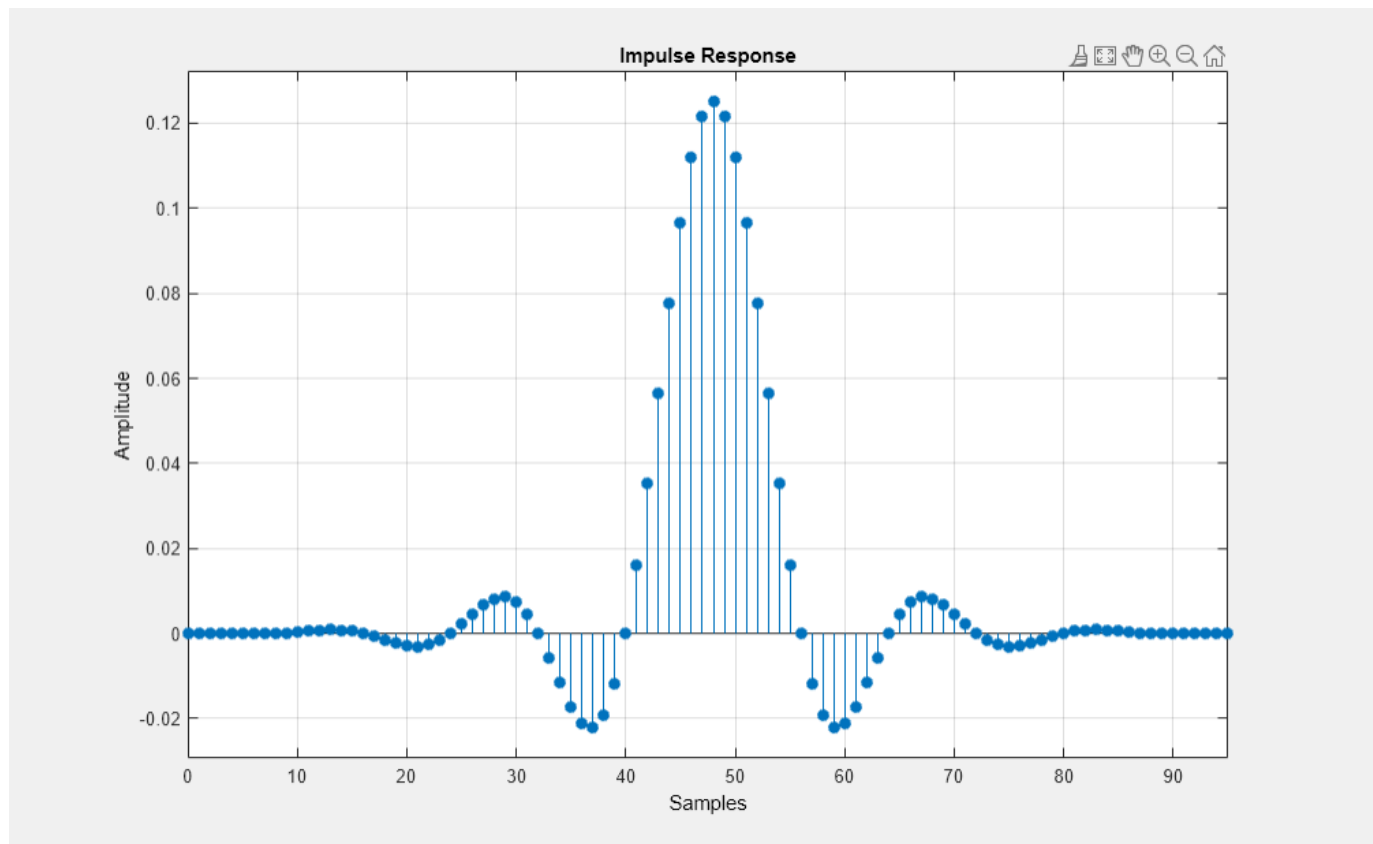
Determine the coefficients of the prototype lowpass filter in the `dsp.Channelizer` object using the `coeffs` function.

The `coeffs` function returns a structure, `c`. The structure field `Numerator` contains the coefficients as a row vector. The number of elements in the row vector, given by `c.Numerator`, equals the number of frequency bands times the number of coefficients per band. These values are given by the `NumFrequencyBands` and `NumTapsPerBand` properties of the `dsp.Channelizer` object.

```
channelizer = dsp.Channelizer;  
c = coeffs(channelizer);
```

Visualize the impulse response of the filter using `fvtool`.

```
fvtool(c.Numerator, 'impulse');
```



## Input Arguments

### **obj** — Input filter System object

`dsp.Channelizer` | `dsp.ChannelSynthesizer`

Input filter, specified as either a `dsp.Channelizer` or a `dsp.ChannelSynthesizer` System object.

Example: `channelizer = dsp.Channelizer;`

Example: `channelizer = dsp.ChannelSynthesizer`

## Output Arguments

### **c** — Lowpass filter coefficients

structure

Lowpass filter coefficients, returned as a structure. The structure field `Numerator` contains the coefficients as a row vector.

## Version History

**Introduced in R2016b**

## See Also

### Functions

`polyphase` | `tf` | `fvtool` | `freqz` | `bandedgeFrequencies` | `centerFrequencies` | `getFilters`

### Objects

`dsp.Channelizer` | `dsp.ChannelSynthesizer`

## **coeread**

Read Xilinx COE file

### **Syntax**

```
hd = coeread(filename)
```

### **Description**

`hd = coeread(filename)` extracts the Distributed Arithmetic FIR filter coefficients defined in the XILINX CORE Generator .COE file specified by `filename`. It returns a `dfilt` object, the fixed-point filter `hd`. If you do not provide the file type extension `.coe` with the `filename`, the function assumes the `.coe` extension.

### **Version History**

**Introduced in R2011a**

# coewrite

Write Xilinx COE file

## Syntax

```
coewrite(hd)
coewrite(hd,radix)
coewrite(...,filename)
```

## Description

`coewrite(hd)` writes a XILINX Distributed Arithmetic FIR filter coefficient .COE file which can be loaded into the XILINX CORE Generator. The coefficients are extracted from the fixed-point `dfilt` object `hd`. Your fixed-point filter must be a direct form FIR structure `dfilt` object with one section and whose Arithmetic property is set to `fixed`. You cannot export single-precision, double-precision, or floating-point filters as .coe files, nor multiple-section filters. To enable you to provide a name for the file, `coewrite` displays a dialog box where you fill in the file name. If you do not specify the name of the output file, the default file name is `untitled.coe`.

`coewrite(hd,radix)` indicates the radix (number base) used to specify the FIR filter coefficients. Valid radix values are 2 for binary, 10 for decimal, and 16 for hexadecimal (default).

`coewrite(...,filename)` writes a XILINX.COE file to `filename`. If you omit the file extension, `coewrite` adds the .coe extension to the name of the file.

The `coewrite` function always generates the XILINX.COE file in your current folder. To use this function, you must have write permission in your current folder.

## Examples

`coewrite` generates an ASCII text file that contains the filter coefficients in a format the XILINX CORE Generator can read and load. In this example, you create a 30th-order fixed-point filter and generate the .coe file that includes the filter coefficients as well as associated information about the filter.

```
b = firceqrip(30,0.4,[0.05 0.03]); hq = dfilt.dffir(b);
set(hq,'arithmetic','fixed'); coewrite(hq,10,'mycoefile');
```

The `coewrite` function generates the output file, `mycoefile.coe`, in your current folder. The .coe file contains the radix, coefficient width, and filter coefficients. The file reports the filter coefficients in column-major order. The radix, coefficient width, and filter coefficients are the minimum set of data needed in a .coe file.

## Version History

Introduced in R2011a

## constraincoeffwl

Constrain coefficient word length

### Syntax

```
Hq = constraincoeffwl(Hd,wordlength)
Hq = constraincoeffwl(Hd,wordlength,'Ntrials',N)
Hq = constraincoeffwl(Hd,wordlength,...,'NoiseShaping',NSFlag)
Hq = constraincoeffwl(Hd,wordlength,...,'Apasstol',Apasstol)
Hq = constraincoeffwl(Hd,wordlength,...,'Astoptol',Astoptol)
```

### Description

`Hq = constraincoeffwl(Hd,wordlength)` returns a fixed-point filter `Hq` meeting the design specifications of the single-stage or multistage FIR filter object `Hd` with a wordlength of at most `wordlength` bits. For multistage filters, `wordlength` can either be a scalar or vector. If `wordlength` is a scalar, the same word length is used for all stages. If `wordlength` is a vector, each stage uses the corresponding element in the vector. The vector length must equal the number of stages. `Hd` must be generated using `fdesign` and `design`. `constraincoeffwl` uses a stochastic noise-shaping procedure by default to minimize the wordlength. To obtain repeatable results on successive function calls, initialize the uniform random number generator `rand`

`Hq = constraincoeffwl(Hd,wordlength,'Ntrials',N)` specifies the number of Monte Carlo trials to use. `Hq` is first filter among the trials to meet the specifications in `Hd` with a wordlength of at most `wordlength`.

`Hq = constraincoeffwl(Hd,wordlength,...,'NoiseShaping',NSFlag)` enables or disables the stochastic noise-shaping procedure in the constraint of the word length. By default `NSFlag` is `true`. Setting `NSFlag` to `false` constrains the word length without using noise-shaping.

`Hq = constraincoeffwl(Hd,wordlength,...,'Apasstol',Apasstol)` specifies the passband ripple tolerance in dB. `'Apasstol'` defaults to `1e-4`.

`Hq = constraincoeffwl(Hd,wordlength,...,'Astoptol',Astoptol)` specifies the stopband tolerance in dB. `'Astoptol'` defaults to `1e-2`

You must have the Fixed-Point Designer software installed to use this function.

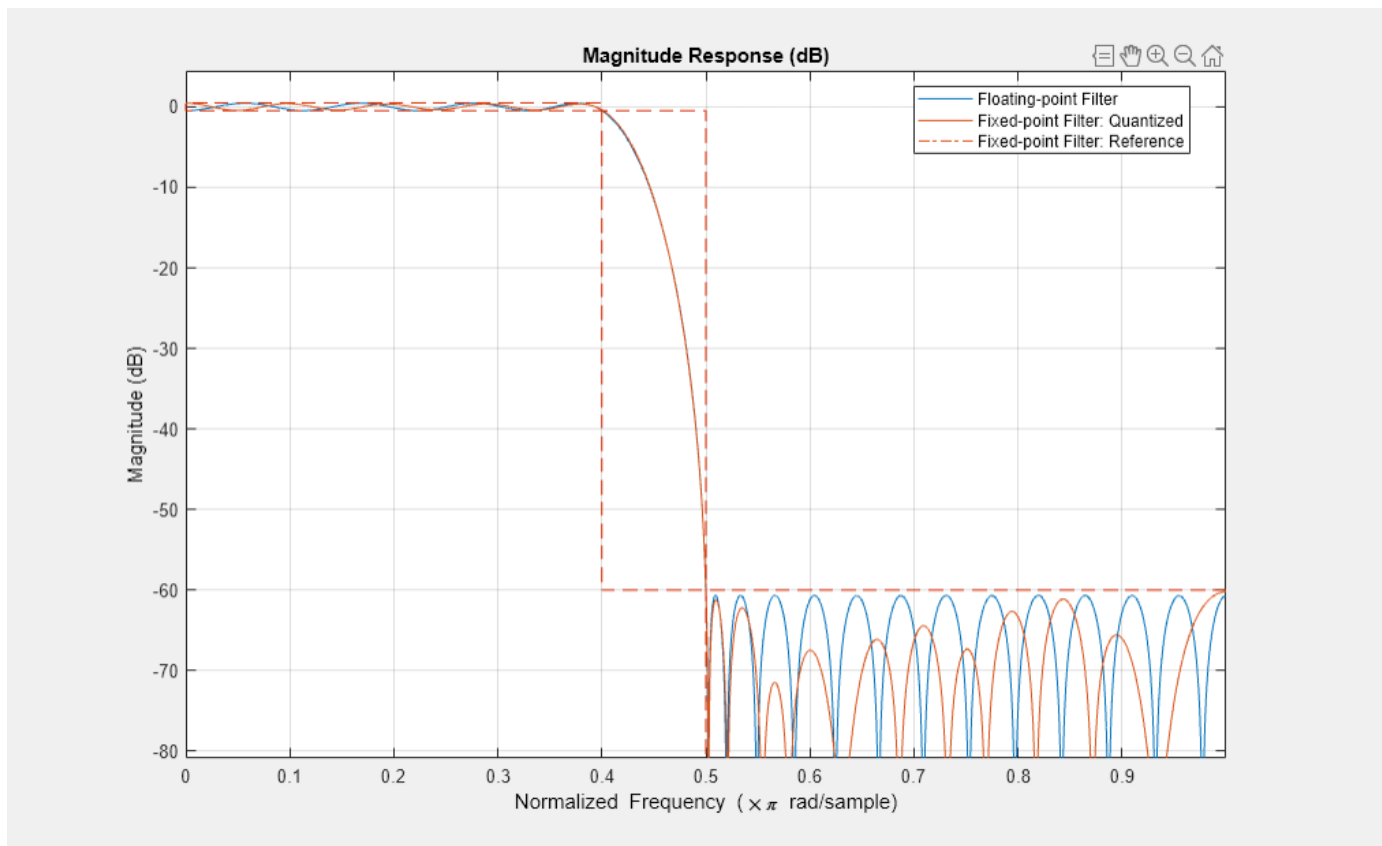
### Examples

#### Design Fixed-Point Filter

Design fixed-point filter with a word length of at most 11 bits using `constraincoeffwl`.

```
Hf = fdesign.lowpass('Fp,Fst,Ap,Ast',.4,.5,1,60);
Hd = design(Hf,'equiripple'); % 43 coefficients
Hq = constraincoeffwl(Hd,11); % 45 11-bit coefficients
hfvt = fvtool(Hd,Hq);
legend(hfvt,'Floating-point Filter','Fixed-point Filter');
```





## Version History

Introduced in R2011a

### See Also

[design](#) | [fdesign](#) | [maximizestopband](#) | [minimizecoeffwl](#) | [measure](#) | [rand](#)

### Topics

"Fixed-Point Overview"

## convert

Convert filter structure of discrete-time filter

### Syntax

```
hq = convert(hq,newstruct)
```

### Description

#### Discrete-Time Filters

`hq = convert(hq,newstruct)` returns a quantized filter whose structure has been transformed to the filter structure specified by `newstruct`. You can enter any one of the following quantized filter structures:

- 'antisymmetricfir': Antisymmetric finite impulse response (FIR)
- 'df1': Direct form I
- 'df1t': Direct form I transposed
- 'df1sos': Direct-Form I, Second-Order Sections
- 'df1tsos': Direct-Form I Transposed, Second-Order Sections
- 'df2': Direct form II
- 'df2t': Direct form II transposed. Default filter structure
- 'df2sos': Direct-Form II, Second-Order Sections
- 'df2tsos': Direct-Form II Transposed, Second-Order Sections
- 'dffir': FIR
- 'dffirt': Direct form FIR transposed
- 'latcallpass': Lattice allpass
- 'latticeca': Lattice coupled-allpass
- 'latticecapc': Lattice coupled-allpass power-complementary
- 'latticear': Lattice autoregressive (AR)
- 'latticemax': Lattice moving average (MA) maximum phase
- 'latticemin': Lattice moving average (MA) minimum phase
- 'latticearma': Lattice ARMA
- 'statespace': Single-input/single-output state-space
- 'symmetricfir': Symmetric FIR. Even and odd forms

All filters can be converted to the following structures:

- 'df1': Direct form I
- 'df1t': Direct form I transposed
- 'df1sos': Direct-Form I, Second-Order Sections
- 'df1tsos': Direct-Form I Transposed, Second-Order Sections

- 'df2': Direct form II
- 'df2t': Direct form II transposed. Default filter structure
- 'df2sos': Direct-Form II, Second-Order Sections
- 'df2tsos': Direct-Form II Transposed, Second-Order Sections
- 'statespace': Single-input/single-output state-space
- 'symmetricfir': Symmetric FIR. Even and odd forms

For the following filter classes, you can specify other conversions as well:

- Minimum phase FIR filters can be converted to `lattice` `mamin`
- Maximum phase FIR filters can be converted to `lattice` `mamax`
- Allpass filters can be converted to `latcallpass`

`convert` generates an error when you specify a conversion that is not possible.

## Examples

### Convert Direct-Form II Transposed Structure to Direct-Form I

```
[b,a]=ellip(5,3,40,.7); hq = dfilt.df2t(b,a)
```

```
hq =
  FilterStructure: 'Direct-Form II Transposed'
  Arithmetic: 'double'
  Numerator: [1x6 double]
  Denominator: [1x6 double]
  PersistentMemory: false
```

```
hq2 = convert(hq,'df1')
```

```
hq2 =
  FilterStructure: 'Direct-Form I'
  Arithmetic: 'double'
  Numerator: [1x6 double]
  Denominator: [1x6 double]
  PersistentMemory: false
```

## Version History

Introduced in R2011a

## cost

**Package:** dsp

Estimate cost of implementing filter System object

### Syntax

```
c = cost(sysobj)
c = cost(sysobj, 'Arithmetic', arithType)
```

### Description

`c = cost(sysobj)` returns a structure, `c`, whose fields contain information about the computational cost of implementing the filter System object, `sysobj`.

`c = cost(sysobj, 'Arithmetic', arithType)` returns a cost estimate `c` for the filter System object `sysobj` in the arithmetic specified by `arithType`.

### Examples

#### Cost of FIR Filter

This example shows how to compute the cost of implementing an FIR Filter created using `dsp.FIRFilter` object.

```
Fs = 8000; Fcutoff = 2000;
firFilt = dsp.FIRFilter('Numerator', fir1(130, Fcutoff/(Fs/2)));
cost(firFilt)
```

```
ans = struct with fields:
    NumCoefficients: 131
    NumStates: 130
    MultiplicationsPerInputSample: 131
    AdditionsPerInputSample: 130
```

### Input Arguments

#### **sysobj** — Input filter

filter System object

Input filter, specified as one of the following filter System objects:

- `dsp.AllpassFilter`
- `dsp.AllpoleFilter`
- `dsp.BiquadFilter`
- `dsp.CICCompensationDecimator`

- `dsp.CICCompensationInterpolator`
- `dsp.CICDecimator`
- `dsp.CICInterpolator`
- `dsp.CoupledAllpassFilter`
- `dsp.Differentiator`
- `dsp.FarrowRateConverter`
- `dsp.FilterCascade`
- `dsp.FIRDecimator`
- `dsp.FIRFilter`
- `dsp.FIRHalfbandDecimator`
- `dsp.FIRHalfbandInterpolator`
- `dsp.FIRInterpolator`
- `dsp.FIRRateConverter`
- `dsp.FourthOrderSectionFilter`
- `dsp.HighpassFilter`
- `dsp.IIRFilter`
- `dsp.IIRHalfbandDecimator`
- `dsp.IIRHalfbandInterpolator`
- `dsp.LowpassFilter`
- `dsp.NotchPeakFilter`
- `dsp.SOSFilter`
- `dsp.VariableBandwidthFIRFilter`
- `dsp.VariableBandwidthIIRFilter`

#### **arithType — Arithmetic type**

`'double'` (default) | `'single'` | `'Fixed'`

Arithmetic used in the filter analysis, specified as `'double'`, `'single'`, or `'Fixed'`. When the arithmetic input is not specified and the filter System object is unlocked, the analysis tool assumes a double-precision filter. When the arithmetic input is not specified and the System object is locked, the function performs the analysis based on the data type of the locked input.

The `'Fixed'` value applies to filter System objects with fixed-point properties only.

When the `'Arithmetic'` input argument is specified as `'Fixed'` and the filter object has the data type of the coefficients set to `'Same word length as input'`, the arithmetic analysis depends on whether the System object is unlocked or locked.

- `unlocked` -- The analysis object function cannot determine the coefficients data type. The function assumes that the coefficients data type is signed, has a 16-bit word length, and is auto scaled. The function performs fixed-point analysis based on this assumption.
- `locked` -- When the input data type is `'double'` or `'single'`, the analysis object function cannot determine the coefficients data type. The function assumes that the data type of the coefficients is signed, has a 16-bit word length, and is auto scaled. The function performs fixed-point analysis based on this assumption.

To check if the System object is locked or unlocked, use the `isLocked` function.

When the arithmetic input is specified as 'Fixed' and the filter object has the data type of the coefficients set to a custom numeric type, the object function performs fixed-point analysis based on the custom numeric data type.

## Output Arguments

### **c** — cost estimate

struct

Cost estimate, `c` contains the following fields:

| Estimated Value               | Description   |
|-------------------------------|---|
| NumCoefficients               | Number of filter coefficients (excluding coefficients with values 0, 1 or -1) |
| NumStates                     | Number of filter states   |
| MultiplicationsPerInputSample | Number of multiplication operations performed for each input sample           |
| AdditionsPerInputSample       | Number of addition operations performed for each input sample                 |

## Version History

Introduced in R2011a

## See Also

### Topics

"Analysis Methods for Filter System Objects" on page 3-2

# cost

**Package:** dsp

Implementation cost of the complex bandpass decimator

## Syntax

```
c = cost(cbd)
```

## Description

`c = cost(cbd)` returns a structure, `c`, whose fields contain information about the computation cost of implementing the complex bandpass decimator object, `cbd`.

## Examples

### Compute Cost of Complex Bandpass Decimator

Compute the implementation cost of a complex bandpass decimator using the `cost` function.

Create a `dsp.ComplexBandpassDecimator` object. Set the `DecimationFactor` to 12, the `CenterFrequency` to 5000 Hz, and the `SampleRate` to 44,100 Hz.

```
cbp = dsp.ComplexBandpassDecimator(12,5000,44100)
```

```
cbp =
  dsp.ComplexBandpassDecimator with properties:
      CenterFrequency: 5000
      Specification: 'Decimation factor'
      DecimationFactor: 12
      StopbandAttenuation: 80
      TransitionWidth: 100
      MinimizeComplexCoefficients: true
      SampleRate: 44100
```

Compute the implementation cost of `cbp` using the `cost` function.

```
c = cost(cbp)

c = struct with fields:
      NumCoefficients: 201
      NumStates: 379
      RealMultiplicationsPerInputSample: 44.3333
      RealAdditionsPerInputSample: 43.8333
```

## Input Arguments

### **cbd** — Filter System object

`dsp.ComplexBandpassDecimator`

Filter System object, specified as a `dsp.ComplexBandpassDecimator` System object.

## Output Arguments

### **c** — Cost estimate

structure

Cost estimate containing these fields:

| Estimated Value                   | Description   |
|-----------------------------------|---|
| NumCoefficients                   | Number of filter coefficients (excluding coefficients with values 0, 1 or -1) |
| NumStates                         | Number of filter states   |
| RealMultiplicationsPerInputSample | Number of real multiplication operations performed for each input sample      |
| RealAdditionsPerInputSample       | Number of real addition operations performed for each input sample            |

The function assumes that a complex-by-complex multiplication requires 3 real multiplications and 5 real additions.

## Version History

Introduced in R2018a

## See Also

### Functions

`freqz` | `info` | `visualizeFilterStages`

### Objects

`dsp.ComplexBandpassDecimator`



# cost

**Package:** dsp

Implementation cost of the sample rate converter

## Syntax

```
c = cost(src)
```

## Description

`c = cost(src)` returns a structure, `c`, whose fields contain information about the computational cost of implementing a multistage sample rate converter, `src`.

## Examples

### Computational Cost of Sample Rate Converter

Create `src`, a multistage sample rate converter with default values. `src` combines three filter stages to convert from 192 kHz to 44.1 kHz. Determine its computational cost: the number of coefficients, the number of states, the number of multiplications per unit sample, and the number of additions per unit sample.

```
src = dsp.SampleRateConverter;  
cst = cost(src)  
  
cst = struct with fields:  
    NumCoefficients: 8631  
    NumStates: 138  
    MultiplicationsPerInputSample: 27.6672  
    AdditionsPerInputSample: 26.6875
```

Repeat the computation allowing a tolerance of 10% in the output sample rate.

```
src.OutputRateTolerance = 0.1;  
ctl = cost(src)  
  
ctl = struct with fields:  
    NumCoefficients: 44  
    NumStates: 80  
    MultiplicationsPerInputSample: 14.2500  
    AdditionsPerInputSample: 13.5000
```

## Input Arguments

**src** — Multistage sample rate converter

SampleRateConverter System object

Multistage sample rate converter, specified as a `dsp.SampleRateConverter` System object.

## Output Arguments

### c — Output structure

structure

Output structure with information about the computational cost of `src`:

| Estimated Value               | Description   |
|-------------------------------|---|
| NumCoefficients               | Number of filter coefficients (excluding coefficients with values 0, 1 or -1) |
| NumStates                     | Number of filter states   |
| MultiplicationsPerInputSample | Number of multiplication operations performed for each input sample           |
| AdditionsPerInputSample       | Number of addition operations performed for each input sample                 |

## Version History

Introduced in R2014b

### See Also

#### Functions

`freqz` | `getActualOutputRate` | `getFilters` | `info` | `visualizeFilterStages` | `getRateChangeFactors`

#### Objects

`dsp.SampleRateConverter`

# getFilters

**Package:** dsp

Obtain single-stage filters

## Syntax

```
c = getFilters(src)
```

## Description

`c = getFilters(src)` returns the multirate filters cascaded together in `src` to perform the overall sample rate conversion. The result is a `FilterCascade` structure, `c`. Each field of `c` holds the filter used at a particular stage and gives access to its coefficients and rate-change factors.

## Examples

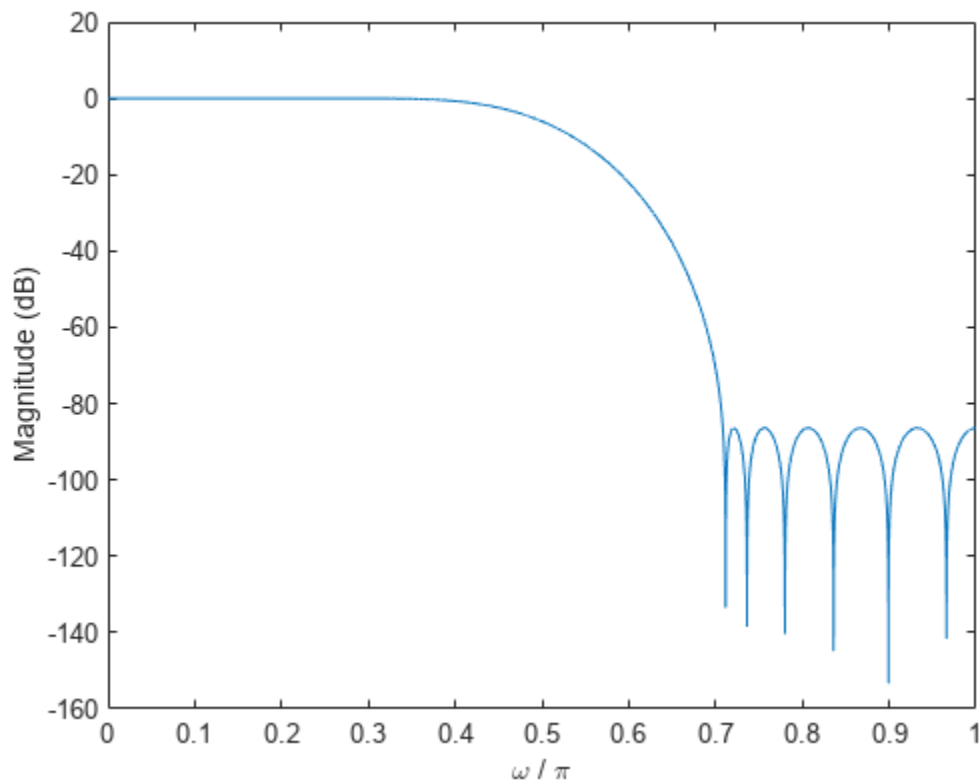
### Single-Stage Filters

Create `src`, a multistage sample rate converter with default properties. `src` converts between 192 kHz and 44.1 kHz. Find the individual filters that are cascaded together to perform the conversion.

```
src = dsp.SampleRateConverter;  
c = getFilters(src);
```

Visualize the frequency response of the decimator used in the first stage of the process.

```
m = c.Stage1;  
  
[h,w] = freqz(m);  
plot(w/pi,20*log10(abs(h)))  
xlabel('\omega / \pi')  
ylabel('Magnitude (dB)')
```



## Input Arguments

**src** — Multistage sample rate converter

`dsp.SampleRateConverter` System object

Multistage sample rate converter, specified as a `dsp.SampleRateConverter` System object.

## Output Arguments

**c** — Single-stage filters

`FilterCascade` structure

Single-stage filters, returned as a `FilterCascade` structure.

## Version History

Introduced in R2014b

## See Also

### Functions

`cost` | `freqz` | `getActualOutputRate` | `info` | `visualizeFilterStages` | `getRateChangeFactors`

**Objects**

dsp.SampleRateConverter

## info

**Package:** dsp

Display information about sample rate converter

### Syntax

```
s = info(src)
```

### Description

`s = info(src)` displays information about the multistage `SampleRateConverter` System object, `src`.

### Examples

#### Default Multistage Sample Rate Converter

Create a multistage sample rate converter with default properties, corresponding to the combined three filter stages used to convert from 192 kHz to 44.1 kHz.

```
src = dsp.SampleRateConverter

src =
  dsp.SampleRateConverter with properties:

      InputSampleRate: 192000
      OutputSampleRate: 44100
      OutputRateTolerance: 0
      Bandwidth: 40000
      StopbandAttenuation: 80
```

Display information about the design.

```
info(src)

ans =
  'Overall Interpolation Factor    : 147
  Overall Decimation Factor      : 640
  Number of Filters              : 3
  Multiplications per Input Sample: 27.667188
  Number of Coefficients         : 8631
  Filters:
    Filter 1:
      dsp.FIRDecimator - Decimation Factor : 2
    Filter 2:
      dsp.FIRDecimator - Decimation Factor : 2
    Filter 3:
      dsp.FIRRateConverter - Interpolation Factor: 147
                          - Decimation Factor   : 160
```

## Input Arguments

### **src** — Multistage sample rate converter

`dsp.SampleRateConverter` System object

Multistage sample rate converter, specified as a `dsp.SampleRateConverter` System object.

## Output Arguments

### **s** — Store filter information

character array

Filter information, returned as a character array with the following fields.

- Overall Interpolation Factor
- Overall Decimation Factor
- Number of Filters
- Multiplications Per Input Sample
- Number of Coefficients
- Filters

## Version History

Introduced in R2014b

## See Also

### Functions

`cost` | `freqz` | `getActualOutputRate` | `getFilters` | `visualizeFilterStages` | `getRateChangeFactors`

### Objects

`dsp.SampleRateConverter`

## cumsec

**Package:** dsp

Cumulative second-order section of the biquadratic filter

### Syntax

```
sect = cumsec(sysobj)
sect = cumsec(sysobj,indices)
sect = cumsec(sysobj,indices,secondary)
cumsec(sysobj,...)
sect = cumsec(sysobj,'Arithmetic',arithType)
```

### Description

`sect = cumsec(sysobj)` returns a cell array, `sect`, which contains cumulative sections of the biquadratic filter System object, `sysobj`. Each element in `sect` is a filter with the structure of the original filter. The first element is the first filter section of the biquadratic filter. The second element of `sect` is a filter that represents the combination of the first and second sections of the biquadratic filter. The third element of `sect` is a filter which combines sections 1, 2, and 3 of the biquadratic filter. This pattern continues until the final element of `sect` contains all the sections of the biquadratic filter and should be identical to the biquadratic filter.

`sect = cumsec(sysobj,indices)` returns the cumulative sections of the biquadratic filter object whose indices in the original filter are in the vector `indices`.

`sect = cumsec(sysobj,indices,secondary)` uses the secondary scaling points `secondary` in the sections to determine where the sections should be split when `secondary` is true. `secondary` is false by default. This option only applies for biquadratic filter objects with 'Direct form II' and 'Direct form I transposed' structures. For these structures, the secondary scaling points refer to the location between the recursive and the nonrecursive part, that is the 'middle' of the section.

`cumsec(sysobj,...)` plots the magnitude response of the cumulative sections using `fvtool`.

`sect = cumsec(sysobj,'Arithmetic',arithType)` returns the cumulative sections of the biquadratic filter object with the arithmetic specified in `arithType`.

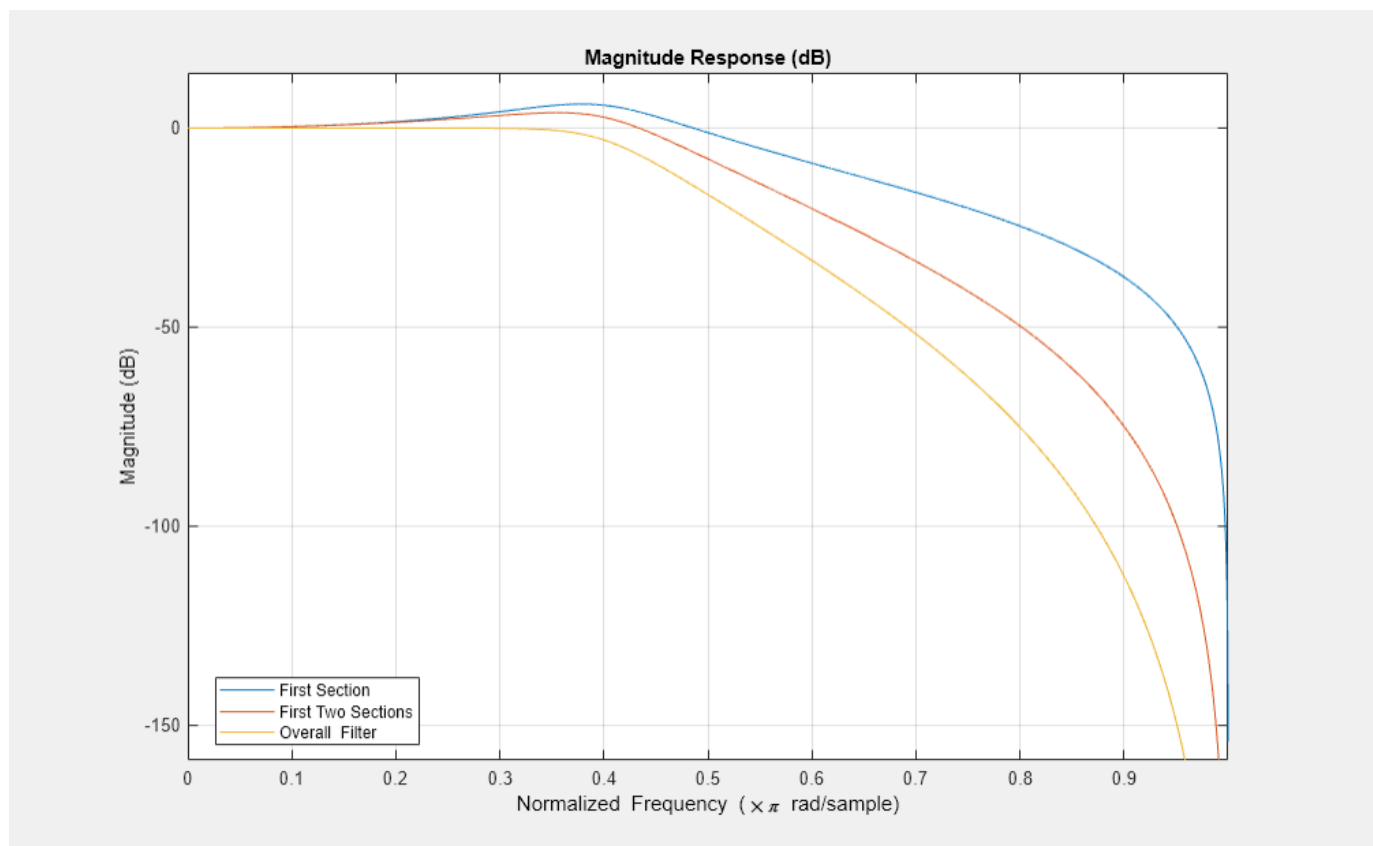
### Examples

#### Frequency Response of SOS Filter

This example plots the relative responses of the sections of a sixth-order filter with three sections. Each curve adds one more section to form the filter response.

```
Lowpass = fdesign.lowpass('n,fc',6,.4); ButterLowpass = butter(Lowpass,'SystemObject',true);
CumSections = cumsec(ButterLowpass); hfvt = fvtool(CumSections{1},CumSections{2},CumSections{3})
legend(hfvt,'First Section','First Two Sections','Overall Filter');
```





## Input Arguments

### **sysobj** — Input filter object

`dsp.BiquadFilter` System object | `dsp.SOSFilter` System object

Input filter, specified as one of the following System objects:

- `dsp.BiquadFilter`
- `dsp.SOSFilter`

### **indices** — Filter indices

scalar | row vector

Filter indices. Use `indices` to specify the filter sections `cumsec` uses to compute the cumulative responses.

### **secondary** — Flag to use secondary scaling points

`false` (default) | `true`

This option applies only when `sysobj` has the `df2sos` and `df1tsos` structures. For these second-order section structures, the secondary scaling points refer to the scaling locations between the recursive and the nonrecursive parts of the section (the "middle" of the section). Argument `secondary` accepts either `true` or `false`. By default, `secondary` is `false`.

**arithType – Arithmetic type**`'double' (default) | 'single' | 'Fixed'`

Arithmetic used in the filter analysis, specified as `'double'`, `'single'`, or `'Fixed'`. When the arithmetic input is not specified and the filter System object is unlocked, the analysis tool assumes a double-precision filter. When the arithmetic input is not specified and the System object is locked, the function performs the analysis based on the data type of the locked input.

The `'Fixed'` value applies to filter System objects with fixed-point properties only.

When the `'Arithmetic'` input argument is specified as `'Fixed'` and the filter object has the data type of the coefficients set to `'Same word length as input'`, the arithmetic analysis depends on whether the System object is unlocked or locked.

- `unlocked` -- The analysis object function cannot determine the coefficients data type. The function assumes that the coefficients data type is signed, has a 16-bit word length, and is auto scaled. The function performs fixed-point analysis based on this assumption.
- `locked` -- When the input data type is `'double'` or `'single'`, the analysis object function cannot determine the coefficients data type. The function assumes that the data type of the coefficients is signed, has a 16-bit word length, and is auto scaled. The function performs fixed-point analysis based on this assumption.

To check if the System object is locked or unlocked, use the `isLocked` function.

When the arithmetic input is specified as `'Fixed'` and the filter object has the data type of the coefficients set to a custom numeric type, the object function performs fixed-point analysis based on the custom numeric data type.

## Version History

Introduced in R2011a

### See Also

**Functions**`scale | scalecheck | scaleopts`**Objects**`dsp.BiquadFilter | dsp.SOSFilter`**Topics**

“Analysis Methods for Filter System Objects” on page 3-2

# deleteCursor

**Package:** dsp

Delete Logic Analyzer cursor

## Syntax

```
deleteCursor(scope, tag)
```

## Description

`deleteCursor(scope, tag)` deletes the Logic Analyzer cursor specified by the input tag.

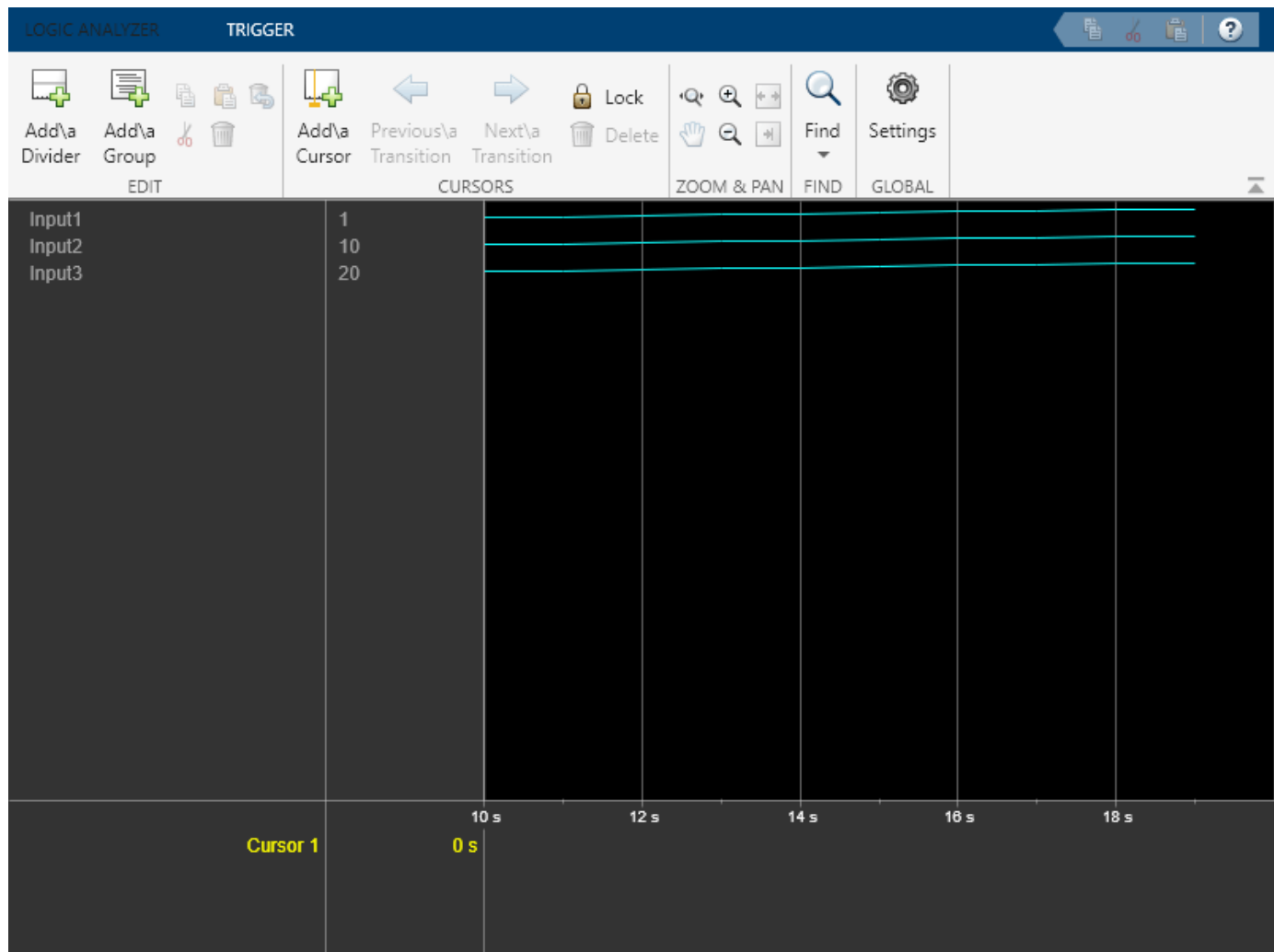
## Examples

### Modify Logic Analyzer Cursors Programmatically

This example shows how to use functions to create, manipulate, and delete cursors in a `dsp.LogicAnalyzer` object.

### Create Logic Analyzer and Signals

```
scope = dsp.LogicAnalyzer('NumInputPorts',3);  
for ii = 1:20  
    scope(ii,10*ii,20*ii);  
end
```



### Add Cursor

Add a cursor at 15 seconds and show the cursor information.

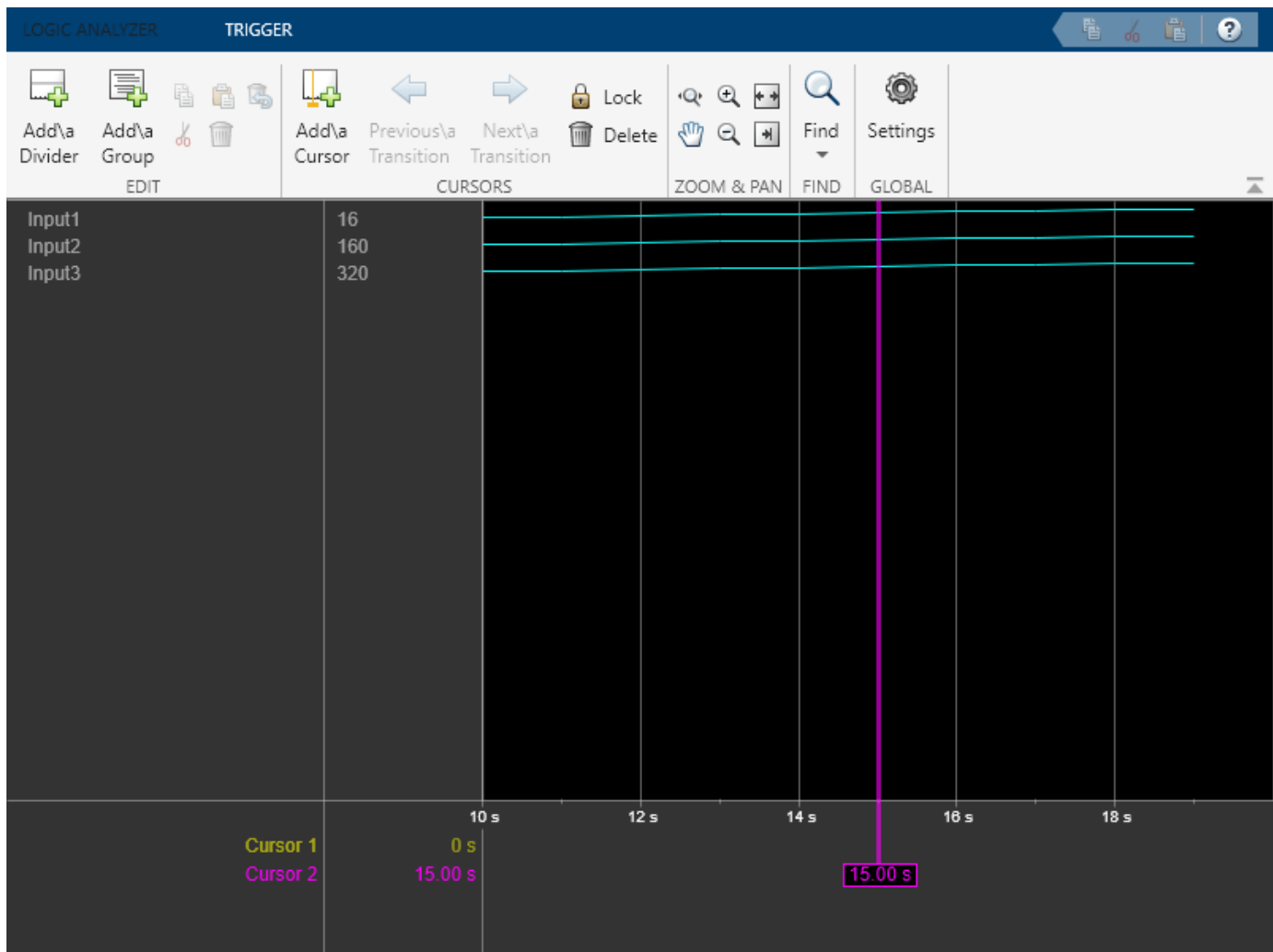
```
cursor = addCursor(scope, 'Location', 15, 'Color', 'Cyan');
getCursorInfo(scope, cursor)
```

```
ans = struct with fields:
    Location: 15
    Color: [0 1 1]
    Locked: 0
    Tag: 'C2'
```

### Modify Cursor

Change the cursor color to magenta.

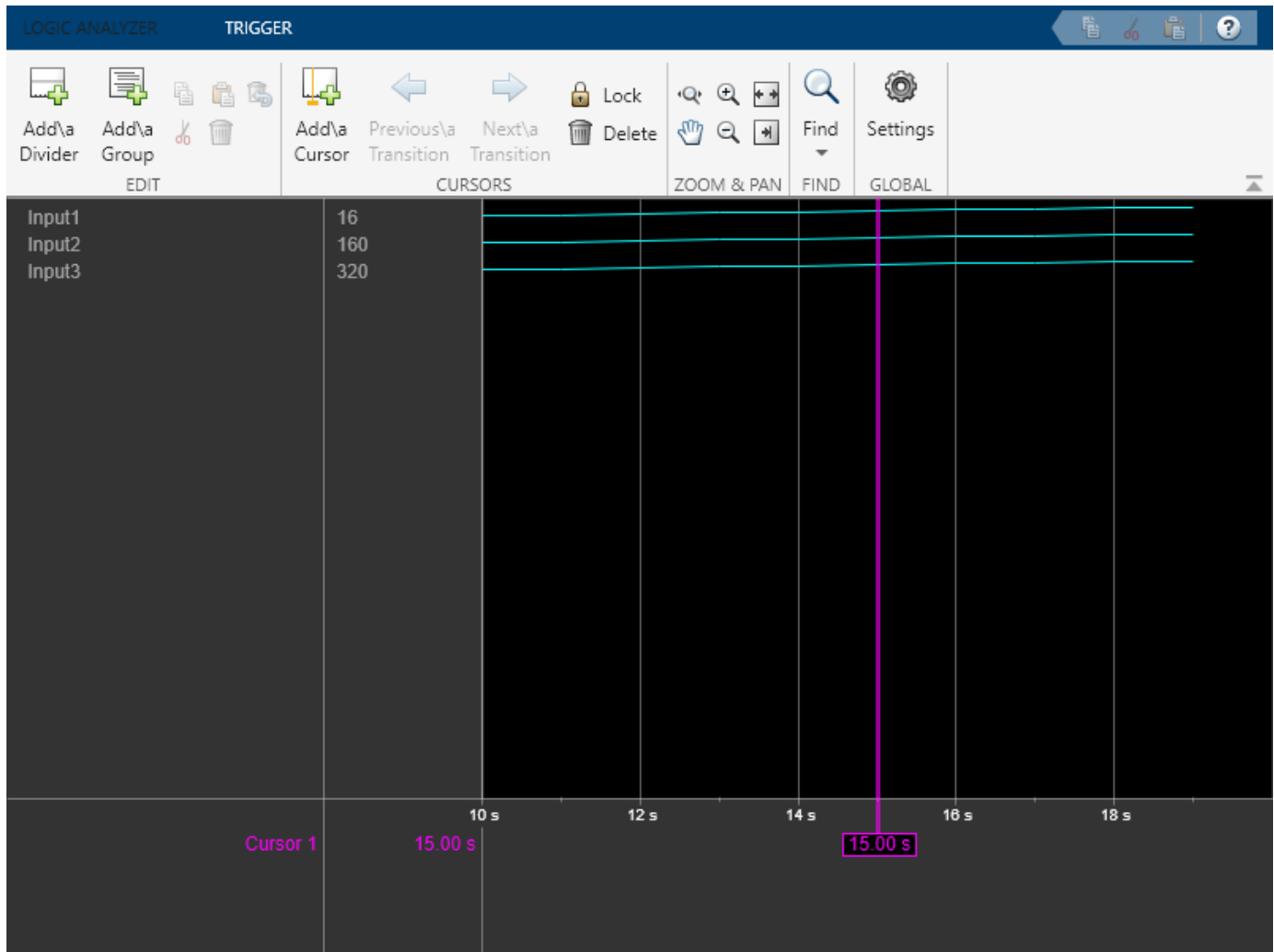
```
hide(scope)
modifyCursor(scope, cursor, 'Color', 'Magenta')
show(scope)
```



### Remove Cursor

Delete the yellow cursor at 0 seconds.

```
hide(scope)
tags = getCursorTags(scope);
deleteCursor(scope, tags{1});
show(scope)
```



## Input Arguments

**scope** — Logic Analyzer object from which you want to delete a cursor

`dsp.LogicAnalyzer` object handle

The Logic Analyzer object from which you want to delete a cursor, specified as a handle to the `dsp.LogicAnalyzer` object.

**tag** — Tag identifying which cursor to delete

randomly assigned character vector

The tag identifying which cursor to delete, specified as a randomly assigned character vector.

Example: `deleteCursor(scope, tag)` deletes a cursor from Logic Analyzer.

Data Types: `char` | `string`

## Version History

Introduced in R2013a

**See Also**

`dsp.LogicAnalyzer` | `addCursor` | `getCursorTags` | `getCursorInfo` | `modifyCursor` | `deleteDisplayChannel`

## deleteDisplayChannel

**Package:** dsp

Delete Logic Analyzer channel

### Syntax

```
deleteDisplayChannel(scope, tag)
```

### Description

`deleteDisplayChannel(scope, tag)` deletes the display channel, either a wave or a divider, specified by the input tag.

### Examples

#### Manipulate Logic Analyzer Programmatically

Use functions to construct and manipulate a `dsp.LogicAnalyzer` System object.

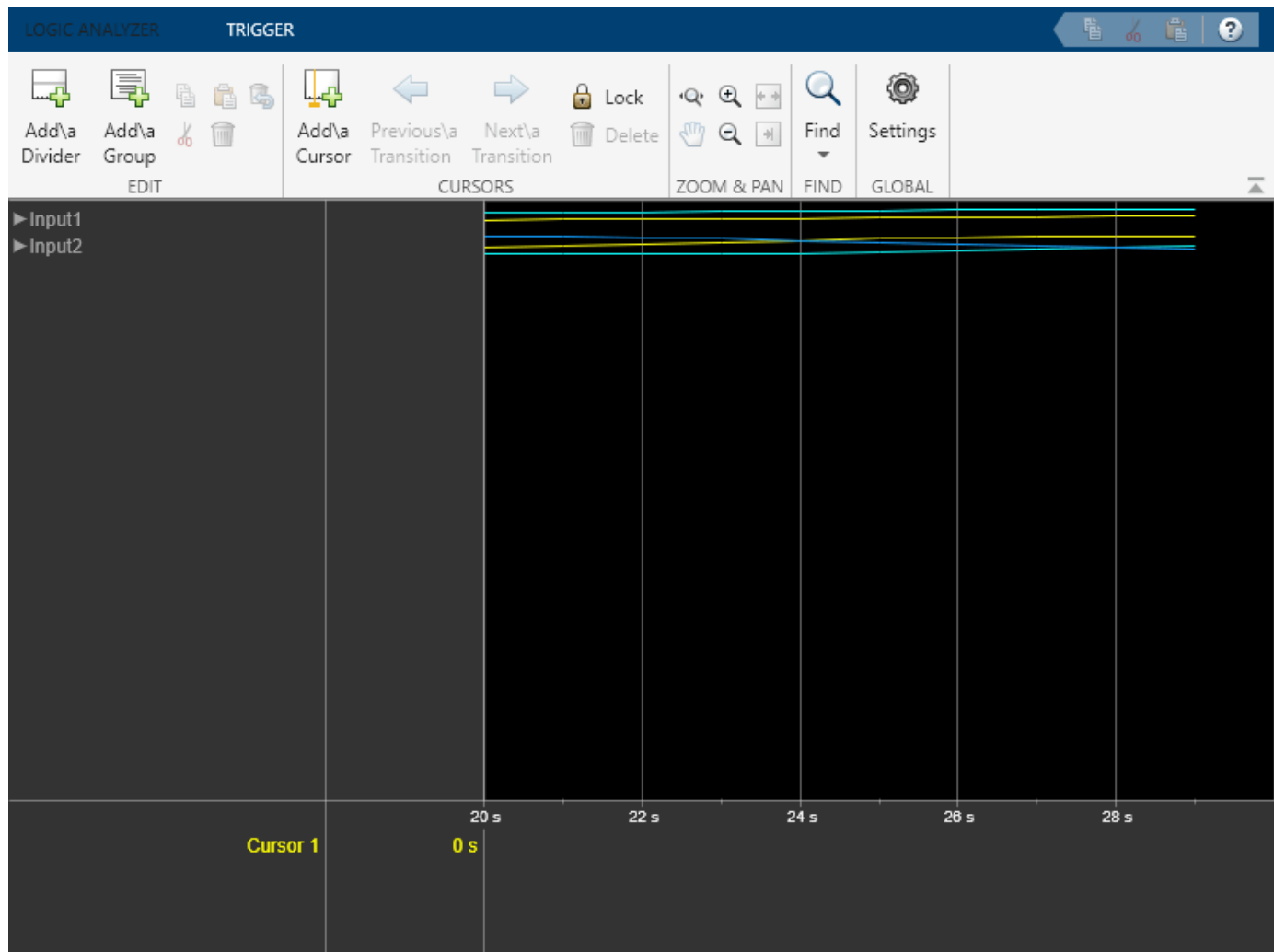
#### Display Waves on Logic Analyzer scope.

```
scope = dsp.LogicAnalyzer('NumInputPorts',2);

stop = 30;
for count = 1:stop
    sinValVec = sin(count/stop*2*pi);
    cosValVec = cos(count/stop*2*pi);
    cosValVecOffset = cos((count+10)/stop*2*pi);

    scope([count (count-(stop/2))],[sinValVec cosValVec cosValVecOffset])
end
```





## Reorganize Display

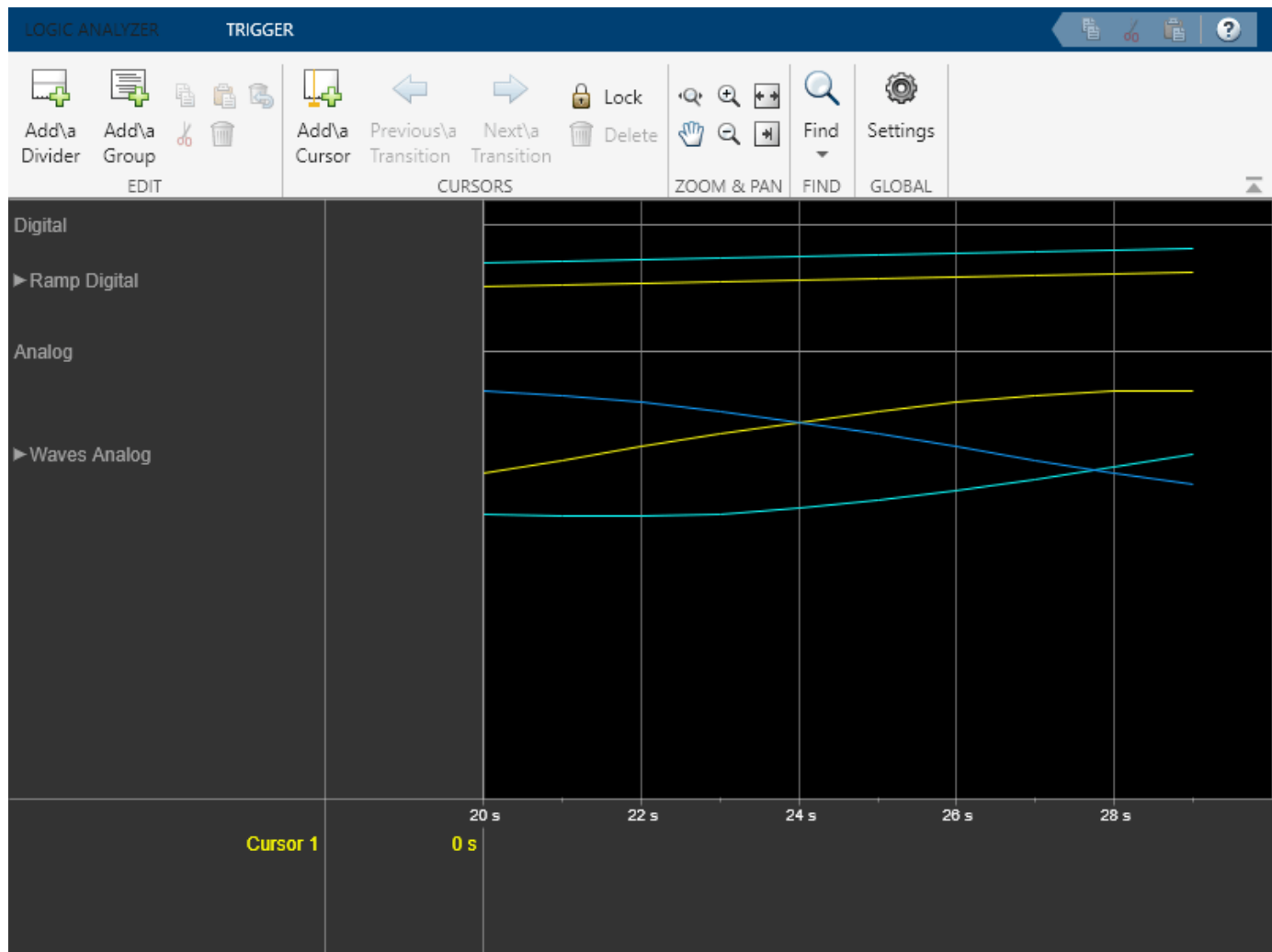
```
hide(scope)
digitalDividerTag = addDivider(scope, 'Name', 'Digital', 'Height', 20);
analogDividerTag = addDivider(scope, 'Name', 'Analog', 'Height', 40);

tags = getDisplayChannelTags(scope);

modifyDisplayChannel(scope, tags{1}, 'InputChannel', 1, ...
    'Name', 'Ramp Digital', 'Height', 40);
modifyDisplayChannel(scope, tags{2}, 'InputChannel', 2, ...
    'Name', 'Waves Analog', 'Format', 'Analog', 'Height', 80);

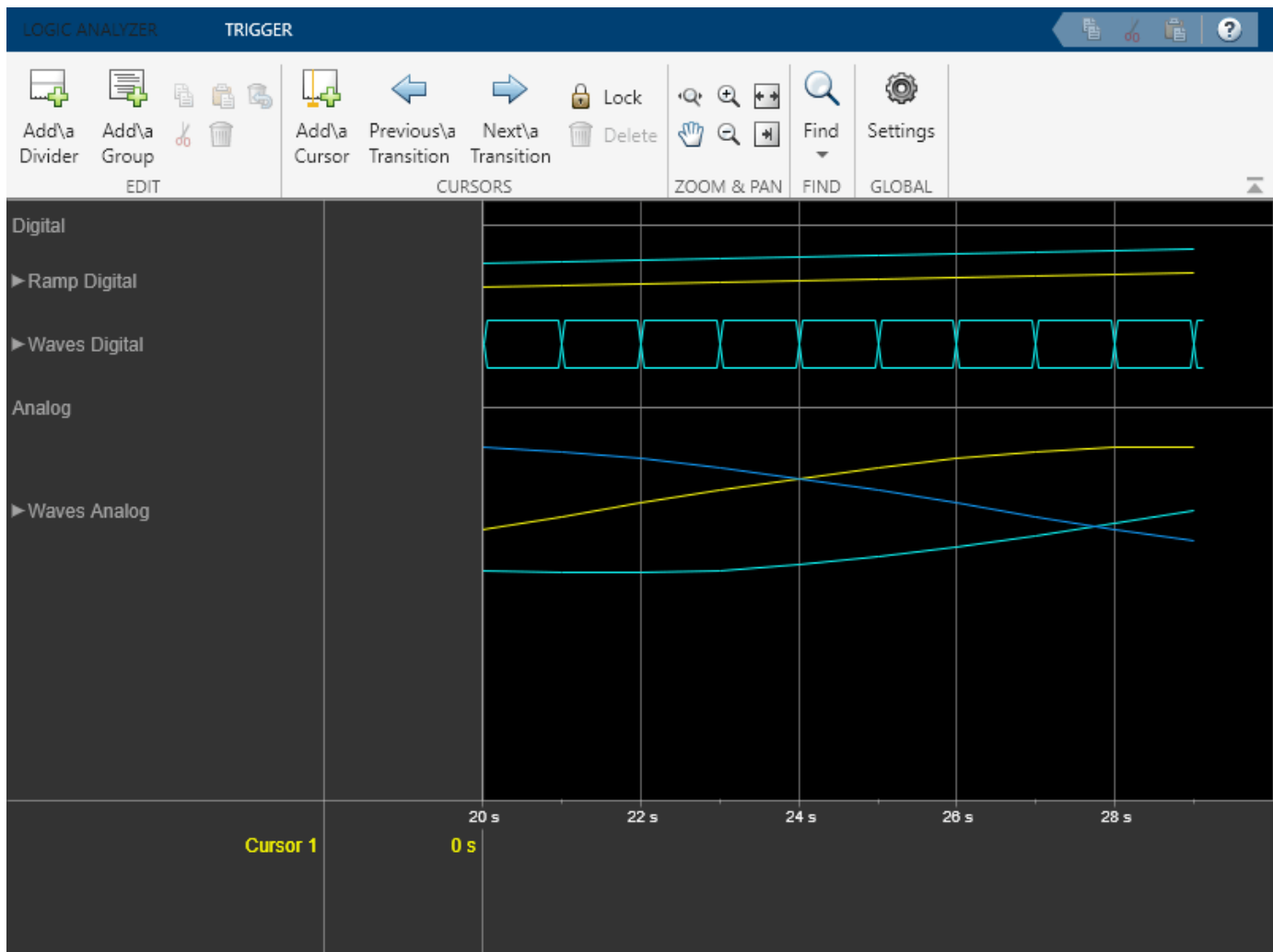
moveDisplayChannel(scope, digitalDividerTag, 'DisplayChannel', 1)
moveDisplayChannel(scope, tags{2}, 'DisplayChannel', length(tags))

show(scope)
```



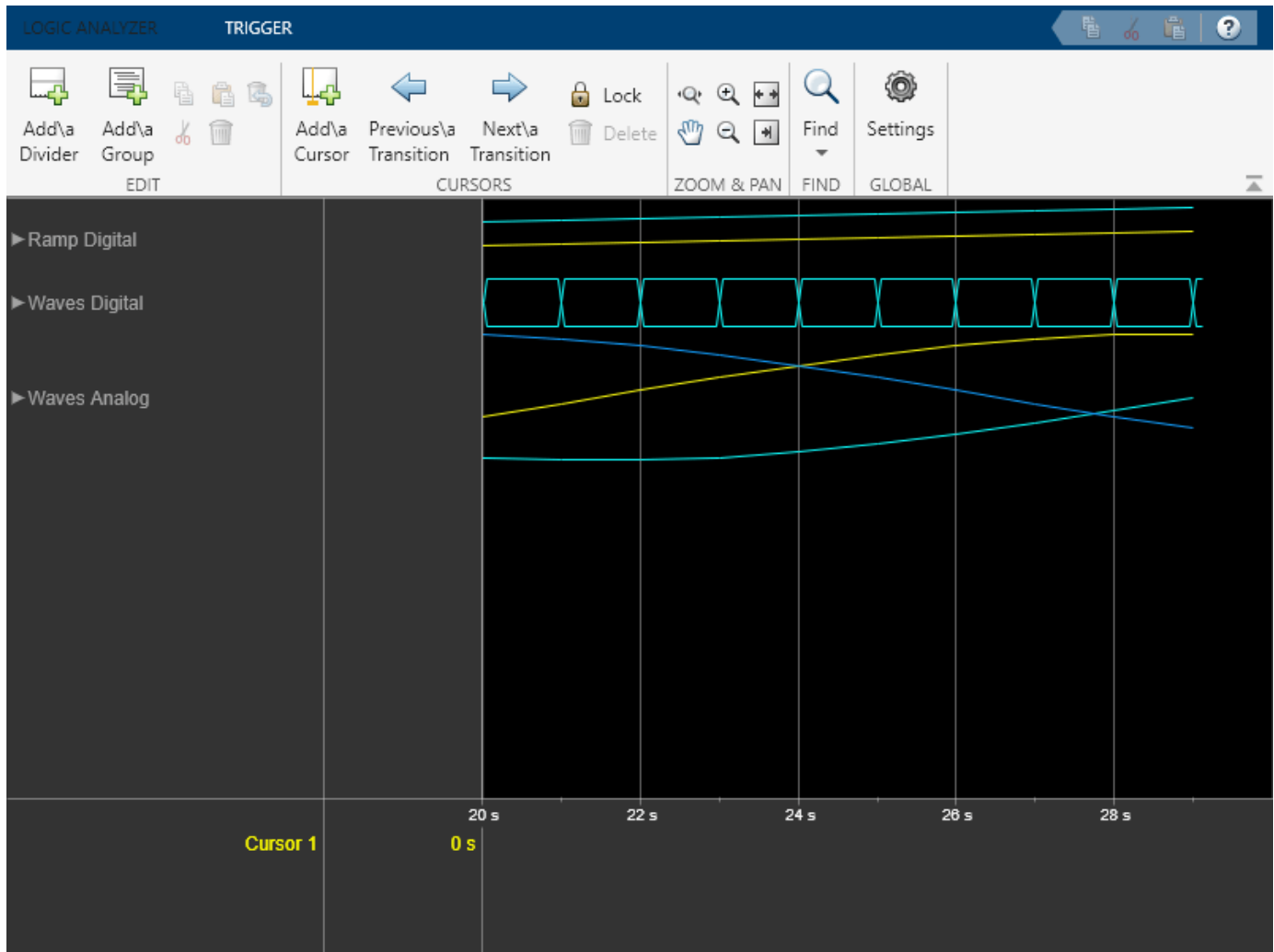
### Duplicate Wave and Check Information

```
hide(scope)
addWave(scope, 'InputChannel', 2, 'Name', 'Waves Digital', 'Format', 'Digital', ...
    'Height', 30, 'DisplayChannel', 3);
show(scope)
```



### Remove Dividers

```
hide(scope)
deleteDisplayChannel(scope,digitalDividerTag)
deleteDisplayChannel(scope,analogDividerTag)
show(scope)
```



### Clear variables

```
clear analogDividerTag cosValVec cosValVecOffset count digitalDividerTag duplicateWave scope sin
```

## Input Arguments

### scope — Logic Analyzer object

dsp.LogicAnalyzer object

The Logic Analyzer object from which you want to delete a display channel, specified as a handle to the dsp.LogicAnalyzer object.

### tag — tag identifier

randomly assigned character vector

The tag identifying which display channel to delete, specified as the randomly assigned character vector.

Example: 'deleteDisplayChannel(scope,tag)' deletes a display channel from Logic Analyzer.

Data Types: char | string

## **Version History**

**Introduced in R2013a**

### **See Also**

`dsp.LogicAnalyzer` | `getDisplayChannelTags` | `addDivider` | `addWave` | `modifyDisplayChannel` | `deleteCursor`

## getFilter

**Package:** dsp

Get underlying FIR filter

### Syntax

```
filter = getFilter(DF)
```

### Description

`filter = getFilter(DF)` returns the underlying FIR filter, `filter`, used to implement the differentiator, `DF`.

### Examples

#### Get the Underlying FIR Filter of the Differentiator

Create a `dsp.Differentiator System` object with default properties.

```
DF = dsp.Differentiator;
```

Use `getFilter` to get the underlying FIR filter, `filter`, which implements the differentiator, `DF`.

```
filter = getFilter(DF)
```

```
filter =
```

```
    dsp.FIRFilter with properties:
```

```
        Structure: 'Direct form'
```

```
    NumeratorSource: 'Property'
```

```
        Numerator: [-0.0019 0.0024 -8.7640e-04 6.6377e-04 -6.2364e-04 6.3164e-04 -6.6022e-04
```

```
    InitialConditions: 0
```

```
    Show all properties
```

### Input Arguments

#### **DF** — Differentiator filter

`dsp.Differentiator System` object

Differentiator filter, specified as a `dsp.Differentiator System` object.

### Output Arguments

#### **filter** — Underlying filter

system object

Underlying filter used to implement the differentiator, returned as a filter System object.

## **Version History**

**Introduced in R2016a**

## **See Also**

### **Objects**

`dsp.Differentiator`

## fvtool

**Package:** dsp

Visualize frequency response of digital down converter or digital up converter filter cascade

### Syntax

```
fvtool(Conv)
fvtool(Conv, 'Arithmetic', arithType)
```

### Description

`fvtool(Conv)` plots the magnitude response of a digital down converter or digital up converter, `Conv`. By default, the object plots the cascade response up to the second CIC null frequency (or to the first when only one CIC null exists). To use this syntax, the object `Conv` must be locked.

`fvtool(Conv, 'Arithmetic', arithType)` specifies the arithmetic type of the filters inside the converter. Set the 'Arithmetic' input to 'double', 'single', or 'fixed-point'. When the `Conv` object is in an unlocked state, you must specify the arithmetic type. When the `Conv` object is in a locked state, it ignores the arithmetic input argument.

For example, to plot the magnitude response of a digital down converter in an unlocked state, set the 'Arithmetic' input.

```
dwnConv = dsp.DigitalDownConverter
fvtool(dwnConv, 'Arithmetic', 'fixed-point')
```

### Examples

#### Magnitude Response of Digital Down Converter

Plot the magnitude response of the digital down converter using the `fvtool` function and the `visualizeFilterStages` function.

Create a `dsp.DigitalDownConverter` System object with the default settings. Using the `fvtool` function, plot the magnitude response of the overall filter cascade. The `visualizeFilterStages` function in addition plots the magnitude response of the individual filters stages.

```
dwnConv = dsp.DigitalDownConverter

dwnConv =
    dsp.DigitalDownConverter with properties:

        DecimationFactor: 100
        MinimumOrderDesign: true
        Bandwidth: 200000
        StopbandFrequencySource: 'Auto'
        PassbandRipple: 0.1000
        StopbandAttenuation: 60
        Oscillator: 'Sine wave'
```



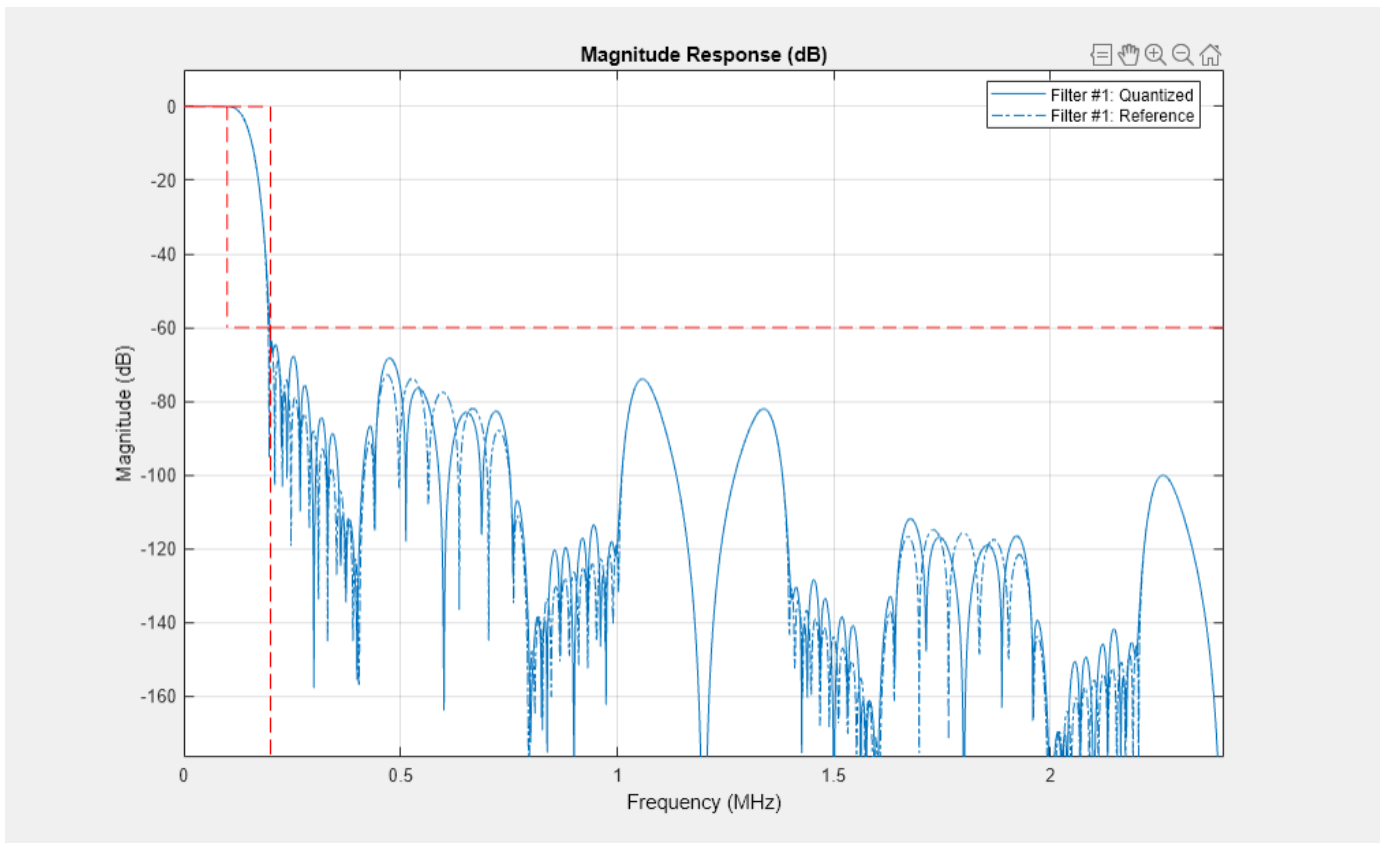
CenterFrequency: 14000000  
 SampleRate: 30000000

Show all properties

## Using fvtool

If the System object is unlocked, you must specify the filter arithmetic through the 'Arithmetic' input of the fvtool function. If the System object is locked, the arithmetic input is ignored.

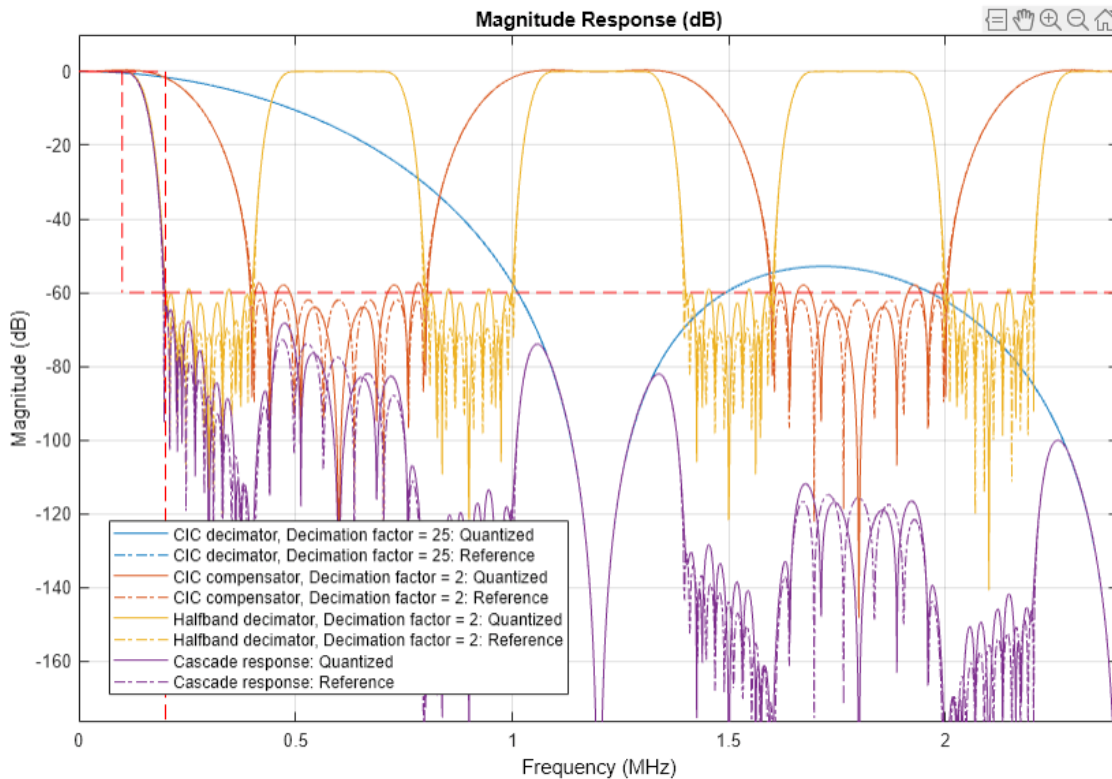
```
fvtool(dwnConv, 'Arithmetic', 'fixed-point')
```



## Using visualizeFilterStages

To view the magnitude response of the individual filter stages, call the visualizeFilterStages function.

```
visualizeFilterStages(dwnConv, 'Arithmetic', 'fixed-point')
```



## Input Arguments

### Conv — Digital down converter or digital up converter

`dsp.DigitalDownConverter` | `dsp.DigitalUpConverter`

Digital down converter or digital up converter, specified as a `dsp.DigitalDownConverter` or `dsp.DigitalUpConverter` System object.

### arithType — Arithmetic type

'double' (default) | 'single' | 'fixed-point'

When the Conv object is in an unlocked state, you must specify the arithmetic type. When the Conv object is in a locked state, it ignores the arithmetic input argument.

## Version History

Introduced in R2012a

## See Also

### Functions

`getInterpolationFactors` | `getDecimationFactors` | `getFilterOrders` | `getFilters` | `groupDelay` | `visualizeFilterStages`

**Objects**

dsp.DigitalDownConverter | dsp.DigitalUpConverter

## denormalize

Undo filter coefficient and gain changes caused by `normalize`

### Syntax

```
denormalize(hq)
```

### Description

`denormalize(hq)` reverses the coefficient changes you make when you use `normalize` with `hq`. The filter coefficients do not change if you call `denormalize(hq)` before you use `normalize(hq)`. Calling `denormalize` more than once on a filter does not change the coefficients after the first `denormalize` call.

### Examples

#### Denormalize Filter Coefficients

Construct a quantized filter `hd`.

```
d=fdesign.highpass('n,F3dB',14,0.45);  
hd =design(d,'butter');  
hd.arithmetic='fixed';
```

Normalize the filter coefficients

```
normalize(hd)  
NormSOSMatrix = hd.sosMatrix;
```

After normalizing the filter coefficients, restore them to their original values by reversing the effects of the `normalize` function.

```
denormalize(hd)  
eqSOSMatrices = isequal(NormSOSMatrix,hd.sosMatrix)
```

```
eqSOSMatrices = logical  
0
```

## Version History

Introduced in R2011a

# design

Apply design method to filter specification object

## Syntax

```
filt = design(designSpecs,'Systemobject',true)
filt = design(designSpecs,method,'Systemobject',true)
filt = design(designSpecs,method,PARAM,VALUE,...,'Systemobject',true)
filt = design(designSpecs,method,opts,'Systemobject',true)
```

## Description

`filt = design(designSpecs,'Systemobject',true)` uses the filter specification object, `designSpecs`, to generate a filter System object, `filt`. When you do not provide a design method as an input argument, `design` uses the default design method. Use `designmethods(designSpecs,'default')` to see the default design method for your filter design specification object. For more information on filter design specifications, see “Design a Filter in Fdesign — Process Overview”.

`filt = design(designSpecs,method,'Systemobject',true)` uses the design method specified by `method`. `method` must be one of the options returned by `designmethods`.

`filt = design(designSpecs,method,PARAM,VALUE,...,'Systemobject',true)` specifies design method options. Use `designoptions(designSpecs,method)` to see a list of available design method options to choose from. For detailed help on each of these options, type `help(designSpecs,method)` in the MATLAB command prompt.

`filt = design(designSpecs,method,opts,'Systemobject',true)` specifies design method options using the structure `opts`. `opts` is usually obtained from the `designopts` function and then specified as an input to the `design` function. Use `help(designSpecs,method)` for more information on optional inputs.

## Examples

### Design FIR Equiripple Lowpass Filters

Design an FIR equiripple lowpass filter. Specify a passband edge frequency of  $0.2\pi$  rad/sample and a stopband edge frequency of  $0.25\pi$  rad/sample. Set the passband ripple to 0.5 dB and the stopband attenuation to 40 dB..

```
designSpecs = fdesign.lowpass('Fp,Fst,Ap,Ast',0.2,0.25,0.5,40)
```

```
designSpecs =
    lowpass with properties:
        Response: 'Lowpass'
        Specification: 'Fp,Fst,Ap,Ast'
        Description: {4x1 cell}
        NormalizedFrequency: 1
```

```

Fpass: 0.2000
Fstop: 0.2500
Apass: 0.5000
Astop: 40

```

Use the default Equiripple method to design the filter.

```
filt = design(designSpecs, 'SystemObject', true)
```

```

filt =
  dsp.FIRFilter with properties:

    Structure: 'Direct form'
    NumeratorSource: 'Property'
    Numerator: [-0.0057 -0.0010 0.0010 0.0040 0.0065 0.0073 0.0055 0.0014 -0.0033 -0.0066]
    InitialConditions: 0

  Show all properties

```

Determine the available design methods by running the `designmethods` function on the filter design specification object, `designSpecs`.

```
designmethods(designSpecs, 'SystemObject', true)
```

```
Design Methods that support System objects for class fdesign.lowpass (Fp,Fst,Ap,Ast):
```

```

butter
cheby1
cheby2
ellip
equiripple
ifir
kaiserwin
multistage

```

You can also specify the design options used in designing the filter. To see a list of available options, run the `designoptions` function on `designSpecs`.

```
designoptions(designSpecs, 'equiripple')
```

```

ans = struct with fields:
  FilterStructure: {'dffir' 'dffirt' 'dfsymfir' 'fftfir'}
  DensityFactor: 'double'
  MinPhase: 'bool'
  MaxPhase: 'bool'
  MinOrder: {'any' 'even' 'odd'}
  StopbandShape: {'flat' 'linear' '1/f'}
  StopbandDecay: 'double'
  UniformGrid: 'bool'
  SystemObject: 'bool'
  DefaultFilterStructure: 'dffir'
  DefaultDensityFactor: 16
  DefaultMaxPhase: 0
  DefaultMinOrder: 'any'
  DefaultMinPhase: 0
  DefaultStopbandDecay: 0

```

```

DefaultStopbandShape: 'flat'
DefaultSystemObject: 0
DefaultUniformGrid: 1

```

Design a minimum-phase FIR equiripple filter by setting 'MinPhase' to true.

```

filtMin = design(designSpecs,'equiripple','MinPhase',true,'SystemObject',true)

```

```

filtMin =
  dsp.FIRFilter with properties:
    Structure: 'Direct form'
    NumeratorSource: 'Property'
    Numerator: [0.0163 0.0381 0.0724 0.1144 0.1562 0.1875 0.1981 0.1814 0.1378 0.0750 0.0163]
    InitialConditions: 0

```

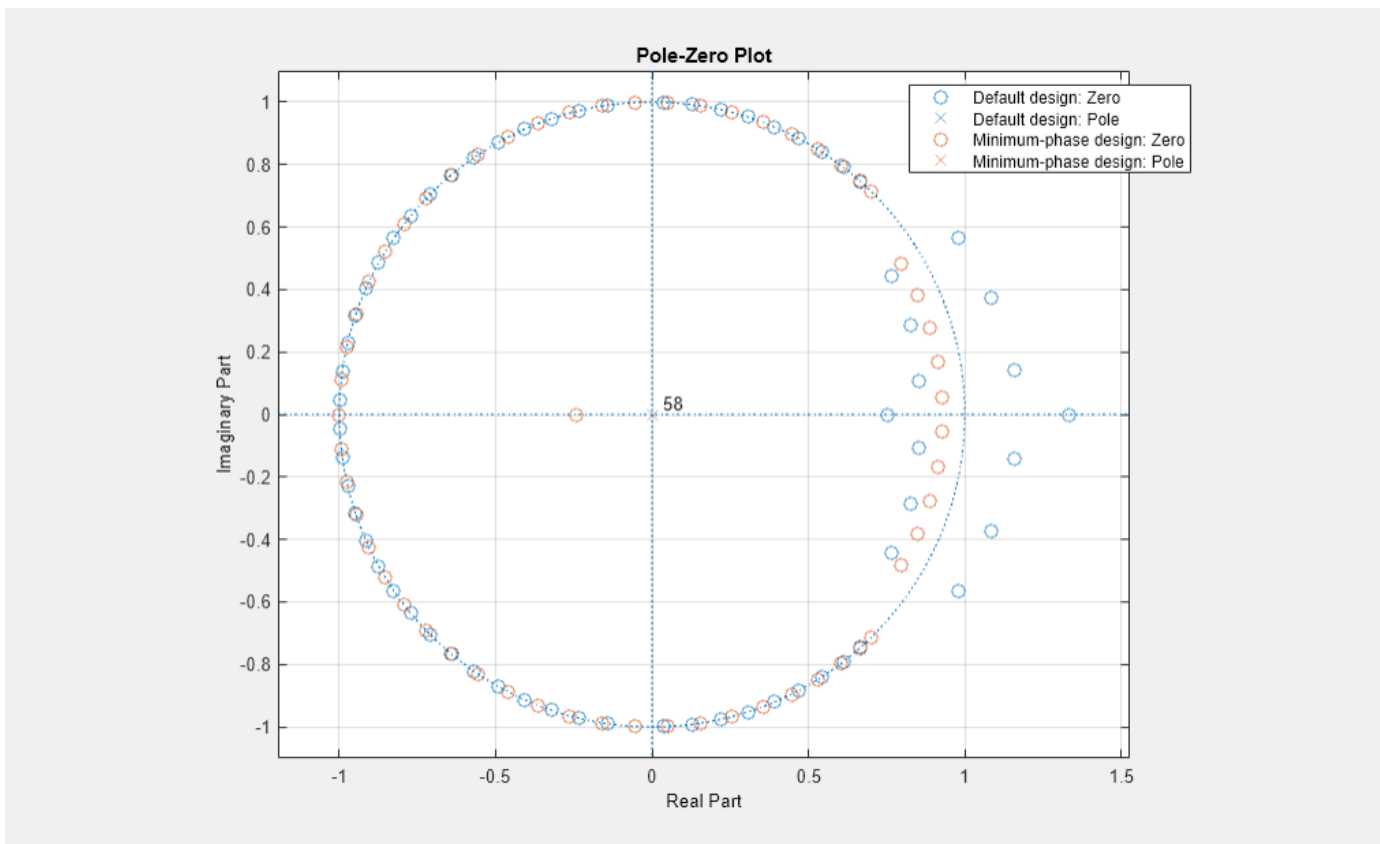
Show all properties

Display pole-zero plots of the default and minimum-phase designs.

```

fvt = fvtool(filt,filtMin,'Analysis','polezero');
legend(fvt,'Default design','Minimum-phase design')

```



Redesign the filter using the elliptic method. Determine the available design options for the elliptic method.

```
designoptions(designSpecs,'ellip')
```

```
ans = struct with fields:
    FilterStructure: {'df1sos' 'df2sos' 'df1tsos' 'df2tsos' 'cascadeallpass' 'cascad
    SOSScaleNorm: 'ustring'
    SOSScaleOpts: 'fdopts.sosscaling'
    MatchExactly: {'passband' 'stopband' 'both'}
    SystemObject: 'bool'
    DefaultFilterStructure: 'df2sos'
    DefaultMatchExactly: 'both'
    DefaultSOSScaleNorm: ''
    DefaultSOSScaleOpts: [1x1 fdopts.sosscaling]
    DefaultSystemObject: 0
```

Match the passband exactly by setting 'MatchExactly' to 'passband'.

```
filt = design(designSpecs,'ellip','MatchExactly','passband','SystemObject',true)
```

```
filt =
    dsp.SOSFilter with properties:
        Structure: 'Direct form II'
        CoefficientSource: 'Property'
        Numerator: [3x3 double]
        Denominator: [3x3 double]
        HasScaleValues: true
        ScaleValues: [0.2868 0.1982 0.2819 1]
```

Show all properties

You can specify the Pth norm scaling on the second-order sections. Use L-infinity norm scaling in the time domain.

```
filtL = design(designSpecs,'ellip','MatchExactly','passband','SOSScaleNorm','linf', ...
    'SystemObject',true)
```

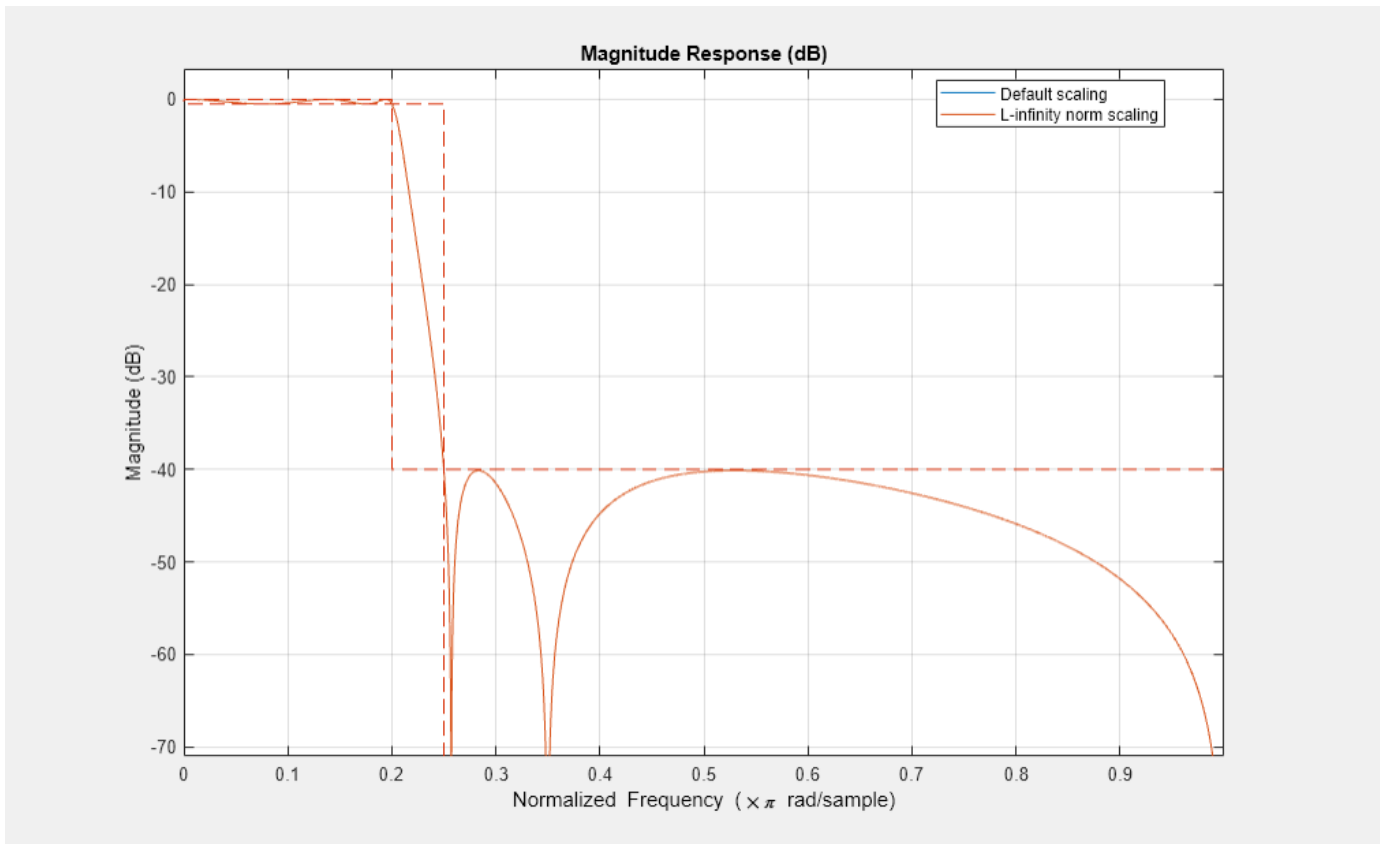
```
filtL =
    dsp.SOSFilter with properties:
        Structure: 'Direct form II'
        CoefficientSource: 'Property'
        Numerator: [3x3 double]
        Denominator: [3x3 double]
        HasScaleValues: false
```

Show all properties

Display the frequency responses of the filters.

```
fvt = fvtool(filt,filtL);
legend(fvt,'Default scaling','L-infinity norm scaling')
```





## Input Arguments

### designSpecs — Filter design specification object

`fdesign.response` object

`fdesign` returns a filter design specification object. Every filter design specification object has these properties.

| Property Name | Default Value                         | Description  |
|---------------|---------------------------------------|--|
| Response      | Depends on the chosen type            | Defines the type of filter to design, such as an interpolator or bandpass filter. This is a read-only value.   |
| Specification | Depends on the chosen type            | Defines the filter characteristics used to define the desired filter performance, such as the cutoff frequency $F_c$ or the filter order $N$ .   |
| Description   | Depends on the filter type you choose | Contains descriptions of the filter specifications used to define the object, and the filter specifications you use when you create a filter from the object. This is a read-only value. |

| Property Name       | Default Value | Description  |
|---------------------|---------------|--|
| NormalizedFrequency | Logical true  | Determines whether the filter calculation uses a normalized frequency from 0 to 1, or the frequency band from 0 to $F_s/2$ , the sampling frequency. Accepts either <code>true</code> or <code>false</code> without single quotation marks. Audio weighting filters do not support normalized frequency. |

In addition to these properties, filter design specification objects may have other properties as well, depending on whether they design single-rate filters or multirate filters.

| Added Properties for Multirate Filters | Description   |
|--|---|
| DecimationFactor                       | Specifies the amount to decrease the sampling rate. Always a positive integer.  |
| InterpolationFactor                    | Specifies the amount to increase the sampling rate. Always a positive integer.  |
| PolyphaseLength                        | Polyphase length is the length of each polyphase subfilter that composes the decimator or interpolator or rate-change factor filters. Total filter length is the product of <code>pl</code> and the rate change factors. <code>pl</code> must be an even integer. |

### method — Design method

character vector

Design method, specified as a character vector. The design method you provide as the input argument must be one of the methods returned by:

```
designmethods(designSpecs, 'Systemobject', true)
```

The table lists all the design methods. A subset of these become available depending on the filter design specification object, `designSpecs`.

| Design methods           | Description                          |
|--------------------------|--------------------------------------|
| <code>butter</code>      | Butterworth filter                   |
| <code>cheby1</code>      | Chebyshev Type I filter              |
| <code>cheby2</code>      | Chebyshev Type II filter             |
| <code>ellip</code>       | Elliptic filter                      |
| <code>equiripple</code>  | Equiripple FIR filter                |
| <code>firls</code>       | Least-square linear-phase FIR filter |
| <code>freqsamp</code>    | Frequency-sampled FIR filter         |
| <code>ifir</code>        | Interpolated FIR filter              |
| <code>iirlinphase</code> | Quasi-linear phase IIR filter        |
| <code>iirlpnorm</code>   | Least P-norm optimal IIR filter      |

| Design methods | Description                                |
|----------------|--|
| iirls          | Least-squares IIR filter                   |
| fircls         | FIR constrained least squares filter       |
| kaiserwin      | Kaiser window filter                       |
| maxflat        | Maxflat FIR filter                         |
| multistage     | Multistage filter                          |
| window         | FIR filter using windowed impulse response |

To help you design filters more quickly, the input argument `method` accepts a variety of special keywords that force `design` to behave in different ways. This table presents the keywords you can use for `method` and how `design` responds to the keyword:

| Design Method Keyword | Description of the Design Response  |
|-----------------------|---|
| 'FIR'                 | Forces <code>design</code> to produce an FIR filter. When no FIR design method exists for object <code>d</code> , <code>design</code> returns an error.   |
| 'IIR'                 | Forces <code>design</code> to produce an IIR filter. When no IIR design method exists for object <code>d</code> , <code>design</code> returns an error.   |
| 'ALLFIR'              | Produces filters from every applicable FIR design method for the specifications in <code>d</code> , one filter for each design method. As a result, <code>design</code> returns multiple filters in the output object.  |
| 'ALLIIR'              | Produces filters from every applicable IIR design method for the specifications in <code>d</code> , one filter for each design method. As a result, <code>design</code> returns multiple filters in the output object.  |
| 'ALL'                 | Designs filters using all applicable design methods for the specifications object <code>d</code> . As a result, <code>design</code> returns multiple filters, one for each design method. <code>design</code> uses the design methods in the order that <code>designmethods(D, 'SystemObject', true)</code> returns them. |

Keywords are not case sensitive.

When `design` returns multiple filters in the output object, use indexing to see the individual filters. For example, to see the third filter in `filt`, enter:

```
filt(3)
```

```
Example: filt = design(designSpecs, 'butter', 'SystemObject', true)
```

```
Example: filt = design(designSpecs, 'ALLFIR', 'SystemObject', true)
```

### opts — Specify design options

structure

Specify design options by passing `opts` structure as an input to the `design` function. The `opts` structure is obtained by running `designopts(designSpecs, method)`.

```
designSpecs = fdesign.notch
opts = designopts(designSpecs, 'butter')
opts.FilterStructure = 'df1sos'
filt = design(designSpecs, 'butter', opts, 'SystemObject', true)
```

## **Version History**

**Introduced in R2009a**

### **See Also**

[fdesign](#) | [designmethods](#) | [designopts](#) | [designoptions](#)

### **Topics**

["Design a Filter in Fdesign — Process Overview"](#)

# designFracDelayFIR

Design band-limited fractional delay FIR filter

## Syntax

```
h = designFracDelayFIR(fd)
h = designFracDelayFIR(fd,N)
h = designFracDelayFIR(fd,TBW)
[h,i0] = designFracDelayFIR(____)
[h,i0,MBW] = designFracDelayFIR(____)
```

## Description

`h = designFracDelayFIR(fd)` designs a fractional delay FIR filter of delay `fd`, default length of 50, approximate bandwidth of 0.9.

`h = designFracDelayFIR(fd,N)` designs a fractional delay FIR filter of delay `fd` and length `N > 1`. The function designs the filter to achieve the specified FIR length.

`h = designFracDelayFIR(fd,TBW)` designs a fractional delay FIR filter of delay `fd` and a combined bandwidth of at least `TBW`, where `TBW < 0.999` and is specified in normalized frequency units. The function automatically determines the appropriate FIR length `N` for which the combined bandwidth is at least `TBW`.

`[h,i0] = designFracDelayFIR(____)` returns the integer latency of the FIR filter. This value is approximately `N/2`. The nominal group delay of the filter is `i0+fd`.

`[h,i0,MBW] = designFracDelayFIR(____)` returns the measured combined bandwidth of the filter `h`.

## Examples

### Design Fractional Delay FIR Filter with Specified Delay and Filter Length

Design a fractional delay FIR filter using the `designFracDelayFIR` function. Pass the delay and the filter length as the input arguments to the function. Vary the filter length and observe the effect on the measured combined bandwidth and the nominal group delay.

#### Vary Filter Length

##### Filter Length Set to 8

Specify the delay to be 0.25 and the filter length to be 8 and design the fractional delay FIR filter.

```
fd = 0.25;
len = [8 32 64];
[h1,i10,bw1] = designFracDelayFIR(fd,len(1))
```

```
h1 = 1×8
```

```
-0.0086  0.0417  -0.1355  0.8793  0.2931  -0.0968  0.0341  -0.0074
```

```
i10 = 3
```

```
bw1 = 0.5810
```

The nominal group delay of the filter `i10+fd` equals 3.25 samples. The measured combined bandwidth of the filter is 0.5810 in normalized frequency units.

### Filter Length Set to 32

Repeat the process with a filter length of 32 taps.

```
[h2,i20,bw2] = designFracDelayFIR(fd,len(2))
```

```
h2 = 1×32
```

```
-0.0001  0.0004  -0.0009  0.0017  -0.0029  0.0046  -0.0071  0.0104  -0.0148  0.0
```

```
i20 = 15
```

```
bw2 = 0.8571
```

The nominal group delay of the filter now equals 15.25 samples. By increasing the filter length, the integer latency `i0` also increases, resulting in an increase in the nominal group delay. The combined bandwidth of the filter has increased to 0.8571 in normalized frequency units.

### Filter Length Set to 64

Increase the filter length to 64 taps. The group delay increases to 31.25 samples, and the integer latency is 31 samples. The measured combined bandwidth of the filter further increases to 0.9219. That is a bandwidth coverage of 92.19% of the overall bandwidth. As the filter length continues to increase, the combined bandwidth tends closer towards 1.

```
[h3,i30,bw3] = designFracDelayFIR(fd,len(3))
```

```
h3 = 1×64
```

```
-0.0000  0.0001  -0.0001  0.0002  -0.0003  0.0004  -0.0006  0.0008  -0.0010  0.
```

```
i30 = 31
```

```
bw3 = 0.9219
```

### Plot Magnitude Response

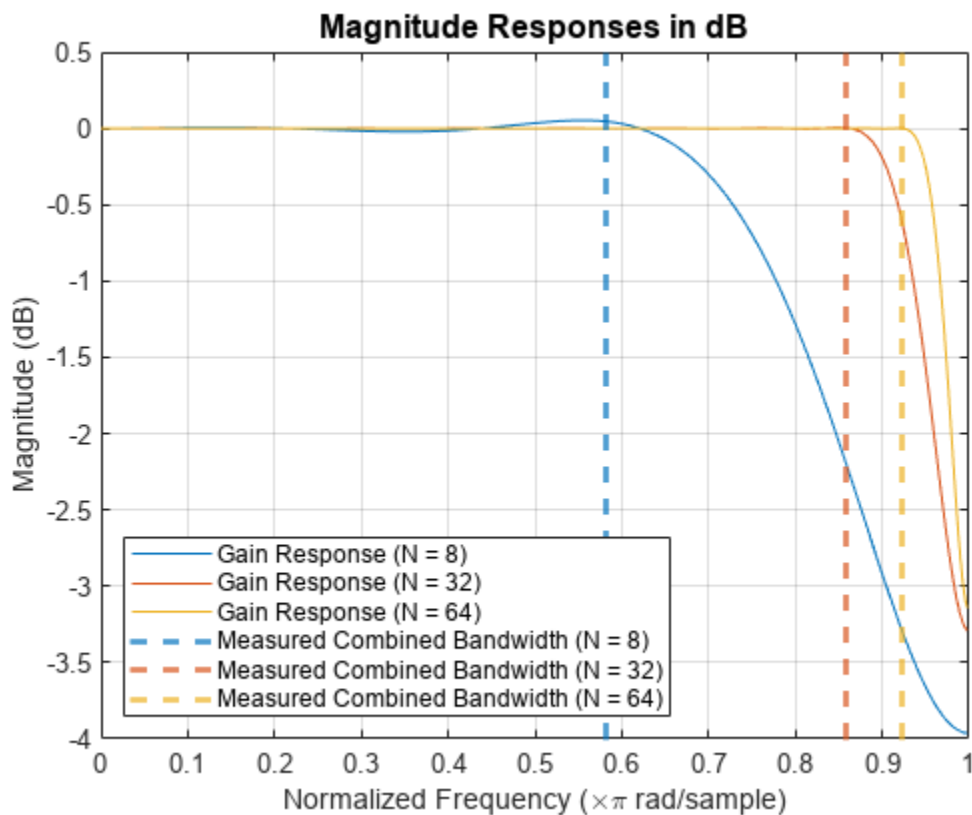
Plot the resulting magnitude response of the three filters. Mark the measured combined bandwidth of the three filters. By increasing the filter length, you can see that the measured combined bandwidth increases.

```
[H1,w] = freqz(h1,1);
H2 = freqz(h2,1);
H3 = freqz(h3,1);
figure;
plot(w/pi,mag2db(abs([H1 H2 H3])))
hold on
```

```

hline = lines;
xline(bw1, LineStyle = '--', LineWidth = 2, Color = hline(1,:))
xline(bw2, LineStyle = '--', LineWidth = 2, Color = hline(2,:))
xline(bw3, LineStyle = '--', LineWidth = 2, Color = hline(3,:))
hold off
title('Magnitude Responses in dB', FontSize = 12)
xlabel("Normalized Frequency (\times\pi rad/sample)")
ylabel("Magnitude (dB)")
grid
legend('Gain Response (N = 8)', 'Gain Response (N = 32)', 'Gain Response (N = 64)', ...
       'Measured Combined Bandwidth (N = 8)', ...
       'Measured Combined Bandwidth (N = 32)', ...
       'Measured Combined Bandwidth (N = 64)', Location = 'Southwest')

```



### Plot Group Delay Response

Plot the group delay response of the three filters. Mark the nominal group delay  $i0 + fd$  of the three filters. By increasing the filter length, you can see that the nominal group delay increases.

```

[g1,w] = grpdelay(h1,1);
g2 = grpdelay(h2,1);
g3 = grpdelay(h3,1);

```

```

figure;
plot(w/pi,[g1 g2 g3])

```

```

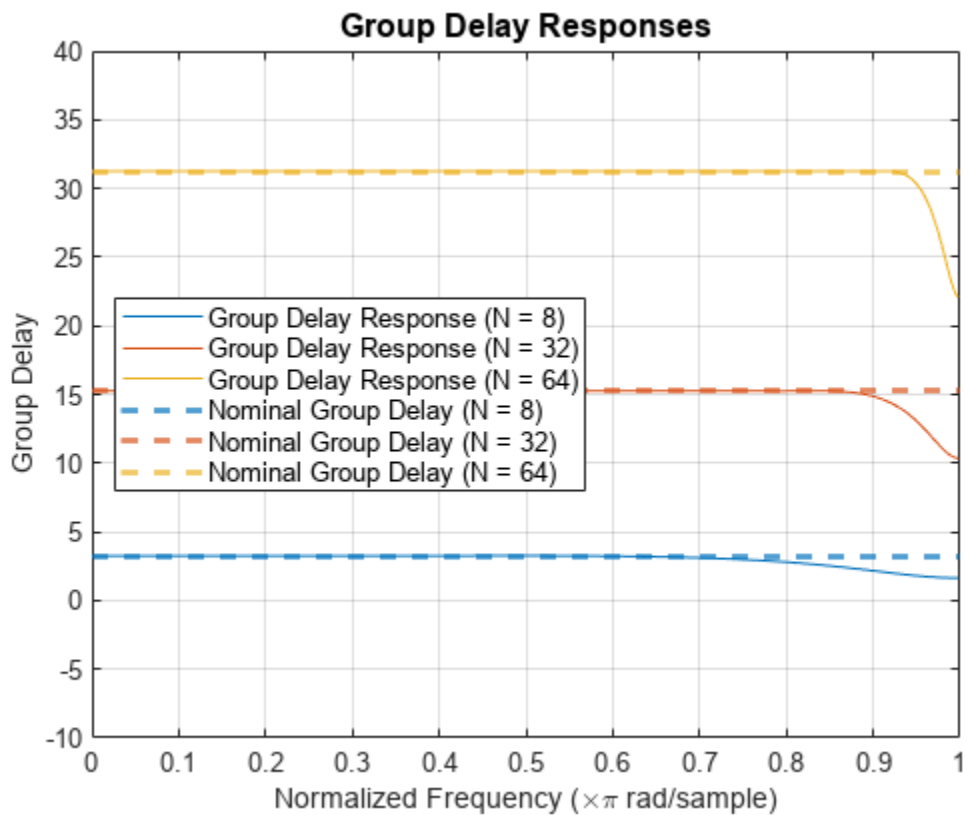
hline = lines;
yline(i10+fd, LineStyle = '--', LineWidth = 2, Color = hline(1,:))
yline(i20+fd, LineStyle = '--', LineWidth = 2, Color = hline(2,:))
yline(i30+fd, LineStyle = '--', LineWidth = 2, Color = hline(3,:))

title('Group Delay Responses',FontSize = 12)

xlabel("Normalized Frequency (\times\pi rad/sample)")
ylabel("Group Delay")
grid

legend('Group Delay Response (N = 8)', 'Group Delay Response (N = 32)', 'Group Delay Response (N = 64)', ...
       'Nominal Group Delay (N = 8)', ...
       'Nominal Group Delay (N = 32)', ...
       'Nominal Group Delay (N = 64)', Location = 'west', FontSize = 10);
ylim([-10,40]);

```



### Design Fractional Delay FIR Filter with Specified Delay and Combined Bandwidth

Design a fractional delay FIR filter using the `designFracDelayFIR` function. Pass the delay and the combined bandwidth as input arguments to the function.

Specify the delay to be 0.786 and the target combined bandwidth to be 0.8. The function designs a filter that has a length of 22 taps, an integer latency  $i0$  of 10 samples, and a combined bandwidth



$mbw$  of 0.8044 in normalized frequency units. This  $mbw$  value makes the combined bandwidth coverage to be 80.44% of the frequency domain and exceeds the specified target combined bandwidth. The nominal group delay of the filter  $i\theta + fd$  equals 10.786.

```
fd = 0.786;
tbw = 0.8;
[h,i0,mbw] = designFracDelayFIR(fd, tbw)
```

```
h = 1×22
```

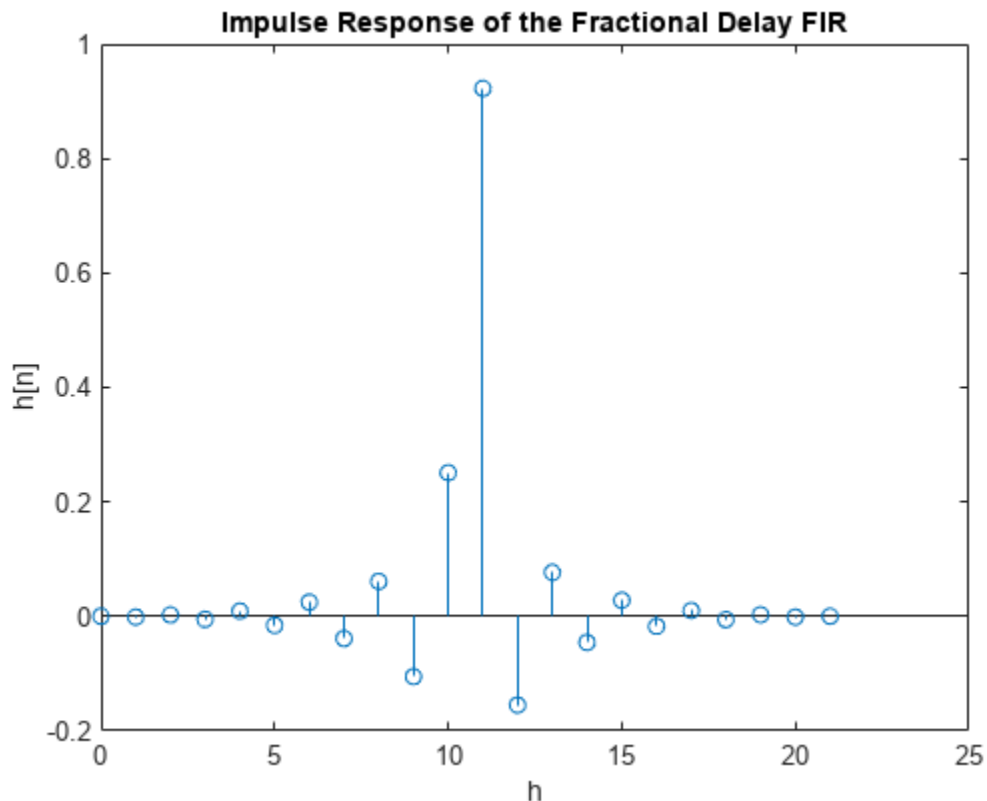
```
    0.0003   -0.0011    0.0026   -0.0052    0.0094   -0.0156    0.0248   -0.0386    0.0611   -0.1
```

```
i0 = 10
```

```
mbw = 0.8044
```

Plot the impulse response of the FIR.

```
stem((0:length(h)-1),h);
xlabel('h'); ylabel('h[n]');
title('Impulse Response of the Fractional Delay FIR')
```



Plot the resulting magnitude response and the group delay response. Mark the nominal group delay and the combined bandwidth of the filter.

```
[H1,w] = freqz(h,1);
G1 = grpdelay(h,1);
```

```

figure;
yyaxis left
plot(w/pi,mag2db(abs(H1)))
ylabel("Magnitude (dB)")

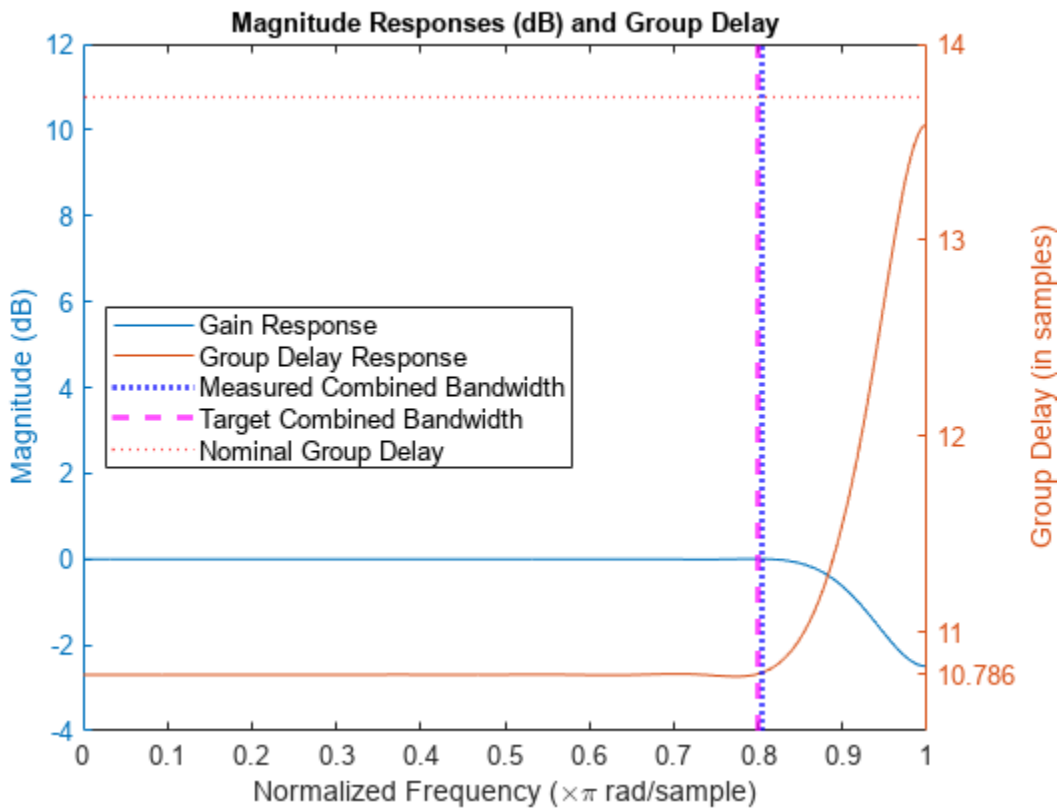
hold on
yyaxis right
plot(w/pi,G1)
ylabel("Group Delay (in samples)")

hline = lines;
xline(mbw,LineStyle=':',Color='b',LineWidth=2)
xline(tbw,LineStyle='--',Color='m',LineWidth=2)
yline(i0+fd,LineStyle=':',Color='r',LineWidth=1)
yticks([i0,i0+fd,i0+1:i0+9]);

hold off
title('Magnitude Responses (dB) and Group Delay',FontSize=10)
xlabel("Normalized Frequency (\times\pi rad/sample)")

legend('Gain Response','Group Delay Response','Measured Combined Bandwidth',...
       'Target Combined Bandwidth','Nominal Group Delay',...
       Location='west',FontSize=10)

```



## Design Fractional Delay FIR Filter and Compare with Shifted Input

Design a fractional delay FIR filter using the `designFracDelayFIR` function. Determine the group delay of the designed filter. Create a `dsp.FIRFilter` object that uses these designed coefficients and hence has the same group delay. Alternately, create a sampled sequence of a known function. Pass the sampled sequence to the FIR filter. Compare the output of the FIR filter to the shifted samples of the known function. Specify this shift to be equal to the group delay of the FIR filter. Verify that the two sequences match.

Specify the delay of the fractional delay FIR filter to be  $1/3$  and the length to be 6 taps.

```
fd = 1/3;  
len = 6;
```

Design the filter using the `designFracDelayFIR` function and determine the center index  $i0$  and the combined bandwidth  $bw$  of the filter. The group delay of the filter you have designed is  $i0 + fd$  or approximately 2.33 for the bandwidth of  $bw$ .

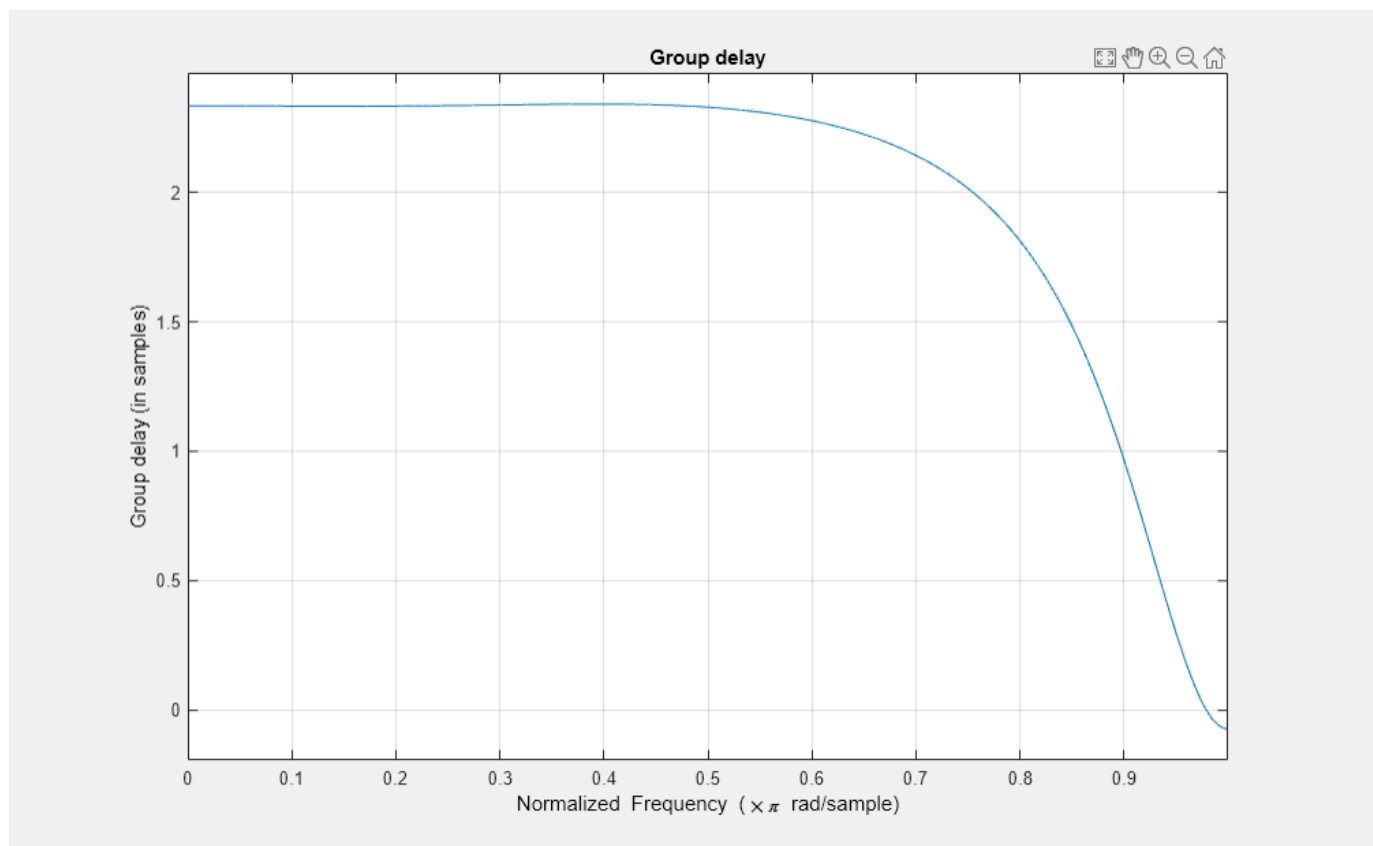
```
[h,i0,bw] = designFracDelayFIR(fd,len)  
  
h = 1×6  
    0.0293    -0.1360    0.7932    0.3966   -0.1088    0.0257
```

```
i0 = 2
```

```
bw = 0.5158
```

Create a `dsp.FIRFilter` object and set its numerator to the filter coefficients  $h$ . This filter is now effectively a fractional delay FIR filter. Verify that the group delay response of this filter is approximately 2.33 for the duration of the bandwidth  $bw$ .

```
fdf = dsp.FIRFilter(h);  
grpdelay(fdf)
```



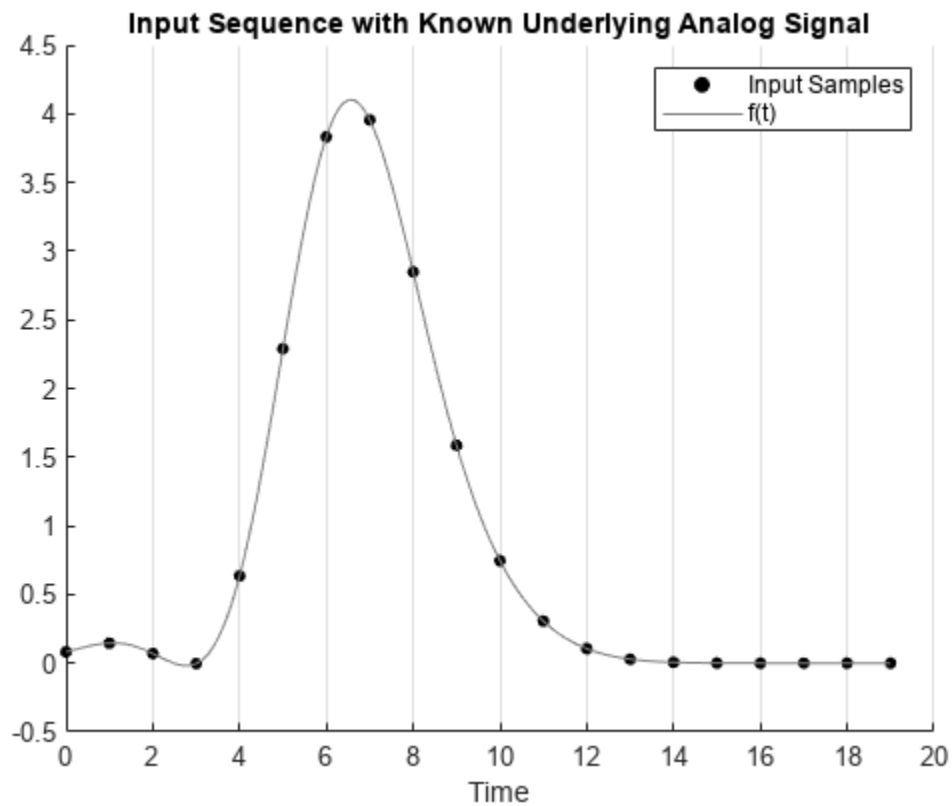
### Compare with Shifted Function

Define a sequence  $x$  as samples of a known function.

```
f = @(t) (0.1*t.^2+cos(0.9*t)).*exp(-0.1*(t-5).^2);
n = (0:19)'; t = linspace(0,19,512);
x = f(n); % Samples
```

Plot the sampled values  $x$  against the original known function  $f(t)$ .

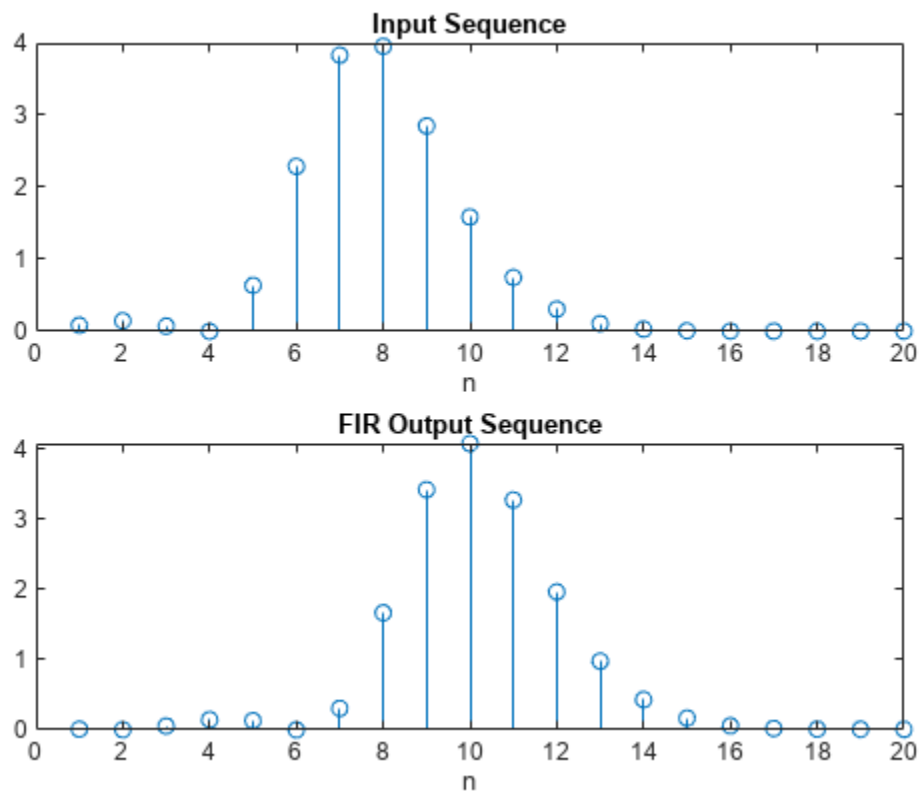
```
scatter(n,x,20,'k','filled');
hold on;
plot(t,f(t),'color',[0.5 0.5 0.5],'LineWidth',0.5)
hold off;
xlabel('Time')
legend(["Input Samples", "f(t)"])
title('Input Sequence with Known Underlying Analog Signal')
ax = gca; ax.XGrid='on';
```



Pass the sampled sequence  $x$  through the FIR filter. Plot the input sequence and output sequence.

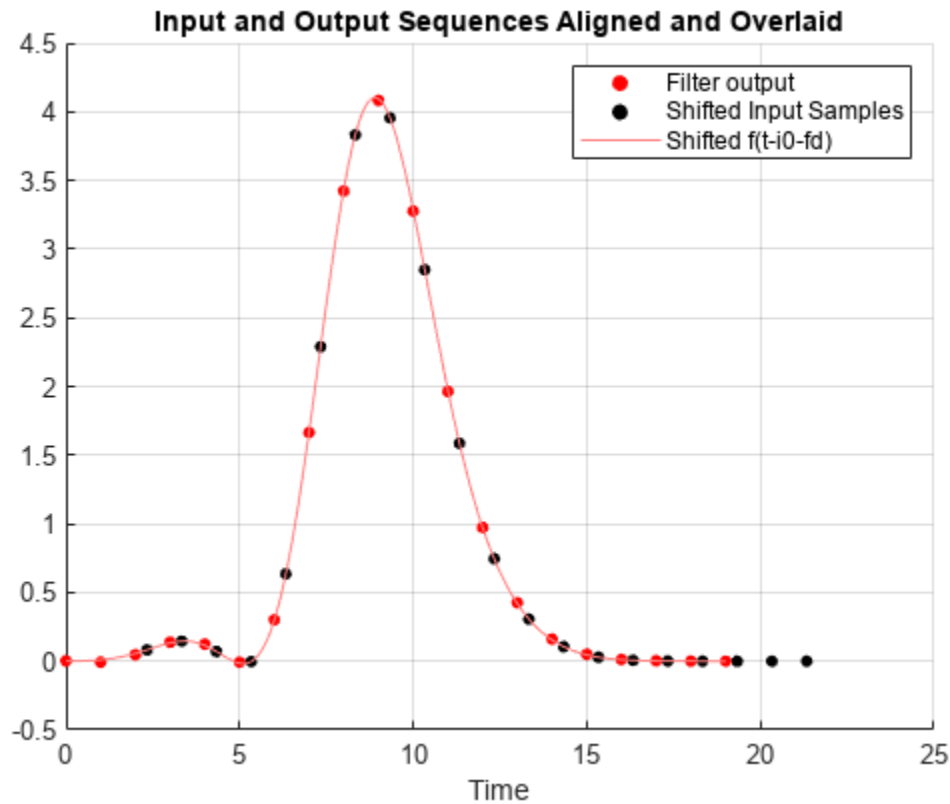
```
y = fdf(x);
```

```
subplot(2,1,1);  
stem(x);  
title('Input Sequence');  
xlabel('n')  
subplot(2,1,2)  
stem(y);  
title('FIR Output Sequence');  
xlabel('n')
```



Shift the input sequence horizontally by  $i0 + fd$ , which is equal to the group delay of the FIR filter. Plot the function  $f(t-i0-FD)$ . Verify that the input and output sequences fall roughly on the shifted function.

```
figure
scatter(n,y,20,'red','filled')
hold on;
scatter(n+i0+fd,x,20,'black','filled')
plot(t,f(t-i0-fd),'Color',[1,0.5,0.5],'LineWidth',0.1)
xlabel('Time')
legend(["Filter output","Shifted Input Samples","Shifted f(t-i0-fd)"])
hold off
grid on
title('Input and Output Sequences Aligned and Overlaid')
```



## Input Arguments

### **fd** — Fractional delay of filter

positive scalar in the range [0,1]

Fractional delay of the filter, specified as a positive scalar in the range [0,1]. The fractional delay value that you specify determines the measured combined bandwidth MBW of the filter. When `fd` is set to 0 or 1, the designed filter has a full bandwidth.

Data Types: `single` | `double`

### **N** — Length of FIR filter

integer greater than 1

Length of the fractional delay FIR filter, specified as an integer greater than 1. In order to specify the length of the filter, you must specify the second input argument of the function to be an integer that is greater than 1. The function treats this value as the desired filter length and returns a filter that is  $N$  taps long.

Data Types: `single` | `double`

### **TBW** — Target combined bandwidth

positive scalar less than 0.999

Target combined bandwidth, specified as a positive scalar less than 0.999. This is the value of the combined bandwidth that the function must satisfy. Combined bandwidth is defined as the minimum of the gain bandwidth on page 5-164 and the group delay bandwidth on page 5-164.

In order to specify the target combined bandwidth, you must specify the second input argument of the function to be a real scalar that is less than 0.999. The function then treats this value as the target combined bandwidth, determines the corresponding filter length, and designs the filter accordingly.

Specify a higher target combined bandwidth for a longer filter. For example TBW set to 0.9 yields a filter of length of 52. For TBW that is set to 0.99, the length is 724. This value is more than 10 times longer. As TBW tends towards 1, the filter length theoretically tends towards infinity.

Data Types: `single` | `double`

## Output Arguments

### **h** — Coefficients of fractional delay FIR filter

row vector

Coefficients of the fractional delay FIR filter, returned as a real-valued N-length row vector. When fractional delay is the only input to the function, the function designs a filter of length 50. The bandwidth for this default filter is approximately 0.9. When the second input argument is specified, the function designs the filter according to the following:

- When the second input argument to the function is specified to be a value greater than 1, the function treats this value as the desired filter length and returns a filter that is N taps long.
- When the second input argument to the function is specified to be a value less than 0.999, the function treats this value as the desired combined bandwidth TBW, determines the corresponding filter length, and designs the filter accordingly.

Data Types: `single` | `double`

### **i0** — Integer latency

integer

Integer latency of the designed FIR filter, returned as an integer value. Integer latency is the smallest integer shift required to make the symmetric Kaiser window causal. This value is approximately equal to half the filter length,  $N/2$ . For more details, see “Integer latency,  $i_0$ ” on page 5-162.

The nominal group delay of the filter is given by  $i_0 + fd$ .

Data Types: `single` | `double`

### **MBW** — Measured combined bandwidth

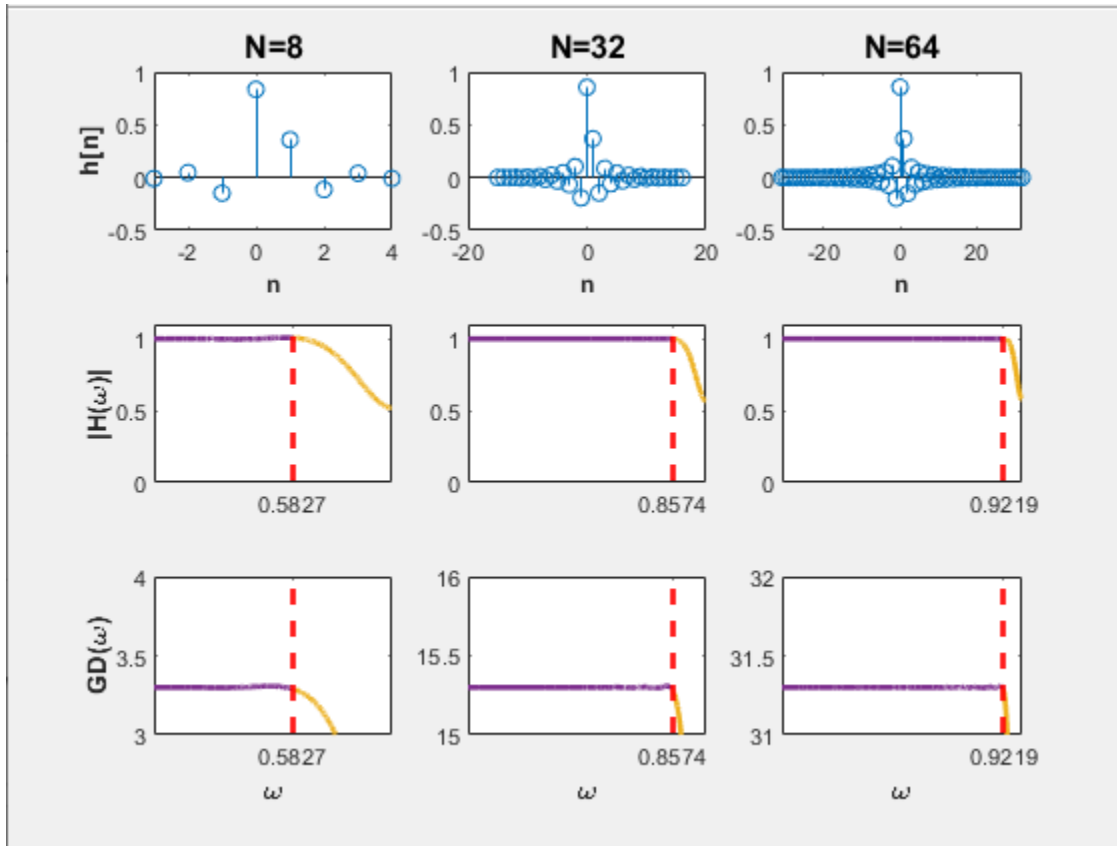
positive scalar less than 0.999

Measured combined bandwidth, returned as a real positive scalar that is less than 0.999. This is the value of the combined bandwidth of the designed filter. Combined bandwidth is defined as the minimum of the gain bandwidth on page 5-164 and the group delay bandwidth on page 5-164.

When you specify the second input argument and the value is less than 0.999, the function treats this value as the target combined bandwidth TBW. The function designs the filter such that the measured combined bandwidth MBW meets or exceeds the target combined bandwidth TBW. The filter length is determined such that this bandwidth constraint is met.



When you specify the second input argument and the value is an integer greater than 1, the function treats this value as the desired length  $N$  of the filter. The measured combined bandwidth in this case varies with the length you specify. Larger the value of  $N$ , higher is the measured combined bandwidth MBW. See the plot below that shows this variation. As the filter length grows, the combined bandwidth of the filter moves closer towards 1. The red dashed vertical line marks the combined bandwidth for each length. The fractional delay value for each of these filters is set to 0.3



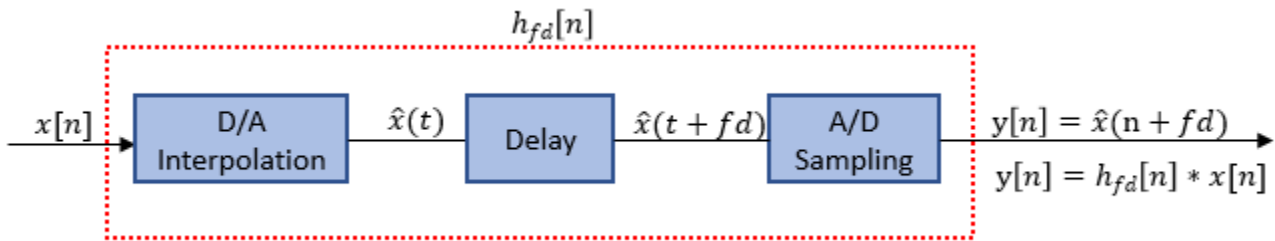
Data Types: single | double

## More About

### Fractional Delay FIR Filter

The fractional delay FIR filter is an FIR approximation of an ideal sinc shift filter with a specified fractional (non-integer) delay value  $fd$  within  $[0,1]$ .

The ideal shift filter models a band-limited D/A interpolator followed by shifted A/D uniform sampling. Assuming uniform sampling rate and shift invariant interpolation, the resulting overall system can be expressed as a convolution filter, approximated by an FIR filter. In other words,  $y[n] = h_{fd}[n] * x[n]$ , which encapsulates the D/A interpolation, shift, and A/D sampling chain as depicted in the figure.



where,

$$\begin{aligned}
 h_{fd}[n] &= \text{sinc}(n - fd), \\
 \hat{x}(t) &= \sum_k x[k] \text{sinc}(t - k) \\
 \Rightarrow \hat{x}(t + fd) &= \sum_k x[k] \text{sinc}(t + fd - k)
 \end{aligned}$$

The frequency response of the ideal shift filter is given by the following equation:

$$H_{fd}(\omega) = e^{-j\omega fd}$$

The ideal shift filter has a flat unity gain response, and a constant group delay of  $fd$ , where  $fd$  is the fractional delay value you specify.

The FIR approximation is obtained by truncating the ideal filter and weighting the truncated filter by a Kaiser window.

$$\hat{x}(n + fd) \approx y[n] = (h * x)[n], \quad \text{where } h[m] = \text{sinc}(m + fd) \cdot K_{N,\beta}[m]$$

where,  $K_{N,\beta}[m]$  is a Kaiser window of length  $N$  and has a shape parameter  $\beta$ . The Kaiser window is designed to optimize the FIR frequency response, maximizing the combined bandwidths of both gain response and group delay response.

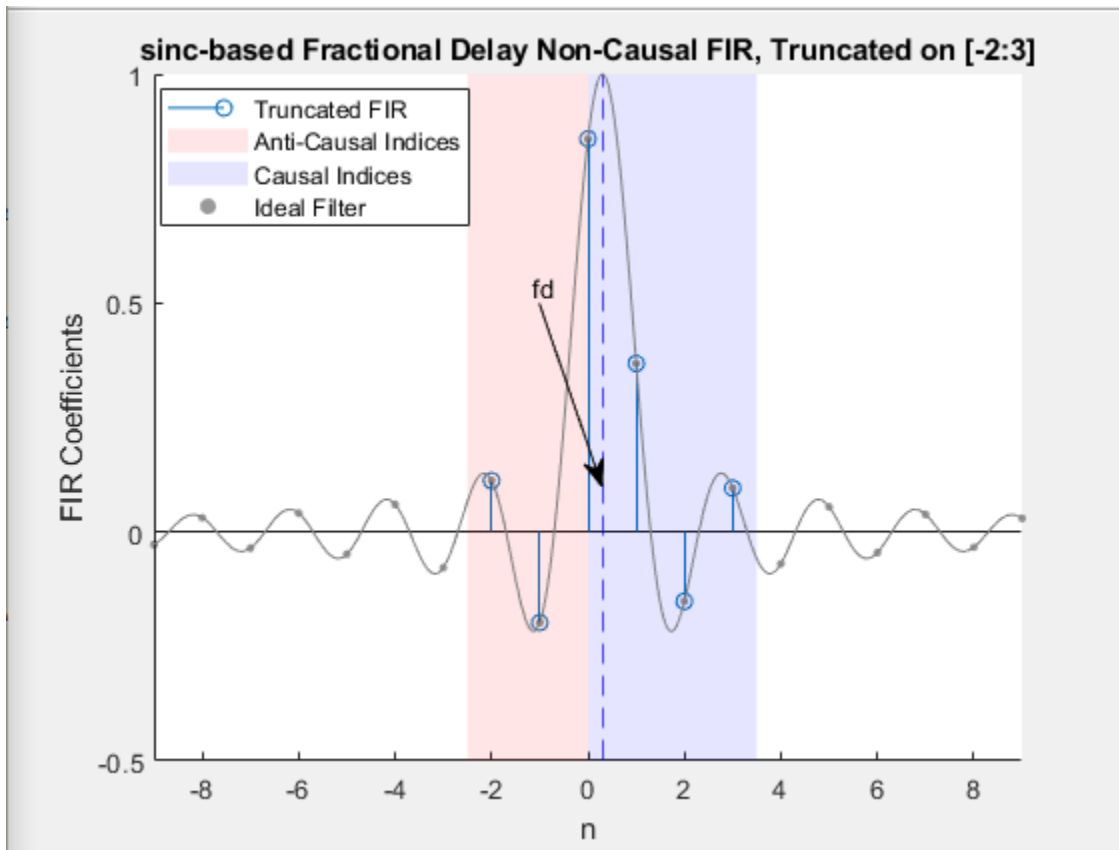
To make the FIR approximation causal, an additional shift of  $i_0$  is introduced, making the nominal group delay of the filter equal to  $i_0 + fd$ . The frequency response of the truncated filter is given by,  $H(\omega) = e^{-j\omega(fd + i_0)}$ .

For more details, see "Integer latency,  $i_0$ " on page 5-162.

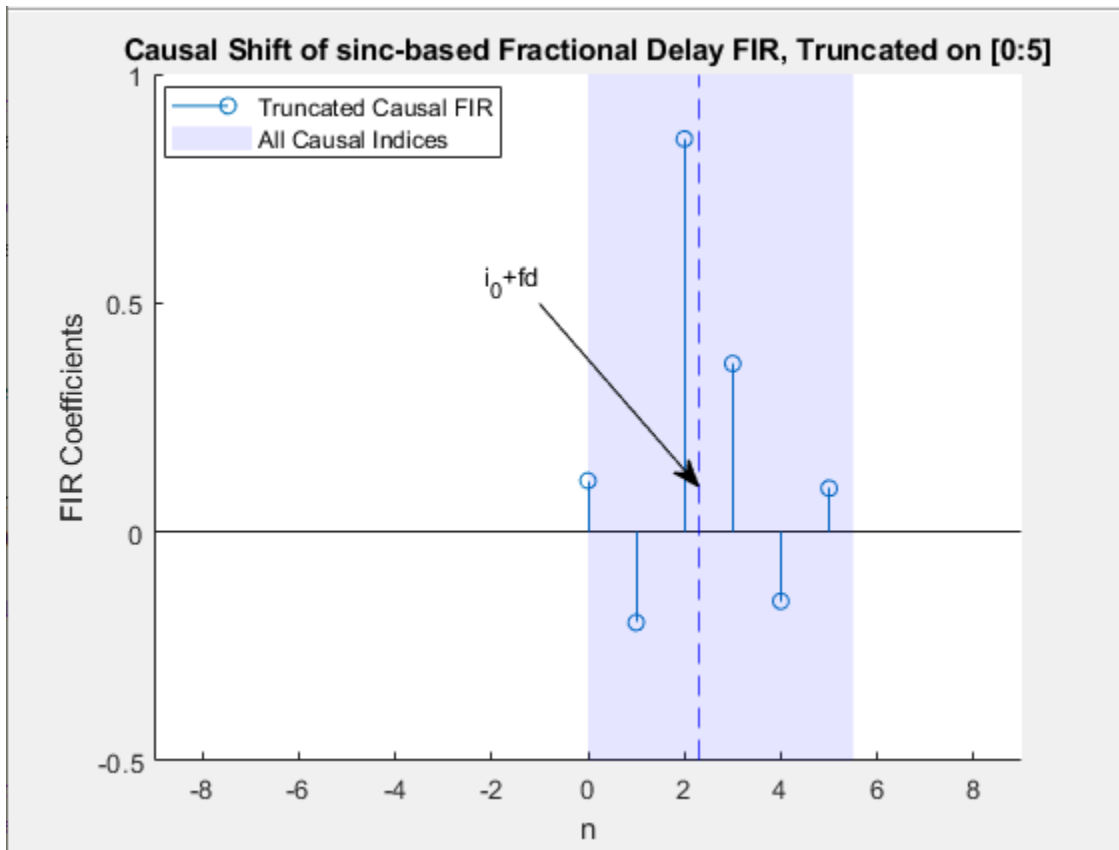
### Integer latency, $i_0$

Integer latency,  $i_0$ , is the smallest integer shift that is required to make the symmetric Kaiser window causal.

The ideal sinc shift filter is an allpass filter, which has an infinite and non-causal impulse response. To approximate this filter, the function uses a finite index Kaiser window of length  $N$  that is symmetric around the origin and captures the main lobe of the sinc function.



Due to the symmetric nature of the window (approximately equal to  $N/2$ ) is on the negative side of the origin making the truncated filter anti-causal. To make the truncated filter causal, shift the anti-causal (negative indices) part of the FIR window by an integer latency,  $i_0$ , that is approximately equal to  $N/2$ .



The overall delay of the causal FIR filter is  $i_0 + fd$ , where  $fd$  is the fractional delay value you specify.

For more details on FIR approximation, see the Causal FIR Approximations of an Ideal sinc Shift Filter section in “Design of Fractional Delay FIR Filters”.

### Gain Bandwidth

Given an FIR frequency response  $H(\omega)$ , the gain bandwidth is the largest interval  $[0 B_a]$  over which the gain response  $|H(\omega)|$  is close to 1 up to a given tolerance value,  $tol$ .

$$B_a = \max_{\omega} \{ | |H(\nu)| - 1 | < tol \ \forall \ 0 \leq \nu \leq \omega \}$$

### Group Delay Bandwidth

Given a tolerance  $tol$  and a group delay response  $G$ , the group delay bandwidth is the largest interval  $[0 B_g]$  such that the group delay is close to the nominal value,  $fd$ .

$$B_g = \max_{\omega} \{ |G(\nu) - fd - i_0| < tol \ \forall \ 0 \leq \nu \leq \omega \}$$

### Combined Bandwidth

Combined bandwidth is defined as the minimum between the gain bandwidth and the group delay bandwidth.

$$B_c = \min(B_a, B_g)$$

Combined bandwidth depends on the fractional delay  $f_d$  and the length of the FIR filter  $N$ .

## Version History

Introduced in R2021a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

### Functions

`designMultirateFIR`

### Objects

`dsp.FIRFilter` | `dsp.VariableFractionalDelay` | `dsp.Delay`

### Topics

“Design of Fractional Delay FIR Filters”

“Fractional Delay Filters Using Farrow Structures”

## designMultirateFIR

Multirate FIR filter design

### Syntax

```
B = designMultirateFIR(L,M)
B = designMultirateFIR(L,M,P)
B = designMultirateFIR(L,M,TW)
B = designMultirateFIR(L,M,P,Astop)
B = designMultirateFIR(L,M,TW,Astop)
B = designMultirateFIR( __ , 'SystemObject', flag)
```

### Description

`B = designMultirateFIR(L,M)` designs a multirate FIR filter with interpolation factor  $L$  and decimation factor  $M$ . The output  $B$  is the vector of designed FIR coefficients. To design a pure interpolator, set  $M$  to 1. To design a pure decimator, set  $L$  to 1.

`B = designMultirateFIR(L,M,P)` designs a multirate FIR filter with half-polyphase length  $P$ . By default, the half-polyphase length is 12.

`B = designMultirateFIR(L,M,TW)` designs a multirate FIR filter with interpolation factor  $L$ , decimation factor  $M$ , and normalized transition width  $TW$ .

`B = designMultirateFIR(L,M,P,Astop)` designs a multirate FIR filter with stopband attenuation  $Astop$ . By default, the stopband attenuation is 80 dB.

`B = designMultirateFIR(L,M,TW,Astop)` designs a multirate FIR filter with interpolation factor  $L$ , decimation factor  $M$ , normalized transition width  $TW$ , and stopband attenuation  $Astop$ , specified in dB.

`B = designMultirateFIR( __ , 'SystemObject', flag)` returns a vector of filter coefficients  $B$  if the flag is set to `false`, or a multirate filter System object if the flag is set to `true`.

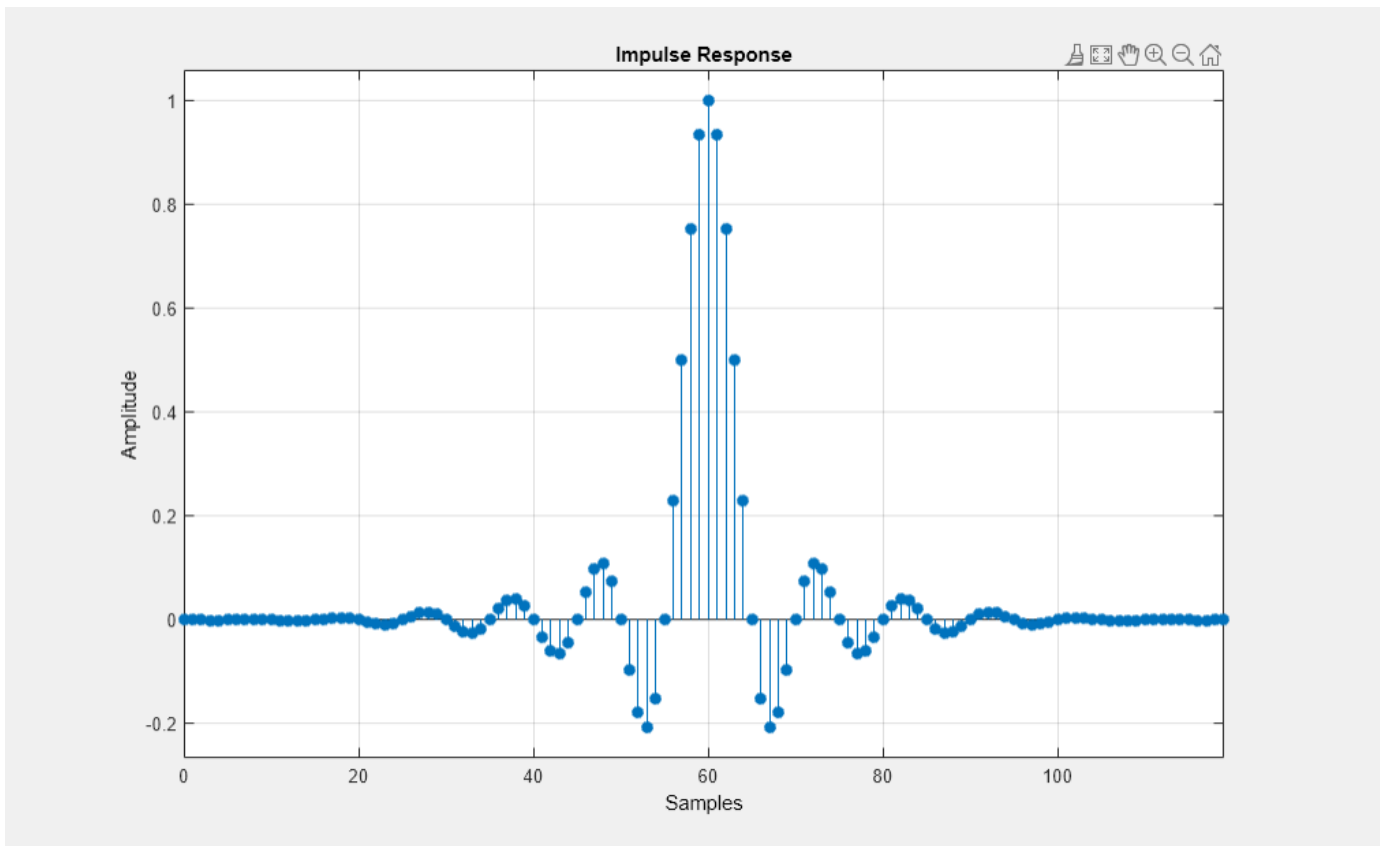
### Examples

#### Design an FIR Interpolator

To design an FIR Interpolator using the `designMultirateFIR` function, you must specify the interpolation factor of interest (usually greater than 1) and a decimation factor equal to 1. You can use the default half-polyphase length of 12 and the default stopband attenuation of 80 dB. Alternately, you can also specify the half-polyphase length and stopband attenuation values.

Design an FIR interpolator with interpolation factor set to 5. Use the default half-polyphase length of 12 and the default stopband attenuation of 80 dB.

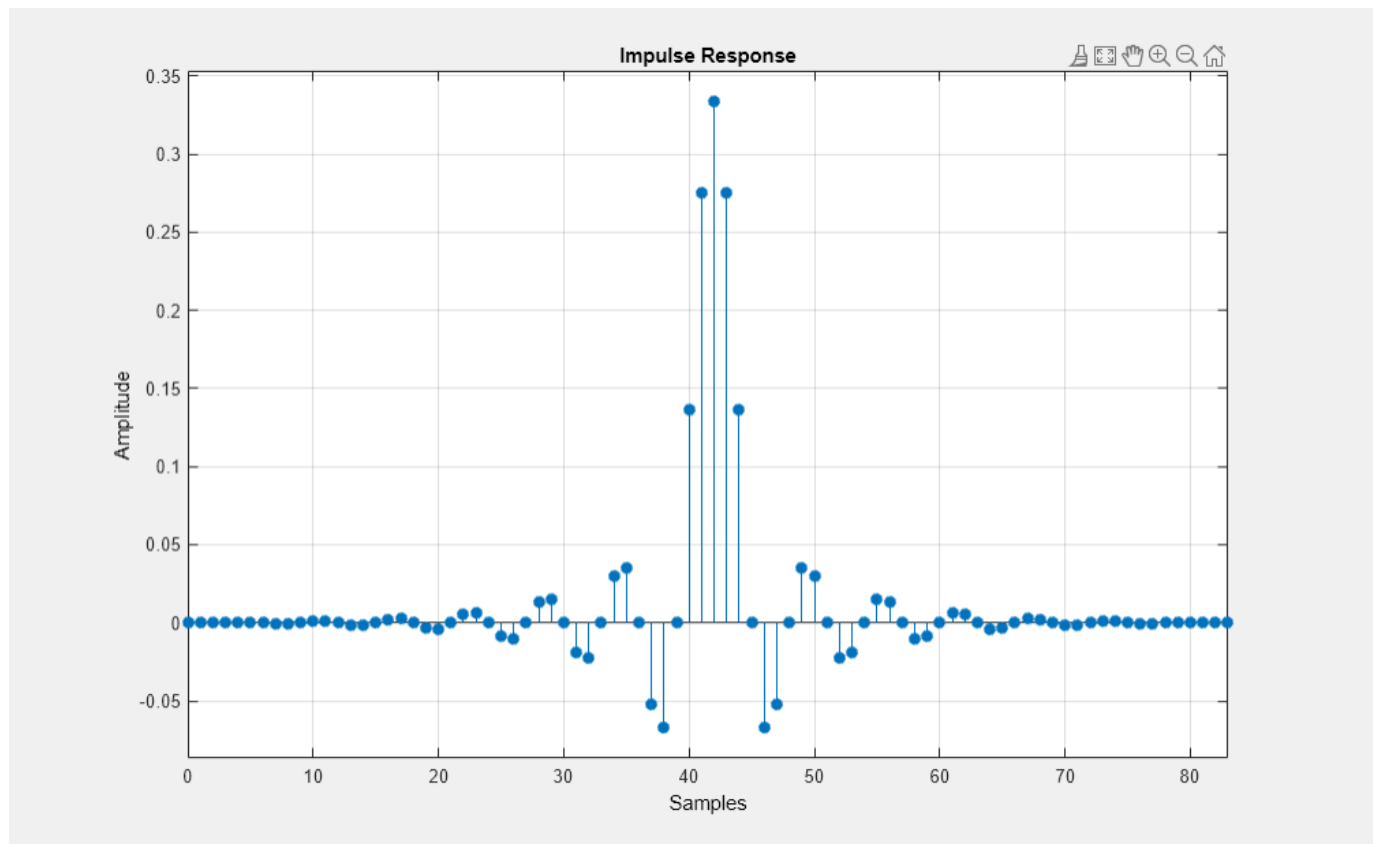
```
b = designMultirateFIR(5,1);
fvtool(b,'impulse')
```



### Design an FIR Decimator

To design an FIR Decimator using the `designMultirateFIR` function, you must specify the decimation factor of interest (usually greater than 1) and an interpolation factor equal to 1. You can use the default half-polyphase length of 12 and the default stopband attenuation of 80 dB. Alternately, you can also specify the half-polyphase length and stopband attenuation values. Design an FIR decimator with decimation factor set to 3, and half-polyphase length set to 14. Use the default stopband attenuation of 80 dB.

```
b = designMultirateFIR(1,3,14);
fvtool(b,'impulse');
```



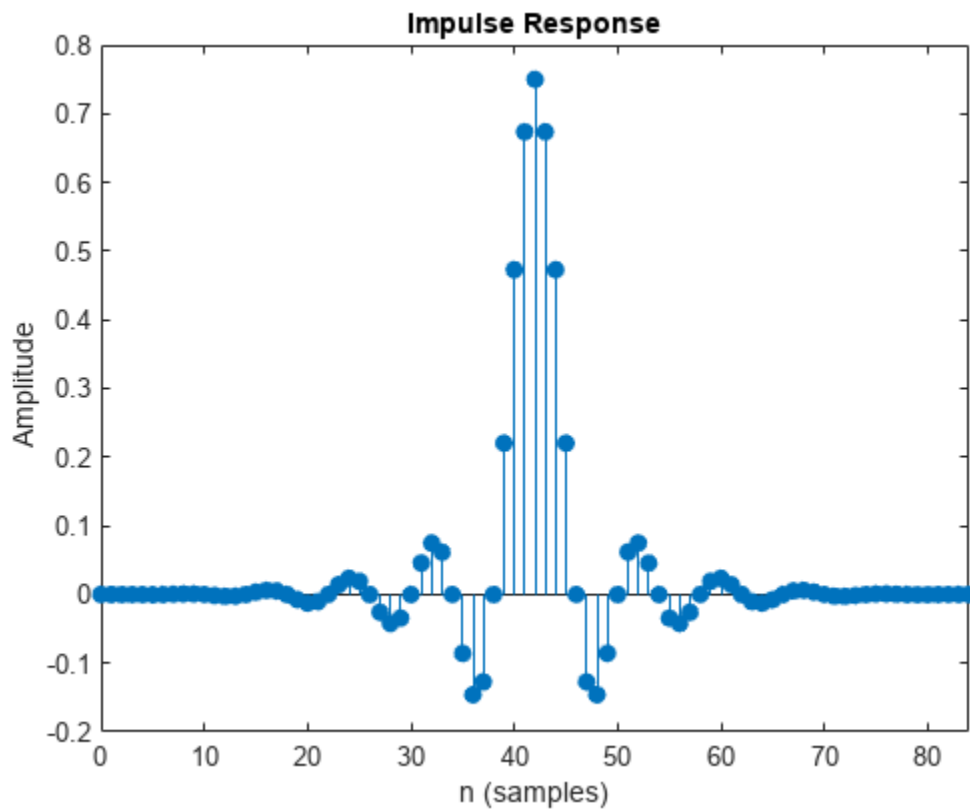
### Design an FIR Rate Converter

To design an FIR Rate Converter using the `designMultirateFIR` function, you must specify the interpolation and decimation factors of interest (usually greater than 1). In addition, you can specify either the half-polyphase length and stopband attenuation values, or the normalized transition width and stopband attenuation values.

Design an FIR rate converter with interpolation factor set to 3, decimation factor set to 4, half-polyphase length set to 14, and stopband attenuation set to 90 dB.

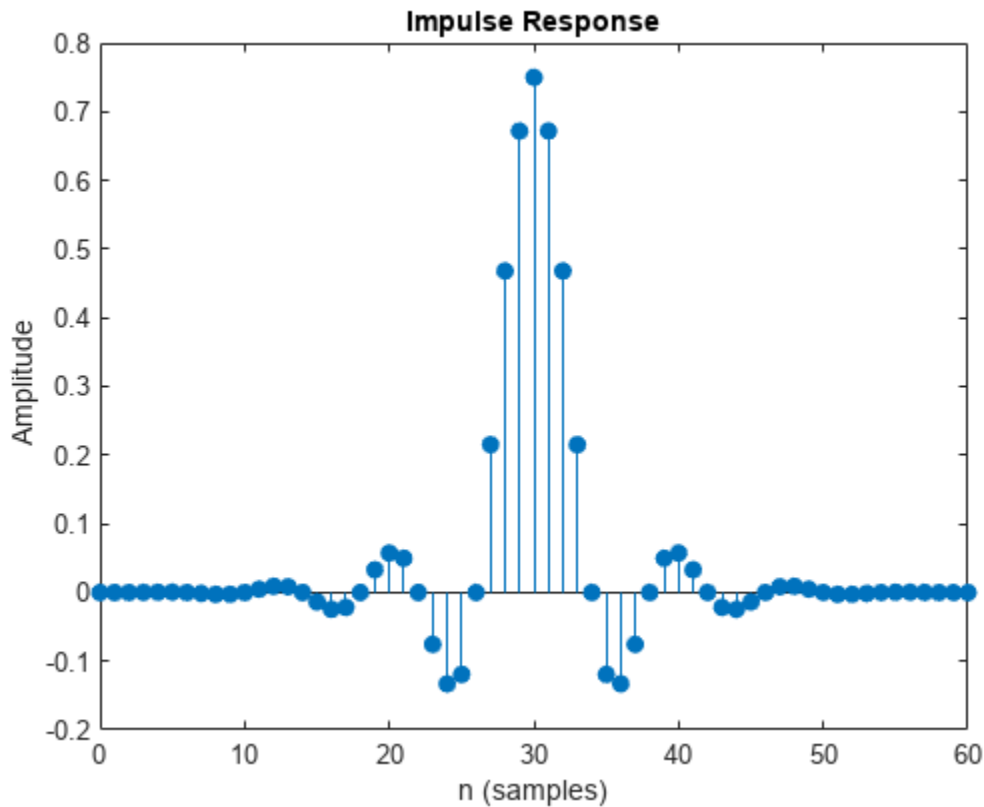
```
b = designMultirateFIR(3,4,14,90);  
impz(b,1)
```





Design an FIR rate converter with interpolation factor set to 3, decimation factor set to 4, normalized transition width set to 0.2, and stopband attenuation set to 90 dB.

```
bTW = designMultirateFIR(3,4,0.2,90);  
impz(bTW,1)
```



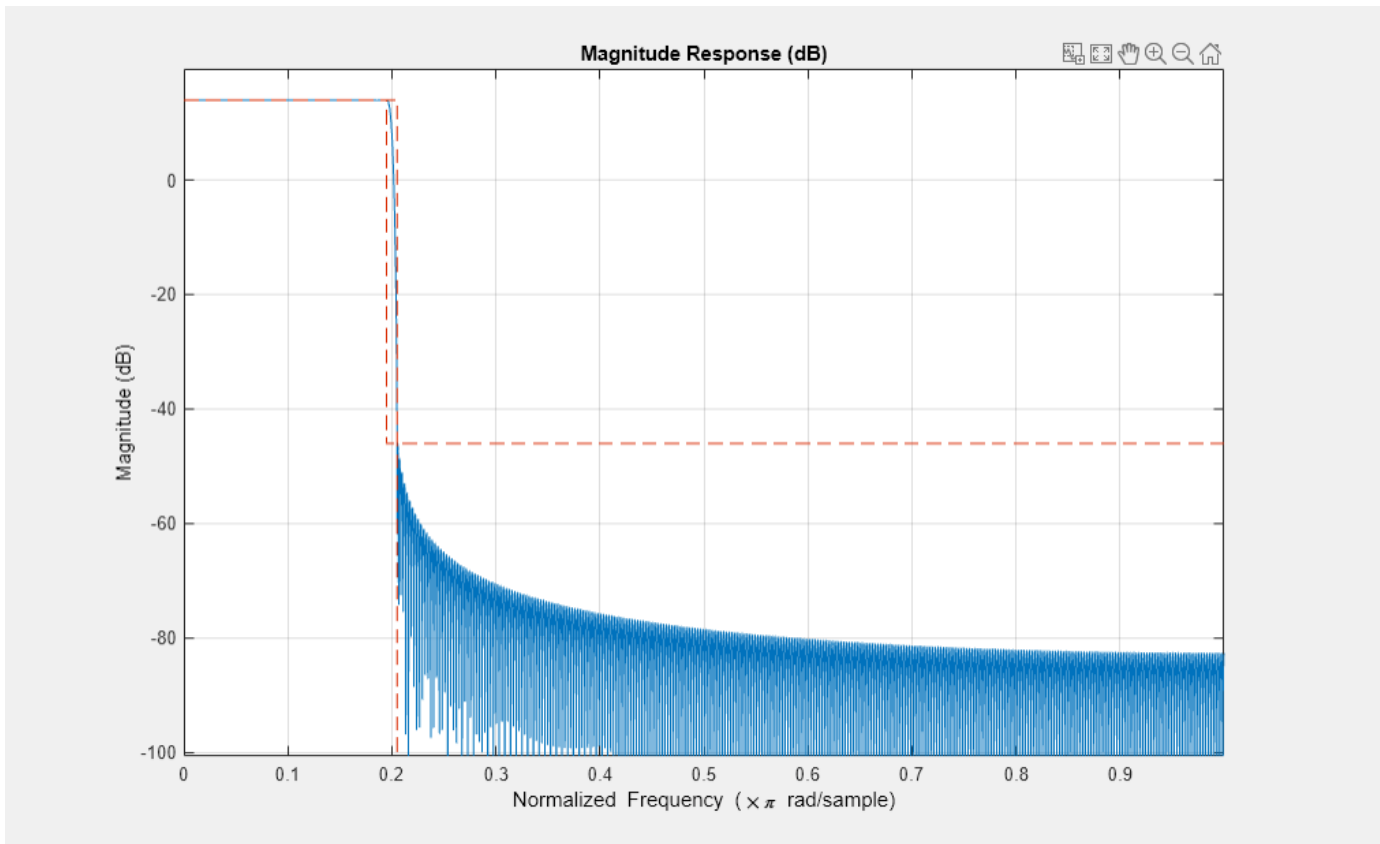
### Design a `dsp.FIRInterpolator` System object

Set the `'SystemObject'` flag to `true` in the `designMultirateFIR` function to design a multirate filter object. The design parameters specified in the function determine the type of System object the function designs.

In this example, the function designs a polyphase FIR interpolator System object™. For more details, see `dsp.FIRInterpolator`.

Create a `dsp.FIRInterpolator` object with the interpolation factor equal to 5, transition width equal to 0.01, and stopband attenuation equal to 60 dB. Set the `'SystemObject'` flag to `true` to design a multirate filter object.

```
firInterp = designMultirateFIR(5,1,0.01,60,'SystemObject',true);  
fvtool(firInterp)
```



Compute the cost of implementing the filter.

```
cost(firInterp)
```

```
ans = struct with fields:
    NumCoefficients: 582
    NumStates: 145
    MultiplicationsPerInputSample: 582
    AdditionsPerInputSample: 578
```

Measure the frequency response characteristics of the filter object.

```
measure(firInterp)
```

```
ans =
Sample Rate      : N/A (normalized frequency)
Passband Edge    : 0.195
3-dB Point       : 0.19884
6-dB Point       : 0.2
Stopband Edge    : 0.205
Passband Ripple  : 0.016474 dB
Stopband Atten.  : 60.1827 dB
Transition Width  : 0.01
```

## Input Arguments

### **L — Interpolation factor**

positive scalar integer

Interpolation factor, specified as a positive scalar integer. To design a decimator only, set  $L$  to 1.

Example: 2

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **M — Decimation factor**

positive scalar integer

Decimation factor, specified as a positive scalar integer. To design an interpolator only, set  $M$  to 1.

Example: 2

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **P — Half-polyphase length**

12 (default) | positive scalar integer

Half-polyphase length, specified as a positive scalar integer that is greater than or equal to 1.

Example: 12

Example: 20

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **TW — Normalized transition width**

real scalar in the range (0 1)

Normalized transition width of the multirate FIR filter, specified as a real scalar in the range (0 1).

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **Astop — Stopband attenuation**

90 (default) | nonnegative scalar

Stopband attenuation in dB, specified as a nonnegative real scalar greater than or equal to 0.

Example: 0.0

Example: 80.5

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **flag — System object flag**

false (default) | true

System object flag set to:

- `false` -- Function returns a vector of filter coefficients.
- `true` -- Function returns one of the following multirate filter System objects:
  - `dsp.FIRInterpolator` -- When  $L > 1$  and  $M = 1$ .
  - `dsp.FIRDecimator` -- When  $L = 1$  and  $M > 1$ .

- `dsp.FIRRateConverter` -- When  $L > 1$  and  $M > 1$ .

Data Types: `logical`

## Output Arguments

### B — Coefficients

real-valued vector

Multirate FIR filter coefficients, returned as a real-valued  $N$ -length vector.

If both  $L$  and  $M$  are equal to 1, then  $N$  equals 1.

If  $L > 1$  or  $M > 1$ , then  $N$  is given by  $N = \begin{cases} 2PR + 1 & M > L > 1 \text{ and } \text{mod}(PL, M) \neq 0 \\ 2PR & \text{otherwise} \end{cases}$ , where  $P$  is the half-polyphase length and  $R$  is defined by the following equations:

- If  $L > 1$ ,  $R = L$ .
- If  $L = 1$ ,  $R = M$ .

For more details, see the “Algorithms” on page 5-173 section.

When the 'SystemObject' flag is set to `true`, the function returns one of the following multirate filter System objects:

- `dsp.FIRInterpolator` -- When  $L > 1$  and  $M = 1$ .
- `dsp.FIRDecimator` -- When  $L = 1$  and  $M > 1$ .
- `dsp.FIRRateConverter` -- When  $L > 1$  and  $M > 1$ .

Data Types: `double`

## Algorithms

`designMultirateFIR` designs an  $R^{\text{th}}$  band Nyquist FIR filter using a Kaiser window vector to window the truncated impulse response of the FIR filter.

The filter length  $N$  is defined as one of the following:

$$N = \begin{cases} 2PR + 1 & M > L > 1 \text{ and } \text{mod}(PL, M) \neq 0 \\ 2PR & \text{otherwise} \end{cases}$$

$P$  is the half-polyphase length and  $R$  is defined as explained in “B” on page 5-0 .

The truncated impulse response  $d(n)$  is delayed by  $N/2$  samples to make it causal. The truncated and delayed impulse response is given by:

$$d(n - N/2) = \frac{\sin(w_c(n - N/2))}{\pi(n - N/2)}, \quad n = 0, \dots, \frac{N}{2}, \dots, N$$

where  $w_c = \pi/R$ .

For every  $R^{\text{th}}$  band, the impulse response of the Nyquist filters is exactly zero. Because of this property, when Nyquist filters are used for pure interpolation, the input samples remain unaltered after interpolating.

A Kaiser window is used because of its near-optimum performance while providing a robust way of designing a Nyquist filter. The window depends on two parameters: length  $N + 1$  and shape parameter  $\beta$ .

The Kaiser window is defined by:

$$w(n) = \frac{I_0\left(\beta\sqrt{1 - \left(\frac{n - N/2}{N/2}\right)^2}\right)}{I_0(\beta)}, \quad 0 \leq n \leq N,$$

where  $I_0$  is the zeroth-order modified Bessel function of the first kind.

The shape parameter  $\beta$  is calculated from:

$$\beta = \begin{cases} 0.1102(A_{\text{stop}} - 8.7) & \text{if } A_{\text{stop}} \geq 50 \\ 0.5842(A_{\text{stop}} - 21)^{0.4} + 0.07886(A_{\text{stop}} - 21) & \text{if } 21 < A_{\text{stop}} < 50 \\ 0 & \text{if } A_{\text{stop}} \leq 21, \end{cases}$$

where  $A_{\text{stop}}$  is the stopband attenuation in dB.

The windowed impulse response is given by

$$h(n) = w(n)d(n - N/2) = w(n)\frac{\sin(w_c(n - N/2))}{\pi(n - N/2)}, \quad n = 0, \dots, \frac{N}{2}, \dots, N$$

$h(n)$  for  $n = 0, \dots, N/2, \dots, N$  are the coefficients of the multirate filter. These coefficients are defined by the interpolation factor,  $L$ , and decimation factor,  $M$ .

## Version History

Introduced in R2016a

## References

- [1] Orfanidis, Sophocles J. *Introduction to Signal Processing*. Upper Saddle River, NJ: Prentice-Hall, 1996.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The function must return a vector of coefficients. Setting the 'SystemObject' flag to true is not supported for code generation.

- The inputs to the function must be constants when generating code for transition width-based filter design. In this design, you cannot specify the polyphase length (and therefore the filter length) and the function determines this value iteratively.

While generating code from the `designMultirateFIR` function, if you specify the inputs to the function as:

- Constants -- The third argument (if specified) is either treated as the transition width or the half polyphase length. If the value of the third argument is less than 1, then it is treated as the transition width. Otherwise, the third argument is treated as the half-polyphase length. This behavior is consistent with the behavior of the function.

```
% Transition width equals 0.1
codegen designMultirateFIR -args {coder.Constant(5),coder.Constant(4),coder.Constant(0.1)}
designMultirateFIR_mex(5,4,.1)
```

```
% Half-polyphase length equals 10
codegen designMultirateFIR -args {coder.Constant(5),coder.Constant(4),coder.Constant(10)}
designMultirateFIR_mex(5,4,10)
```

- Nonconstants -- The third argument (if specified) is always treated as the half-polyphase length, irrespective of its value.

For example, specify the third argument as 0.1 while generating the code. Even though the function treats a value less than 1 as the transition width, the generated code treats this value as the half-polyphase length.

```
codegen designMultirateFIR -args {5,4,0.1}
```

While running the generated code, specify the third argument as a positive integer since the half-polyphase length input must be a positive integer.

```
designMultirateFIR_mex(5,4,1)
```

However, if you specify the third argument as a scalar less than 1, the function displays the following error message.

```
designMultirateFIR_mex(5,4,0.1)
```

Expected P to be a scalar with value  $\geq 1$ .

## See Also

### Functions

`firnyquist` | `firhalfband` | `rcosdesign` | `fdesign.decimator` | `fdesign.interpolator` | `fdesign.halfband` | `designMultistageDecimator` | `designFracDelayFIR`

### Objects

`dsp.FIRInterpolator` | `dsp.FIRDecimator` | `dsp.FIRRateConverter`

## designMultistageDecimator

Multistage decimator design

### Syntax

```
C = designMultistageDecimator(M)
C = designMultistageDecimator(M,Fs,TW)
C = designMultistageDecimator(M,Fs,TW,Astop)
C = designMultistageDecimator( ____,Name,Value)
```

### Description

`C = designMultistageDecimator(M)` designs a multistage decimator that has an overall decimation factor of  $M$ . In order for  $C$  to be multistage,  $M$  must not be a prime number. For details, see “Algorithms” on page 5-186. The design process can take a while if  $M$  has many factors.

`C = designMultistageDecimator(M,Fs,TW)` designs a multistage decimator with a sampling rate of  $F_s$  and a transition width of  $TW$ . Sampling rate in this case refers to the input sampling rate of the signal before the multistage decimator.

The multistage decimator has a cutoff frequency of  $F_s/(2M)$ .

`C = designMultistageDecimator(M,Fs,TW,Astop)` specifies a minimum attenuation of  $A_{stop}$  dB for the resulting design.

`C = designMultistageDecimator( ____,Name,Value)` specifies additional design parameters using one or more name-value pair arguments.

Example: `C = designMultistageDecimator(48,48000,200,80,'NumStages','auto')` designs a multistage decimator with the least number of multiplications per input sample (MPIS).

### Examples

#### Design Efficient Decimator

Design a single-stage decimator using the `designMultirateFIR` function and a multistage decimator using the `designMultistageDecimator` function. Determine the efficiency of the two designs using the `cost` function. The implementation efficiency is characterized by two cost metrics - `NumCoefficients` and `MultiplicationsPerInputSample`.

Compute the cost of implementing both designs, and determine which design is more efficient. To make a comparison, design the filters such that their transition width is the same.

#### Initialization

Choose a decimation factor of 48, input sample rate of  $30.72 \times 48$  MHz, one-sided bandwidth of 10 MHz, and a stopband attenuation of 90 dB.

```
M = 48;
Fin = 30.72e6*M;
```



```
Astop = 90;
BW = 1e7;
```

### Using the designMultirateFIR Function

Designing the decimation filter using the `designMultirateFIR` function yields a single-stage design. Set the half-polyphase length to a finite integer, in this case 8.

```
HalfPolyLength = 8;
b = designMultirateFIR(1,M,HalfPolyLength,Astop);
d = dsp.FIRDecimator(M,b)
```

```
d =
dsp.FIRDecimator with properties:

    Main
    DecimationFactor: 48
    NumeratorSource: 'Property'
        Numerator: [0 -5.7242e-08 -1.2617e-07 -2.0736e-07 -3.0130e-07 -4.0841e-07 -5.2899e-07]
        Structure: 'Direct form'

    Show all properties
```

Compute the cost of implementing the decimator. The decimation filter requires 753 coefficients and 720 states. The number of multiplications per input sample and additions per input sample are 15.6875 and 15.6667, respectively.

```
cost(d)

ans = struct with fields:
    NumCoefficients: 753
    NumStates: 720
    MultiplicationsPerInputSample: 15.6875
    AdditionsPerInputSample: 15.6667
```

### Using the designMultistageDecimator Function

Design a multistage decimator with the same filter specifications as the single-stage design. Compute the transition width using the following relationship:

```
Fc = Fin/(2*M);
TW = 2*(Fc-BW);
```

By default, the number of stages given by the `NumStages` argument is set to 'Auto', yielding an optimal design that tries to minimize the number of multiplications per input sample.

```
c = designMultistageDecimator(M,Fin,TW,Astop)
```

```
c =
dsp.FilterCascade with properties:

    Stage1: [1x1 dsp.FIRDecimator]
    Stage2: [1x1 dsp.FIRDecimator]
    Stage3: [1x1 dsp.FIRDecimator]
    Stage4: [1x1 dsp.FIRDecimator]
    CloneStages: false
```

Calling the `info` function on `c` shows that the filter is implemented as a cascade of four `dsp.FIRDecimator` objects, with decimation factors of 3, 2, 2, and 4, respectively.

Compute the cost of implementing the decimator.

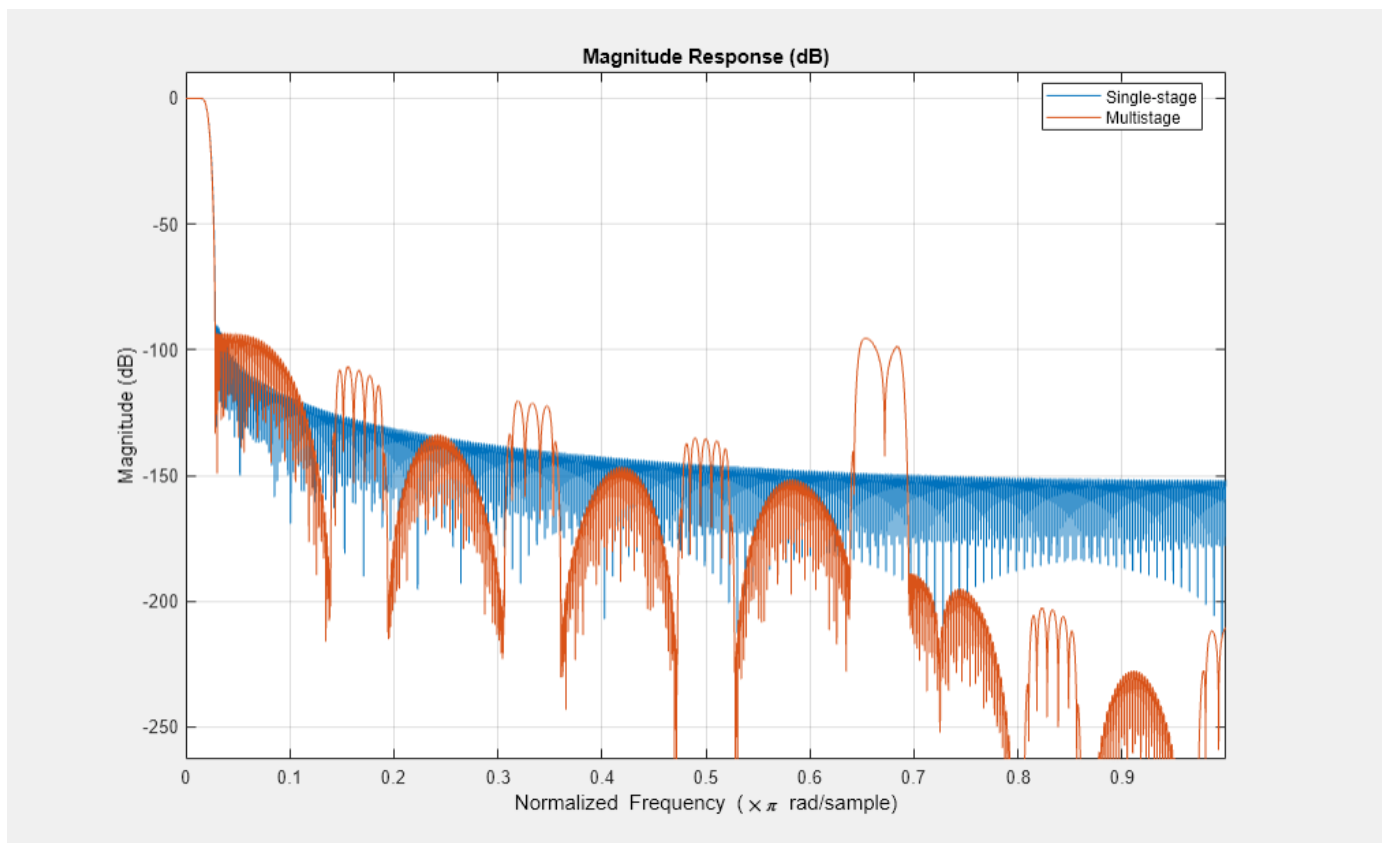
```
cost(c)
```

```
ans = struct with fields:
    NumCoefficients: 78
    NumStates: 99
    MultiplicationsPerInputSample: 7.2708
    AdditionsPerInputSample: 6.6667
```

The `NumCoefficients` and the `MultiplicationsPerInputSample` parameters are lower for the four-stage filter designed by the `designMultistageDecimator` function, making it more efficient.

Compare the magnitude response of both the designs.

```
hvft = fvtool(b,c);
legend(hvft, 'Single-stage', 'Multistage')
```



The magnitude response shows that the transition width of both the filters is the same, making the filters comparable. The `cost` function shows that implementing the multistage design is more efficient compared to implementing the single-stage design.

## Using the 'design' Option in the designMultistageDecimator Function

The filter can be made even more efficient by setting the 'CostMethod' argument of the designMultistageDecimator function to 'design'. By default, this argument is set to 'estimate'.

In the 'design' mode, the function designs each stage and computes the filter order. This yields an optimal design compared to the 'estimate' mode, where the function estimates the filter order for each stage and designs the filter based on the estimate.

Note that the 'design' option can take much longer compared to the 'estimate' option.

```
cOptimal = designMultistageDecimator(M,Fin,TW,Astop,'CostMethod','design')
```

```
cOptimal =
  dsp.FilterCascade with properties:

    Stage1: [1x1 dsp.FIRDecimator]
    Stage2: [1x1 dsp.FIRDecimator]
    Stage3: [1x1 dsp.FIRDecimator]
    Stage4: [1x1 dsp.FIRDecimator]
    CloneStages: false
```

```
cost(cOptimal)
```

```
ans = struct with fields:
    NumCoefficients: 70
    NumStates: 93
    MultiplicationsPerInputSample: 7
    AdditionsPerInputSample: 6.5417
```

## Compare Multistage Decimator Designs

Design a decimator with an overall decimation factor of 24 using the designMultistageDecimator function. Design the filter in two configurations:

- Two-stage configuration - NumStages is set to 2.
- Auto configuration - NumStages is set to 'Auto'. This configuration designs a filter with the lowest number of multiplications per input sample.

Compare the cost of implementing both the configurations.

### Initialization

Choose a decimation factor of 24, input sample rate of 6 kHz, stopband attenuation of 90 dB, and a transition width of  $0.03 \times \frac{6000}{2}$ .

```
M = 24;
Fs = 6000;
Astop = 90;
TW = 0.03*Fs/2;
```

## Design the Filter

Design the two filters using the `designMultistageDecimator` function.

```
cAuto = designMultistageDecimator(M,Fs,TW,Astop,'NumStages','Auto')
```

```
cAuto =
  dsp.FilterCascade with properties:
    Stage1: [1x1 dsp.FIRDecimator]
    Stage2: [1x1 dsp.FIRDecimator]
    Stage3: [1x1 dsp.FIRDecimator]
    CloneStages: false
```

```
cTwo = designMultistageDecimator(M,Fs,TW,Astop,'NumStages',2)
```

```
cTwo =
  dsp.FilterCascade with properties:
    Stage1: [1x1 dsp.FIRDecimator]
    Stage2: [1x1 dsp.FIRDecimator]
    CloneStages: false
```

View the filter information using the `info` function. The 'Auto' configuration designs a cascade of three FIR decimators with decimation factors 2, 3, and 4, respectively. The two-stage configuration designs a cascade of two FIR decimators with decimation factors 4 and 6, respectively.

## Compare the Cost

Compare the cost of implementing the two designs using the `cost` function.

```
cost(cAuto)
ans = struct with fields:
    NumCoefficients: 73
    NumStates: 94
    MultiplicationsPerInputSample: 8.6250
    AdditionsPerInputSample: 7.9167
```

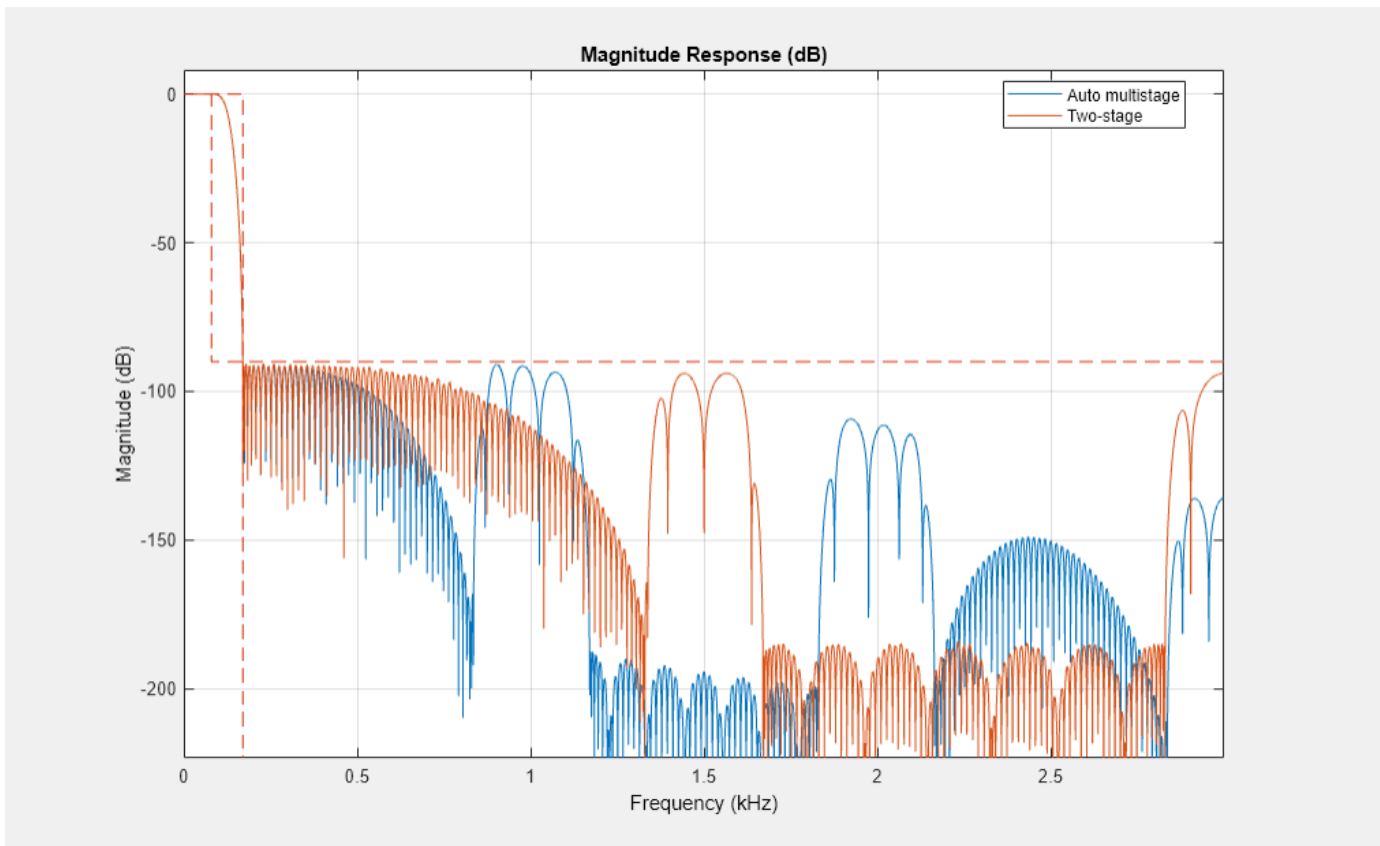
```
cost(cTwo)
ans = struct with fields:
    NumCoefficients: 100
    NumStates: 118
    MultiplicationsPerInputSample: 8.9583
    AdditionsPerInputSample: 8.6667
```

The 'Auto' configuration decimation filter yields a three-stage design that out-performs the two-stage design on all cost metrics.

## Compare the Magnitude Response

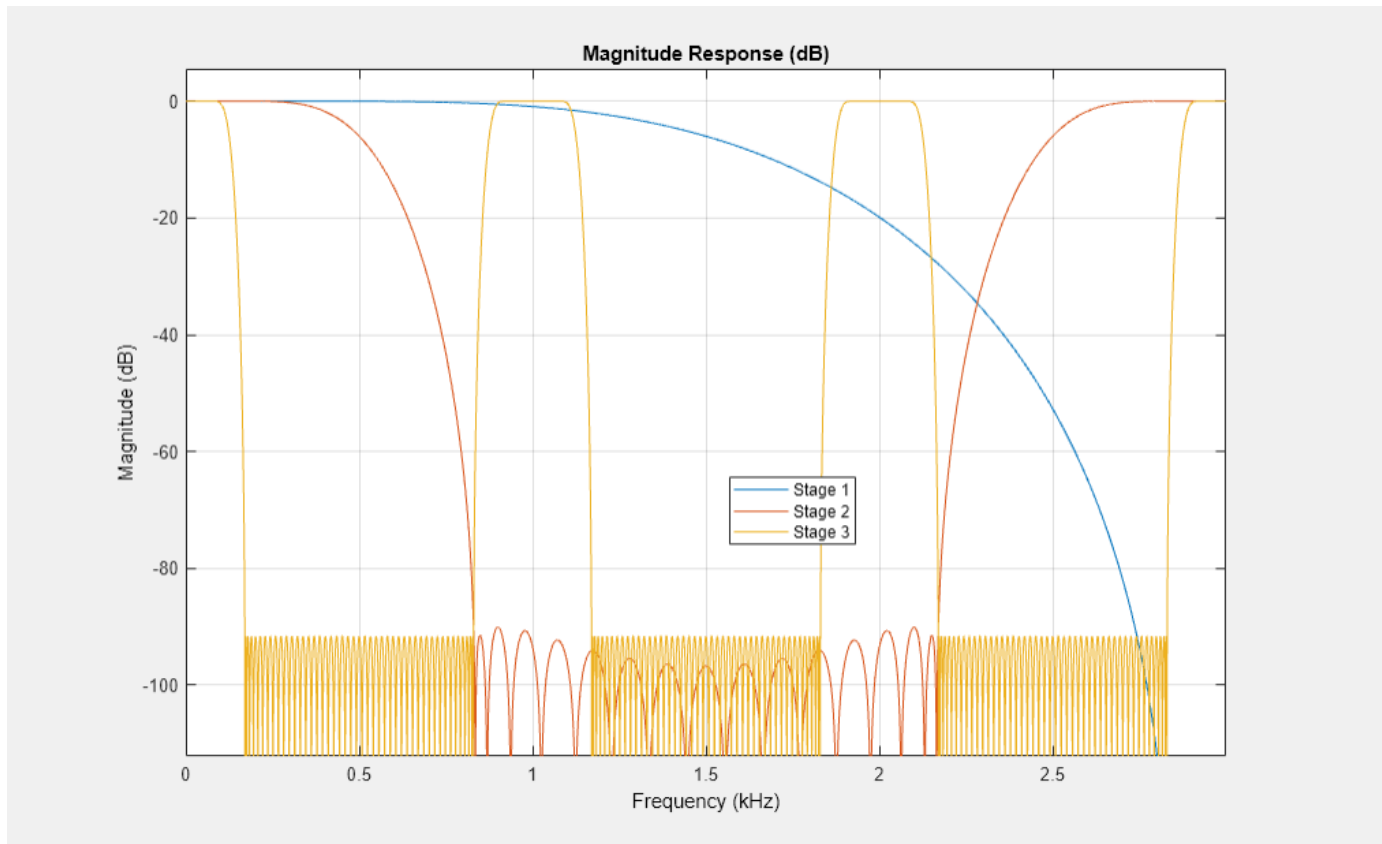
Comparing the magnitude response of the two filters, both the filters have the same transition-band behavior and follow the design specifications.

```
hfvt = fvtool(cAuto,cTwo,Analysis="magnitude");
legend(hfvt,"Auto multistage","Two-stage")
```



However, to understand where the computational savings are coming from in the three-stage design, look at the magnitude response of the three stages individually.

```
autoSt1 = cAuto.Stage1;
autoSt2 = cAuto.Stage2;
autoSt3 = cAuto.Stage3;
hfvt = fvtool(autoSt1, autoSt2, autoSt3,Analysis="magnitude");
legend(hfvt,"Stage 1","Stage 2","Stage 3")
```



The third stage provides the narrow transition width required for the overall design ( $0.03 \times F_s/2$ ). However, the third stage operates at 1.5 kHz and has spectral replicas centered at that frequency and its harmonics.

The first stage removes such replicas. This first and second stages operate at a faster rate but can afford a wide transition width. The result is a decimate-by-2 first-stage filter with only 7 nonzero coefficients and a decimate-by-3 second-stage filter with only 19 nonzero coefficients. The third stage requires 47 coefficients. Overall, there are 73 nonzero coefficients for the three-stage design and 100 nonzero coefficients for the two-stage design.

### Determining Best Multistage Decimator Design

The filters in the multistage design satisfy the following conditions:

- The combined response must meet or exceed the given design specifications.
- The combined decimation must equal the overall decimation required.

For an overall decimation factor of 48, there are several combinations of individual stages.

To obtain a design with the least number of total coefficients, set the 'MinTotalCoeffs' argument to true.

```
Astop = 80;
M = 48;
```

```

Fs = 6000;
TW = 0.03*Fs/2;
cMinCoeffs = designMultistageDecimator(M,Fs,TW,Astop,'MinTotalCoeffs',true)

```

```

cMinCoeffs =
    dsp.FilterCascade with properties:
        Stage1: [1x1 dsp.FIRDecimator]
        Stage2: [1x1 dsp.FIRDecimator]
        Stage3: [1x1 dsp.FIRDecimator]
        Stage4: [1x1 dsp.FIRDecimator]
    CloneStages: false

```

```
cost(cMinCoeffs)
```

```

ans = struct with fields:
    NumCoefficients: 48
    NumStates: 59
    MultiplicationsPerInputSample: 5.8542
    AdditionsPerInputSample: 5.0833

```

To obtain the design with the least number of multiplications per input sample, set 'NumStages' to 'auto'.

```
cMinMulti = designMultistageDecimator(M,Fs,TW,Astop,'NumStages','auto')
```

```

cMinMulti =
    dsp.FilterCascade with properties:
        Stage1: [1x1 dsp.FIRDecimator]
        Stage2: [1x1 dsp.FIRDecimator]
    CloneStages: false

```

```
cost(cMinMulti)
```

```

ans = struct with fields:
    NumCoefficients: 158
    NumStates: 150
    MultiplicationsPerInputSample: 5.6875
    AdditionsPerInputSample: 5.1667

```

Compare the magnitude response of both filters using `fvtool`. Both filters have the same transition-band behavior and a stopband attenuation that is below 80 dB.

```
hvft = fvtool(cMinCoeffs,cMinMulti)
```

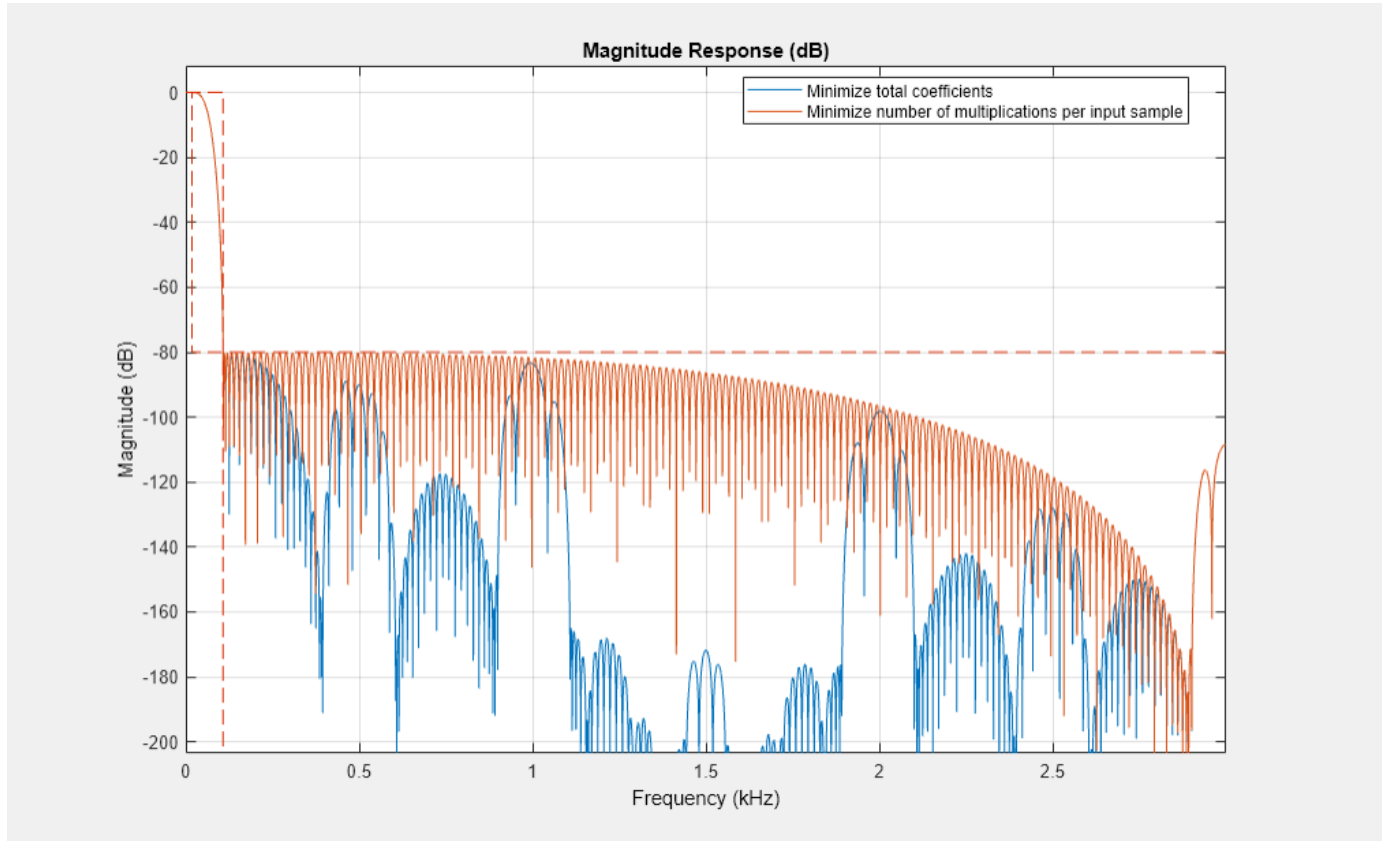
```

hvft =
    Figure (filtervisualizationtool) with properties:
        Number: []
        Name: 'Figure 1: Magnitude Response (dB)'
        Color: [0.9400 0.9400 0.9400]
        Position: [360 402 560 420]
        Units: 'pixels'

```

Use `get` to show all properties

```
legend(hvft, 'Minimize total coefficients', 'Minimize number of multiplications per input sample')
```



## Input Arguments

### M — Overall decimation factor

positive integer

Overall decimation factor, specified as a positive integer greater than one. In order for `C` to be multistage, `M` must not be a prime number. For details, see “Algorithms” on page 5-186.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### Fs — Input sampling rate

48000 (default) | positive real scalar

Input sampling rate prior to the multistage decimator, specified as a positive real scalar. If not specified, `Fs` defaults to 48,000 Hz. The multistage decimator has a cutoff frequency of  $F_s/(2M)$ .

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### TW — Transition width

$0.2 \times F_s/M$  (default) | positive real scalar



Transition width, specified as a positive real scalar less than  $F_s/M$ . If not specified,  $TW$  defaults to  $0.2 \times F_s/M$ . Transition width must be less than  $F_s/M$ .

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **Astop** — Minimum stopband attenuation

positive real scalar

Minimum stopband attenuation for the resulting design, specified as a positive real scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **Name-Value Pair Arguments**

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: `C = designMultistageDecimator(48,48000,200,80,'NumStages','auto')` designs a multistage decimator with the lowest number of multiplications per input sample.

### **NumStages** — Number of decimator stages

'auto' (default) | positive integer

Number of decimator stages, specified as a positive integer. If set to 'auto', the design algorithm determines the number of stages that result in the lowest number of multiplications per input sample. If specified as a positive integer,  $N$ , the overall decimation factor,  $M$ , must be able to factor into at least  $N$  factors, not counting 1 or  $M$  as factors.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **MinTotalCoeffs** — Minimize total number of coefficients

false (default) | true

When false, the design algorithm minimizes the number of multiplications per input sample. When true, the design algorithm minimizes the total number of coefficients.

Data Types: `logical`

### **CostMethod** — Cost computation method

'estimate' (default) | 'design'

Cost computation method, specified as either:

- 'estimate' -- The function estimates the filter order required for each stage and designs the filter based on the estimate. This method is faster than 'design', but can lead to suboptimal designs.
- 'design' -- The function designs each stage and computes the filter order. This method leads to an optimal overall design.

Data Types: `char`

### **CostTolerance** — Tolerance

1e-6 (default) | positive scalar

Tolerance, specified as a positive scalar. The tolerance is used to determine the multistage configuration with the least MPIS. When multiple configurations result in the same lowest MPIS within the tolerance specified, the configuration that yields the lowest number of coefficients overall is chosen. To view the total number of coefficients and MPIS for a specific filter, use the `cost` function.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

## Output Arguments

### C — Designed filter

`dsp.FilterCascade`

Designed filter, returned as a `dsp.FilterCascade` System object. The filter is a cascade of the multiple stages designed by the function. The number of stages is determined by the `'NumStages'` argument.

To get information about each filter stage, call the `info` function on the C argument.

## Algorithms

The overall decimation factor is split into smaller factors with each factor being the decimation factor of the corresponding individual stage. The combined decimation of all the individual stages must equal the overall decimation. The combined response must meet or exceed the given design specifications.

The function determines the number of decimator stages through the `'NumStages'` argument. The sequence of stages is determined based on the implementation cost. By default, `'NumStages'` is set to `'auto'`, resulting in a sequence that gives the lowest number of MPIS. When multiple configurations result in the same lowest MPIS within the tolerance specified, the configuration that yields the lowest number of coefficients overall is chosen. If `'MinTotalCoeffs'` is set to `true`, the function determines the sequence that requires the lowest number of total coefficients.

By default, the `'CostMethod'` is set to `'estimate'`. In this mode, the function estimates the filter order required for each stage and designs the filter based on the estimate. This method is faster than `'design'`, but can lead to suboptimal designs. For an optimal design, set `'CostMethod'` to `'design'`. In this mode, the function designs each stage and computes the filter order.

## Version History

Introduced in R2018b

## See Also

### Objects

`dsp.FilterCascade` | `dsp.FIRDecimator`

### Functions

`designMultirateFIR` | `designMultistageInterpolator` | `info` | `cost`

**Topics**

“Multistage Rate Conversion”

## designMultistageInterpolator

Multistage interpolator design

### Syntax

```
C = designMultistageInterpolator(L)
C = designMultistageInterpolator(L,Fs,TW)
C = designMultistageInterpolator(L,Fs,TW,Astop)
C = designMultistageInterpolator( ___,Name,Value)
```

### Description

`C = designMultistageInterpolator(L)` designs a multistage interpolator that has an overall interpolation factor of  $L$ . In order for  $C$  to be multistage,  $L$  must not be a prime number. For details, see “Algorithms” on page 5-196. The design process can take a while if  $L$  has many factors.

`C = designMultistageInterpolator(L,Fs,TW)` designs a multistage interpolator with a sampling rate of  $F_s$  and a transition width of  $TW$ . Sampling rate in this case refers to the output sampling rate of the signal after the multistage interpolator.

The multistage interpolator has a cutoff frequency of  $F_s/(2L)$ .

`C = designMultistageInterpolator(L,Fs,TW,Astop)` specifies a minimum attenuation of  $A_{stop}$  dB for the resulting design.

`C = designMultistageInterpolator( ___,Name,Value)` specifies additional design parameters using one or more name-value pair arguments.

Example: `C = designMultistageInterpolator(48,48000,200,80,'NumStages','auto')` designs a multistage interpolator with the least number of multiplications per input sample (MPIS).

### Examples

#### Design Efficient Interpolator

Design a single-stage interpolator using the `designMultirateFIR` function and a multistage interpolator using the `designMultistageInterpolator` function. Determine the efficiency of the two designs using the `cost` function. The implementation efficiency is characterized by two cost metrics: `NumCoefficients` and `MultiplicationsPerInputSample`.

Compute the cost of implementing both designs, and determine which design is more efficient. To make a comparison, design the filters such that their transition width is the same.

#### Initialization

Choose an interpolation factor of 48, input sample rate of 30.72 MHz, one-sided bandwidth of 100 kHz, and a stopband attenuation of 90 dB.

```
L = 48;
Fin = 30.72e6;
```

```
Astop = 90;
BW = 1e5;
```

### Using the designMultirateFIR Function

Designing the interpolation filter using the `designMultirateFIR` function yields a single-stage design. Set the half-polyphase length to a finite integer, in this case 4.

```
HalfPolyLength = 4;
b = designMultirateFIR(L,1,HalfPolyLength,Astop);
d = dsp.FIRInterpolator(L,b)
```

```
d =
dsp.FIRInterpolator with properties:
```

```
InterpolationFactor: 48
  NumeratorSource: 'Property'
      Numerator: [0 -6.0692e-06 -1.4587e-05 -2.5889e-05 -4.0313e-05 -5.8200e-05 -7.9886e-05]
```

```
Show all properties
```

Compute the cost of implementing the interpolator. The interpolation filter requires 376 coefficients and 7 states. The number of multiplications per input sample and additions per input sample are 376 and 329, respectively.

```
cost(d)
```

```
ans = struct with fields:
      NumCoefficients: 376
      NumStates: 7
  MultiplicationsPerInputSample: 376
    AdditionsPerInputSample: 329
```

### Using the designMultistageInterpolator Function

Design a multistage interpolator with the same filter specifications as the single-stage design. Compute the transition width using the following relationship:

```
Fc = Fin/(2*L);
TW = 2*(Fc-BW);
```

By default, the number of stages given by the `NumStages` argument is set to 'Auto', yielding an optimal design that tries to minimize the number of multiplications per input sample.

```
c = designMultistageInterpolator(L,Fin,TW,Astop)
```

```
c =
dsp.FilterCascade with properties:
```

```
Stage1: [1x1 dsp.FIRInterpolator]
Stage2: [1x1 dsp.FIRInterpolator]
CloneStages: false
```

Calling the `info` function on `c` shows that the filter is implemented as a cascade of two `dsp.FIRInterpolator` objects with interpolation factors of 24 and 2, respectively.

Compute the cost of implementing the interpolator.

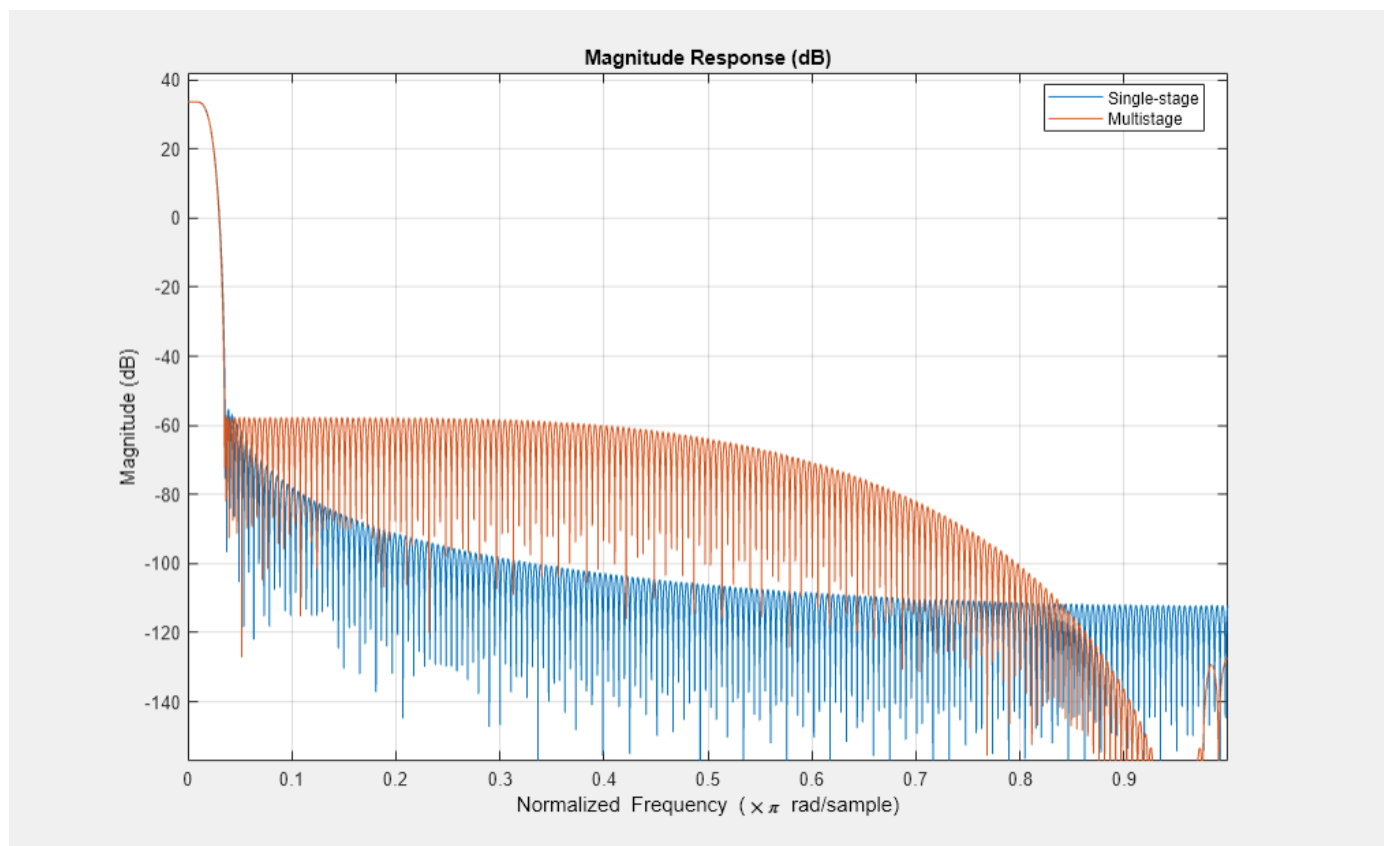
```
cost(c)
```

```
ans = struct with fields:
    NumCoefficients: 184
    NumStates: 12
    MultiplicationsPerInputSample: 322
    AdditionsPerInputSample: 275
```

The `NumCoefficients` and the `MultiplicationsPerInputSample` parameters are lower for the two-stage filter designed by the `designMultistageInterpolator` function, making it more efficient.

Compare the magnitude response of both designs.

```
hvft = fvtool(b,c);
legend(hvft, 'Single-stage', 'Multistage')
```



The magnitude response shows that the transition width of both the filters is the same, making the filters comparable. The `cost` function shows that implementing the multistage design is more efficient compared to implementing the single-stage design.

## Using the 'design' Option in the designMultistageInterpolator Function

The filter can be made even more efficient by setting the 'CostMethod' argument of the designMultistageInterpolator function to 'design'. By default, this argument is set to 'estimate'.

In the 'design' mode, the function designs each stage and computes the filter order. This yields an optimal design compared to the 'estimate' mode, where the function estimates the filter order for each stage and designs the filter based on the estimate.

Note that the 'design' option can take much longer compared to the 'estimate' option.

```
cOptimal = designMultistageInterpolator(L,Fin,TW,Astop,'CostMethod','design')
```

```
cOptimal =
  dsp.FilterCascade with properties:

    Stage1: [1x1 dsp.FIRInterpolator]
    Stage2: [1x1 dsp.FIRInterpolator]
    Stage3: [1x1 dsp.FIRInterpolator]
  CloneStages: false
```

```
cost(cOptimal)
```

```
ans = struct with fields:
    NumCoefficients: 74
    NumStates: 17
  MultiplicationsPerInputSample: 296
    AdditionsPerInputSample: 249
```

## Compare Multistage Interpolator Designs

Design an interpolator with an overall interpolation factor of 24 using the designMultistageInterpolator function. Design the filter in two configurations:

- Two-stage configuration - NumStages is set to 2.
- Auto configuration - NumStages is set to 'Auto'. This configuration designs a filter with the lowest number of multiplications per input sample.

Compare the cost of implementing both the configurations.

### Initialization

Choose an interpolation factor of 24, input sample rate of 6 kHz, stopband attenuation of 90 dB, and a transition width of  $0.03 \times 6000/2$ .

```
L = 24;
Fs = 6000;
Astop = 90;
TW = 0.03*Fs/2;
```

### Design the Filter

Design the two filters using the designMultistageInterpolator function.

```
cAuto = designMultistageInterpolator(L,Fs,TW,Astop,'NumStages','Auto')
```

```
cAuto =
  dsp.FilterCascade with properties:

    Stage1: [1x1 dsp.FIRInterpolator]
    Stage2: [1x1 dsp.FIRInterpolator]
    Stage3: [1x1 dsp.FIRInterpolator]
  CloneStages: false
```

```
cTwo = designMultistageInterpolator(L,Fs,TW,Astop,'NumStages',2)
```

```
cTwo =
  dsp.FilterCascade with properties:

    Stage1: [1x1 dsp.FIRInterpolator]
    Stage2: [1x1 dsp.FIRInterpolator]
  CloneStages: false
```

View the filter information using the `info` function. The 'Auto' configuration designs a cascade of three FIR interpolators with interpolation factors 4, 3, and 2, respectively. The two-stage configuration designs a cascade of two FIR interpolators with interpolation factors 6 and 4, respectively.

### Compare the Cost

Compare the cost of implementing the two designs using the `cost` function.

```
cost(cAuto)
```

```
ans = struct with fields:
    NumCoefficients: 70
    NumStates: 28
  MultiplicationsPerInputSample: 190
  AdditionsPerInputSample: 167
```

```
cost(cTwo)
```

```
ans = struct with fields:
    NumCoefficients: 98
    NumStates: 22
  MultiplicationsPerInputSample: 208
  AdditionsPerInputSample: 185
```

The 'Auto' configuration interpolation filter yields a three-stage design that out-performs the two-stage design in terms of `NumCoefficients` and `MultiplicationsPerInputSample` metrics.

### Determining Best Multistage Interpolator Design

The filters in the multistage design satisfy the following conditions:

- The combined response must meet or exceed the given design specifications.



- The combined interpolation must equal the overall interpolation required.

For an overall interpolation factor of 50, there are several combinations of individual stages.

To obtain a design with the least number of total coefficients, set the 'MinTotalCoeffs' argument to true.

```
Astop = 80;
L = 50;
Fs = 6000;
TW = 0.03*Fs/2;
cMinCoeffs = designMultistageInterpolator(L,Fs,TW,Astop,'MinTotalCoeffs',true)
```

```
cMinCoeffs =
  dsp.FilterCascade with properties:

      Stage1: [1x1 dsp.FIRInterpolator]
      Stage2: [1x1 dsp.FIRInterpolator]
      Stage3: [1x1 dsp.FIRInterpolator]
  CloneStages: false
```

```
cost(cMinCoeffs)
```

```
ans = struct with fields:
      NumCoefficients: 58
      NumStates: 18
  MultiplicationsPerInputSample: 306
  AdditionsPerInputSample: 257
```

To obtain the design with the lowest number of multiplications per input sample, set 'NumStages' to 'auto'.

```
cMinMulti = designMultistageInterpolator(L,Fs,TW,Astop,'NumStages','auto')
```

```
cMinMulti =
  dsp.FilterCascade with properties:

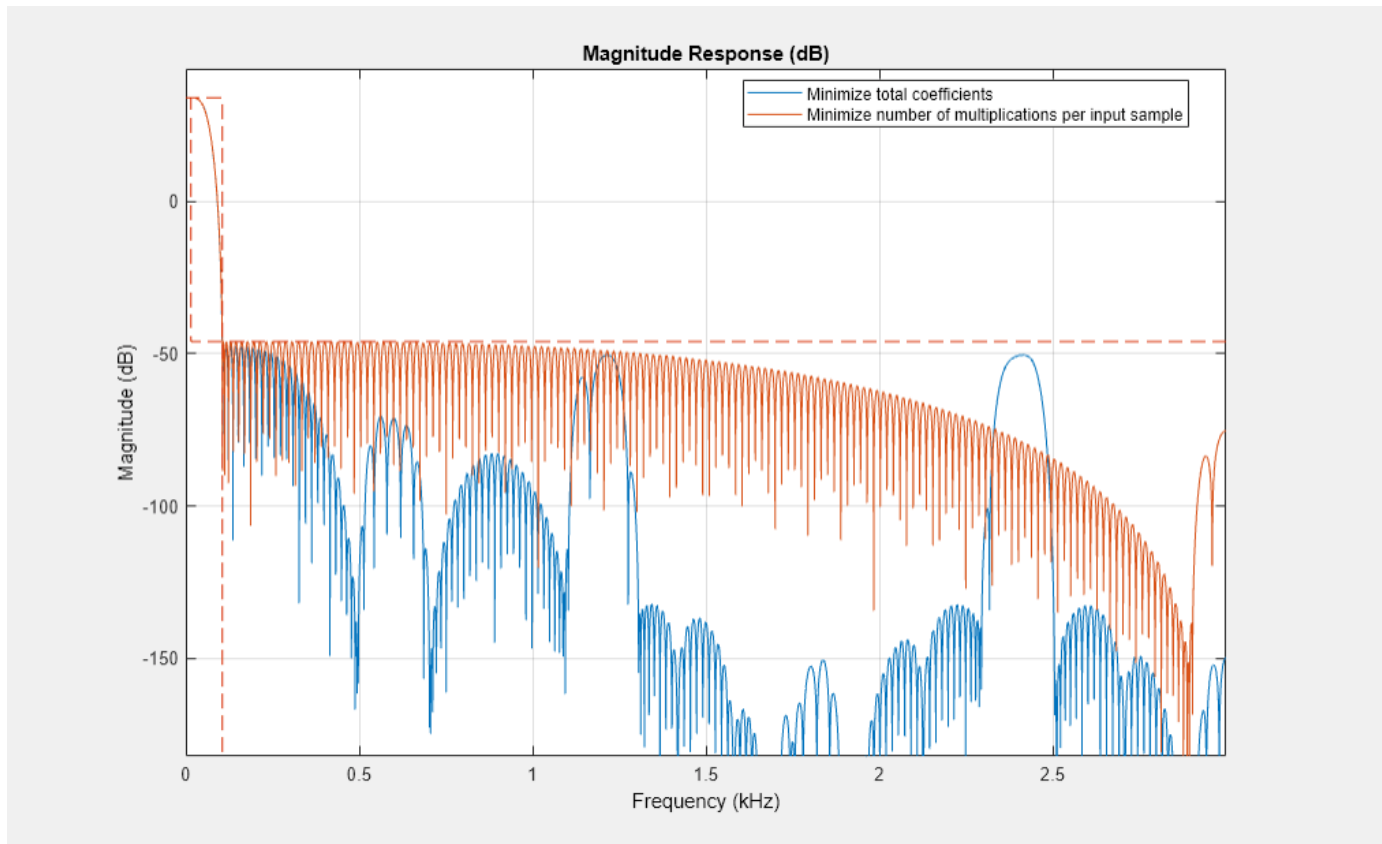
      Stage1: [1x1 dsp.FIRInterpolator]
      Stage2: [1x1 dsp.FIRInterpolator]
  CloneStages: false
```

```
cost(cMinMulti)
```

```
ans = struct with fields:
      NumCoefficients: 156
      NumStates: 9
  MultiplicationsPerInputSample: 252
  AdditionsPerInputSample: 203
```

Compare the magnitude response of both the filters using `fvtool`. Both filters have the same transition-band behavior and a stopband attenuation that is below 80 dB.

```
hvft = fvtool(cMinCoeffs,cMinMulti);
legend(hvft,'Minimize total coefficients','Minimize number of multiplications per input sample')
```



## Input Arguments

### L — Overall interpolation factor

positive integer

Overall interpolation factor, specified as a positive integer greater than one. In order for C to be multistage, L must not be a prime number. For details, see “Algorithms” on page 5-196.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### Fs — Input sampling rate

48000 (default) | positive real scalar

Sampling rate of the output signal after the multistage interpolator, specified as a positive real scalar. If not specified, Fs defaults to 48,000 Hz. The multistage interpolator has a cutoff frequency of  $F_s/(2L)$ .

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### TW — Transition width

$0.2 \times F_s/L$  (default) | positive real scalar

Transition width, specified as a positive real scalar less than  $F_s/L$ . If not specified, TW defaults to  $0.2 \times F_s/L$ . Transition width must be less than  $F_s/L$ .

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**Astop — Minimum stopband attenuation**

positive real scalar

Minimum stopband attenuation for the resulting design, specified as a positive real scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**Name-Value Pair Arguments**

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: `C = designMultistageInterpolator(48,48000,200,80,'NumStages','auto')` designs a multistage interpolator with the lowest number of multiplications per input sample.

**NumStages — Number of interpolator stages**`'auto'` (default) | positive integer

Number of interpolator stages, specified as a positive integer. If set to `'auto'`, the design algorithm determines the number of stages that result in the lowest number of multiplications per input sample. If specified as a positive integer,  $N$ , the overall interpolation factor,  $L$ , must be able to factor into at least  $N$  factors, not counting 1 or  $L$  as factors.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**MinTotalCoeffs — Minimize total number of coefficients**`false` (default) | `true`

When `false`, the design algorithm minimizes the number of multiplications per input sample. When `true`, the design algorithm minimizes the total number of coefficients.

Data Types: `logical`

**CostMethod — Cost computation method**`'estimate'` (default) | `'design'`

Cost computation method, specified as either:

- `'estimate'` -- The function estimates the filter order required for each stage and designs the filter based on the estimate. This method is faster than `'design'`, but can lead to suboptimal designs.
- `'design'` -- The function designs each stage and computes the filter order.

Data Types: `char`

**CostTolerance — Tolerance**`1e-6` (default) | positive scalar

Tolerance, specified as a positive scalar. The tolerance is used to determine the multistage configuration with the least number of MPIS. When multiple configurations result in the same lowest MPIS within the tolerance specified, the configuration that yields the lowest number of coefficients overall is chosen. To view the total number of coefficients and MPIS for a specific filter, use the `cost` function.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

## Output Arguments

### C — Designed filter

`dsp.FilterCascade`

Designed filter, returned as a `dsp.FilterCascade` System object. The filter is a cascade of the multiple stages designed by the function. The number of stages is determined by the `'NumStages'` argument.

To get information about each filter stage, call the `info` function on the C argument.

## Algorithms

The overall interpolation factor is split into smaller factors with each factor being the interpolation factor of the corresponding individual stage. The combined interpolation of all the individual stages must equal the overall interpolation. The combined response must meet or exceed the given design specifications.

The function determines the number of interpolator stages through the `'NumStages'` argument. The sequence of stages is determined based on the implementation cost. By default, `'NumStages'` is set to `'auto'`, resulting in a sequence that gives the lowest number of MPIS. When multiple configurations result in the same lowest MPIS within the tolerance specified, the configuration that yields the lowest number of coefficients overall is chosen. If `'MinTotalCoeffs'` is set to `true`, the function determines the sequence that requires the lowest number of total coefficients.

By default, the `'CostMethod'` is set to `'estimate'`. In this mode, the function estimates the filter order required for each stage and designs the filter based on the estimate. This method is faster but can lead to suboptimal designs. For an optimal design, set `'CostMethod'` to `'design'`. In this mode, the function designs each stage and computes the filter order.

## Version History

Introduced in R2018b

## See Also

### Objects

`dsp.FilterCascade` | `dsp.FIRInterpolator`

### Functions

`designMultirateFIR` | `info` | `cost` | `designMultistageDecimator`

### Topics

“Multistage Rate Conversion”

# designmethods

Methods available for designing filter from specification object

## Syntax

```
methods = designmethods(designSpecs, 'SystemObject', true)
methods = designmethods(designSpecs, 'default')
methods = designmethods(designSpecs, TYPE, 'SystemObject', true)
methods = designmethods(designSpecs, 'full', 'SystemObject', true)
```

## Description

`methods = designmethods(designSpecs, 'SystemObject', true)` returns the available design methods for designing filter System objects for the filter specification object, `designSpecs`.

`methods = designmethods(designSpecs, 'default')` returns the default design method for the filter specification object `designSpecs`.

`methods = designmethods(designSpecs, TYPE, 'SystemObject', true)` returns the TYPE design methods for the filter specification object, `designSpecs`. TYPE can be either 'FIR' or 'IIR'.

`methods = designmethods(designSpecs, 'full', 'SystemObject', true)` returns the full name for each of the available design methods. For example, `designmethods` with the 'full' argument returns Butterworth for the `butter` method.

## Examples

### Valid Design Methods for Lowpass Filter

Construct a lowpass filter design specification object and determine the valid design methods.

```
designSpecs = fdesign.lowpass('Fp,Fst,Ap,Ast',500,600,0.5,60,1e4);
methods = designmethods(designSpecs, 'SystemObject', true)
```

```
methods = 8x1 cell
    {'butter'    }
    {'cheby1'   }
    {'cheby2'   }
    {'ellip'    }
    {'equiripple'}
    {'ifir'     }
    {'kaiserwin' }
    {'multistage'}
```

Use `help` to get more information on the Chebyshev type I design method.

```
help(designSpecs, methods{2})
```

DESIGN Design a Chebyshev type I iir filter.

HD = DESIGN(D, 'cheby1') designs a Chebyshev type I filter specified by the FDESIGN object D, and returns the DFILT/MFILT object HD.

HD = DESIGN(D, ..., 'SystemObject', true) implements the filter, HD, using a System object instead of a DFILT/MFILT object.

HD = DESIGN(..., 'FilterStructure', STRUCTURE) returns a filter with the structure STRUCTURE. STRUCTURE is 'df2sos' by default and can be any of the following:

```
'df1sos'
'df2sos'
'dfltsos'
'df2tsos'
'cascadeallpass'
'cascadewdfallpass'
```

Some of the listed structures may not be supported by System object filters. Type validstructures(D, 'cheby1', 'SystemObject', true) to get a list of structures supported by System objects.

HD = DESIGN(..., 'MatchExactly', MATCH) designs a Chebyshev type I filter and matches the frequency and magnitude specification for the band MATCH exactly. The other band will exceed the specification. MATCH can be 'stopband' or 'passband' and is 'passband' by default.

HD = DESIGN(..., 'SOSScaleNorm', NORM) designs an SOS filter and scales the coefficients using the P-Norm NORM. NORM can be either a discrete-time-domain norm or a frequency-domain norm. Valid time-domain norms are 'l1', 'l2', and 'linf'. Valid frequency-domain norms are 'L1', 'L2', and 'Linf'. Note that L2-norm is equal to l2-norm (Parseval's theorem) but the same is not true for other norms.

The different norms can be ordered in terms of how stringent they are as follows: 'l1' >= 'Linf' >= 'L2' = 'l2' >= 'L1' >= 'linf'. Using the most stringent scaling, 'l1', the filter is the least prone to overflow, but also has the worst signal-to-noise ratio. Linf-scaling is the most commonly used scaling in practice.

Scaling is turned off by default, which is equivalent to setting SOSScaleNorm = ''.

HD = DESIGN(..., 'SOSScaleOpts', OPTS) designs an SOS filter and scales the coefficients using an FDOPTS.SOSSCALING object OPTS. Scaling options are:

| Property               | Default    | Description/Valid values   |
|------------------------|------------|--|
| 'sosReorder'           | 'auto'     | Reorder section prior to scaling.<br>{'auto', 'none', 'up', 'down', 'lowpass', 'highpass', 'bandpass', 'bandstop'} |
| 'MaxNumerator'         | 2          | Maximum value for numerator coefficients   |
| 'NumeratorConstraint'  | 'none'     | {'none', 'unit', 'normalize', 'po2'}   |
| 'OverflowMode'         | 'wrap'     | {'wrap', 'saturate'}   |
| 'ScaleValueConstraint' | 'unit'     | {'unit', 'none', 'po2'}  |
| 'MaxScaleValue'        | 'Not used' | Maximum value for scale values   |

When sosReorder is set to 'auto', the sections will be automatically

reordered depending on the response type of the design (lowpass, highpass, etc.).

Note that 'MaxScaleValue' will only be used when 'ScaleValueConstraint' is set to something other than 'unit'. If 'MaxScaleValue' is set to a number, the 'ScaleValueConstraint' will be changed to 'none'. Further, if SOSScaleNorm is off (as it is by default), then all the SOSScaleOpts will be ignored.

For more information about P-Norm and scaling options see help for DFILT\SCALE.

```
% Example #1 - Compare passband and stopband MatchExactly.
h      = fdesign.lowpass('Fp,Fst,Ap,Ast', .1, .3, 1, 60);
Hd     = design(h, 'cheby1', 'MatchExactly', 'passband');
Hd(2) = design(h, 'cheby1', 'MatchExactly', 'stopband');

% Compare the passband edges in FVTool.
fvtool(Hd);
axis([.09 .11 -2 0]);
```

## Input Arguments

### **designSpecs** — Filter specification object

object

Filter specification object, specified as one of the `fdesign` functions.

### **TYPE** — Filter impulse response type

'FIR' | 'IIR'

Impulse response of the designed filter, specified as 'FIR' or 'IIR'. When not specified, the function outputs design methods which support both 'FIR' and 'IIR' response types.

Example: `designmethods(designSpecs, 'FIR', 'SystemObject', true)`

## Output Arguments

### **methods** — Available design methods

cell array

Available design methods, returned as a cell array. Each cell contains the name of the method and is determined by the arguments input to the function.

## Version History

**Introduced in R2009a**

## See Also

`design` | `designopts` | `fdesign` | `designoptions`

## designoptions

Show all options available for specified design

### Syntax

```
options = designoptions(designSpecs,method)
```

### Description

`options = designoptions(designSpecs,method)` returns all design options available for a specification object, `designSpecs`, using a particular design method, `method`.

### Examples

#### Design Butterworth Filter

Design a butterworth filter with lowpass and highpass frequency responses. The filter design procedure is:

- 1 Specify the filter design specifications using a `fdesign` function.
- 2 Pick a design method provided by the `designmethods` function.
- 3 To determine the available design options to choose from, use the `designoptions` function.
- 4 Design the filter using the `design` function.

#### Lowpass Filter

Construct a default lowpass filter design specification object using `fdesign.lowpass`.

```
designSpecs = fdesign.lowpass
designSpecs =
    lowpass with properties:
        Response: 'Lowpass'
        Specification: 'Fp,Fst,Ap,Ast'
        Description: {4x1 cell}
        NormalizedFrequency: 1
            Fpass: 0.4500
            Fstop: 0.5500
            Apass: 1
            Astop: 60
```

Determine the available design methods using the `designmethods` function. To design a butterworth filter, pick `butter`.

```
designmethods(designSpecs,SystemObject=true)
```

```
Design Methods that support System objects for class fdesign.lowpass (Fp,Fst,Ap,Ast):
```



```

butter
cheby1
cheby2
ellip
equiripple
ifir
kaiserwin
multistage

```

While designing the filter, you can specify additional design options. View a list of the options using the `designoptions` function. This function also shows the default design options the filter uses.

```
designoptions(designSpecs, 'butter', SystemObject=true)
```

```
ans = struct with fields:
```

```

    FilterStructure: {'df1sos' 'df2sos' 'df1tsos' 'df2tsos' 'cascadeallpass' 'cascadeallpass'}
    SOSScaleNorm: 'ustring'
    SOSScale0pts: 'fdopts.sosscaling'
    MatchExactly: {'passband' 'stopband'}
    DefaultFilterStructure: 'df2sos'
    DefaultMatchExactly: 'stopband'
    DefaultSOSScaleNorm: ''
    DefaultSOSScale0pts: [1x1 fdopts.sosscaling]

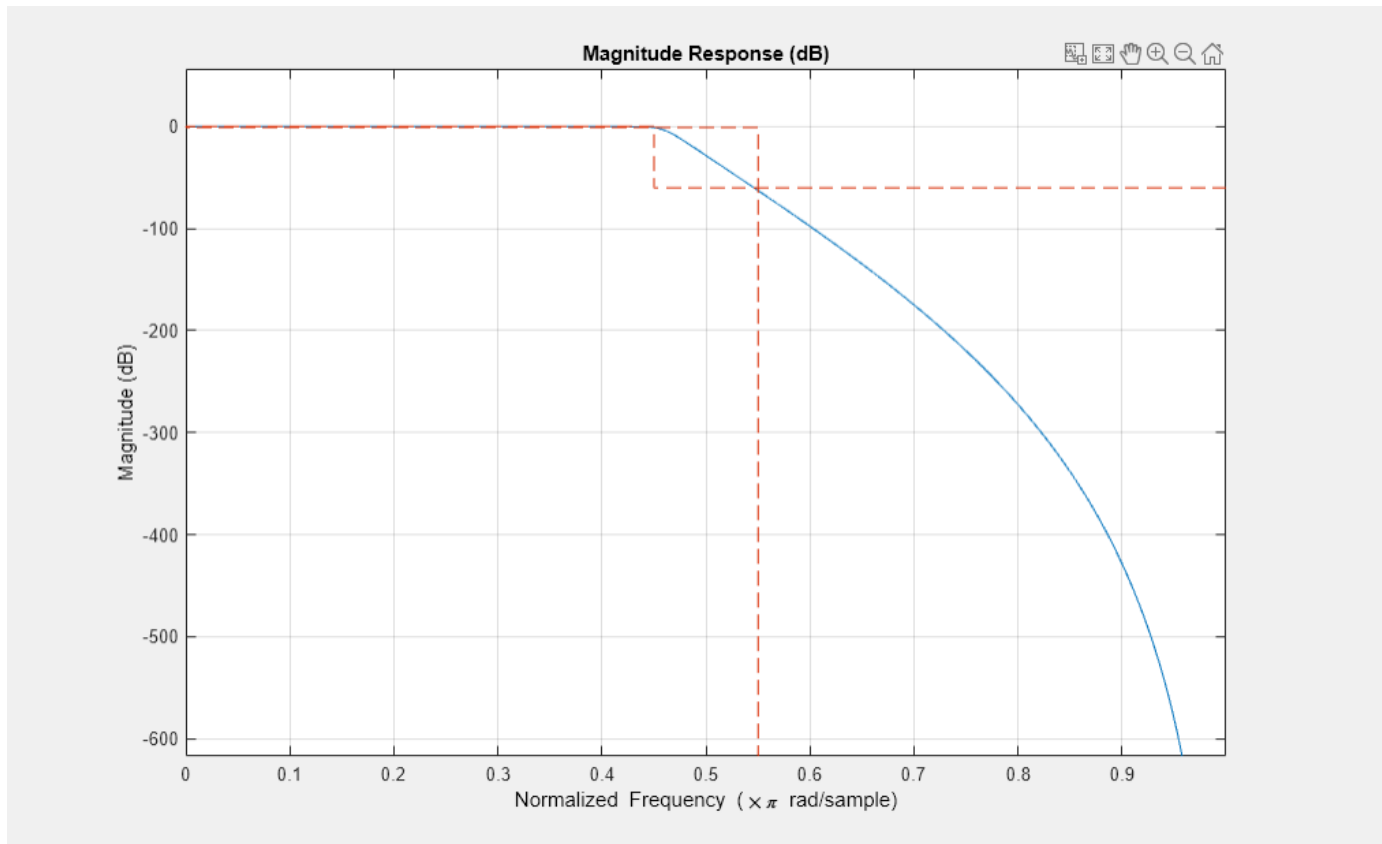
```

Use the `design` function to design the filter. Pass `'butter'` and the specifications given by variable `designSpecs`, as input arguments. Specify the `'matchexactly'` design option to `'passband'`.

```
lpFilter = design(designSpecs, 'butter', matchexactly='passband', SystemObject=true);
```

Visualize the frequency response of the designed filter.

```
fvtool(lpFilter)
```



## Highpass Filter

Construct a highpass filter design specification object using `fdesign.highpass`. Specify the order to be 7 and the 3 dB frequency to be  $0.6\pi$  radians/sample.

```
designSpecs = fdesign.highpass('N,F3dB',7,.6);
```

Determine the available design methods. To design a butterworth filter, pick `butter`.

```
designMethods(designSpecs,SystemObject=true)
```

Design Methods that support System objects for class `fdesign.highpass(N,F3dB)`:

```
butter
maxflat
```

While designing the filter, you can specify additional design options. View a list of the options using the `designoptions` function. This function also shows the default design options the filter uses.

```
designoptions(designSpecs,'butter',SystemObject=true)
```

```
ans = struct with fields:
```

```
    FilterStructure: {'df1sos' 'df2sos' 'df1tsos' 'df2tsos' 'cascadeallpass' 'cascadeallpass'}
    SOSScaleNorm: 'ustring'
    SOSScaleOpts: 'fdopts.sosscaling'
    DefaultFilterStructure: 'df2sos'
    DefaultSOSScaleNorm: ''
```

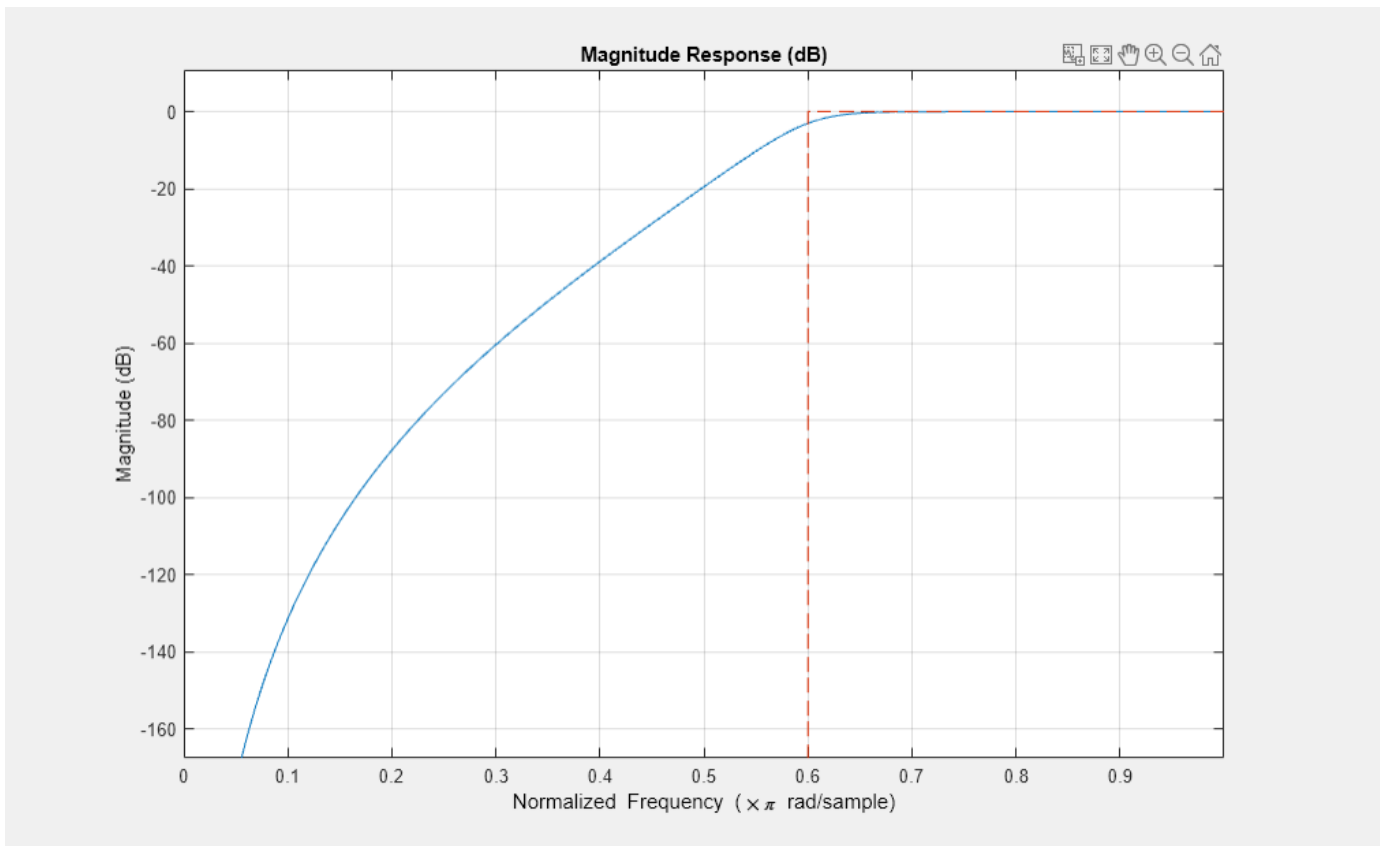
```
DefaultSOSScale0pts: [1x1 fdopts.sosscaling]
```

To design the butterworth filter, use the design function and specify 'butter' as an input. Set 'FilterStructure' to 'cascadeallpass'.

```
hpFilter = design(designSpecs, 'butter', FilterStructure='cascadeallpass', SystemObject=true);
```

Visualize the highpass frequency response.

```
fvtool(hpFilter)
```



## Design Notch Filter

Design a direct-form I notching filter that has a filter order of 6, center frequency of 0.5, quality factor of 10, and a passband ripple of 1 dB.

Create a notch filter design specification object using the `fdesign.notch` function and specify these design parameters.

```
notchSpecs = fdesign.notch('N,F0,Q,Ap',6,0.5,10,1);
```

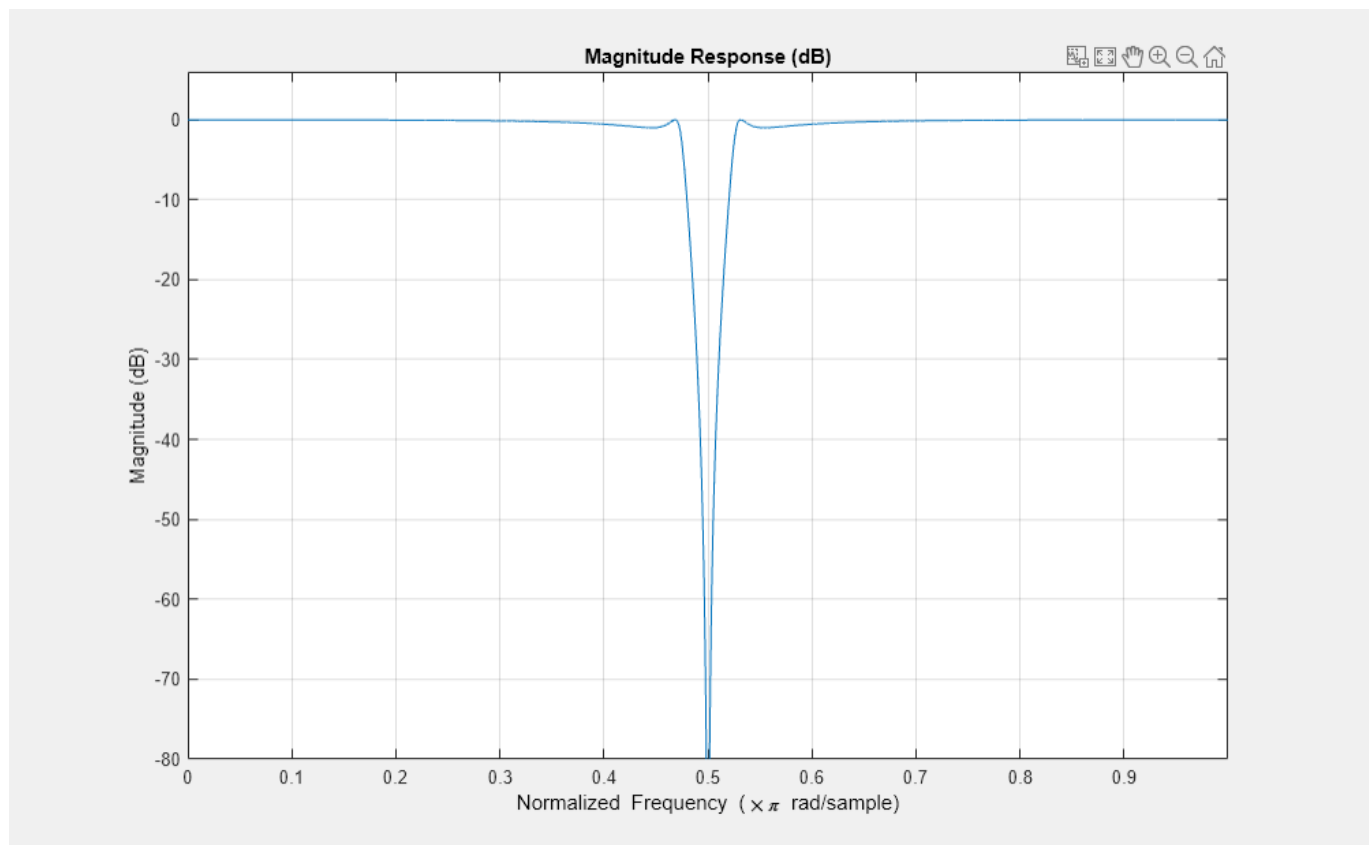
Design the notch filter using the `design` function. The resulting filter is a `dsp.SOSFilter` System object™. For details on how to apply this filter on streaming data, refer to `dsp.SOSFilter`.

```
notchFilt = design(notchSpecs, SystemObject=true)
```

```
notchFilt =  
  dsp.SOSFilter with properties:  
  
    Structure: 'Direct form II'  
  CoefficientSource: 'Property'  
    Numerator: [3x3 double]  
    Denominator: [3x3 double]  
  HasScaleValues: true  
    ScaleValues: [0.7722 0.7722 1.3597 1]  
  
  Show all properties
```

Visualize the frequency response of the designed filter using `fvtool`.

```
fvtool(notchFilt)
```



## Input Arguments

**designSpecs** — Filter specification object

object

Filter specification object, specified as one of the `fdesign` functions.

**method** — Design method

character vector

Design method, specified as a character vector. You can pick a design method from the available methods given by the `designmethods` function.

## Output Arguments

### **options** — Available design options

structure

Available design options, returned as a structure with the fields determined by the input filter specification object, `designSpecs`, and the design method chosen.

## Version History

Introduced in R2007b

## See Also

### Functions

`fdesign` | `designmethods` | `design`

## designopts

Valid input arguments and values for specification object and method

### Syntax

```
OPTS = designopts(D,METHOD)
```

### Description

`OPTS = designopts(D,METHOD)` returns a structure array with the default design parameters used by the design method `METHOD`. `METHOD` must be one of the options returned by `designmethods`.

Use `help(D,METHOD)` to get a description of the design parameters.

If you have DSP System Toolbox software installed, `OPTS` has the `SystemObject` property if at least one of the structures available for that design method is supported by System objects. However, not all structures for that design method are supported by System objects.

### Examples

#### Butterworth Filter Design Options

Create a lowpass filter with a numerator and denominator order of 10 and a 3-dB frequency of  $0.2\pi$  rad/sample. Obtain the default design parameters for a Butterworth design. Test whether the filter structure is a direct-form II biquad.

```
D = fdesign.lowpass('Nb,Na,F3dB',10,10,0.2);  
OPTS = designopts(D,'butter')
```

```
OPTS = struct with fields:  
  FilterStructure: 'df2sos'  
  SOSScaleNorm: ''  
  SOSScaleOpts: [1x1 fdopts.sosscaling]  
  SystemObject: 0
```

```
if isequal(OPTS.FilterStructure,'df2sos')  
    fprintf('The default filter structure is Direct-Form II\n');  
    fprintf('with second-order sections.\n');  
end
```

The default filter structure is Direct-Form II  
with second-order sections.

### Version History

Introduced in R2009a

**See Also**

[design](#) | [designmethods](#) | [fdesign](#) | [validstructures](#)

## dfilt

Discrete-time filter

### Syntax

```
hd = dfilt.structure(input1,...)
hd = design(d,'designmethod')
```

### Description

`hd = dfilt.structure(input1,...)` returns a discrete-time filter, `hd`, of type *structure*. Each structure takes one or more inputs. When you specify a `dfilt.structure` with no inputs, a default filter is created.

You must use a *structure* with `dfilt`.

You can get a vector of `dfilt` filters by concatenating `dfilt.structure(input1,...)` as an array.

```
hd = [dfilt.dffir, dfilt.df1, dfilt.df2]
```

```
hd =
```

```
dfilt.dffir
dfilt.df1
dfilt.df2
```

### Structures

Structures for `dfilt.structure` specify the type of filter structure. Available types of structures for `dfilt` are shown below.

| <b>dfilt.structure</b>               | <b>Description</b>                              | <b>Coefficient Mapping Support in <code>realizemdl</code></b> |
|--------------------------------------|---|---|
| <code>dfilt.allpass</code>           | Allpass filter                                  | Supported   |
| <code>dfilt.cascadeallpass</code>    | Cascade of allpass filter sections              | Supported   |
| <code>dfilt.cascadewdfallpass</code> | Cascade of allpass wave digital filters         | Supported   |
| <code>dfilt.delay</code>             | Delay   | Not supported   |
| <code>dfilt.df1</code>               | Direct-form I                                   | Supported   |
| <code>dfilt.df1sos</code>            | Direct-form I, second-order sections            | Supported   |
| <code>dfilt.df1t</code>              | Direct-form I transposed                        | Supported   |
| <code>dfilt.df1tsos</code>           | Direct-form I transposed, second-order sections | Supported   |
| <code>dfilt.df2</code>               | Direct-form II                                  | Supported   |
| <code>dfilt.df2sos</code>            | Direct-form II, second-order sections           | Supported   |



| <b>dfilt.structure</b> | <b>Description</b>                                       | <b>Coefficient Mapping Support in realizedl</b> |
|------------------------|--|---|
| dfilt.df2t             | Direct-form II transposed                                | Supported                                       |
| dfilt.df2tsos          | Direct-form II transposed, second-order sections         | Supported                                       |
| dfilt.dffir            | Direct-form FIR  | Supported                                       |
| dfilt.dffirt           | Direct-form FIR transposed                               | Supported                                       |
| dfilt.dfsymfir         | Direct-form symmetric FIR                                | Supported                                       |
| dfilt.dfasymfir        | Direct-form antisymmetric FIR                            | Supported                                       |
| dfilt.farrowfd         | Generic fractional delay Farrow filter                   | Supported                                       |
| dfilt.farrowlinearfd   | Linear fractional delay Farrow filter                    | Not supported                                   |
| dfilt.fftir            | Overlap-add FIR  | Not supported                                   |
| dfilt.latticeallpass   | Lattice allpass  | Supported                                       |
| dfilt.latticear        | Lattice autoregressive (AR)                              | Supported                                       |
| dfilt.latticearma      | Lattice autoregressive moving- average (ARMA)            | Supported                                       |
| dfilt.latticemamax     | Lattice moving-average (MA) for maximum phase            | Supported                                       |
| dfilt.latticemamin     | Lattice moving-average (MA) for minimum phase            | Supported                                       |
| dfilt.calattice        | Coupled, allpass lattice                                 | Supported                                       |
| dfilt.calatticepc      | Coupled, allpass lattice with power complementary output | Supported                                       |
| dfilt.statespace       | State-space  | Supported                                       |
| dfilt.scalar           | Scalar gain object                                       | Supported                                       |
| dfilt.wdfallpass       | Allpass wave digital filter object                       | Supported                                       |
| dfilt.cascade          | Filters arranged in series                               | Supported                                       |
| dfilt.parallel         | Filters arranged in parallel                             | Supported                                       |

For more information on each structure, refer to its reference page.

`hd = design(d, 'designmethod')` returns the `dfilt` object `hd` resulting from the filter specification object `d` and the design method you specify in *designmethod*. When you omit the `designmethod` argument, `design` uses the default design method to construct a filter from the object `d`.

With this syntax, you design filters by:

- 1 Specifying the filter specifications, such as the response shape (perhaps highpass) and details (passband edges and attenuation).
- 2 Selecting a method (such as `equiripple`) to design the filter.
- 3 Applying the method to the specifications object with `design(d, 'designmethod')`.

Using the specification-based technique can be more effective than the coefficient-based filter design techniques.

## Design Methods for Design Syntax

When you use the `hd = design(d, 'designmethod')` syntax, you have a range of design methods available depending on `d`, the filter specification object. The next table lists all of the design methods in the toolbox.

| Design Method           | Filter Design Result                                      |
|-------------------------|---|
| <code>butter</code>     | Butterworth IIR   |
| <code>cheby1</code>     | Chebyshev Type I IIR                                      |
| <code>cheby2</code>     | Chebyshev Type II IIR                                     |
| <code>ellip</code>      | Elliptic IIR  |
| <code>equiripple</code> | Equiripple with the same ripple in the pass and stopbands |
| <code>firls</code>      | Least-squares FIR   |
| <code>fregsamp</code>   | Frequency-Sampled FIR                                     |
| <code>ifir</code>       | Interpolated FIR  |
| <code>iirlpnorm</code>  | Least Pth norm IIR  |
| <code>iirls</code>      | Least-Squares IIR   |
| <code>kaiserwin</code>  | Kaiser-windowed FIR                                       |
| <code>lagrange</code>   | Fractional delay filter                                   |
| <code>multistage</code> | Multistage FIR  |
| <code>window</code>     | Windowed FIR  |

As the specifications object `d` changes, the available methods for designing filters from `d` also change. For instance, if `d` is a lowpass filter with the default specification `'Fp,Fst,Ap,Ast'`, the applicable methods are:

```
% Create an object to design a lowpass filter.
d=fdesign.lowpass;
designmethods(d) % What design methods apply to object d?
```

If you change the specification to `'N,F3dB'`, the available design methods change:

```
d=fdesign.lowpass('N,F3dB');
designmethods(d)
```

## Analysis Methods

Methods provide ways of performing functions directly on your `dfilt` object without having to specify the filter parameters again. You can apply these methods directly on the variable you assigned to your `dfilt` object.

For example, if you create a `dfilt` object, `hd`, you can check whether it has linear phase with `islinphase(hd)`, view its frequency response plot with `fvtool(hd)`, or obtain its frequency response values with `h = freqz(hd)`. You can use all of the methods described here in this way.

---

**Note** If your variable `hd` is a 1-D array of `dfilt` filters, the method is applied to each object in the array. Only `freqz`, `grpdelay`, `impz`, `is*`, `order`, and `stepz` methods can be applied to arrays. The `zplane` method can be applied to an array only if `zplane` is used without outputs.

---

Some of the methods listed here have the same name as functions in Signal Processing Toolbox software. They behave similarly.

| Method                 | Description   |
|------------------------|---|
| <code>addstage</code>  | Adds a stage to a <code>cascade</code> or <code>parallel</code> object, where a stage is a separate, modular filter. Refer to <code>dfilt.cascade</code> and <code>dfilt.parallel</code> .  |
| <code>block</code>     | <p><code>block(hd)</code> creates a block of the <code>dfilt</code> object. The <code>block</code> method can specify these properties and values:</p> <p>'Destination' indicates where to place the block. 'Current' places the block in the current Simulink model. 'New' creates a new model. Default value is 'Current'.</p> <p>'Blockname' assigns the entered character vector to the block name. Default name is 'Filter'.</p> <p>'OverwriteBlock' indicates whether to overwrite the block generated by the <code>block</code> method ('on') and defined by <code>Blockname</code>. Default is 'off'.</p> <p>'MapStates' specifies initial conditions in the block ('on'). Default is 'off'. Refer to "Using Filter States" in Signal Processing Toolbox documentation.</p> |
| <code>cascade</code>   | Returns the series combination of two <code>dfilt</code> objects. Refer to <code>dfilt.cascade</code> .   |
| <code>coeffs</code>    | Returns the filter coefficients in a structure containing fields that use the same property names as those in the original <code>dfilt</code> .   |
| <code>convert</code>   | Converts a <code>dfilt</code> object from one filter structure, to another filter structure.  |
| <code>fcfwrite</code>  | <p>Writes a filter coefficient ASCII file. The file can contain a single filter or a vector of objects. The default file name is <code>untitled.fcf</code>.</p> <p><code>fcfwrite(hd, filename)</code> writes to a disk file named <code>filename</code> in the current working folder. The <code>.fcf</code> extension is added automatically.</p> <p><code>fcfwrite(..., fmt)</code> writes the coefficients in the format <code>fmt</code>, where valid <code>fmt</code> character vectors are:</p> <p>'hex' for hexadecimal</p> <p>'dec' for decimal</p> <p>'bin' for binary representation</p>   |
| <code>fftcoeffs</code> | Returns the frequency-domain coefficients used when filtering with a <code>dfilt.fftir</code>   |
| <code>filter</code>    | Performs filtering using the <code>dfilt</code> object.   |
| <code>firtype</code>   | Returns the type (1-4) of a linear phase FIR filter.  |

| Method                  | Description  |
|-------------------------|--|
| <code>freqz</code>      | Plots the frequency response in <b>FVTool</b> . Unlike the <code>freqz</code> function, this <code>dfilt freqz</code> method has a default length of 8192.   |
| <code>grpdelay</code>   | Plots the group delay in <b>FVTool</b> .   |
| <code>impz</code>       | Plots the impulse response in <b>FVTool</b> .  |
| <code>impzlength</code> | Returns the length of the impulse response.  |
| <code>info</code>       | Displays <code>dfilt</code> information, such as filter structure, length, stability, linear phase, and, when appropriate, lattice and ladder length.  |
| <code>isallpass</code>  | Returns a logical 1 (i.e., true) if the <code>dfilt</code> object is an allpass filter or a logical 0 (i.e., false) if it is not.  |
| <code>iscascade</code>  | Returns a logical 1 if the <code>dfilt</code> object is cascaded or a logical 0 if it is not.  |
| <code>isfir</code>      | Returns a logical 1 if the <code>dfilt</code> object has finite impulse response (FIR) or a logical 0 if it does not.  |
| <code>islinphase</code> | Returns a logical 1 if the <code>dfilt</code> object is linear phase or a logical 0 if it is not.  |
| <code>ismaxphase</code> | Returns a logical 1 if the <code>dfilt</code> object is maximum-phase or a logical 0 if it is not.   |
| <code>isminphase</code> | Returns a logical 1 if the <code>dfilt</code> object is minimum-phase or a logical 0 if it is not.   |
| <code>isparallel</code> | Returns a logical 1 if the <code>dfilt</code> object has parallel stages or a logical 0 if it does not.  |
| <code>isreal</code>     | Returns a logical 1 if the <code>dfilt</code> object has real-valued coefficients or a logical 0 if it does not.   |
| <code>isscalar</code>   | Returns a logical 1 if the <code>dfilt</code> object is a scalar or a logical 0 if it is not scalar.   |
| <code>issos</code>      | Returns a logical 1 if the <code>dfilt</code> object has second-order sections or a logical 0 if it does not.  |
| <code>isstable</code>   | Returns a logical 1 if the <code>dfilt</code> object is stable or a logical 0 if it are not.   |
| <code>nsections</code>  | Returns the number of sections in a second-order sections filter. If a multistage filter contains stages with multiple sections, using <code>nsections</code> returns the total number of sections in all the stages (a stage with a single section returns 1).  |
| <code>nstages</code>    | Returns the number of stages of the filter, where a stage is a separate, modular filter.   |
| <code>nstates</code>    | Returns the number of states for an object.  |
| <code>order</code>      | Returns the filter order. If <code>hd</code> is a single-stage filter, the order is given by the number of delays needed for a minimum realization of the filter. If <code>hd</code> has multiple stages, the order is given by the number of delays needed for a minimum realization of the overall filter. |

| Method      | Description  |
|-------------|--|
| parallel    | Returns the parallel combination of two <code>dfilt</code> filters. Refer to <code>dfilt.parallel</code> .   |
| phasez      | Plots the phase response in <b>FVTool</b> .  |
| realizemdl  | <p>(Available only with Simulink.)</p> <p><code>realizemdl(hd)</code> creates a Simulink model containing a subsystem block realization of your <code>dfilt</code>.</p> <p><code>realizemdl(hd,p1,v1,p2,v2,...)</code> creates the block using the properties <code>p1, p2,...</code> and values <code>v1, v2,...</code> specified.</p> <p>The following properties are available:</p> <p>'Blockname' specifies the name of the block. The default value is 'Filter'.</p> <p>'Destination' specifies whether to add the block to a current Simulink model or create a new model. Valid values are 'Current' and 'New'.</p> <p>'OverwriteBlock' specifies whether to overwrite an existing block that was created by <code>realizemdl</code> or create a new block. Valid values are 'on' and 'off'. Only blocks created by <code>realizemdl</code> are overwritten.</p> <p>The following properties optimize the block structure. Specifying 'on' turns the optimization on and 'off' creates the block without optimization. The default for each block is 'off'.</p> <p>'OptimizeZeros' removes zero-gain blocks.</p> <p>'OptimizeOnes' replaces unity-gain blocks with a direct connection.</p> <p>'OptimizeNegOnes' replaces negative unity-gain blocks with a sign change at the nearest summation block.</p> <p>'OptimizeDelayChains' replaces delay chains made up of <math>n</math> unit delays with a single delay by <math>n</math>.</p> |
| removestage | Removes a stage from a cascade or parallel <code>dfilt</code> . Refer to <code>dfilt.cascade</code> and <code>dfilt.parallel</code> .  |
| setstage    | Overwrites a stage of a cascade or parallel <code>dfilt</code> . Refer to <code>dfilt.cascade</code> and <code>dfilt.parallel</code> .   |

| Method    | Description  |
|-----------|--|
| sos       | <p>Converts the <code>dfilt</code> to a second-order sections <code>dfilt</code>. If <code>hd</code> has a single section, the returned filter has the same class.</p> <p><code>sos(hd, flag)</code> specifies the ordering of the second-order sections. If <code>flag='UP'</code>, the first row contains the poles closest to the origin, and the last row contains the poles closest to the unit circle. If <code>flag='down'</code>, the sections are ordered in the opposite direction. The zeros are always paired with the poles closest to them.</p> <p><code>sos(hd, flag, scale)</code> specifies the scaling of the gain and the numerator coefficients of all second-order sections. <code>scale</code> can be <code>'none'</code>, <code>'inf'</code> (infinity-norm), or <code>'two'</code> (2-norm). Using infinity-norm scaling with <code>up</code> ordering minimizes the probability of overflow in the realization. Using 2-norm scaling with <code>down</code> ordering minimizes the peak roundoff noise.</p> |
| ss        | Converts the <code>dfilt</code> to state-space. To see the separate <code>A, B, C, D</code> matrices for the state-space model, use <code>[A, B, C, D]=ss(hd)</code> .   |
| stepz     | <p>Plots the step response in <b>FVTool</b>.</p> <p><code>stepz(hd, n)</code> computes the first <code>n</code> samples of the step response.</p> <p><code>stepz(hd, n, Fs)</code> separates the time samples by <math>T = 1/Fs</math>, where <code>Fs</code> is assumed to be in hertz.</p>   |
| sysobj    | Converts the <code>dfilt</code> to a filter System object. See the reference page for a list of supported objects.   |
| tf        | Converts the <code>dfilt</code> to a transfer function.  |
| zerophase | Plots the zero-phase response in <b>FVTool</b> .   |
| zpk       | Converts the <code>dfilt</code> to zeros-pole-gain form.   |
| zplane    | Plots a pole-zero plot in <b>FVTool</b> .  |

### Viewing Properties

As with any object, use `get` to view a `dfilt` properties. To see a specific property, use

```
get(hd, 'property')
```

To see all properties for an object, use

```
get(hd)
```

---

**Note** `dfilt` objects include an `arithmetic` property. You can change the internal arithmetic of the filter from double-precision to single-precision using: `hd.arithmetic = 'single'`.

If you have Fixed-Point Designer software, you can change the `arithmetic` property to fixed-point using: `hd.arithmetic = 'fixed'`

---

### Changing Properties

To set specific properties, use

```
set(hd, 'property1', value, 'property2', value, ...)
```

You must use single quotation marks around the property name. Use single quotation marks around the `value` argument when the value is a character vector, such as `specifyall` or `fixed`.

### Copying an Object

To create a copy of an object, use the `copy` method.

```
h2 = copy(hd)
```

---

**Note** Using the syntax `H2 = hd` copies only the object handle and does not create a new, independent object.

---

### Converting Between Filter Structures

To change the filter structure of a `dfilt` object `hd`, use

```
hd2 = convert(hd, 'structure_charactervector');
```

where `structure_charactervector` is any valid structure name in single quotation marks. If `hd` is a `cascade` or `parallel` structure, each stage is converted to the new structure.

### Using Filter States

Two properties control the filter states:

- `states` — Stores the current states of the filter. Before the filter is applied, the states correspond to the initial conditions and after the filter is applied, the states correspond to the final conditions. For `df1`, `df1t`, `df1sos` and `df1tsos` structures, `states` returns a `filtstates` object.
- `PersistentMemory` — Controls whether filter states are saved. The default value is `'false'`, which causes the initial conditions to be reset to zero before filtering and turns off the display of states information. Setting `PersistentMemory` to `'true'` allows the filter to use your initial conditions or to reuse the final conditions from a previous filtering operation as the initial conditions of the next filtering operation. The `true` setting also displays information about the filter states.

---

**Note** If you set the `states` and want to use them for filtering, you must set `PersistentMemory` to `'true'` before you use the filter.

---

## Examples

Create a direct-form I filter, and use a method to see if it is stable.

```
[b,a] = butter(8,0.25);  
hd = dfilt.df1(b,a);  
isstable(hd)
```

If a `dfilt`'s numerator values do not fit on a single line, a description of the vector is displayed. To see the specific numerator values for this example, use

```
B = get(hd, 'numerator');  
% or  
B1 = hd.numerator;
```

Create an array containing two `dfilt` objects, apply a method and verify that the method acts on both objects. Use a method to test whether the objects are FIR objects.

```
b = fir1(5, .5);  
hd = dfilt.dffir(b);           % Create an FIR filter object  
[b,a] = butter(5, .5);       % Create IIR filter  
hd(2) = dfilt.df2t(b,a);     % Create DF2T object and place  
                             % in the second column of hd.  
  
[h,w] = freqz(hd);  
test_fir = isfir(hd)  
% hd(1) is FIR and hd(2) is not.
```

Refer to the reference pages for each structure for more examples.

## Version History

**Introduced in R2011a**

### See Also

[design](#) | [fdesign](#) | [realizemdl](#) | [sos](#) | [stepz](#) | [freqz](#) | [grpdelay](#) | [impz](#) | [zplane](#)



## dfilt.allpass

Allpass filter

### Syntax

```
hd = dfilt.allpass(c)
```

### Description

`hd = dfilt.allpass(c)` constructs an allpass filter with the minimum number of multipliers from the elements in vector `c`. To be valid, `c` must contain one, two, three, or four real elements. The number of elements in `c` determines the order of the filter. For example, `c` with two elements creates a second-order filter and `c` with four elements creates a fourth-order filter.

The transfer function for the allpass filter is defined by

$$H(z) = \frac{c(n) + c(n-1)z^{-1} + \dots + z^{-n}}{1 + c(1)z^{-1} + \dots + c(n)z^{-n}}$$

given the coefficients in `c`.

To construct a cascade of allpass filter objects, use `dfilt.cascadeallpass`.

### Properties

The following table provides a list of all the properties associated with an allpass `dfilt` object.

| Property Name       | Brief Description  |
|---------------------|--|
| AllpassCoefficients | Contains the coefficients for the allpass filter object  |
| FilterStructure     | Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering — gains, delays, sums, products, and input/output.  |
| PersistentMemory    | Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. <code>False</code> is the default setting.   |
| States              | This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. They also provide linkage between the sections of a multisection filter, such as a cascade filter. For details, refer to <code>filtstates</code> in Signal Processing Toolbox documentation or in the Help system. |

## Examples

This example constructs and displays the information about a second-order allpass filter that uses the minimum number of multipliers.

```
c = [1.5, 0.7];  
% Create a second-order dfilt object.  
hd = dfilt.allpass(c);
```

## Version History

Introduced in R2011a

## See Also

[dsp.AllpassFilter](#) | [dsp.CICInterpolator](#) | [dsp.IIRHalfbandDecimator](#)

# dfilt.calattice

Coupled-allpass, lattice filter

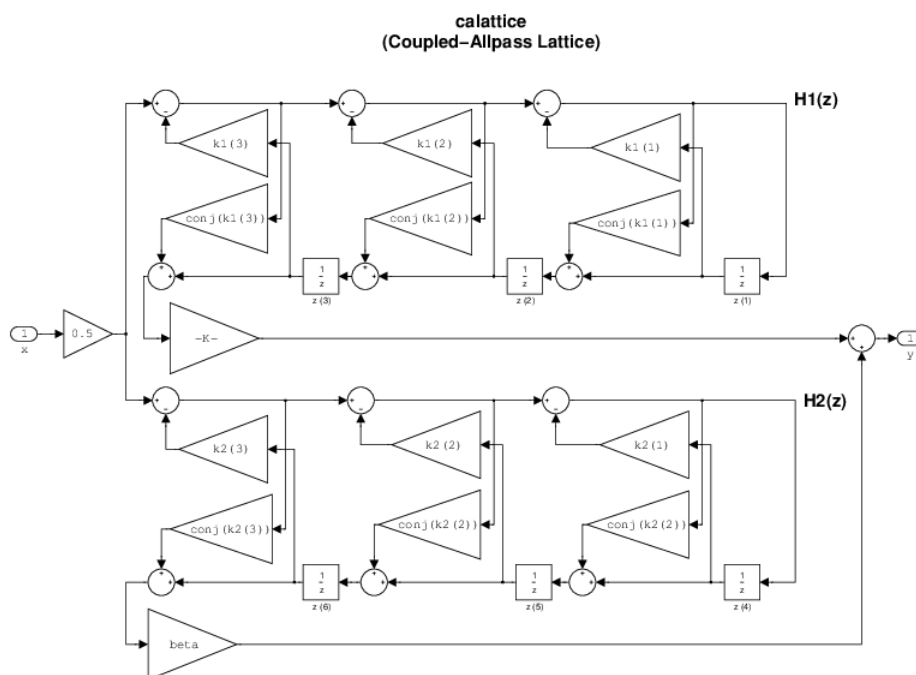
## Syntax

```
hd = dfilt.calattice(k1,k2,beta)
hd = dfilt.calattice
```

## Description

`hd = dfilt.calattice(k1,k2,beta)` returns a discrete-time, coupled-allpass, lattice filter object `hd`, which is two allpass, lattice filter structures coupled together. The lattice coefficients for each structure are vectors `k1` and `k2`. Input argument `beta` is shown in the diagram below.

`hd = dfilt.calattice` returns a default, discrete-time coupled-allpass, lattice filter object, `hd`. The default values are `k1 = k2 = []`, and `beta = 1`. This filter passes the input through to the output unchanged.



## Examples

Specify a third-order lattice coupled-allpass filter structure for a `dfilt` filter, `hd` with the following code.

```
k1 = [0.9511 + 1j*0.3088; 0.7511 + 1j*0.1158];
k2 = 0.7502 - 1j*0.1218;
```

```
beta = 0.1385 + 1j*0.9904;  
hd = dfilt.calattice(k1,k2,beta);
```

The Allpass1 and Allpass2 properties store vectors of coefficients.

## **Version History**

**Introduced in R2011a**

## **See Also**

`dsp.CoupledAllpassFilter`

# dfilt.calatticepc

Coupled-allpass, power-complementary lattice filter

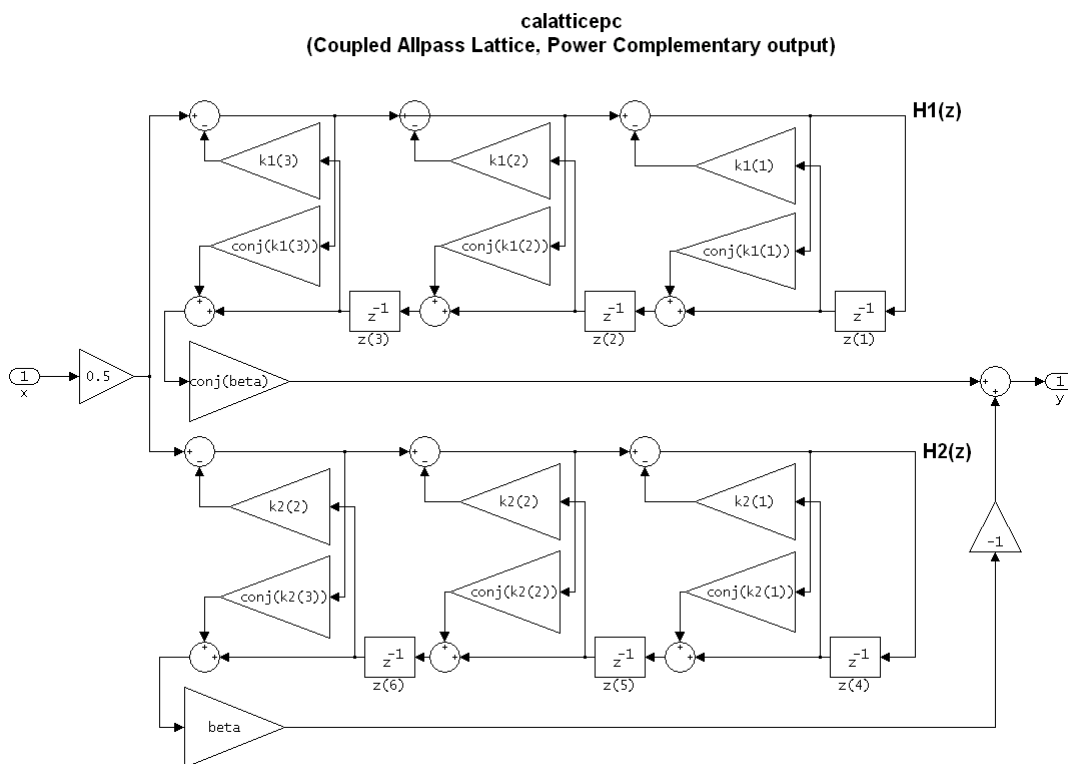
## Syntax

```
hd = dfilt.calatticepc(k1,k2)
hd = dfilt.calatticepc
```

## Description

`hd = dfilt.calatticepc(k1,k2)` returns a discrete-time, coupled-allpass, lattice filter object `hd`, with power-complementary output. This object is two allpass lattice filter structures coupled together to produce complementary output. The lattice coefficients for each structure are vectors, `k1` and `k2`, respectively. `beta` is shown in the following diagram.

`hd = dfilt.calatticepc` returns a default, discrete-time, coupled-allpass, lattice filter object `hd`, with power-complementary output. The default values are `k1 = k2 = []`, which is the default value for the `dfilt.latticeallpass`. The default for `beta = 1`. This filter passes the input through to the output unchanged.



## Examples

Specify a third-order lattice coupled-allpass power complementary filter structure for a filter `hd` with the following code. You see from the returned properties that `Allpass1` and `Allpass2` contain vectors of coefficients for the constituent filters.

```
k1 = [0.9511 + 0.3088i; 0.7511 + 0.1158i];  
k2 = 0.7502 - 0.1218i;  
beta = 0.1385 + 0.9904i;  
hd = dfilt.calatticepc(k1,k2,beta);
```

To see the coefficients for `Allpass1`, check the property values.

```
get(hd, 'Allpass1')
```

## Version History

**Introduced in R2011a**

## See Also

`dsp.CoupledAllpassFilter`

# dfilt.cascade

Cascade of discrete-time filters

## Syntax

```
hd = dfilt.cascade(filterobject1,filterobject2,...)
```

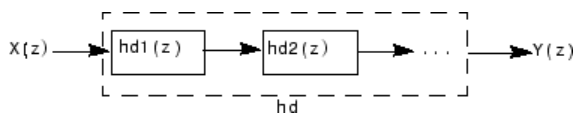
## Description

`hd = dfilt.cascade(filterobject1,filterobject2,...)` returns a discrete-time filter object `hd` of type `cascade`, which is a serial interconnection of two or more filter objects `filterobject1`, `filterobject2`, and so on. `dfilt.cascade` accepts any combination of `dfilt` objects (discrete time filters) to cascade, as well as Farrow filter objects.

You can use the standard notation to cascade one or more filters:

```
cascade(hd1,hd2,...)
```

where `hd1`, `hd2`, and so on can be mixed types, such as `dfilt` objects and other filtering objects.



`hd1`, `hd2`, and so on can be fixed-point filters. All filters in the cascade must be the same arithmetic format — `double`, `single`, or `fixed`. `hd`, the filter object returned, inherits the format of the cascaded filters.

## Examples

Cascade a lowpass filter and a highpass filter to produce a bandpass filter.

```
[b1,a1]=butter(8,0.6);           % Lowpass
[b2,a2]=butter(8,0.4,'high');    % Highpass
h1=dfilt.df2t(b1,a1);
h2=dfilt.df2t(b2,a2);
hcas=dfilt.cascade(h1,h2);      % Bandpass with passband 0.4-0.6
% View stage 1 with hcas.Stage(1)
```

## Version History

Introduced in R2011a

## See Also

`dsp.FilterCascade`

## dfilt.cascadeallpass

Cascade of allpass discrete-time filters

### Syntax

```
hd = dfilt.cascadeallpass(c1,c2,...)
```

### Description

`hd = dfilt.cascadeallpass(c1,c2,...)` constructs a cascade of allpass filters, each of which uses the minimum number of multipliers, given the filter coefficients provided in `c1`, `c2`, and so on.

Each vector `c` represents one section in the cascade filter. `c` vectors must contain one, two, three, or four elements as the filter coefficients for each section. As a result of the design algorithm, each section is a `dfilt.allpass` structure whose coefficients are given in the matching `c` vector, such as the `c1` vector contains the coefficients for the first stage.

States for each section are shared between sections.

Vectors `c` do not have to be the same length. You can combine various length vectors in the input arguments. For example, you can cascade fourth-order sections with second-order sections, or first-order sections.

For more information about the vectors `ci` and about the transfer function of each section, refer to `dfilt.allpass`.

Generally, you do not construct these allpass cascade filters directly. Instead, they result from the design process for an IIR filter. Refer to the first example in Examples for more about using `dfilt.cascadeallpass` to design an IIR filter.

### Properties

In the next table, the row entries are the filter properties and a brief description of each property.

| Property Name       | Brief Description  |
|---------------------|--|
| AllpassCoefficients | Contains the coefficients for the allpass filter object  |
| FilterStructure     | Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering — gains, delays, sums, products, and input/output.                                    |
| PersistentMemory    | Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. <code>False</code> is the default setting. |



| Property Name | Brief Description  |
|---------------|--|
| States        | This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. They also provide linkage between the sections of a multisection filter, such as a cascade filter. For details, refer to <code>filtstates</code> in Signal Processing Toolbox documentation or in the Help system. |

## Examples

Two examples show how `dfilt.cascadeallpass` works in very different applications — designing a halfband IIR filter and constructing an allpass cascade of `dfilt` objects.

First, design the IIR halfband filter using cascaded allpass filters. Each branch of the parallel cascade construction is a `cascadeallpass` filter object.

```
tw = 100; % Transition width of filter to be designed, 100 Hz.
ast = 80; % Stopband attenuation of filter to be designed, 80dB.
fs = 2000; % Sampling frequency of signal to be filtered.
% Store halfband design specs in the specifications object d.
d = fdesign.halfband('tw,ast',tw,ast,fs);
```

Now perform the actual filter design. `hd` contains two `dfilt.cascadeallpass` objects.

```
hd = design(d,'ellip','filterstructure','cascadeallpass');
% Get summary information about one dfilt.cascadeallpass stage.
StageInfo = hd.Stage(1).Stage(1);
```

This second example constructs a `dfilt.cascadeallpass` filter object directly given allpass coefficients for the input vectors.

```
section1 = 0.8;
section2 = [1.2,0.7];
section3 = [1.3,0.9];
hd = dfilt.cascadeallpass(section1,section2,section3);
% Get information about the filter
% return informatio in character array
S = info(hd);
```

## Version History

Introduced in R2011a

## See Also

`dsp.IIRHalfbandDecimator` | `dsp.IIRHalfbandDecimator`

## dfilt.cascadewdfallpass

Cascade allpass WDF filters to construct allpass WDF

### Syntax

```
hd = dfilt.cascadewdfallpass(c1,c2,...)
```

### Description

`hd = dfilt.cascadewdfallpass(c1,c2,...)` constructs a cascade of allpass wave digital filters given the allpass coefficients in the vectors `c1`, `c2`, and so on.

Each `c` vector contains the coefficients for one section of the cascaded filter. `C` vectors must have one, two, or four elements (coefficients). Three element vectors are not supported.

When the `c` vector has four elements, the first and third elements of the vector must be 0. Each section of the cascade is an allpass wave digital filter, from `dfilt.wdfallpass`, with the coefficients given by the corresponding `c` vector. That is, the first section has coefficients from vector `c1`, the second section coefficients come from `c2`, and on until all of the `c` vectors are used.

You can mix the lengths of the `c` vectors. They do not need to be the same length. For example, you can cascade several fourth-order sections (`length(c) = 4`) with first or second-order sections.

Wave digital filters are usually used to create other filters. This toolbox uses them to implement halfband filters, which the first example in Examples demonstrates. They are most often building blocks for filters.

Generally, you do not construct these WDF allpass cascade filters directly. Instead, they result from the design process for an IIR filter. Refer to the first example in Examples for more about using `dfilt.cascadewdfallpass` to design an IIR filter.

For more information about the `c` vectors and the transfer function for the allpass filters, refer to `dfilt.wdfallpass`.

### Properties

In the next table, the row entries are the filter properties and a brief description of each property.

| Property Name       | Brief Description  |
|---------------------|--|
| AllpassCoefficients | Contains the coefficients for the allpass wave digital filter object   |
| FilterStructure     | Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering — gains, delays, sums, products, and input/output.                                    |
| PersistentMemory    | Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. <code>False</code> is the default setting. |

| Property Name | Brief Description  |
|---------------|--|
| States        | This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. They also provide linkage between the sections of a multisection filter, such as a cascade filter. For details, refer to <code>filtstates</code> in Signal Processing Toolbox documentation or in the Help system. |

## Examples

To demonstrate two approaches to using `dfilt.cascadewdfallpass` to design a filter, these examples show both direct construction and construction as part of another filter.

The first design shown creates an IIR halfband filter that uses lattice wave digital filters. Each branch of the parallel connection in the lattice is an allpass cascade wave digital filter.

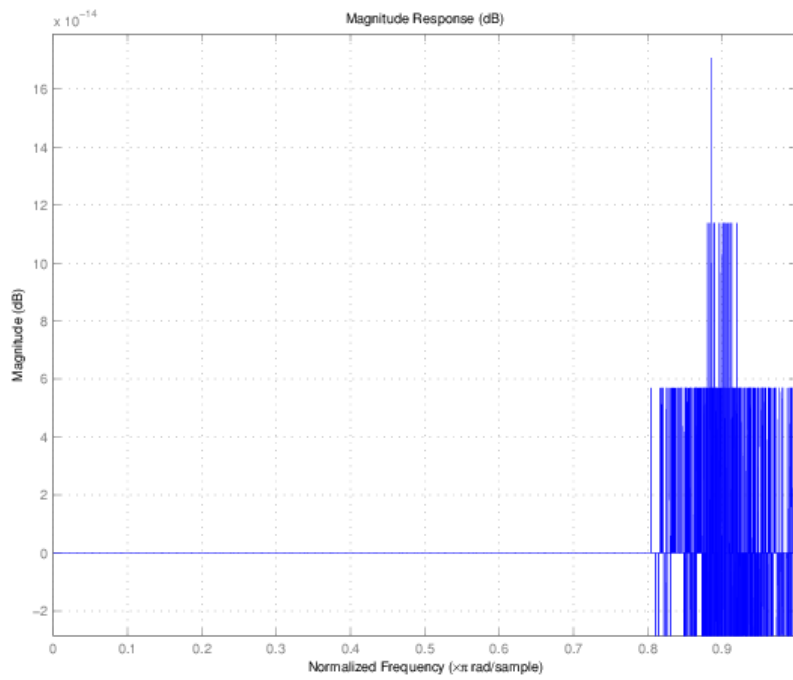
```
tw = 100; % Transition width of filter, 100 Hz.
ast = 80; % Stopband attenuation of filter, 80 dB.
fs = 2000; % Sampling frequency of signal to filter.
% Store halfband specs.
d = fdesign.halfband('tw,ast',tw,ast,fs);
```

Now perform the actual halfband design process. `hd` contains two `dfilt.cascadewdfallpass` filters.

```
hd = design(d,'ellip','filterstructure','cascadewdfallpass');
% Summary info on dfilt.cascadewdfallpass.
StageSummary = hd.stage(1).stage(2);
```

This example demonstrates direct construction of a `dfilt.cascadewdfallpass` filter with allpass coefficients.

```
section1 = 0.8;
section2 = [1.5,0.7];
section3 = [1.8,0.9];
hd = dfilt.cascadewdfallpass(section1,section2,section3);
```



## Version History

Introduced in R2011a

## dfilt.delay

Delay filter

### Syntax

```
Hd = dfilt.delay  
Hd = dfilt.delay(latency)
```

### Description

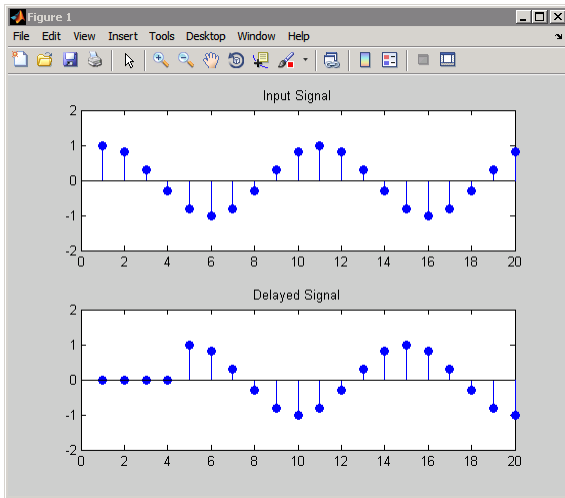
`Hd = dfilt.delay` returns a discrete-time filter, `Hd`, of type `delay`, which adds a single delay to any signal filtered with `Hd`. The filtered signal has its values shifted by one sample.

`Hd = dfilt.delay(latency)` returns a discrete-time filter, `Hd`, of type `delay`, which adds the number of delay units specified in `latency` to any signal filtered with `Hd`. The filtered signal has its values shifted by the `latency` number of samples. The values that appear before the shifted signal are the filter states.

### Examples

Create a delay filter with a latency of 4 and filter a simple signal to view the impact of applying a delay.

```
h = dfilt.delay(4);  
Fs = 1000;  
t = 0:1/Fs:1;  
sig = cos(2*pi*100*t);  
y = filter(h,sig);  
subplot(211);  
stem(sig,'markerfacecolor',[0 0 1]);  
axis([0 20 -2 2]);  
title('Input Signal');  
subplot(212);  
stem(y,'markerfacecolor',[0 0 1]);  
axis([0 20 -2 2]);  
title('Delayed Signal');
```



## Version History

Introduced in R2011a

## dfilt.df1

Discrete-time, direct-form I filter

### Syntax

```
hd = dfilt.df1
```

### Description

`hd = dfilt.df1` returns a default discrete-time, direct-form I filter object that uses double-precision arithmetic. By default, the numerator and denominator coefficients `b` and `a` are set to 1. With these coefficients the filter passes the input to the output without changes.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

```
set(hd,'arithmetic','single');
```
- To change to fixed-point filtering, enter

```
set(hd,'arithmetic','fixed');
```

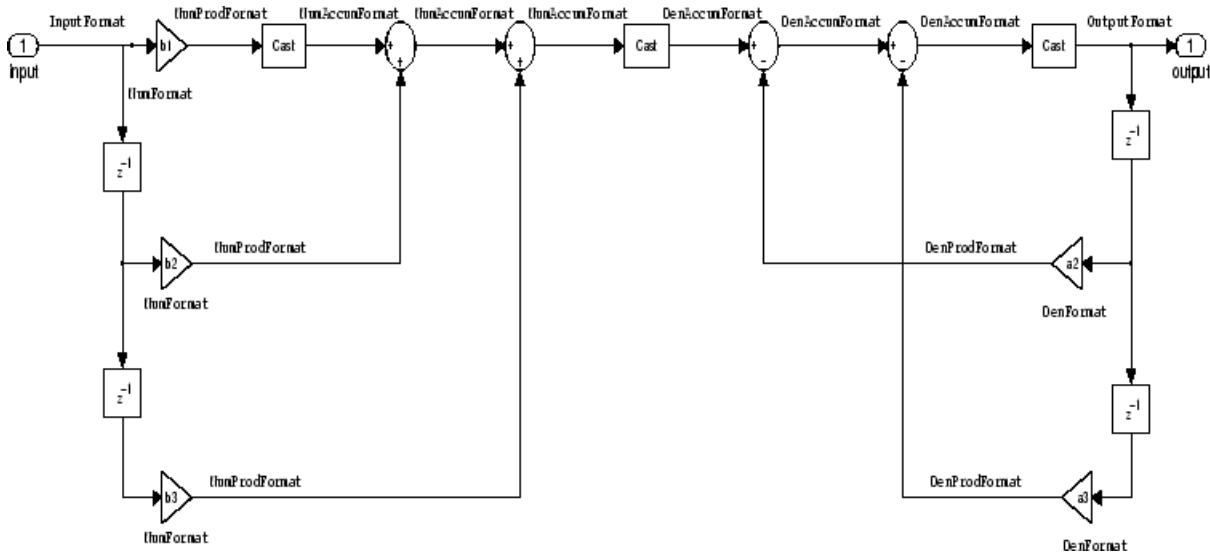
---

**Note** `a(1)`, the leading denominator coefficient, cannot be 0. To allow you to change the arithmetic setting to `fixed` or `single`, `a(1)` must be equal to 1.

---

### Fixed-Point Filter Structure

The following figure shows the signal flow for the direct-form I filter implemented by `dfilt.df1`. To help you see how the filter processes the coefficients, input, output, and states of the filter, as well as numerical operations, the figure includes the locations of the arithmetic and data type format elements within the signal flow.



### Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the word “format.” In this use, “format” means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFormat label refers to the word length and fraction length used to interpret the data input to the filter. The format properties InputWordLength and InputFracLength (as shown in the table) store the word length and the fraction length in bits. Or consider NumFormat, which refers to the word and fraction lengths (CoeffWordLength, NumFracLength) associated with representing filter numerator coefficients.

| Signal Flow Label | Corresponding Word Length Property | Corresponding Fraction Length Property | Related Properties                |
|-------------------|------------------------------------|--|-----------------------------------|
| DenAccumFormat    | AccumWordLength                    | DenAccumFracLength                     | AccumMode, CastBeforeSum          |
| DenFormat         | CoeffWordLength                    | DenFracLength                          | CoeffAutoScale, SignedDenominator |
| DenProdFormat     | CoeffWordLength                    | DenProdFracLength                      | ProductMode, ProductWordLength    |
| InputFormat       | InputWordLength                    | InputFracLength                        | None                              |
| NumAccumFormat    | AccumWordLength                    | NumAccumFracLength                     | AccumMode, CastBeforeSum          |
| NumFormat         | CoeffWordLength                    | NumFracLength                          | CoeffAutoScale, Signed, Numerator |
| NumProdFormat     | CoeffWordLength                    | NumProdFracLength                      | ProductWordLength, ProductMode    |



| Signal Flow Label | Corresponding Word Length Property | Corresponding Fraction Length Property | Related Properties |
|-------------------|------------------------------------|--|--------------------|
| OutputFormat      | OutputWordLength                   | OutputFracLength                       | OutputMode         |

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label DenProdFormat, which always follows a denominator coefficient multiplication element in the signal flow. The label indicates that denominator coefficients leave the multiplication element with the word length and fraction length associated with product operations that include denominator coefficients. From reviewing the table, you see that the DenProdFormat refers to the properties ProdWordLength, ProductMode and DenProdFracLength that fully define the denominator format after multiply (or product) operations.

## Properties

In this table you see the properties associated with df1 implementations of dfilt objects.

**Note** The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use `get(hd)` where `hd` is a filter.

| Property Name   | Brief Description   |
|-----------------|---|
| AccumMode       | Determines how the accumulator outputs stored values. Choose from full precision ( <code>FullPrecision</code> ), or whether to keep the most significant bits ( <code>KeepMSB</code> ) or least significant bits ( <code>KeepLSB</code> ) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set <code>AccumMode</code> to <code>SpecifyPrecision</code> . |
| AccumWordLength | Sets the word length used to store data in the accumulator/buffer.  |
| Arithmetic      | Defines the arithmetic the filter uses. Gives you the options <code>double</code> , <code>single</code> , and <code>fixed</code> . In short, this property defines the operating mode for your filter.  |
| CastBeforeSum   | Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.   |
| CoeffAutoScale  | Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to <code>false</code> enables you to change the <code>NumFracLength</code> and <code>DenFracLength</code> properties to specify the precision used.   |
| CoeffWordLength | Specifies the word length to apply to filter coefficients.  |

| Property Name      | Brief Description  |
|--------------------|--|
| DenAccumFracLength | Specifies the fraction length the filter algorithm uses to interpret the results of product operations involving denominator coefficients. You can change the value for this property when you set AccumMode to SpecifyPrecision.  |
| DenFracLength      | Set the fraction length the filter uses to interpret denominator coefficients. DenFracLength is always available, but it is read-only until you set CoeffAutoScale to false.   |
| Denominator        | Stores the denominator coefficients for the IIR filter.  |
| DenProdFracLength  | Specifies how the filter algorithm interprets the results of product operations involving denominator coefficients. You can change this property value when you set ProductMode to SpecifyPrecision.   |
| FilterStructure    | Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering — gains, delays, sums, products, and input/output.  |
| InputFracLength    | Specifies the fraction length the filter uses to interpret input data.   |
| InputWordLength    | Specifies the word length applied to interpret input data.   |
| NumAccumFracLength | Specifies how the filter algorithm interprets the results of addition operations involving numerator coefficients. You can change the value of this property after you set AccumMode to SpecifyPrecision.  |
| Numerator          | Holds the numerator coefficient values for the filter.   |
| NumFracLength      | Sets the fraction length used to interpret the value of numerator coefficients.  |
| NumProdFracLength  | Specifies how the filter algorithm interprets the results of product operations involving numerator coefficients. Available to be changed when you set ProductMode to SpecifyPrecision.  |
| OutputFracLength   | Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set OutputMode to SpecifyPrecision.   |
| OutputWordLength   | Determines the word length used for the output data.   |
| OverflowMode       | Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision. |

| Property Name     | Brief Description   |
|-------------------|---|
| ProductMode       | Determines how the filter handles the output of product operations. Choose from full precision ( <code>FullPrecision</code> ), or whether to keep the most significant bit ( <code>KeepMSB</code> ) or least significant bit ( <code>KeepLSB</code> ) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set <code>ProductMode</code> to <code>SpecifyPrecision</code> .  |
| ProductWordLength | Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set <code>ProductMode</code> to <code>SpecifyPrecision</code> .   |
| PersistentMemory  | Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. <code>False</code> is the default setting.  |
| RoundMode         | <p>Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths).</p> <ul style="list-style-type: none"> <li><code>ceil</code> - Round toward positive infinity.</li> <li><code>convergent</code> - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.</li> <li><code>fix</code> - Round toward zero.</li> <li><code>floor</code> - Round toward negative infinity.</li> <li><code>nearest</code> - Round toward nearest. Ties round toward positive infinity.</li> <li><code>round</code> - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.</li> </ul> <p>The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.</p> |
| Signed            | Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.   |
| States            | This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use <code>fi</code> objects, with the associated properties from those objects. For details, refer to <code>filtstates</code> in Signal Processing Toolbox documentation or in the Help system.  |

## Examples

Specify a second-order direct-form I structure for a `dfilt` object, `hd`, with the following code:

```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
hd = dfilt.df1(b,a);
```

```
% Convert hd to fixed-point filter  
set(hd, 'arithmetic', 'fixed')
```

## **Version History**

**Introduced in R2011a**

### **See Also**

dsp.IIRFilter

## dfilt.df1sos

Discrete-time, SOS direct-form I filter

### Syntax

```
hd = dfilt.df1sos(s)
hd = dfilt.df1sos(b1,a1,b2,a2,...)
hd = dfilt.df1sos(...,g)
hd = dfilt.df1sos
```

### Description

`hd = dfilt.df1sos(s)` returns a discrete-time, second-order section, direct-form I filter object `hd`, with coefficients given in the `s` matrix.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

```
set(hd,'arithmetic','single');
```

- To change to fixed-point filtering, enter

```
set(hd,'arithmetic','fixed');
```

`hd = dfilt.df1sos(b1,a1,b2,a2,...)` returns a discrete-time, second-order section, direct-form I filter object `hd`, with coefficients for the first section given in the `b1` and `a1` vectors, for the second section given in the `b2` and `a2` vectors, and so on.

`hd = dfilt.df1sos(...,g)` includes a gain vector `g`. The elements of `g` are the gains for each section. The maximum length of `g` is the number of sections plus one. When you do not specify `g`, all gains default to one.

`hd = dfilt.df1sos` returns a default, discrete-time, second-order section, direct-form I filter object, `hd`. This filter passes the input through to the output unchanged.

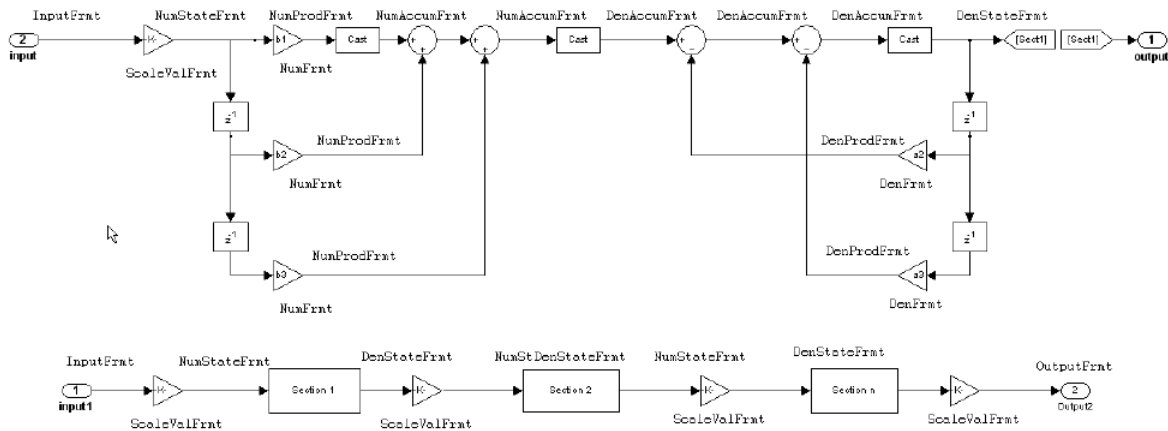
---

**Note** The leading coefficient of the denominator `a(1)` cannot be 0. To allow you to change the arithmetic setting to `fixed` or `single`, `a(1)` must be equal to 1.

---

### Fixed-Point Filter Structure

The following figure shows the signal flow for the direct-form I filter implemented in second-order sections by `dfilt.df1sos`. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.



**Notes About the Signal Flow Diagram**

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the letters “frmt” (format). In this use, “frmt” means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFrmt label refers to the word length and fraction length used to interpret the data input to the filter. The format properties InputWordLength and InputFracLength (as shown in the table) store the word length and the fraction length in bits. Similarly consider NumFrmt, which refers to the word and fraction lengths (CoeffWordLength, NumFracLength) associated with representing filter numerator coefficients.

| Signal Flow Label | Corresponding Word Length Property | Corresponding Fraction Length Property | Related Properties                  |
|-------------------|------------------------------------|--|-------------------------------------|
| DenAccumFrmt      | AccumWordLength                    | DenAccumFracLength                     | AccumMode, CastBeforeSum            |
| DenFrmt           | CoeffWordLength                    | DenFracLength                          | CoeffAutoScale, Signed, Denominator |
| DenProdFrmt       | ProductWordLength                  | DenProdFracLength                      | ProductMode                         |
| DenStateFrmt      | DenStateWordLength                 | DenStateFracLength                     | CastBeforeSum, States               |
| InputFrmt         | InputWordLength                    | InputFracLength                        | None                                |
| NumAccumFrmt      | AccumWordLength                    | NumAccumFracLength                     | AccumMode, CastBeforeSum            |
| NumFrmt           | CoeffWordLength                    | NumFracLength                          | CoeffAutoScale, Signed, Numerator   |
| NumProdFrmt       | ProductWordLength                  | NumProdFracLength                      | ProductMode                         |
| NumStateFrmt      | NumStateWordLength                 | NumStateFracLength                     | States                              |
| OutputFrmt        | OutputWordLength                   | OutputFracLength                       | OutputMode                          |

| Signal Flow Label | Corresponding Word Length Property | Corresponding Fraction Length Property | Related Properties          |
|-------------------|------------------------------------|--|-----------------------------|
| ScaleValueFrmt    | CoeffWordLength                    | ScaleValueFracLength                   | CoeffAutoScale, ScaleValues |

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label DenProdFrmt, which always follows a denominator coefficient multiplication element in the signal flow. The label indicates that denominator coefficients leave the multiplication element with the word length and fraction length associated with product operations that include denominator coefficients. From reviewing the table, you see that the DenProdFrmt refers to the properties ProductWordLength, DenProdFracLength and ProductMode that fully define the denominator format after multiply (or product) operations.

## Properties

In this table you see the properties associated with SOS implementation of direct-form I `dfilt` objects.

**Note** The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

```
get(hd)
```

where `hd` is a filter.

| Property Name   | Brief Description   |
|-----------------|---|
| AccumMode       | Determines how the accumulator outputs stored values. Choose from full precision ( <code>FullPrecision</code> ), or whether to keep the most significant bits ( <code>KeepMSB</code> ) or least significant bits ( <code>KeepLSB</code> ) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set <code>AccumMode</code> to <code>SpecifyPrecision</code> . |
| AccumWordLength | Sets the word length used to store data in the accumulator/buffer.  |
| Arithmetic      | Defines the arithmetic the filter uses. Gives you the options <code>double</code> , <code>single</code> , and <code>fixed</code> . In short, this property defines the operating mode for your filter.  |
| CastBeforeSum   | Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.   |
| CoeffAutoScale  | Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to <code>false</code> enables you to change the <code>NumFracLength</code> and <code>DenFracLength</code> properties to specify the precision used.   |

| Property Name       | Brief Description  |
|---------------------|--|
| CoeffWordLength     | Specifies the word length to apply to filter coefficients.   |
| DenAccumFracLength  | Specifies the fraction length used to interpret data in the accumulator used to hold the results of sum operations. You can change the value for this property when you set AccumMode to SpecifyPrecision. |
| DenFracLength       | Set the fraction length the filter uses to interpret denominator coefficients. DenFracLength is always available, but it is read-only until you set CoeffAutoScale to false.                               |
| DenProdFracLength   | Specifies how the filter algorithm interprets the results of product operations involving denominator coefficients. You can change this property value when you set ProductMode to SpecifyPrecision.       |
| DenStateFracLength  | Specifies the fraction length used to interpret the states associated with denominator coefficients in the filter.   |
| DenStateWordLength  | Specifies the word length used to represent the states associated with denominator coefficients in the filter.   |
| FilterStructure     | Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering—gains, delays, sums, products, and input/output.                            |
| InputFracLength     | Specifies the fraction length the filter uses to interpret input data.   |
| InputWordLength     | Specifies the word length applied to interpret input data.   |
| NumAccumFracLength  | Specifies how the filter algorithm interprets the results of addition operations involving numerator coefficients. You can change the value of this property after you set AccumMode to SpecifyPrecision.  |
| NumFracLength       | Sets the fraction length used to interpret the value of numerator coefficients.  |
| NumStateFracLength  | Specifies the fraction length used to interpret the states associated with numerator coefficient operations in the filter.   |
| NumWordFracLength   | Specifies the word length used to interpret the states associated with numerator coefficient operations in the filter.   |
| OptimizeScaleValues | When true, the filter skips multiplication-by-one scaling. When false, the filter performs multiplication-by-one scaling.  |
| OutputFracLength    | Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set OutputMode to SpecifyPrecision.   |



| Property Name     | Brief Description   |
|-------------------|---|
| OutputMode        | <p>Sets the mode the filter uses to scale the filtered data for output. You have the following choices:</p> <ul style="list-style-type: none"> <li>• <b>AvoidOverflow</b> — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow.</li> <li>• <b>BestPrecision</b> — directs the filter to set the output data word length and fraction length to maximize the precision in the output data.</li> <li>• <b>SpecifyPrecision</b> — lets you set the word and fraction lengths used by the output data from filtering.</li> </ul> |
| OutputWordLength  | Determines the word length applied for the output data.   |
| OverflowMode      | <p>Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either <b>saturate</b> (limit the output to the largest positive or negative representable value) or <b>wrap</b> (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow—they maintain full precision.</p>   |
| ProductMode       | <p>Determines how the filter handles the output of product operations. Choose from full precision (<b>FullPrecision</b>), or whether to keep the most significant bit (<b>KeepMSB</b>) or least significant bit (<b>KeepLSB</b>) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set <b>ProductMode</b> to <b>SpecifyPrecision</b>.</p>  |
| ProductWordLength | <p>Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set <b>ProductMode</b> to <b>SpecifyPrecision</b>.</p>   |
| PersistentMemory  | <p>Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. <b>False</b> is the default setting.</p>   |

| Property Name        | Brief Description   |
|----------------------|---|
| RoundMode            | <p>Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths).</p> <ul style="list-style-type: none"> <li><code>ceil</code> - Round toward positive infinity.</li> <li><code>convergent</code> - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.</li> <li><code>fix</code> - Round toward zero.</li> <li><code>floor</code> - Round toward negative infinity.</li> <li><code>nearest</code> - Round toward nearest. Ties round toward positive infinity.</li> <li><code>round</code> - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.</li> </ul> <p>The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.</p> |
| ScaleValueFracLength | Scale values work with SOS filters. Setting this property controls how your filter interprets the scale values by setting the fraction length. Only available when you disable <code>AutoScaleMode</code> by setting it to <code>false</code> .   |
| ScaleValues          | Scaling for the filter objects in SOS filters.  |
| Signed               | Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.   |
| SosMatrix            | Holds the filter coefficients as property values. Displays the matrix in the format [sections x coefficients/section datatype]. A [15x6 double] SOS matrix represents a filter with 6 coefficients per section and 15 sections, using data type <code>double</code> to represent the coefficients.  |
| States               | This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use <code>fi</code> objects, with the associated properties from those objects. For details, refer to <code>filtstates</code> in Signal Processing Toolbox documentation or in the Help system.  |
| StateWordLength      | Sets the word length used to represent the filter states.   |

## Examples

Specify a fixed-point, second-order section, direct-form I `dfilt` object with the following code:

```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
hd = dfilt.df1sos(b,a);
% Convert to fixed-point filter
hd.arithmetic = 'fixed';
```

## **Version History**

Introduced in R2011a

### **See Also**

`dsp.BiquadFilter`

## dfilt.df1t

Discrete-time, direct-form I transposed filter

### Syntax

```
hd = dfilt.df1t(b,a)
hd = dfilt.df1t
```

### Description

`hd = dfilt.df1t(b,a)` returns a discrete-time, direct-form I transposed filter object `hd`, with numerator coefficients `b` and denominator coefficients `a`.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

```
set(hd,'arithmetic','single');
```

- To change to fixed-point filtering, enter

```
set(hd,'arithmetic','fixed');
```

`hd = dfilt.df1t` returns a default, discrete-time, direct-form I transposed filter object `hd`, with `b=1` and `a=1`. This filter passes the input through to the output unchanged.

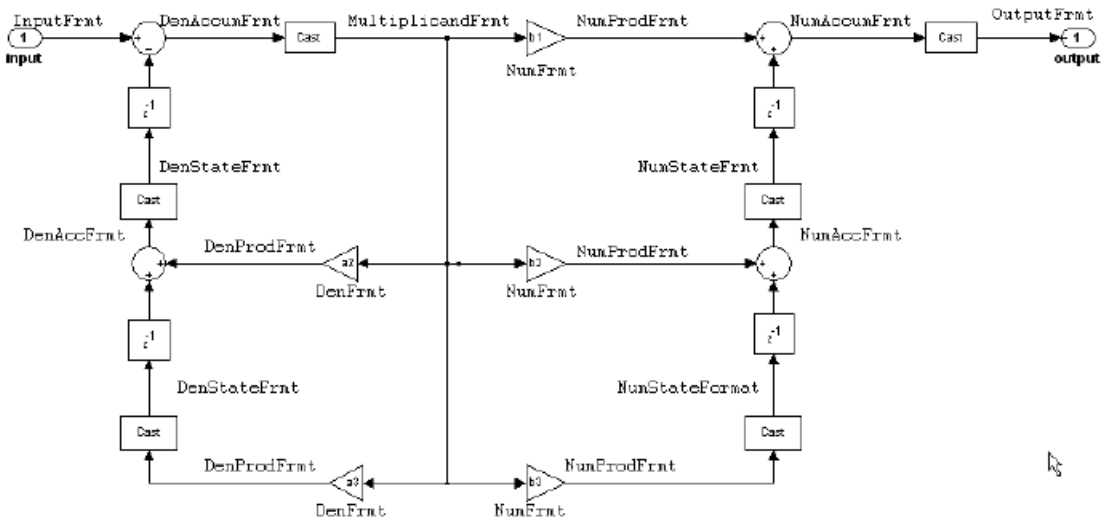
---

**Note** The leading coefficient of the denominator `a(1)` cannot be 0. To allow you to change the arithmetic setting to `fixed` or `single`, `a(1)` must be equal to 1.

---

### Fixed-Point Filter Structure

The following figure shows the signal flow for the transposed direct-form I filter implemented by `dfilt.df1t`. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.



### Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the letters “frmt” (format). In this use, “frmt” means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFrmt label refers to the word length and fraction length used to interpret the data input to the filter. The format properties InputWordLength and InputFracLength (as shown in the table) store the word length and the fraction length in bits. Or consider NumFrmt, which refers to the word and fraction lengths (CoeffWordLength, NumFracLength) associated with representing filter numerator coefficients.

| Signal Flow Label | Corresponding Word Length Property | Corresponding Fraction Length Property | Related Properties                   |
|-------------------|------------------------------------|--|--------------------------------------|
| DenAccumFrmt      | AccumWordLength                    | DenAccumFracLength                     | AccumMode, CastBeforeSum             |
| DenFrmt           | CoeffWordLength                    | DenFracLength                          | CoeffAutoScale , Signed, Denominator |
| DenProdFrmt       | CoeffWordLength                    | DenProdFracLength                      | ProductMode, ProductWordLength       |
| DenStateFrmt      | DenStateWordLength                 | DenStateFracLength                     | CastBeforeSum, States                |
| InputFrmt         | InputWordLength                    | InputFracLength                        | None                                 |
| Multiplicandfrmt  | Multiplicand-WordLength            | Multiplicand-FracLength                | CastBeforeSum                        |
| NumAccumFrmt      | AccumWordLength                    | NumAccumFracLength                     | AccumMode,CastBeforeSum              |

| Signal Flow Label | Corresponding Word Length Property | Corresponding Fraction Length Property | Related Properties                |
|-------------------|------------------------------------|--|-----------------------------------|
| NumFrmt           | CoeffWordLength                    | NumFracLength                          | CoeffAutoScale, Signed, Numerator |
| NumProdFrmt       | CoeffWordLength                    | NumProdFracLength                      | ProductWordLength, ProductMode    |
| NumStateFrmt      | NumStateWordLength                 | NumStateFracLength                     | States                            |
| OutputFrmt        | OutputWordLength                   | OutputFracLength                       | OutputMode                        |

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label DenProdFrmt, which always follows a denominator coefficient multiplication element in the signal flow. The label indicates that denominator coefficients leave the multiplication element with the word length and fraction length associated with product operations that include denominator coefficients. From reviewing the table, you see that the DenProdFrmt refers to the properties ProdWordLength, ProductMode and DenProdFracLength that fully define the denominator format after multiply (or product) operations.

## Properties

In this table you see the properties associated with `df1t` implementation of `dfilt` objects.

**Note** The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

```
get(hd)
```

where `hd` is a filter.

| Property Name   | Brief Description   |
|-----------------|---|
| AccumMode       | Determines how the accumulator outputs stored values. Choose from full precision ( <code>FullPrecision</code> ), or whether to keep the most significant bits ( <code>KeepMSB</code> ) or least significant bits ( <code>KeepLSB</code> ) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set <code>AccumMode</code> to <code>SpecifyPrecision</code> . |
| AccumWordLength | Sets the word length used to store data in the accumulator/buffer.  |
| Arithmetic      | Defines the arithmetic the filter uses. Gives you the options <code>double</code> , <code>single</code> , and <code>fixed</code> . In short, this property defines the operating mode for your filter.  |
| CastBeforeSum   | Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.   |

| Property Name          | Brief Description   |
|------------------------|---|
| CoeffAutoScale         | Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to <code>false</code> enables you to change the <code>NumFracLength</code> and <code>DenFracLength</code> properties to specify the precision used. |
| CoeffWordLength        | Specifies the word length to apply to filter coefficients.  |
| DenAccumFracLength     | Specifies the fraction length used to interpret data in the accumulator used to hold the results of sum operations. You can change the value for this property when you set <code>AccumMode</code> to <code>SpecifyPrecision</code> .   |
| DenFracLength          | Set the fraction length the filter uses to interpret denominator coefficients. <code>DenFracLength</code> is always available, but it is read-only until you set <code>CoeffAutoScale</code> to <code>false</code> .  |
| Denominator            | Holds the denominator coefficients for the filter.  |
| DenProdFracLength      | Specifies how the filter algorithm interprets the results of product operations involving denominator coefficients. You can change this property value when you set <code>ProductMode</code> to <code>SpecifyPrecision</code> .   |
| DenStateFracLength     | Specifies the fraction length used to interpret the states associated with denominator coefficients in the filter.  |
| FilterStructure        | Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering — gains, delays, sums, products, and input/output.   |
| InputFracLength        | Specifies the fraction length the filter uses to interpret input data.  |
| InputWordLength        | Specifies the word length applied to interpret input data.  |
| MultiplicandFracLength | Sets the fraction length for values (multiplicands) used in multiply operations in the filter.  |
| MultiplicandWordLength | Sets the word length applied to the values input to a multiply operation (the multiplicands).   |
| NumAccumFracLength     | Specifies how the filter algorithm interprets the results of addition operations involving numerator coefficients. You can change the value of this property after you set <code>AccumMode</code> to <code>SpecifyPrecision</code> .  |
| Numerator              | Holds the numerator coefficient values for the filter.  |
| NumFracLength          | Sets the fraction length used to interpret the value of numerator coefficients.   |
| NumProdFracLength      | Specifies how the filter algorithm interprets the results of product operations involving numerator coefficients. Available to be changed when you set <code>ProductMode</code> to <code>SpecifyPrecision</code> .  |

| Property Name      | Brief Description  |
|--------------------|--|
| NumStateFracLength | For IIR filters, this defines the binary point location applied to the numerator states of the filter. Specifies the fraction length used to interpret the states associated with numerator coefficient operations in the filter.  |
| OutputFracLength   | Determines how the filter interprets the filter output data. You can change the value of <code>OutputFracLength</code> when you set <code>OutputMode</code> to <code>SpecifyPrecision</code> .   |
| OutputMode         | Sets the mode the filter uses to scale the filtered data for output. You have the following choices: <ul style="list-style-type: none"> <li>• <code>AvoidOverflow</code> — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow.</li> <li>• <code>BestPrecision</code> — directs the filter to set the output data word length and fraction length to maximize the precision in the output data.</li> <li>• <code>SpecifyPrecision</code> — lets you set the word and fraction lengths used by the output data from filtering.</li> </ul> |
| OutputWordLength   | Determines the word length used for the output data.   |
| OverflowMode       | Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either <code>saturate</code> (limit the output to the largest positive or negative representable value) or <code>wrap</code> (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow—they maintain full precision.   |
| ProductMode        | Determines how the filter handles the output of product operations. Choose from full precision ( <code>FullPrecision</code> ), or whether to keep the most significant bit ( <code>KeepMSB</code> ) or least significant bit ( <code>KeepLSB</code> ) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set <code>ProductMode</code> to <code>SpecifyPrecision</code> .   |
| ProductWordLength  | Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set <code>ProductMode</code> to <code>SpecifyPrecision</code> .  |
| PersistentMemory   | Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. <code>False</code> is the default setting.   |



| Property Name   | Brief Description   |
|-----------------|---|
| RoundMode       | <p>Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths).</p> <ul style="list-style-type: none"> <li><code>ceil</code> - Round toward positive infinity.</li> <li><code>convergent</code> - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.</li> <li><code>fix</code> - Round toward zero.</li> <li><code>floor</code> - Round toward negative infinity.</li> <li><code>nearest</code> - Round toward nearest. Ties round toward positive infinity.</li> <li><code>round</code> - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.</li> </ul> <p>The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.</p> |
| Signed          | Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.   |
| StateAutoScale  | Setting autoscaling for filter states to <code>true</code> reduces the possibility of overflows occurring during fixed-point operations. Set to <code>false</code> , <code>StateAutoScale</code> lets the filter select the fraction length to limit the overflow potential.  |
| States          | This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions.   |
| StateWordLength | Sets the word length used to represent the filter states.   |

## Examples

Specify a second-order direct-form I transposed filter structure for a `dfilt` object, `hd`, with the following code:

```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
hd = dfilt.dflt(b,a);
% Convert filter to single-precision arithmetic
set(hd,'arithmetic','single')
```

## Version History

Introduced in R2011a

**See Also**

`dsp.IIRFilter`

## dfilt.df1tsos

Discrete-time, SOS direct-form I transposed filter

### Syntax

```
hd = dfilt.df1tsos(s)
hd = dfilt.df1tsos(b1,a1,b2,a2,...)
hd = dfilt.df1tsos(...,g)
hd = dfilt.df1tsos
```

### Description

`hd = dfilt.df1tsos(s)` returns a discrete-time, second-order section, direct-form I, transposed filter object `hd`, with coefficients given in the `s` matrix.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

```
set(hd,'arithmetic','single');
```

- To change to fixed-point filtering, enter

```
set(hd,'arithmetic','fixed');
```

For more information about the property `Arithmetic`, refer to .

`hd = dfilt.df1tsos(b1,a1,b2,a2,...)` returns a discrete-time, second-order section, direct-form I, transposed filter object `hd`, with coefficients for the first section given in the `b1` and `a1` vectors, for the second section given in the `b2` and `a2` vectors, etc.

`hd = dfilt.df1tsos(...,g)` includes a gain vector `g`. The elements of `g` are the gains for each section. The maximum length of `g` is the number of sections plus one. If `g` is not specified, all gains default to one.

`hd = dfilt.df1tsos` returns a default, discrete-time, second-order section, direct-form I, transposed filter object, `hd`. This filter passes the input through to the output unchanged.

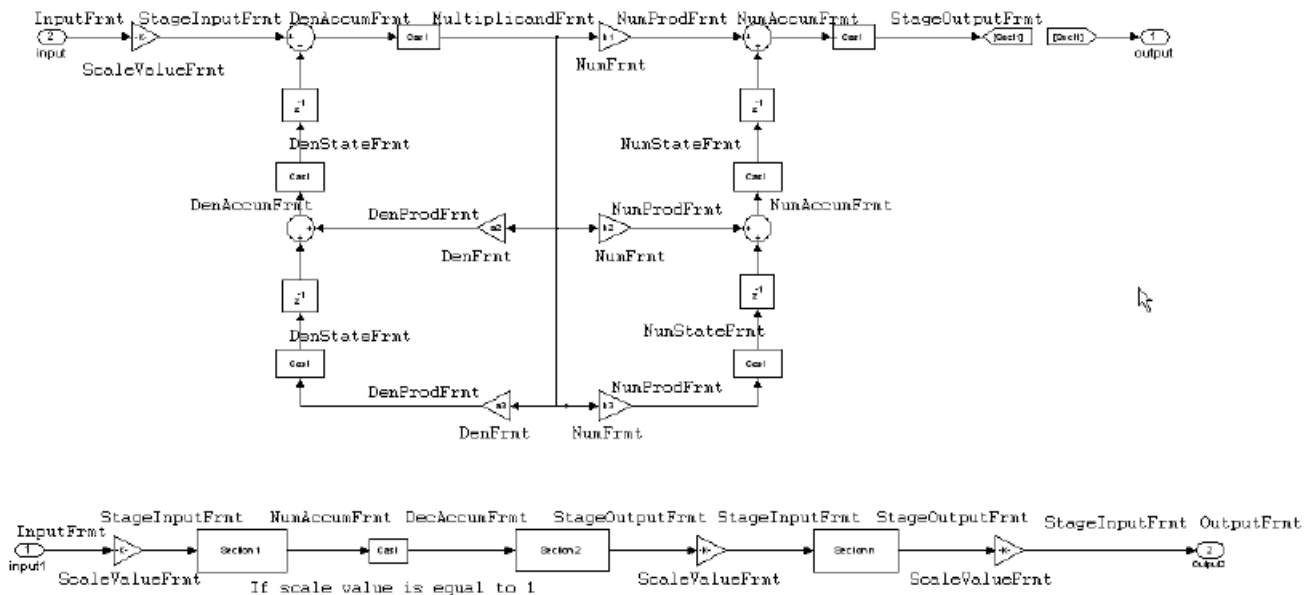
---

**Note** The leading coefficient of the denominator `a(1)` cannot be 0. To allow you to change the arithmetic setting to `fixed` or `single`, `a(1)` must be equal to 1.

---

### Fixed-Point Filter Structure

The following figure shows the signal flow for the direct-form I transposed filter implemented using second-order sections by `dfilt.df1tsos`. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.



### Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the letters “frmt” (format). In this use, “frmt” means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFrmt label refers to the word length and fraction length used to interpret the data input to the filter. The format properties InputWordLength and InputFracLength (as shown in the table) store the word length and the fraction length in bits. Or consider NumFrmt, which refers to the word and fraction lengths (CoeffWordLength, NumFracLength) associated with representing filter numerator coefficients.

| Signal Flow Label | Corresponding Word Length Property | Corresponding Fraction Length Property | Related Properties                  |
|-------------------|------------------------------------|--|-------------------------------------|
| DenAccumFrmt      | AccumWordLength                    | DenAccumFracLength                     | AccumMode, CastBeforeSum            |
| DenFrmt           | CoeffWordLength                    | DenFracLength                          | CoeffAutoScale, Signed, Denominator |
| DenProdFrmt       | CoeffWordLength                    | DenProdFracLength                      | ProductMode, ProductWordLength      |
| DenStateFrmt      | DenStateWordLength                 | DenStateFracLength                     | CastBeforeSum, States               |
| InputFrmt         | InputWordLength                    | InputFracLength                        | None                                |
| MultiplicandFrmt  | Multiplicand-WordLength            | Multiplicand-FracLength                | CastBeforeSum                       |

| Signal Flow Label | Corresponding Word Length Property | Corresponding Fraction Length Property | Related Properties                |
|-------------------|------------------------------------|--|-----------------------------------|
| NumAccumFrmt      | AccumWordLength                    | NumAccumFracLength                     | AccumMode, CastBeforeSum          |
| NumFrmt           | CoeffWordLength                    | NumFracLength                          | CoeffAutoScale, Signed, Numerator |
| NumProdFrmt       | CoeffWordLength                    | NumProdFracLength                      | ProductWordLength, ProductMode    |
| NumStateFrmt      | NumStateWordLength                 | NumStateFracLength                     | States                            |
| OutputFrmt        | OutputWordLength                   | OutputFracLength                       | OutputMode                        |
| ScaleValueFrmt    | CoeffWordLength                    | ScaleValueFracLength                   | CoeffAutoScale, ScaleValues       |

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label DenProdFrmt, which always follows a denominator coefficient multiplication element in the signal flow. The label indicates that denominator coefficients leave the multiplication element with the word length and fraction length associated with product operations that include denominator coefficients. From reviewing the table, you see that the DenProdFrmt refers to the properties ProdWordLength, ProductMode and DenProdFracLength that fully define the denominator format after multiply (or product) operations.

## Properties

In this table you see the properties associated with SOS implementation of transposed direct-form I `dfilt` objects.

**Note** The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

```
get(hd)
```

where `hd` is a filter.

| Property Name   | Brief Description   |
|-----------------|---|
| AccumMode       | Determines how the accumulator outputs stored values. Choose from full precision ( <code>FullPrecision</code> ), or whether to keep the most significant bits ( <code>KeepMSB</code> ) or least significant bits ( <code>KeepLSB</code> ) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set <code>AccumMode</code> to <code>SpecifyPrecision</code> . |
| AccumWordLength | Sets the word length used to store data in the accumulator/buffer.  |

| Property Name          | Brief Description   |
|------------------------|---|
| Arithmetic             | Defines the arithmetic the filter uses. Gives you the options <code>double</code> , <code>single</code> , and <code>fixed</code> . In short, this property defines the operating mode for your filter.  |
| CastBeforeSum          | Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.   |
| CoeffAutoScale         | Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to <code>false</code> enables you to change the <code>NumFracLength</code> and <code>DenFracLength</code> properties to specify the precision used. |
| CoeffWordLength        | Specifies the word length to apply to filter coefficients.  |
| DenAccumFracLength     | Specifies the fraction length used to interpret data in the accumulator used to hold the results of sum operations. You can change the value for this property when you set <code>AccumMode</code> to <code>SpecifyPrecision</code> .   |
| DenFracLength          | Set the fraction length the filter uses to interpret denominator coefficients. <code>DenFracLength</code> is always available, but it is read-only until you set <code>CoeffAutoScale</code> to <code>false</code> .  |
| DenProdFracLength      | Specifies how the filter algorithm interprets the results of product operations involving denominator coefficients. You can change this property value when you set <code>ProductMode</code> to <code>SpecifyPrecision</code> .   |
| DenStateFracLength     | Specifies the fraction length used to interpret the states associated with denominator coefficients in the filter.  |
| FilterStructure        | Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering—gains, delays, sums, products, and input/output.   |
| InputFracLength        | Specifies the fraction length the filter uses to interpret input data.  |
| InputWordLength        | Specifies the word length applied to interpret input data.  |
| MultiplicandFracLength | Sets the fraction length for values (multiplicands) used in multiply operations in the filter.  |
| MultiplicandWordLength | Sets the word length applied to the values input to a multiply operation (the multiplicands)  |

| Property Name       | Brief Description  |
|---------------------|--|
| NumAccumFracLength  | Specifies how the filter algorithm interprets the results of addition operations involving numerator coefficients. You can change the value of this property after you set AccumMode to SpecifyPrecision.  |
| Numerator           | Holds the numerator coefficient values for the filter.   |
| NumProdFracLength   | Specifies how the filter algorithm interprets the results of product operations involving numerator coefficients. Available to be changed when you set ProductMode to SpecifyPrecision.  |
| NumStateFracLength  | For IIR filters, this defines the binary point location applied to the numerator states of the filter. Specifies the fraction length used to interpret the states associated with numerator coefficient operations in the filter.  |
| NumStateWordLength  | For IIR filters, this defines the word length applied to the numerator states of the filter. Specifies the word length used to interpret the states associated with numerator coefficient operations in the filter.  |
| OptimizeScaleValues | When true, the filter skips multiplication-by-one scaling. When false, the filter performs multiplication-by-one scaling.  |
| OutputFracLength    | Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set OutputMode to SpecifyPrecision.   |
| OutputMode          | Sets the mode the filter uses to scale the filtered data for output. You have the following choices: <ul style="list-style-type: none"> <li>• <b>AvoidOverflow</b> — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow.</li> <li>• <b>BestPrecision</b> — directs the filter to set the output data word length and fraction length to maximize the precision in the output data.</li> <li>• <b>SpecifyPrecision</b> — lets you set the word and fraction lengths used by the output data from filtering.</li> </ul> |
| OutputWordLength    | Determines the word length used for the output data.   |

| Property Name     | Brief Description  |
|-------------------|--|
| OverflowMode      | Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either <b>saturate</b> (limit the output to the largest positive or negative representable value) or <b>wrap</b> (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow—they maintain full precision. |
| ProductMode       | Determines how the filter handles the output of product operations. Choose from full precision ( <b>FullPrecision</b> ), or whether to keep the most significant bit ( <b>KeepMSB</b> ) or least significant bit ( <b>KeepLSB</b> ) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set <b>ProductMode</b> to <b>SpecifyPrecision</b> .                                     |
| ProductWordLength | Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set <b>ProductMode</b> to <b>SpecifyPrecision</b> .  |
| PersistentMemory  | Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. <b>False</b> is the default setting.   |



| Property Name        | Brief Description   |
|----------------------|---|
| RoundMode            | <p>Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths).</p> <ul style="list-style-type: none"> <li>• <code>ceil</code> - Round toward positive infinity.</li> <li>• <code>convergent</code> - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.</li> <li>• <code>fix</code> - Round toward zero.</li> <li>• <code>floor</code> - Round toward negative infinity.</li> <li>• <code>nearest</code> - Round toward nearest. Ties round toward positive infinity.</li> <li>• <code>round</code> - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.</li> </ul> <p>The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.</p> |
| ScaleValueFracLength | <p>Scale values work with SOS filters. Setting this property controls how your filter interprets the scale values by setting the fraction length. Only available when you disable <code>AutoScaleMode</code> by setting it to <code>false</code>.</p>   |
| ScaleValues          | <p>Scaling for the filter objects in SOS filters.</p>   |
| Signed               | <p>Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.</p>  |
| SosMatrix            | <p>Holds the filter coefficients as property values. Displays the matrix in the format [sections x coefficients/sectiondatatype]. A [15x6 <code>double</code>] SOS matrix represents a filter with 6 coefficients per section and 15 sections, using data type <code>double</code> to represent the coefficients.</p>   |
| StateAutoScale       | <p>Setting autoscaling for filter states to <code>true</code> reduces the possibility of overflows occurring during fixed-point operations. Set to <code>false</code>, <code>StateAutoScale</code> lets the filter select the fraction length to limit the overflow potential.</p>  |
| States               | <p>This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions.</p>  |

| Property Name   | Brief Description   |
|-----------------|---|
| StateWordLength | Sets the word length used to represent the filter states. |

## Examples

With the following code, this example specifies a second-order section, direct-form I transposed `dfilt` object for a filter. Then convert the filter to fixed-point operation.

```
b = [0.3 0.6 0.3];  
a = [1 0 0.2];  
hd = dfilt.df1tsos(b,a);  
set(hd,'arithmetic','fixed')
```

## Version History

Introduced in R2011a

## See Also

`dsp.BiquadFilter`

## dfilt.df2

Discrete-time, direct-form II filter

### Syntax

```
hd = dfilt.df2(b,a)
hd = dfilt.df2
```

### Description

`hd = dfilt.df2(b,a)` returns a discrete-time, direct-form II filter object `hd`, with numerator coefficients `b` and denominator coefficients `a`.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

```
set(hd,'arithmetic','single');
```

- To change to fixed-point filtering, enter

```
set(hd,'arithmetic','fixed');
```

`hd = dfilt.df2` returns a default, discrete-time, direct-form II filter object `hd`, with `b = 1` and `a = 1`. This filter passes the input through to the output unchanged.

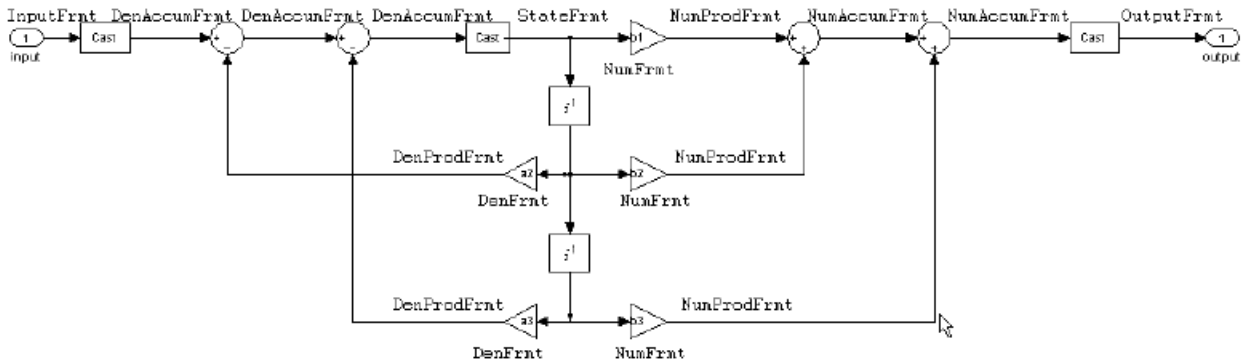
---

**Note** The leading coefficient of the denominator `a(1)` cannot be 0. To allow you to change the arithmetic setting to `fixed` or `single`, `a(1)` must be equal to 1.

---

### Fixed-Point Filter Structure

The following figure shows the signal flow for the direct-form II filter implemented by `dfilt.df2`. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.



### Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the letters “frmt” (format). In this use, “frmt” means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFrmt label refers to the word length and fraction length used to interpret the data input to the filter. The format properties InputWordLength and InputFracLength (as shown in the table) store the word length and the fraction length in bits. Or consider NumFrmt, which refers to the word and fraction lengths (CoeffWordLength, NumFracLength) associated with representing filter numerator coefficients.

| Signal Flow Label | Corresponding Word Length Property | Corresponding Fraction Length Property | Related Properties                  |
|-------------------|------------------------------------|--|-------------------------------------|
| DenAccumFrmt      | AccumWordLength                    | DenAccumFracLength                     | AccumMode, CastBeforeSum            |
| DenFrmt           | CoeffWordLength                    | DenFracLength                          | CoeffAutoScale, Signed, Denominator |
| DenProdFrmt       | CoeffWordLength                    | DenProdFracLength                      | ProductMode, ProductWordLength      |
| InputFrmt         | InputWordLength                    | InputFracLength                        | None                                |
| NumAccumFrmt      | AccumWordLength                    | NumAccumFracLength                     | AccumMode, CastBeforeSum            |
| NumFrmt           | CoeffWordLength                    | NumFracLength                          | CoeffAutoScale, Signed, Numerator   |
| NumProdFrmt       | CoeffWordLength                    | NumProdFracLength                      | ProductWordLength, ProductMode      |
| OutputFrmt        | OutputWordLength                   | OutputFracLength                       | OutputMode                          |
| StateFrmt         | StateWordLength                    | StateFracLength                        | States                              |

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label DenProdFrmt, which always follows a denominator coefficient multiplication element in the signal flow. The label indicates that denominator coefficients leave the multiplication element with the word length and fraction length associated with product operations that include denominator coefficients. From reviewing the table, you see that the DenProdFrmt refers to the properties ProdWordLength, ProductMode and DenProdFracLength that fully define the denominator format after multiply (or product) operations.

## Properties

In this table you see the properties associated with the df2 implementation of dfilt objects.

**Note** The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

```
get(hd)
```

where `hd` is a filter.

| Property Name      | Brief Description   |
|--------------------|---|
| AccumMode          | Determines how the accumulator outputs stored values. Choose from full precision ( <code>FullPrecision</code> ), or whether to keep the most significant bits ( <code>KeepMSB</code> ) or least significant bits ( <code>KeepLSB</code> ) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set <code>AccumMode</code> to <code>SpecifyPrecision</code> . |
| AccumWordLength    | Sets the word length used to store data in the accumulator/buffer.  |
| Arithmetic         | Defines the arithmetic the filter uses. Gives you the options <code>double</code> , <code>single</code> , and <code>fixed</code> . In short, this property defines the operating mode for your filter.  |
| CastBeforeSum      | Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.   |
| CoeffAutoScale     | Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to <code>false</code> enables you to change the <code>NumFracLength</code> and <code>DenFracLength</code> properties to specify the precision used.   |
| CoeffWordLength    | Specifies the word length to apply to filter coefficients.  |
| DenAccumFracLength | Specifies the fraction length used to interpret data in the accumulator used to hold the results of sum operations. You can change the value for this property when you set <code>AccumMode</code> to <code>SpecifyPrecision</code> .   |

| Property Name      | Brief Description  |
|--------------------|--|
| DenFracLength      | Set the fraction length the filter uses to interpret denominator coefficients. DenFracLength is always available, but it is read-only until you set CoeffAutoScale to false.   |
| Denominator        | Holds the denominator coefficients for IIR filters.  |
| DenProdFracLength  | Specifies how the filter algorithm interprets the results of product operations involving denominator coefficients. You can change this property value when you set ProductMode to SpecifyPrecision.   |
| FilterStructure    | Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering — gains, delays, sums, products, and input/output.  |
| InputFracLength    | Specifies the fraction length the filter uses to interpret input data.   |
| InputWordLength    | Specifies the word length applied to interpret input data.   |
| NumAccumFracLength | Specifies how the filter algorithm interprets the results of addition operations involving numerator coefficients. You can change the value of this property after you set AccumMode to SpecifyPrecision.  |
| Numerator          | Holds the numerator coefficient values for the filter.   |
| NumFracLength      | Sets the fraction length used to interpret the value of numerator coefficients.  |
| NumProdFracLength  | Specifies how the filter algorithm interprets the results of product operations involving numerator coefficients. Available to be changed when you set ProductMode to SpecifyPrecision.  |
| OutputFracLength   | Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set OutputMode to SpecifyPrecision.   |
| OutputMode         | Sets the mode the filter uses to scale the filtered data for output. You have the following choices: <ul style="list-style-type: none"> <li>• <b>AvoidOverflow</b> — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow.</li> <li>• <b>BestPrecision</b> — directs the filter to set the output data word length and fraction length to maximize the precision in the output data.</li> <li>• <b>SpecifyPrecision</b> — lets you set the word and fraction lengths used by the output data from filtering.</li> </ul> |
| OutputWordLength   | Determines the word length used for the output data.   |

| Property Name    | Brief Description   |
|------------------|---|
| OverflowMode     | Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either <code>saturate</code> (limit the output to the largest positive or negative representable value) or <code>wrap</code> (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow—they maintain full precision.  |
| ProductMode      | Determines how the filter handles the output of product operations. Choose from full precision ( <code>FullPrecision</code> ), or whether to keep the most significant bit ( <code>KeepMSB</code> ) or least significant bit ( <code>KeepLSB</code> ) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set <code>ProductMode</code> to <code>SpecifyPrecision</code> .  |
| PersistentMemory | Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. <code>False</code> is the default setting.  |
| RoundMode        | <p>Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths).</p> <ul style="list-style-type: none"> <li>• <code>ceil</code> - Round toward positive infinity.</li> <li>• <code>convergent</code> - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.</li> <li>• <code>fix</code> - Round toward zero.</li> <li>• <code>floor</code> - Round toward negative infinity.</li> <li>• <code>nearest</code> - Round toward nearest. Ties round toward positive infinity.</li> <li>• <code>round</code> - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.</li> </ul> <p>The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.</p> |
| Signed           | Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.   |
| StateFracLength  | When you set <code>StateAutoScale</code> to <code>false</code> , you enable the <code>StateFracLength</code> property that lets you set the fraction length applied to interpret the filter states.   |
| States           | This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions.   |

| Property Name   | Brief Description   |
|-----------------|---|
| StateWordLength | Sets the word length used to represent the filter states. |

## Examples

Specify a second-order direct-form II filter structure for a `dfilt` object, `hd`, with the following code:

```
b = [0.3 0.6 0.3];  
a = [1 0 0.2];  
hd = dfilt.df2(b,a);  
% Change filter to fixed-point  
set(hd,'arithmetic','fixed')
```

## Version History

Introduced in R2011a

## See Also

`dsp.IIRFilter`



## dfilt.df2sos

Discrete-time, SOS, direct-form II filter

### Syntax

```
hd = dfilt.df2sos(s)
hd = dfilt.df2sos(b1,a1,b2,a2,...)
hd = dfilt.df2sos(...,g)
hd = dfilt.df2sos
```

### Description

`hd = dfilt.df2sos(s)` returns a discrete-time, second-order section, direct-form II filter object `hd`, with coefficients given in the `s` matrix.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

```
set(hd,'arithmetic','single');
```

- To change to fixed-point filtering, enter

```
set(hd,'arithmetic','fixed');
```

`hd = dfilt.df2sos(b1,a1,b2,a2,...)` returns a discrete-time, second-order section, direct-form II object, `hd`, with coefficients for the first section given in the `b1` and `a1` vectors, for the second section given in the `b2` and `a2` vectors, etc.

`hd = dfilt.df2sos(...,g)` includes a gain vector `g`. The elements of `g` are the gains for each section. The maximum length of `g` is the number of sections plus one. If `g` is not specified, all gains default to one.

`hd = dfilt.df2sos` returns a default, discrete-time, second-order section, direct-form II filter object, `hd`. This filter passes the input through to the output unchanged.

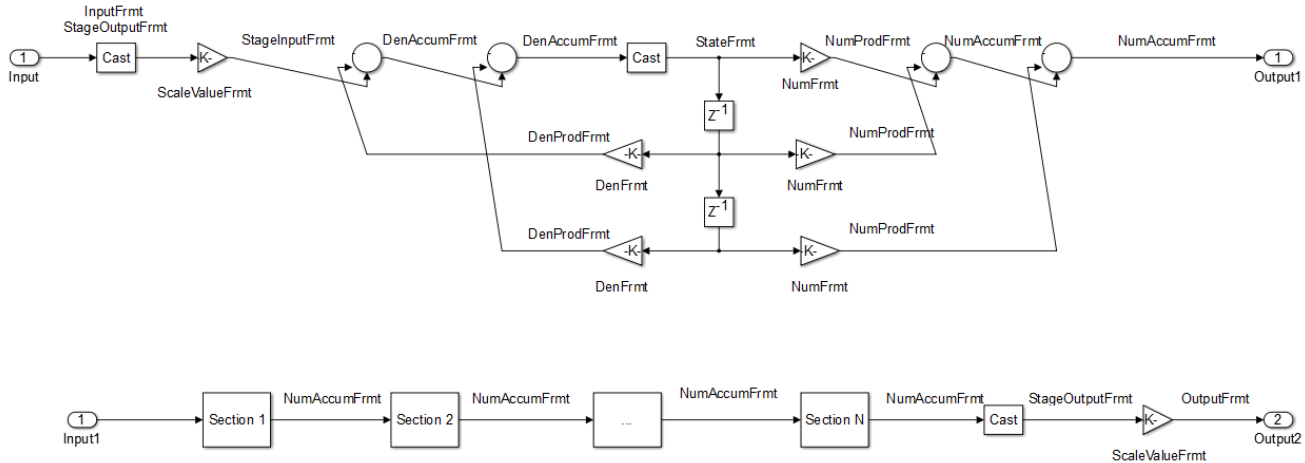
---

**Note** The leading coefficient of the denominator `a(1)` cannot be 0. To allow you to change the arithmetic setting to `fixed` or `single`, `a(1)` must be equal to 1.

---

### Fixed-Point Filter Structure

The figure below shows the signal flow for the direct-form II filter implemented with second-order sections by `dfilt.df2sos`. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.



### Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the letters “frmt” (format). In this use, “frmt” means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFrmt label refers to the word length and fraction length used to interpret the data input to the filter. The frmt properties InputWordLength and InputFracLength (as shown in the table) store the word length and the fraction length in bits. Or consider NumFrmt, which refers to the word and fraction lengths (CoeffWordLength, NumFracLength) associated with representing filter numerator coefficients.

| Signal Flow Label | Corresponding Word Length Property | Corresponding Fraction Length Property | Related Properties                        |
|-------------------|------------------------------------|--|---|
| DenAccumFrmt      | AccumWordLength                    | DenAccumFracLength                     | AccumMode, CastBeforeSum                  |
| DenFrmt           | CoeffWordLength                    | DenFracLength                          | CoeffAutoScale, Signed, sosMatrix         |
| DenProdFrmt       | CoeffWordLength                    | DenProdFracLength                      | ProductMode, ProductWordLength, sosMatrix |
| InputFrmt         | InputWordLength                    | InputFracLength                        | None                                      |
| NumAccumFrmt      | AccumWordLength                    | NumAccumFracLength                     | AccumMode, CastBeforeSum                  |

| Signal Flow Label | Corresponding Word Length Property | Corresponding Fraction Length Property | Related Properties                |
|-------------------|------------------------------------|--|-----------------------------------|
| NumFrmt           | CoeffWordLength                    | NumFracLength                          | CoeffAutoScale, Signed, sosMatrix |
| NumProdFrmt       | CoeffWordLength                    | NumProdFracLength                      | ProductWordLength, ProductMode    |
| OutputFrmt        | OutputWordLength                   | OutputFracLength                       | OutputMode                        |
| ScaleValueFrmt    | CoeffWordLength                    | ScaleValueFracLength                   | CoeffAutoScale, ScaleValues       |
| SectionInputFrmt  | SectionInput-WordLength            | SectionInput-FracLength                | SectionInput-AutoScale            |
| SectionOutputFrmt | SectionOutput-WordLength           | SectionOutput-FracLength               | SectionOutput-AutoScale           |
| StateFrmt         | StateWordLength                    | StateFracLength                        | CastBeforeSum, States             |

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label DenProdFrmt, which always follows a denominator coefficient multiplication element in the signal flow. The label indicates that denominator coefficients leave the multiplication element with the word length and fraction length associated with product operations that include denominator coefficients. From reviewing the table, you see that the DenProdFrmt refers to the properties ProdWordLength, ProductMode and DenProdFracLength that fully define the denominator format after multiply (or product) operations.

## Properties

In this table you see the properties associated with second-order section implementation of direct-form II dfilt objects.

**Note** The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

```
get(hd)
```

where `hd` is a filter.

| Property Name | Brief Description   |
|---------------|---|
| AccumMode     | Determines how the accumulator outputs stored values. Choose from full precision ( <code>FullPrecision</code> ), or whether to keep the most significant bits ( <code>KeepMSB</code> ) or least significant bits ( <code>KeepLSB</code> ) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set <code>AccumMode</code> to <code>SpecifyPrecision</code> . |

| Property Name      | Brief Description   |
|--------------------|---|
| AccumWordLength    | Sets the word length used to store data in the accumulator/buffer.  |
| Arithmetic         | Defines the arithmetic the filter uses. Gives you the options <code>double</code> , <code>single</code> , and <code>fixed</code> . In short, this property defines the operating mode for your filter.  |
| CastBeforeSum      | Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.   |
| CoeffAutoScale     | Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to <code>false</code> enables you to change the <code>NumFracLength</code> and <code>DenFracLength</code> properties to specify the precision used. |
| CoeffWordLength    | Specifies the word length to apply to filter coefficients.  |
| DenAccumFracLength | Specifies the fraction length used to interpret data in the accumulator used to hold the results of sum operations. You can change the value for this property when you set <code>AccumMode</code> to <code>SpecifyPrecision</code> .   |
| DenFracLength      | Set the fraction length the filter uses to interpret denominator coefficients. <code>DenFracLength</code> is always available, but it is read-only until you set <code>CoeffAutoScale</code> to <code>false</code> .  |
| DenProdFracLength  | Specifies how the filter algorithm interprets the results of product operations involving denominator coefficients. You can change this property value when you set <code>ProductMode</code> to <code>SpecifyPrecision</code> .   |
| FilterStructure    | Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering—gains, delays, sums, products, and input/output.   |
| InputFracLength    | Specifies the fraction length the filter uses to interpret input data.  |
| InputWordLength    | Specifies the word length applied to interpret input data.  |
| NumAccumFracLength | Specifies how the filter algorithm interprets the results of addition operations involving numerator coefficients. You can change the value of this property after you set <code>AccumMode</code> to <code>SpecifyPrecision</code> .  |
| NumFracLength      | Sets the fraction length used to interpret the value of numerator coefficients.<br><br><code>NumFracLength</code> is always available, but it is read-only until you set <code>CoeffAutoScale</code> to <code>false</code> .  |
| NumProdFracLength  | Specifies how the filter algorithm interprets the results of product operations involving numerator coefficients. Available to be changed when you set <code>ProductMode</code> to <code>SpecifyPrecision</code> .  |

| Property Name       | Brief Description  |
|---------------------|--|
| OptimizeScaleValues | When true, the filter skips multiplication-by-one scaling. When false, the filter performs multiplication-by-one scaling.  |
| OutputFracLength    | Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set OutputMode to SpecifyPrecision.   |
| OutputMode          | Sets the mode the filter uses to scale the filtered data for output. You have the following choices: <ul style="list-style-type: none"> <li>• <b>AvoidOverflow</b> — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow.</li> <li>• <b>BestPrecision</b> — directs the filter to set the output data word length and fraction length to maximize the precision in the output data.</li> <li>• <b>SpecifyPrecision</b> — lets you set the word and fraction lengths used by the output data from filtering.</li> </ul> |
| OutputWordLength    | Determines the word length used for the output data.   |
| OverflowMode        | Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either <b>saturate</b> (limit the output to the largest positive or negative representable value) or <b>wrap</b> (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.   |
| ProductMode         | Determines how the filter handles the output of product operations. Choose from full precision ( <b>FullPrecision</b> ), or whether to keep the most significant bit ( <b>KeepMSB</b> ) or least significant bit ( <b>KeepLSB</b> ) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set ProductMode to SpecifyPrecision.  |
| ProductWordLength   | Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set ProductMode to SpecifyPrecision.   |
| PersistentMemory    | Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. <b>False</b> is the default setting.   |

| Property Name           | Brief Description   |
|-------------------------|---|
| RoundMode               | <p>Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths).</p> <ul style="list-style-type: none"> <li>• <code>ceil</code> - Round toward positive infinity.</li> <li>• <code>convergent</code> - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.</li> <li>• <code>fix</code> - Round toward zero.</li> <li>• <code>floor</code> - Round toward negative infinity.</li> <li>• <code>nearest</code> - Round toward nearest. Ties round toward positive infinity.</li> <li>• <code>round</code> - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.</li> </ul> <p>The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.</p> |
| ScaleValueFracLength    | <p>Scale values work with SOS filters. Setting this property controls how your filter interprets the scale values by setting the fraction length.</p> <p><code>ScaleValueFracLength</code> is always available, but it is read-only until you set <code>CoeffAutoScale</code> to <code>false</code>.</p>  |
| ScaleValues             | Scaling for the filter objects in SOS filters.  |
| SectionInputAutoScale   | Tells the filter whether to set the stage input data format to minimize the occurrence of overflow conditions.  |
| SectionInputFracLength  | Lets you set the fraction length for section inputs in SOS filters, if you set <code>SectionInputAutoScale</code> to <code>false</code> .   |
| SectionInputWordLength  | Lets you set the word length for section inputs in SOS filters, if you set <code>SectionInputAutoScale</code> to <code>false</code> .   |
| SectionOutputAutoScale  | Tells the filter whether to set the section output data format to minimize the occurrence of overflow conditions.   |
| SectionOutputFracLength | Lets you set the fraction length for section outputs in SOS filters, if you set <code>SectionOutputAutoScale</code> to <code>off</code> .   |
| SectionOutputWordLength | Lets you set the word length for section outputs in SOS filters, if you set <code>SectionOutputAutoScale</code> to <code>false</code> .   |
| Signed                  | Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.   |
| SosMatrix               | Holds the filter coefficients as property values. Displays the matrix in the format [sections x coefficients/section datatype]. A [15x6 double] SOS matrix represents a filter with 6 coefficients per section and 15 sections, using data type <code>double</code> to represent the coefficients.  |

| Property Name   | Brief Description   |
|-----------------|---|
| StateFracLength | StateFracLength property that lets you set the fraction length applied to interpret the filter states.  |
| States          | This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. |
| StateWordLength | Sets the word length used to represent the filter states.   |

## Examples

Specify a second-order section, direct-form II `dfilt` object for a Butterworth filter converted to second-order sections, with the following code:

```
[z,p,k] = butter(30,0.5);
[s,g] = zp2sos(z,p,k);
hd = dfilt.df2sos(s,g);
% Convert filter to fixed-point
hd.arithmetic='fixed';
```

## Version History

Introduced in R2011a

## See Also

`dsp.BiquadFilter`

## dfilt.df2t

Discrete-time, direct-form II transposed filter

### Syntax

```
hd = dfilt.df2t(b,a)
hd = dfilt.df2t
```

### Description

`hd = dfilt.df2t(b,a)` returns a discrete-time, direct-form II transposed filter object `hd`, with numerator coefficients `b` and denominator coefficients `a`.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

```
set(hd,'arithmetic','single');
```

- To change to fixed-point filtering, enter

```
set(hd,'arithmetic','fixed');
```

`hd = dfilt.df2t` returns a default, discrete-time, direct-form II transposed filter object `hd`, with `b = 1` and `a = 1`. This filter passes the input through to the output unchanged.

---

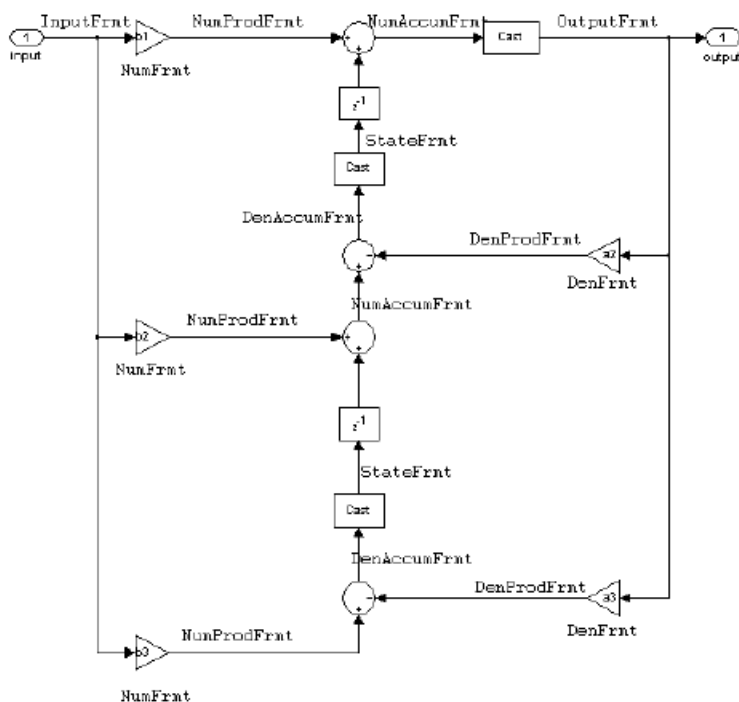
**Note** The leading coefficient of the denominator `a(1)` cannot be 0. To allow you to change the arithmetic setting to `fixed` or `single`, `a(1)` must be equal to 1.

---

### Fixed-Point Filter Structure

The following figure shows the signal flow for the direct-form II transposed filter implemented by `dfilt.df2t`. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.





### Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the letters “frmt.” In this use, “frmt” means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFrmt label refers to the word length and fraction length used to interpret the data input to the filter. The format properties InputWordLength and InputFracLength (as shown in the table) store the word length and the fraction length in bits. Or consider NumFrmt, which refers to the word and fraction lengths (CoeffWordLength, NumFracLength) associated with representing filter numerator coefficients.

| Signal Flow Label | Corresponding Word Length Property | Corresponding Fraction Length Property | Related Properties                  |
|-------------------|------------------------------------|--|-------------------------------------|
| DenAccumFrmt      | AccumWordLength                    | DenAccumFracLength                     | AccumMode, CastBeforeSum            |
| DenFrmt           | CoeffWordLength                    | DenFracLength                          | CoeffAutoScale, Signed, Denominator |
| DenProdFrmt       | CoeffWordLength                    | DenProdFracLength                      | ProductMode, ProductWordLength      |
| InputFrmt         | InputWordLength                    | InputFracLength                        | None                                |
| NumAccumFrmt      | AccumWordLength                    | NumAccumFracLength                     | AccumMode, CastBeforeSum            |

| Signal Flow Label | Corresponding Word Length Property | Corresponding Fraction Length Property | Related Properties                |
|-------------------|------------------------------------|--|-----------------------------------|
| NumFrmt           | CoeffWordLength                    | NumFracLength                          | CoeffAutoScale, Signed, Numerator |
| NumProdFrmt       | CoeffWordLength                    | NumProdFracLength                      | ProductWordLength, ProductMode    |
| OutputFrmt        | OutputWordLength                   | OutputFracLength                       | None                              |
| StateFrmt         | StateWordLength                    | StateFracLength                        | States                            |

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label DenProdFrmt, which always follows a denominator coefficient multiplication element in the signal flow. The label indicates that denominator coefficients leave the multiplication element with the word length and fraction length associated with product operations that include denominator coefficients. From reviewing the table, you see that the DenProdFrmt refers to the properties ProdWordLength, ProductMode and DenProdFracLength that fully define the denominator format after multiply (or product) operations.

## Properties

In this table you see the properties associated with `df2t` implementation of `dfilt` objects.

**Note** The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

```
get(hd)
```

where `hd` is a filter.

| Property Name   | Brief Description   |
|-----------------|---|
| AccumMode       | Determines how the accumulator outputs stored values. Choose from full precision ( <code>FullPrecision</code> ), or whether to keep the most significant bits ( <code>KeepMSB</code> ) or least significant bits ( <code>KeepLSB</code> ) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set <code>AccumMode</code> to <code>SpecifyPrecision</code> . |
| AccumWordLength | Sets the word length used to store data in the accumulator/buffer.  |
| Arithmetic      | Defines the arithmetic the filter uses. Gives you the options <code>double</code> , <code>single</code> , and <code>fixed</code> . In short, this property defines the operating mode for your filter.  |
| CastBeforeSum   | Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.   |

| Property Name      | Brief Description  |
|--------------------|--|
| CoeffAutoScale     | Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to <code>false</code> enables you to change the <code>NumFracLength</code> and <code>DenFracLength</code> properties to specify the precision used.  |
| CoeffWordLength    | Specifies the word length to apply to filter coefficients.   |
| DenAccumFracLength | Specifies the fraction length used to interpret data in the accumulator used to hold the results of sum operations. You can change the value for this property when you set <code>AccumMode</code> to <code>SpecifyPrecision</code> .  |
| DenFracLength      | Set the fraction length the filter uses to interpret denominator coefficients. <code>DenFracLength</code> is always available, but it is read-only until you set <code>CoeffAutoScale</code> to <code>false</code> .   |
| Denominator        | Holds the denominator coefficients for IIR filters.  |
| DenProdFracLength  | Specifies how the filter algorithm interprets the results of product operations involving denominator coefficients. You can change this property value when you set <code>ProductMode</code> to <code>SpecifyPrecision</code> .  |
| FilterStructure    | Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering—gains, delays, sums, products, and input/output.  |
| InputFracLength    | Specifies the fraction length the filter uses to interpret input data.   |
| InputWordLength    | Specifies the word length applied to interpret input data.   |
| NumAccumFracLength | Specifies how the filter algorithm interprets the results of addition operations involving numerator coefficients. You can change the value of this property after you set <code>AccumMode</code> to <code>SpecifyPrecision</code> .   |
| Numerator          | Holds the numerator coefficient values for the filter.   |
| NumFracLength      | Sets the fraction length used to interpret the value of numerator coefficients.  |
| NumProdFracLength  | Specifies how the filter algorithm interprets the results of product operations involving numerator coefficients. Available to be changed when you set <code>ProductMode</code> to <code>SpecifyPrecision</code> .   |
| OutputFracLength   | Determines how the filter interprets the filter output data.   |
| OutputWordLength   | Determines the word length used for the output data.   |
| OverflowMode       | Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either <code>saturate</code> (limit the output to the largest positive or negative representable value) or <code>wrap</code> (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision. |

| Property Name     | Brief Description   |
|-------------------|---|
| ProductMode       | Determines how the filter handles the output of product operations. Choose from full precision ( <code>FullPrecision</code> ), or whether to keep the most significant bit ( <code>KeepMSB</code> ) or least significant bit ( <code>KeepLSB</code> ) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set <code>ProductMode</code> to <code>SpecifyPrecision</code> .  |
| ProductWordLength | Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set <code>ProductMode</code> to <code>SpecifyPrecision</code> .   |
| PersistentMemory  | Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. <code>False</code> is the default setting.  |
| RoundMode         | <p>Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths).</p> <ul style="list-style-type: none"> <li><code>ceil</code> - Round toward positive infinity.</li> <li><code>convergent</code> - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.</li> <li><code>fix</code> - Round toward zero.</li> <li><code>floor</code> - Round toward negative infinity.</li> <li><code>nearest</code> - Round toward nearest. Ties round toward positive infinity.</li> <li><code>round</code> - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.</li> </ul> <p>The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.</p> |
| Signed            | Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.   |
| StateAutoScale    | Setting autoscaling for filter states to <code>true</code> reduces the possibility of overflows occurring during fixed-point operations. Set to <code>false</code> , <code>StateAutoScale</code> lets the filter select the fraction length to limit the overflow potential.  |
| StateFracLength   | When you set <code>StateAutoScale</code> to <code>false</code> , you enable the <code>StateFracLength</code> property that lets you set the fraction length applied to interpret the filter states.   |
| States            | This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions.   |
| StateWordLength   | Sets the word length used to represent the filter states.   |

## Examples

Create a fixed-point filter by specifying a second-order direct-form II transposed filter structure for a `dfilt` object, and then converting the double-precision arithmetic setting to fixed-point.

```
b = [0.3 0.6 0.3];  
a = [1 0 0.2];  
hd = dfilt.df2t(b,a);  
% Convert filter to fixed-point  
set(hd,'arithmetic','fixed')
```

## Version History

**Introduced in R2011a**

### See Also

`dsp.IIRFilter`

## dfilt.df2tsos

Discrete-time, SOS direct-form II transposed filter

### Syntax

```
hd = dfilt.df2tsos(s)
hd = dfilt.df2tsos(b1,a1,b2,a2,...)
hd = dfilt.df2tsos(...,g)
hd = dfilt.df2tsos
```

### Description

`hd = dfilt.df2tsos(s)` returns a discrete-time, second-order section, direct-form II, transposed filter object `hd`, with coefficients given in the matrix `s`.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

```
set(hd,'arithmetic','single');
```

- To change to fixed-point filtering, enter

```
set(hd,'arithmetic','fixed');
```

`hd = dfilt.df2tsos(b1,a1,b2,a2,...)` returns a discrete-time, second-order section, direct-form II, transposed filter object `hd`, with coefficients for the first section given in the `b1` and `a1` vectors, for the second section given in the `b2` and `a2` vectors, etc.

`hd = dfilt.df2tsos(...,g)` includes a gain vector `g`. The elements of `g` are the gains for each section. The maximum length of `g` is the number of sections plus one. If `g` is not specified, all gains default to one.

`hd = dfilt.df2tsos` returns a default, discrete-time, second-order section, direct-form II, transposed filter object, `hd`. This filter passes the input through to the output unchanged.

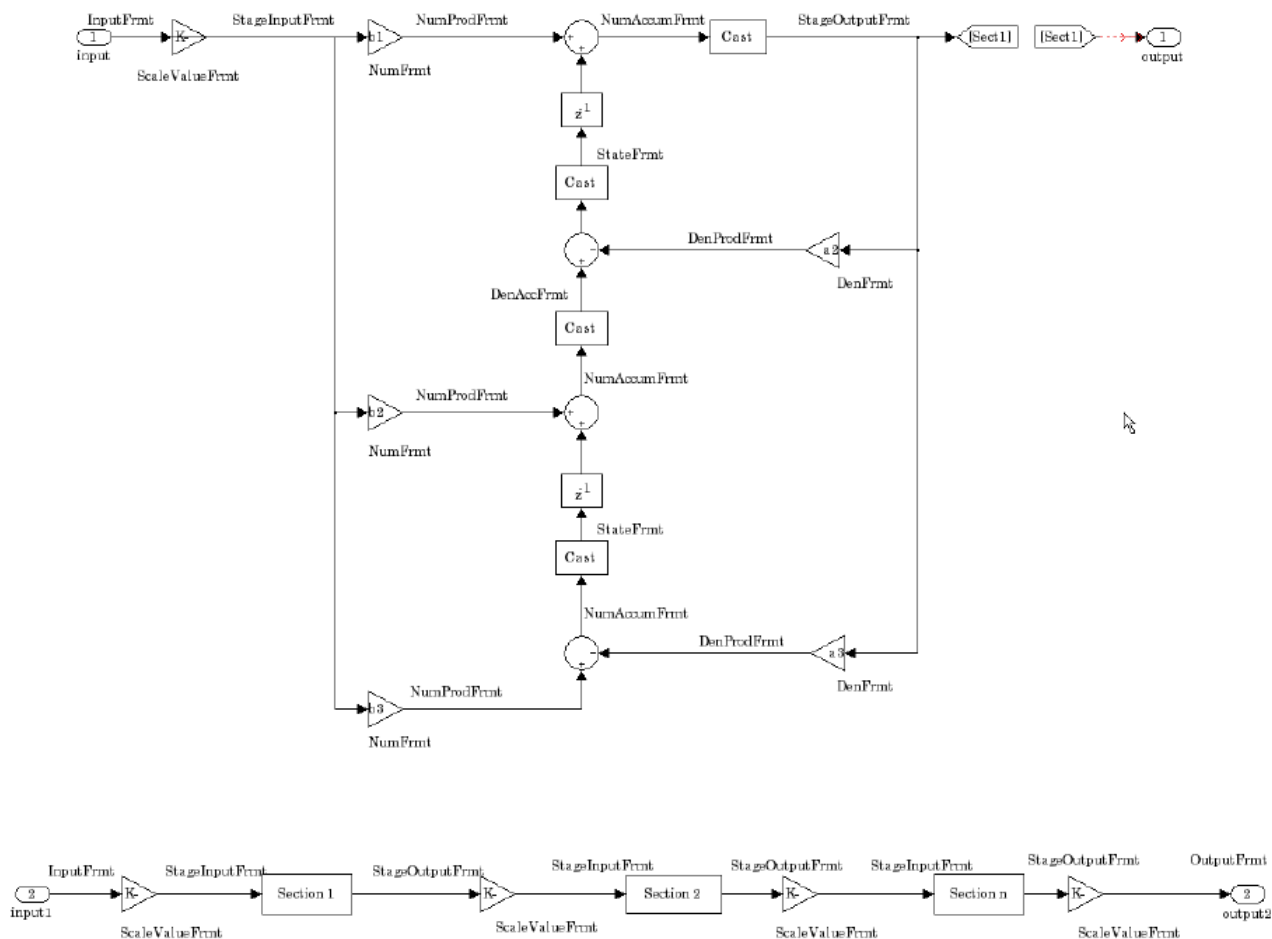
---

**Note** The leading coefficient of the denominator `a(1)` cannot be 0. To allow you to change the arithmetic setting to `fixed` or `single`, `a(1)` must be equal to 1.

---

### Fixed-Point Filter Structure

The figure below shows the signal flow for the second-order section transposed direct-form II filter implemented by `dfilt.dftsos`. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.



### Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the letters “frmt” (format). In this use, “frmt” indicates the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFrmt label refers to the word length and fraction length used to interpret the data input to the filter. The format properties InputWordLength and InputFracLength (as shown in the table) store the word length and the fraction length in bits. Or consider NumFrmt, which refers to the word and fraction lengths (CoeffWordLength, NumFracLength) associated with representing filter numerator coefficients.

| Signal Flow Label | Corresponding Word Length Property | Corresponding Fraction Length Property | Related Properties       |
|-------------------|------------------------------------|--|--------------------------|
| DenAccumFrmt      | AccumWordLength                    | DenAccumFracLength                     | AccumMode, CastBeforeSum |

| Signal Flow Label | Corresponding Word Length Property | Corresponding Fraction Length Property | Related Properties                  |
|-------------------|------------------------------------|--|-------------------------------------|
| DenFrmt           | CoeffWordLength                    | DenFracLength                          | CoeffAutoScale, Signed, Denominator |
| DenProdFrmt       | CoeffWordLength                    | DenProdFracLength                      | ProductMode, ProductWordLength      |
| InputFrmt         | InputWordLength                    | InputFracLength                        | None                                |
| NumAccumFrmt      | AccumWordLength                    | NumAccumFracLength                     | AccumMode, CastBeforeSum            |
| NumFrmt           | CoeffWordLength                    | NumFracLength                          | CoeffAutoScale, SignedNumerator     |
| NumProdFrmt       | CoeffWordLength                    | NumProdFracLength                      | ProductWordLength, ProductMode      |
| OutputFrmt        | OutputWordLength                   | OutputFracLength                       | OutputMode                          |
| ScaleValueFrmt    | CoeffWordLength                    | ScaleValueFracLength                   | CoeffAutoScale, ScaleValues         |
| SectionInputFrmt  | SectionInput-WordLength            | SectionInput-FracLength                |                                     |
| SectionOutputFrmt | SectionOutput-WordLength           | SectionOutput-FracLength               |                                     |
| StateFrmt         | StateWordLength                    | StateFracLength                        | States                              |

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label DenProdFrmt, which always follows a denominator coefficient multiplication element in the signal flow. The label indicates that denominator coefficients leave the multiplication element with the word length and fraction length associated with product operations that include denominator coefficients. From reviewing the table, you see that the DenProdFrmt refers to the properties ProdWordLength, ProductMode and DenProdFracLength that fully define the denominator format after multiply (or product) operations.

## Properties

In this table you see the properties associated with second-order section implementation of transposed direct-form II `dfilt` objects.

---

**Note** The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

```
get(hd)
```

where `hd` is a filter.

---



| Property Name      | Brief Description   |
|--------------------|---|
| AccumMode          | Determines how the accumulator outputs stored values. Choose from full precision ( <code>FullPrecision</code> ), or whether to keep the most significant bits ( <code>KeepMSB</code> ) or least significant bits ( <code>KeepLSB</code> ) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set <code>AccumMode</code> to <code>SpecifyPrecision</code> . |
| AccumWordLength    | Sets the word length used to store data in the accumulator/buffer.  |
| Arithmetic         | Defines the arithmetic the filter uses. Gives you the options <code>double</code> , <code>single</code> , and <code>fixed</code> . In short, this property defines the operating mode for your filter.  |
| CastBeforeSum      | Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.   |
| CoeffAutoScale     | Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to <code>false</code> enables you to change the <code>NumFracLength</code> and <code>DenFracLength</code> properties to specify the precision used.   |
| CoeffWordLength    | Specifies the word length to apply to filter coefficients.  |
| DenAccumFracLength | Specifies the fraction length used to interpret data in the accumulator used to hold the results of sum operations. You can change the value for this property when you set <code>AccumMode</code> to <code>SpecifyPrecision</code> .   |
| DenFracLength      | Set the fraction length the filter uses to interpret denominator coefficients. <code>DenFracLength</code> is always available, but it is read-only until you set <code>CoeffAutoScale</code> to <code>false</code> .  |
| DenProdFracLength  | Specifies how the filter algorithm interprets the results of product operations involving denominator coefficients. You can change this property value when you set <code>ProductMode</code> to <code>SpecifyPrecision</code> .   |
| FilterStructure    | Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering — gains, delays, sums, products, and input/output.   |
| InputFracLength    | Specifies the fraction length the filter uses to interpret input data.  |
| InputWordLength    | Specifies the word length applied to interpret input data.  |
| NumAccumFracLength | Specifies how the filter algorithm interprets the results of addition operations involving numerator coefficients. You can change the value of this property after you set <code>AccumMode</code> to <code>SpecifyPrecision</code> .  |
| NumFracLength      | Sets the fraction length used to interpret the value of numerator coefficients.   |
| NumProdFracLength  | Specifies how the filter algorithm interprets the results of product operations involving numerator coefficients. Available to be changed when you set <code>ProductMode</code> to <code>SpecifyPrecision</code> .  |

| Property Name       | Brief Description  |
|---------------------|--|
| OptimizeScaleValues | When true, the filter skips multiplication-by-one scaling. When false, the filter performs multiplication-by-one scaling.  |
| OutputFracLength    | Determines how the filter interprets the filter output data. You can change the value of <code>OutputFracLength</code> when you set <code>OutputMode</code> to <code>SpecifyPrecision</code> .   |
| OutputMode          | Sets the mode the filter uses to scale the filtered data for output. You have the following choices: <ul style="list-style-type: none"> <li>• <code>AvoidOverflow</code> — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow.</li> <li>• <code>BestPrecision</code> — directs the filter to set the output data word length and fraction length to maximize the precision in the output data.</li> <li>• <code>SpecifyPrecision</code> — lets you set the word and fraction lengths used by the output data from filtering.</li> </ul> |
| OutputWordLength    | Determines the word length used for the output data.   |
| OverflowMode        | Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either <code>saturate</code> (limit the output to the largest positive or negative representable value) or <code>wrap</code> (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.   |
| ProductMode         | Determines how the filter handles the output of product operations. Choose from full precision ( <code>FullPrecision</code> ), or whether to keep the most significant bit ( <code>KeepMSB</code> ) or least significant bit ( <code>KeepLSB</code> ) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set <code>ProductMode</code> to <code>SpecifyPrecision</code> .   |
| ProductWordLength   | Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set <code>ProductMode</code> to <code>SpecifyPrecision</code> .  |
| PersistentMemory    | Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. <code>False</code> is the default setting.   |

| Property Name           | Brief Description   |
|-------------------------|---|
| RoundMode               | <p>Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths).</p> <ul style="list-style-type: none"> <li>• <code>ceil</code> - Round toward positive infinity.</li> <li>• <code>convergent</code> - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.</li> <li>• <code>fix</code> - Round toward zero.</li> <li>• <code>floor</code> - Round toward negative infinity.</li> <li>• <code>nearest</code> - Round toward nearest. Ties round toward positive infinity.</li> <li>• <code>round</code> - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.</li> </ul> <p>The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.</p> |
| ScaleValueFracLength    | <p>Scale values work with SOS filters. Setting this property controls how your filter interprets the scale values by setting the fraction length. Only available when you disable <code>AutoScaleMode</code> by setting it to <code>false</code>.</p>   |
| ScaleValues             | <p>Scaling for the filter objects in SOS filters.</p>   |
| Signed                  | <p>Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.</p>  |
| SosMatrix               | <p>Holds the filter coefficients as property values — you use <code>set</code> and <code>get</code> to modify them. Displays the matrix in the format [sections x coefficients/section data type]. A [15x6 <code>double</code>] SOS matrix represents a filter with 6 coefficients per section and 15 sections, using data type <code>double</code> to represent the coefficients.</p>  |
| SectionInputFracLength  | <p>Lets you set the fraction length for section inputs in SOS filters, if you set <code>SectionInputAutoScale</code> to <code>false</code>.</p>   |
| SectionInputWordLength  | <p>Lets you set the word length for section inputs in SOS filters, if you set <code>SectionInputAutoScale</code> to <code>false</code>.</p>   |
| SectionOutputFracLength | <p>Lets you set the fraction length for section outputs in SOS filters, if you set <code>SectionOutputAutoScale</code> to <code>off</code>.</p>   |
| SectionOutputWordLength | <p>Lets you set the word length for section outputs in SOS filters, if you set <code>SectionOutputAutoScale</code> to <code>false</code>.</p>   |
| StateAutoScale          | <p>Setting autoscaling for filter states to <code>true</code> reduces the possibility of overflows occurring during fixed-point operations. Set to <code>false</code>, <code>StateAutoScale</code> lets the filter select the fraction length to limit the overflow potential.</p>  |

| Property Name   | Brief Description   |
|-----------------|---|
| StateFracLength | When you set <code>StateAutoScale</code> to <code>false</code> , you enable the <code>StateFracLength</code> property that lets you set the fraction length applied to interpret the filter states. |
| States          | This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions.   |
| StateWordLength | Sets the word length used to represent the filter states.   |

## Examples

Construct a second-order section Butterworth filter for fixed-point filtering. Start by specifying a Butterworth filter, and then convert the filter to second-order sections, with the following code:

```
[z,p,k] = butter(30,0.5);  
[s,g] = zp2sos(z,p,k);  
hd = dfilt.df2tsos(s,g);  
% convert filter to fixed-point  
hd.arithmetic='fixed';
```

## Version History

Introduced in R2011a

## See Also

`dsp.BiquadFilter`

## dfilt.dfasymfir

Discrete-time, direct-form antisymmetric FIR filter

### Syntax

```
hd = dfilt.dfasymfir(b)
hd = dfilt.dfasymfir
```

### Description

`hd = dfilt.dfasymfir(b)` returns a discrete-time, direct-form, antisymmetric FIR filter object `hd`, with numerator coefficients `b`.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter  
`set(hd,'arithmetic','single');`
- To change to fixed-point filtering, enter  
`set(hd,'arithmetic','fixed');`

`hd = dfilt.dfasymfir` returns a default, discrete-time, direct-form, antisymmetric FIR filter object `hd`, with `b=1`. This filter passes the input through to the output unchanged.

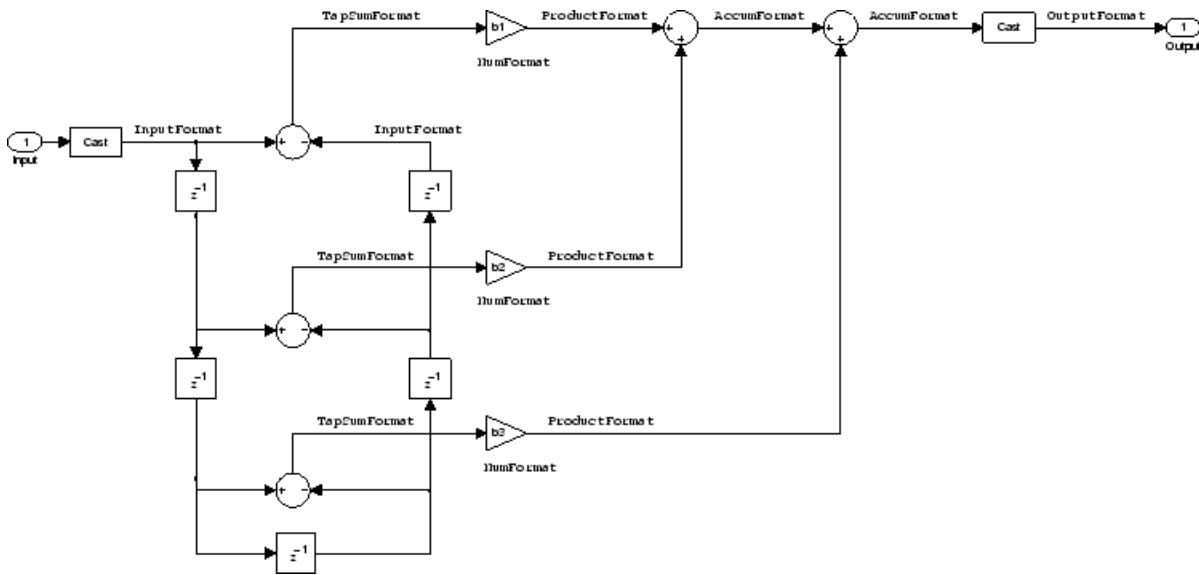
---

**Note** Only the coefficients in the first half of vector `b` are used because `dfilt.dfasymfir` assumes the coefficients in the second half are antisymmetric to those in the first half. For example, in the figure coefficients,  $b(4) = -b(3)$ ,  $b(5) = -b(2)$ , and  $b(6) = -b(1)$ .

---

### Fixed-Point Filter Structure

The following figure shows the signal flow for the odd-order antisymmetric FIR filter implemented by `dfilt.dfasymfir`. The even-order filter uses similar flow. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.



### Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the word “format.” In this use, “format” means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFormat label refers to the word length and fraction length used to interpret the data input to the filter. The format properties InputWordLength and InputFracLength (as shown in the table) store the word length and the fraction length in bits. Or consider NumFormat, which refers to the word and fraction lengths (CoeffWordLength, NumFracLength) associated with representing filter numerator coefficients.

| Signal Flow Label | Corresponding Word Length Property | Corresponding Fraction Length Property | Related Properties                  |
|-------------------|------------------------------------|--|-------------------------------------|
| AccumFormat       | AccumWordLength                    | AccumFracLength                        | None                                |
| InputFormat       | InputWordLength                    | InputFracLength                        | None                                |
| NumFormat         | CoeffWordLength                    | NumFracLength                          | CoeffAutoScale, , Signed, Numerator |
| OutputFormat      | OutputWordLength                   | OutputFracLength                       | None                                |
| ProductFormat     | ProductWordLength                  | ProductFracLength                      | None                                |
| TapSumFormat      | InputWordLength                    | InputFracLength                        | InputFormat                         |

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label ProductFormat, which always follows a coefficient multiplication element in the signal flow. The label indicates that coefficients leave the multiplication element with the word length and fraction length associated with product operations that include coefficients. From reviewing the table, you see that the ProductFormat refers to the properties

ProductFracLength and ProductWordLength that fully define the coefficient format after multiply (or product) operations.

## Properties

In this table you see the properties associated with an antisymmetric FIR implementation of `dfilt` objects.

**Note** The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

```
get(hd)
```

where `hd` is a filter.

| Name            | Values   | Description   |
|-----------------|--|---|
| AccumFracLength | Any positive or negative integer number of bits [27] | Specifies the fraction length used to interpret data output by the accumulator.   |
| AccumWordLength | Any integer number of bits[33]                       | Sets the word length used to store data in the accumulator.   |
| Arithmetic      | fixed for fixed-point filters                        | Setting this to <code>fixed</code> allows you to modify other filter properties to customize your fixed-point filter.   |
| CoeffAutoScale  | [true], false  | Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to <code>false</code> enables you to change the <code>NumFracLength</code> property value to specify the precision used.  |
| CoeffWordLength | Any integer number of bits [16]                      | Specifies the word length to apply to filter coefficients.  |
| FilterInternals | [FullPrecision], SpecifyPrecision                    | Controls whether the filter automatically sets the output word and fraction lengths, product word and fraction lengths, and the accumulator word and fraction lengths to maintain the best precision results during filtering. The default value, <code>FullPrecision</code> , sets automatic word and fraction length determination by the filter. <code>SpecifyPrecision</code> makes the output and accumulator-related properties available so you can set your own word and fraction lengths for them. |
| InputFracLength | Any positive or negative integer number of bits [15] | Specifies the fraction length the filter uses to interpret input data. Also controls <code>TapSumFracLength</code> .  |
| InputWordLength | Any integer number of bits [16]                      | Specifies the word length applied to interpret input data. Also determines <code>TapSumWordLength</code> .  |

| Name              | Values   | Description  |
|-------------------|--|--|
| NumFracLength     | Any positive or negative integer number of bits [14] | Sets the fraction length used to interpret the numerator coefficients.   |
| OutputFracLength  | Any positive or negative integer number of bits [29] | Determines how the filter interprets the filter output data. You can change the value of <code>OutputFracLength</code> when you set <code>FilterInternals</code> to <code>SpecifyPrecision</code> .  |
| OutputWordLength  | Any integer number of bits [33]                      | Determines the word length used for the output data. You make this property editable by setting <code>FilterInternals</code> to <code>SpecifyPrecision</code> .  |
| OverflowMode      | saturate, [wrap]                                     | Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either <code>saturate</code> (limit the output to the largest positive or negative representable value) or <code>wrap</code> (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision. |
| ProductFracLength | Any positive or negative integer number of bits [27] | Specifies the fraction length to use for multiplication operation results. This property becomes writable (you can change the value) when you set <code>ProductMode</code> to <code>SpecifyPrecision</code> .  |
| ProductWordLength | Any integer number of bits [33]                      | Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set <code>ProductMode</code> to <code>SpecifyPrecision</code> .  |



| Name      | Values   | Description   |
|-----------|--|---|
| RoundMode | [convergent], ceil, fix, floor, nearest, round   | <p>Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths).</p> <ul style="list-style-type: none"> <li>• <b>ceil</b> - Round toward positive infinity.</li> <li>• <b>convergent</b> - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.</li> <li>• <b>fix</b> - Round toward zero.</li> <li>• <b>floor</b> - Round toward negative infinity.</li> <li>• <b>nearest</b> - Round toward nearest. Ties round toward positive infinity.</li> <li>• <b>round</b> - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.</li> </ul> <p>The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.</p> |
| Signed    | [true], false                                    | Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.   |
| States    | fi object to match the filter arithmetic setting | Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use fi objects, with the associated properties from those objects. For details, refer to fixed-point objects in Fixed-Point Designer documentation.  |

## Examples

Specify a fifth-order direct-form antisymmetric FIR filter structure for a `dfilt` object, `hd`, with the following code:

```
b = [-0.008 0.06 -0.44 0.44 -0.06 0.008];
hd = dfilt.dfasymfir(b);
```

## Version History

Introduced in R2011a

## See Also

`dsp.FIRFilter`

## dfilt.dffir

Discrete-time, direct-form FIR filter

### Syntax

```
hd = dfilt.dffir(b)
hd = dfilt.dffir
```

### Description

`hd = dfilt.dffir(b)` returns a discrete-time, direct-form finite impulse response (FIR) filter object `hd`, with numerator coefficients `b`.

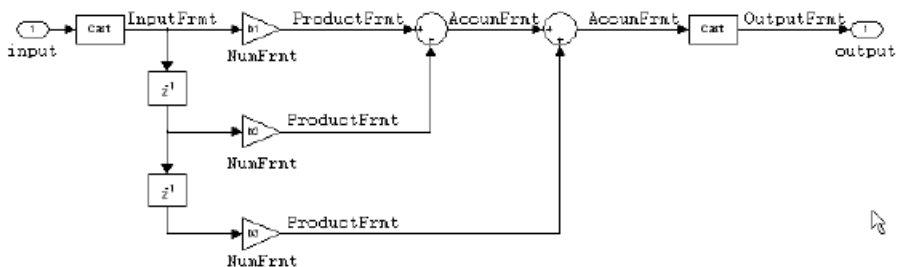
Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter  
`set(hd,'arithmetic','single');`
- To change to fixed-point filtering, enter  
`set(hd,'arithmetic','fixed');`

`hd = dfilt.dffir` returns a default, discrete-time, direct-form FIR filter object `hd`, with `b=1`. This filter passes the input through to the output unchanged.

### Fixed-Point Filter Structure

The following figure shows the signal flow for the direct-form FIR filter implemented by `dfilt.dffir`. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.



### Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following

table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the letters “frmt” (format). In this use, “frmt” indicates the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFrmt label refers to the word length and fraction length used to interpret the data input to the filter. The format properties InputWordLength and InputFracLength (as shown in the table) store the word length and the fraction length in bits. Or consider NumFrmt, which refers to the word and fraction lengths (CoeffWordLength, NumFracLength) associated with representing filter numerator coefficients.

| Signal Flow Label | Corresponding Word Length Property | Corresponding Fraction Length Property | Related Properties                |
|-------------------|------------------------------------|--|-----------------------------------|
| AccumFrmt         | AccumWordLength                    | AccumFracLength                        | None                              |
| InputFrmt         | InputWordLength                    | InputFracLength                        | None                              |
| NumFrmt           | CoeffWordLength                    | NumFracLength                          | CoeffAutoScale, Signed, Numerator |
| OutputFrmt        | OutputWordLength                   | OutputFracLength                       | None                              |
| ProductFrmt       | ProductWordLength                  | ProductFracLength                      | None                              |

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label ProductFrmt, which always follows a coefficient multiplication element in the signal flow. The label indicates that coefficients leave the multiplication element with the word length and fraction length associated with product operations that include coefficients. From reviewing the table, you see that the ProductFrmt refers to the properties ProductFracLength and ProductWordLength that fully define the coefficient format after multiply (or product) operations.

## Properties

In this table you see the properties associated with direct-form FIR implementation of `dfilt` objects.

**Note** The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

```
get(hd)
```

where `hd` is a filter.

| Name            | Values   | Description   |
|-----------------|--|---|
| AccumFracLength | Any positive or negative integer number of bits [30] | Specifies the fraction length used to interpret data output by the accumulator. |
| AccumWordLength | Any integer number of bits[34]                       | Sets the word length used to store data in the accumulator.                     |

| Name              | Values  | Description   |
|-------------------|---|---|
| Arithmetic        | fixed for fixed-point filters                                 | Setting this to <code>fixed</code> allows you to modify other filter properties to customize your fixed-point filter.   |
| CoeffAutoScale    | [ <code>true</code> ], <code>false</code>                     | Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to <code>false</code> enables you to change the <code>NumFracLength</code> property value to specify the precision used.  |
| CoeffWordLength   | Any integer number of bits [16]                               | Specifies the word length to apply to filter coefficients.  |
| FilterInternals   | [ <code>FullPrecision</code> ], <code>SpecifyPrecision</code> | Controls whether the filter automatically sets the output word and fraction lengths, product word and fraction lengths, and the accumulator word and fraction lengths to maintain the best precision results during filtering. The default value, <code>FullPrecision</code> , sets automatic word and fraction length determination by the filter. <code>SpecifyPrecision</code> makes the output and accumulator-related properties available so you can set your own word and fraction lengths for them. |
| InputFracLength   | Any positive or negative integer number of bits [15]          | Specifies the fraction length the filter uses to interpret input data.  |
| InputWordLength   | Any integer number of bits [16]                               | Specifies the word length applied to interpret input data.  |
| NumFracLength     | Any positive or negative integer number of bits [14]          | Sets the fraction length used to interpret the numerator coefficients.  |
| OutputFracLength  | Any positive or negative integer number of bits [32]          | Determines how the filter interprets the filter output data. You can change the value of <code>OutputFracLength</code> when you set <code>FilterInternals</code> to <code>SpecifyPrecision</code> .   |
| OutputWordLength  | Any integer number of bits [39]                               | Determines the word length used for the output data. You make this property editable by setting <code>FilterInternals</code> to <code>SpecifyPrecision</code> .   |
| OverflowMode      | <code>saturate</code> , [ <code>wrap</code> ]                 | Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either <code>saturate</code> (limit the output to the largest positive or negative representable value) or <code>wrap</code> (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.  |
| ProductFracLength | Any positive or negative integer number of bits [30]          | Specifies the fraction length to use for multiplication operation results. This property becomes writable (you can change the value) when you set <code>ProductMode</code> to <code>SpecifyPrecision</code> .   |

| Name              | Values   | Description   |
|-------------------|--|---|
| ProductWordLength | Any integer number of bits [32]                  | Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set ProductMode to SpecifyPrecision.  |
| RoundMode         | [convergent], ceil, fix, floor, nearest, round   | <p>Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths).</p> <ul style="list-style-type: none"> <li>• <code>ceil</code> - Round toward positive infinity.</li> <li>• <code>convergent</code> - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.</li> <li>• <code>fix</code> - Round toward zero.</li> <li>• <code>floor</code> - Round toward negative infinity.</li> <li>• <code>nearest</code> - Round toward nearest. Ties round toward positive infinity.</li> <li>• <code>round</code> - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.</li> </ul> <p>The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.</p> |
| Signed            | [true], false                                    | Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.   |
| States            | fi object to match the filter arithmetic setting | Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use fi objects, with the associated properties from those objects. For details, refer to fixed-point objects in Fixed-Point Designer documentation.  |

## Examples

Specify a second-order direct-form FIR filter structure for a `dfilt` object `hd`, with the following code that constructs the filter in double-precision format and then converts the filter to fixed-point operation:

```
b = [0.05 0.9 0.05];
hd = dfilt.dffir(b);
% Create fixed-point filter
hd.arithmetic='fixed';
% Change FilterInternals property to
% SpecifyPrecision enabling other properties
hd.FilterInternals='SpecifyPrecision';
```

## **Version History**

**Introduced in R2011a**

### **See Also**

`dsp.FIRFilter`

## dfilt.dffirt

Discrete-time, direct-form FIR transposed filter

### Syntax

```
hd = dfilt.dffirt(b)
hd = dfilt.dffirt
```

### Description

`hd = dfilt.dffirt(b)` returns a discrete-time, direct-form FIR transposed filter object `hd`, with numerator coefficients `b`.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

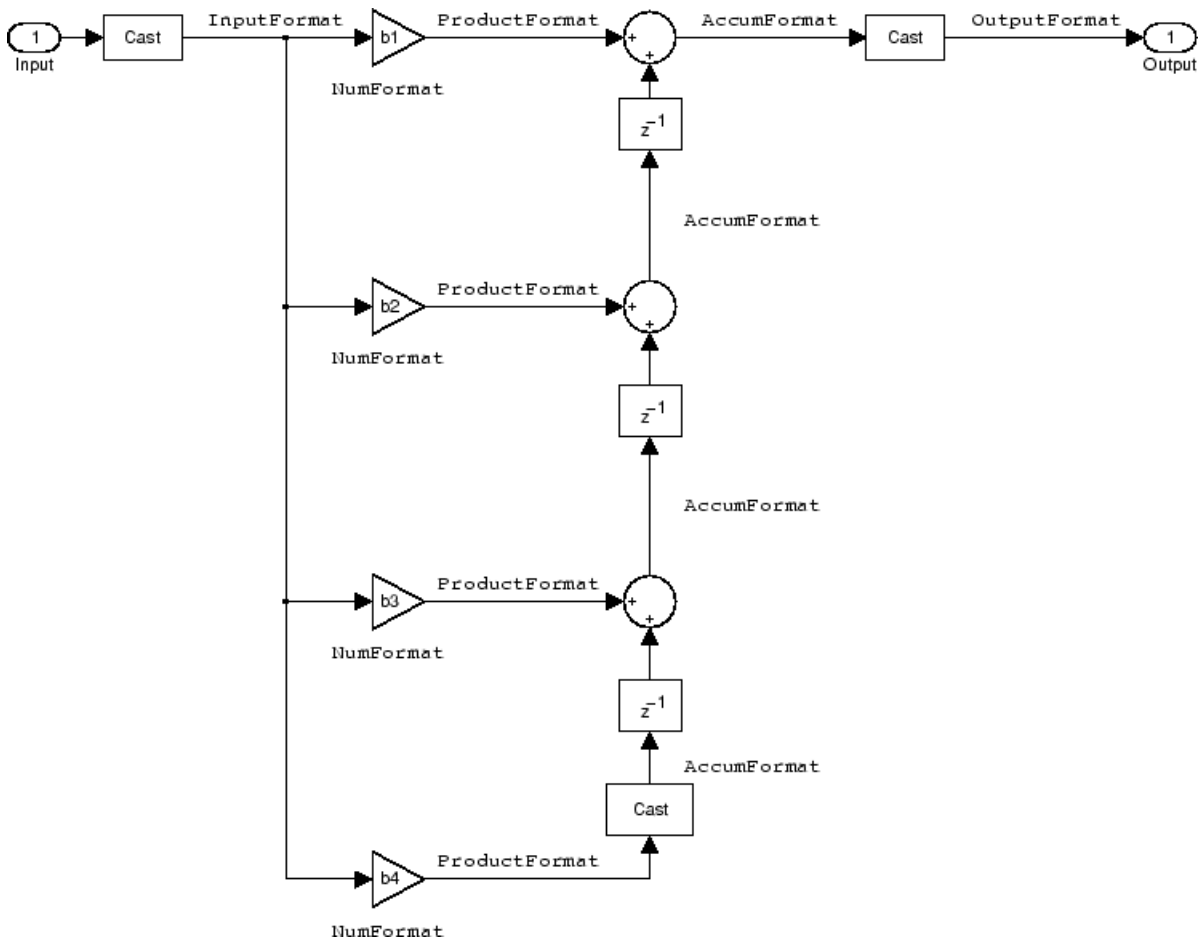
```
set(hd,'arithmetic','single');
```
- To change to fixed-point filtering, enter

```
set(hd,'arithmetic','fixed');
```

`hd = dfilt.dffirt` returns a default, discrete-time, direct-form FIR transposed filter object `hd`, with `b = 1`. This filter passes the input through to the output unchanged.

### Fixed-Point Filter Structure

The following figure shows the signal flow for the transposed direct-form FIR filter implemented by `dfilt.dffirt`. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.



**Notes About the Signal Flow Diagram**

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the word “format.” In this use, “format” means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFormat label refers to the word length and fraction length used to interpret the data input to the filter. The format properties InputWordLength and InputFracLength (as shown in the table) store the word length and the fraction length in bits. Or consider NumFormat, which refers to the word and fraction lengths (CoeffWordLength, NumFracLength) associated with representing filter numerator coefficients.

| Signal Flow Label | Corresponding Word Length Property | Corresponding Fraction Length Property | Related Properties |
|-------------------|------------------------------------|--|--------------------|
| AccumFormat       | AccumWordLength                    | AccumFracLength                        | None               |
| InputFormat       | InputWordLength                    | InputFracLength                        | None               |



| Signal Flow Label | Corresponding Word Length Property | Corresponding Fraction Length Property | Related Properties                |
|-------------------|------------------------------------|--|-----------------------------------|
| NumFormat         | CoeffWordLength                    | NumFracLength                          | CoeffAutoScale, Signed, Numerator |
| OutputFormat      | OutputWordLength                   | OutputFracLength                       | None                              |
| ProductFormat     | ProductWordLength                  | ProductFracLength                      | None                              |

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label ProductFormat, which always follows a coefficient multiplication element in the signal flow. The label indicates that coefficients leave the multiplication element with the word length and fraction length associated with product operations that include coefficients. From reviewing the table, you see that the ProductFormat refers to the properties ProductFracLength and ProductWordLength that fully define the coefficient format after multiply (or product) operations.

## Properties

In this table you see the properties associated with the transposed direct-form FIR implementation of `dfilt` objects.

**Note** The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

```
get(hd)
```

where `hd` is a filter.

| Name            | Values   | Description  |
|-----------------|--|--|
| AccumFracLength | Any positive or negative integer number of bits [30] | Specifies the fraction length used to interpret data output by the accumulator.  |
| AccumWordLength | Any integer number of bits [34]                      | Sets the word length used to store data in the accumulator.  |
| Arithmetic      | fixed for fixed-point filters                        | Setting this to <code>fixed</code> allows you to modify other filter properties to customize your fixed-point filter.  |
| CoeffAutoScale  | [true], false  | Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to <code>false</code> enables you to change the <code>NumFracLength</code> property value to specify the precision used. |
| CoeffWordLength | Any integer number of bits [16]                      | Specifies the word length to apply to filter coefficients.   |

| Name             | Values   | Description   |
|------------------|--|---|
| FilterInternals  | [FullPrecision],<br>SpecifyPrecision                 | Controls whether the filter automatically sets the output word and fraction lengths, product word and fraction lengths, and the accumulator word and fraction lengths to maintain the best precision results during filtering. The default value, <code>FullPrecision</code> , sets automatic word and fraction length determination by the filter. <code>SpecifyPrecision</code> makes the output and accumulator-related properties available so you can set your own word and fraction lengths for them. |
| InputFracLength  | Any positive or negative integer number of bits [15] | Specifies the fraction length the filter uses to interpret input data.  |
| InputWordLength  | Any integer number of bits [16]                      | Specifies the word length applied to interpret input data.  |
| NumFracLength    | Any positive or negative integer number of bits [14] | Sets the fraction length used to interpret the numerator coefficients.  |
| OutputFracLength | Any positive or negative integer number of bits [30] | Determines how the filter interprets the filter output data. You can change the value of <code>OutputFracLength</code> when you set <code>FilterInternals</code> to <code>SpecifyPrecision</code> .   |
| OutputWordLength | Any integer number of bits [34]                      | Determines the word length used for the output data. You make this property editable by setting <code>FilterInternals</code> to <code>SpecifyPrecision</code> .   |
| OverflowMode     | saturate, [wrap]                                     | Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either <code>saturate</code> (limit the output to the largest positive or negative representable value) or <code>wrap</code> (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow—they maintain full precision.    |

| Name      | Values  | Description   |
|-----------|---|---|
| RoundMode | [convergent], ceil, fix, floor, nearest, round                | <p>Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths).</p> <ul style="list-style-type: none"> <li>• <code>ceil</code> - Round toward positive infinity.</li> <li>• <code>convergent</code> - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.</li> <li>• <code>fix</code> - Round toward zero.</li> <li>• <code>floor</code> - Round toward negative infinity.</li> <li>• <code>nearest</code> - Round toward nearest. Ties round toward positive infinity.</li> <li>• <code>round</code> - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.</li> </ul> <p>The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.</p> |
| Signed    | [true], false   | Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.   |
| States    | <code>fi</code> object to match the filter arithmetic setting | Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use <code>fi</code> objects, with the associated properties from those objects. For details, refer to fixed-point objects in Fixed-Point Designer documentation.   |

## Examples

Specify a second-order direct-form FIR transposed filter structure for a `dfilt` object, `hd`, with the following code:

```
b = [0.05 0.9 0.05];
hd = dfilt.dffirt(b);
set(hd, 'arithmetic', 'fixed')
```

## Version History

Introduced in R2011a

## See Also

`dsp.FIRFilter`

## dfilt.dfsymfir

Discrete-time, direct-form symmetric FIR filter

### Syntax

```
hd = dfilt.dfsymfir(b)
hd = dfilt.dfsymfir
```

### Description

`hd = dfilt.dfsymfir(b)` returns a discrete-time, direct-form symmetric FIR filter object `hd`, with numerator coefficients `b`.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

```
set(hd,'arithmetic','single');
```
- To change to fixed-point filtering, enter

```
set(hd,'arithmetic','fixed');
```

`hd = dfilt.dfsymfir` returns a default, discrete-time, direct-form symmetric FIR filter object `hd`, with `b=1`. This filter passes the input through to the output unchanged.

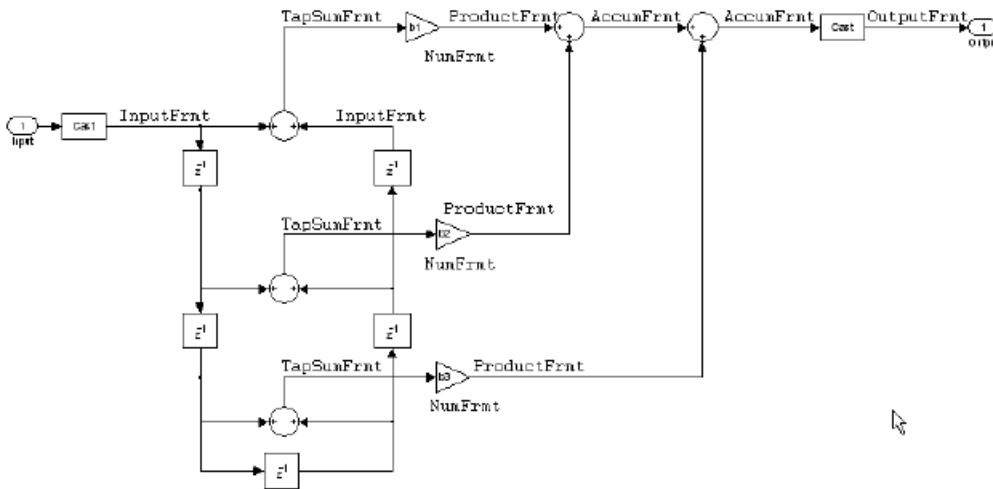
---

**Note** Only the coefficients in the first half of vector `b` are used because `dfilt.dfsymfir` assumes the coefficients in the second half are symmetric to those in the first half. In the following figure, for example,  $b(3) = 0$ ,  $b(4) = b(2)$  and  $b(5) = b(1)$ .

---

### Fixed-Point Filter Structure

In the following figure you see the signal flow diagram for the symmetric FIR filter that `dfilt.dfsymfir` implements.



### Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the letters “frmt” (format). In this use, “frmt” indicates the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFrmt label refers to the word length and fraction length used to interpret the data input to the filter. The format properties InputWordLength and InputFracLength (as shown in the table) store the word length and the fraction length in bits. Or consider NumFrmt, which refers to the word and fraction lengths (CoeffWordLength, NumFracLength) associated with representing filter numerator coefficients.

| Signal Flow Label | Corresponding Word Length Property | Corresponding Fraction Length Property | Related Properties                |
|-------------------|------------------------------------|--|-----------------------------------|
| AccumFrmt         | AccumWordLength                    | AccumFracLength                        | None                              |
| InputFrmt         | InputWordLength                    | InputFracLength                        | None                              |
| NumFrmt           | CoeffWordLength                    | NumFracLength                          | CoeffAutoScale, Signed, Numerator |
| OutputFrmt        | OutputWordLength                   | OutputFracLength                       | None                              |
| ProductFrmt       | ProductWordLength                  | ProductFracLength                      | None                              |
| TapSumFrmt        | InputWordLength                    | InputFracLength                        | None                              |

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label ProductFrmt, which always follows a coefficient multiplication element in the signal flow. The label indicates that coefficients leave the multiplication element with the word length and fraction length associated with product operations that include coefficients. From reviewing the table, you see that the ProductFrmt refers to the properties

ProductFracLength and ProductWordLength that fully define the coefficient format after multiply (or product) operations.

## Properties

In this table you see the properties associated with the symmetric FIR implementation of `dfilt` objects.

**Note** The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

```
get(hd)
```

where `hd` is a filter.

| Name            | Values   | Description   |
|-----------------|--|---|
| AccumFracLength | Any positive or negative integer number of bits [27] | Specifies the fraction length used to interpret data output by the accumulator.   |
| AccumWordLength | Any integer number of bits[33]                       | Sets the word length used to store data in the accumulator.   |
| Arithmetic      | fixed for fixed-point filters                        | Setting this to <code>fixed</code> allows you to modify other filter properties to customize your fixed-point filter.   |
| CoeffAutoScale  | [true], false  | Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to <code>false</code> enables you to change the <code>NumFracLength</code> property value to specify the precision used.  |
| CoeffWordLength | Any integer number of bits [16]                      | Specifies the word length to apply to filter coefficients.  |
| FilterInternals | [FullPrecision], SpecifyPrecision                    | Controls whether the filter automatically sets the output word and fraction lengths, product word and fraction lengths, and the accumulator word and fraction lengths to maintain the best precision results during filtering. The default value, <code>FullPrecision</code> , sets automatic word and fraction length determination by the filter. <code>SpecifyPrecision</code> makes the output and accumulator-related properties available so you can set your own word and fraction lengths for them. |
| InputFracLength | Any positive or negative integer number of bits [15] | Specifies the fraction length the filter uses to interpret input data.  |
| InputWordLength | Any integer number of bits [16]                      | Specifies the word length applied to interpret input data.  |

| Name              | Values   | Description  |
|-------------------|--|--|
| NumFracLength     | Any positive or negative integer number of bits [14] | Sets the fraction length used to interpret the numerator coefficients.   |
| OutputFracLength  | Any positive or negative integer number of bits [29] | Determines how the filter interprets the filter output data. You can change the value of <code>OutputFracLength</code> when you set <code>FilterInternals</code> to <code>SpecifyPrecision</code> .  |
| OutputWordLength  | Any integer number of bits [33]                      | Determines the word length used for the output data. You make this property editable by setting <code>FilterInternals</code> to <code>SpecifyPrecision</code> .  |
| OverflowMode      | saturate, [wrap]                                     | Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either <code>saturate</code> (limit the output to the largest positive or negative representable value) or <code>wrap</code> (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow—they maintain full precision. |
| ProductFracLength | Any positive or negative integer number of bits [29] | Specifies the fraction length to use for multiplication operation results. This property becomes writable (you can change the value) when you set <code>ProductMode</code> to <code>SpecifyPrecision</code> .  |
| ProductWordLength | Any integer number of bits [33]                      | Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set <code>ProductMode</code> to <code>SpecifyPrecision</code> .  |

| Name      | Values  | Description   |
|-----------|---|---|
| RoundMode | [convergent], ceil, fix, floor, nearest, round                | <p>Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths).</p> <ul style="list-style-type: none"> <li>• <code>ceil</code> - Round toward positive infinity.</li> <li>• <code>convergent</code> - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.</li> <li>• <code>fix</code> - Round toward zero.</li> <li>• <code>floor</code> - Round toward negative infinity.</li> <li>• <code>nearest</code> - Round toward nearest. Ties round toward positive infinity.</li> <li>• <code>round</code> - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.</li> </ul> <p>The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.</p> |
| Signed    | [true], false   | Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.   |
| States    | <code>fi</code> object to match the filter arithmetic setting | Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use <code>fi</code> objects, with the associated properties from those objects. For details, refer to fixed-point objects in Fixed-Point Designer documentation.   |

## Examples

Specify a fifth-order direct-form symmetric FIR filter structure for a `dfilt` object `hd`, with the following code:

```
b = [-0.008 0.06 0.44 0.44 0.06 -0.008];
hd = dfilt.dfsymfir(b);
% Create fixed-point filter
set(hd,'arithmetic','fixed')
% Change FilterInternals property to
% SpecifyPrecision
hd.FilterInternals='SpecifyPrecision';
```

## Version History

Introduced in R2011a



**See Also**

`dsp.FIRFilter`

## dfilt.farrowfd

Fractional Delay Farrow filter

### Syntax

```
Hd = dfilt.farrowfd(D, COEFFS)
```

### Description

`Hd = dfilt.farrowfd(D, COEFFS)` Constructs a discrete-time fractional delay Farrow filter with `COEFFS` coefficients and `D` delay.

### Examples

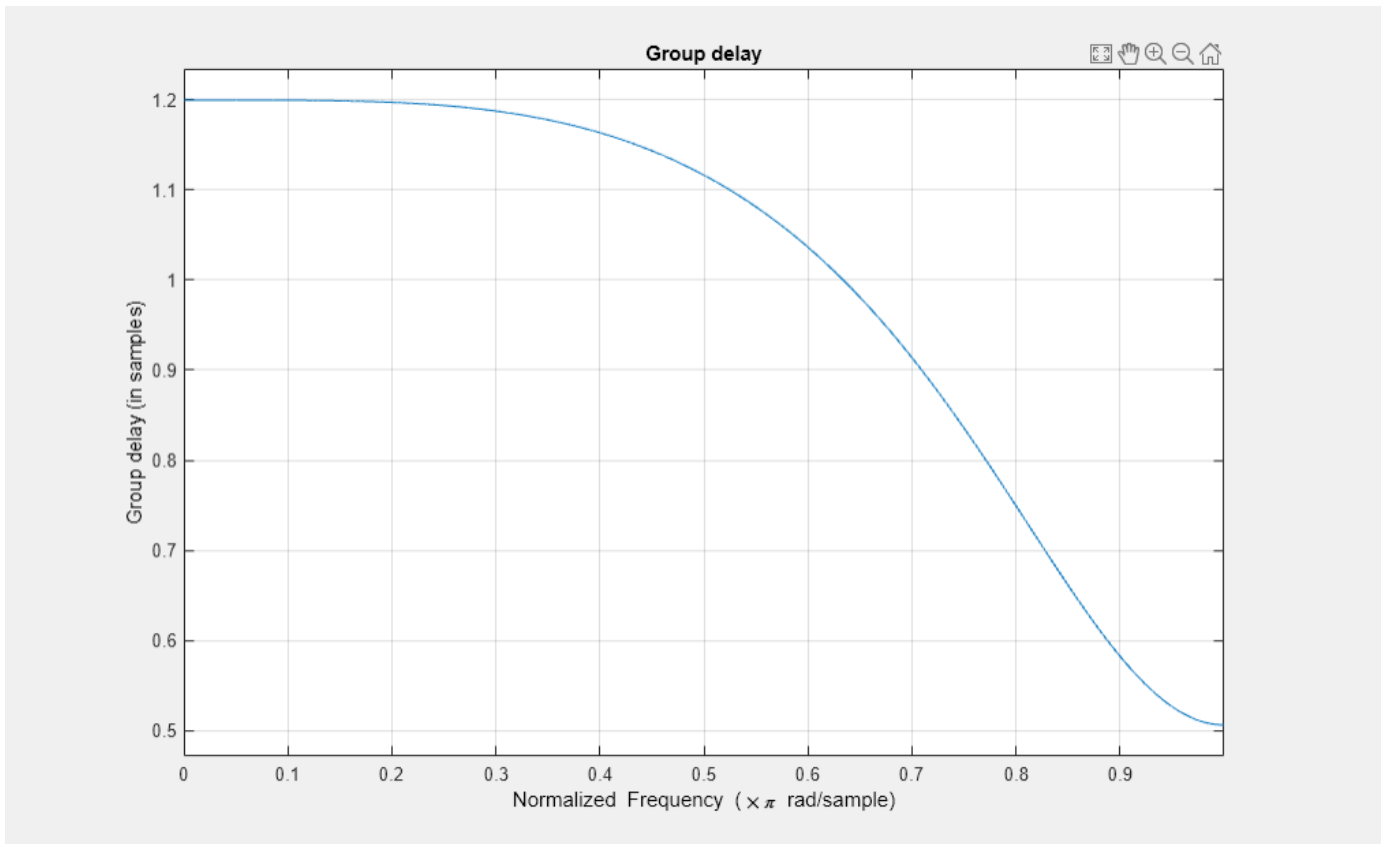
#### Design a Fractional Delay Farrow Filter

Farrow filters can be designed with the `dfilt.farrowfd` filter designer.

```
coeffs = [-1/6 1/2 -1/3 0;1/2 -1 -1/2 1;  
-1/2 1/2 1 0;1/6 0 -1/6 0];  
farrow = dfilt.farrowfd(0.5, coeffs);
```

Design a cubic fractional delay filter with the Lagrange method.

```
fdelay = .2; % Fractional delay  
d = fdesign.fracdelay(fdelay,'N',3);  
cubicfarrow = design(d, 'lagrange', 'FilterStructure', 'farrowfd');  
fvtool(cubicfarrow, 'Analysis', 'grpdelay');
```



## Version History

Introduced in R2011a

## dfilt.farrowlinearfd

Farrow Linear Fractional Delay filter

### Syntax

```
Hd = dfilt.farrowlinearfd(D)
```

### Description

`Hd = dfilt.farrowlinearfd(D)` Constructs a discrete-time linear fractional delay Farrow filter with the delay `D`.

### Examples

Farrow linear fractional delay filter with 1/2 sample delay:

```
Hd = dfilt.farrowlinearfd(0.5);  
x = cos(pi/10*(0:159));  
y = filter(Hd,x);  
stem(x(1:40));  
axis([0 40 -2 2]);  
hold on;  
stem(y(1:40), 'color',[1 0 0], 'markerfacecolor',[1 0 0]);  
legend('Original Signal', 'Filtered Signal', 'Location', 'best');
```

For more information about fractional delay filter implementations, see “Fractional Delay Filters Using Farrow Structures”.

## Version History

**Introduced in R2011a**

## dfilt.fftfir

Discrete-time, overlap-add, FIR filter

### Syntax

```
Hd = dfilt.fftfir(b, len)
Hd = dfilt.fftfir(b)
Hd = dfilt.fftfir
```

### Description

This object uses the overlap-add method of block FIR filtering, which is very efficient for streaming data.

`Hd = dfilt.fftfir(b, len)` returns a discrete-time, FFT, FIR filter, `Hd`, with numerator coefficients, `b` and block length, `len`. The block length is the number of input points to use for each overlap-add computation.

`Hd = dfilt.fftfir(b)` returns a discrete-time, FFT, FIR filter, `Hd`, with numerator coefficients, `b` and block length, `len=100`.

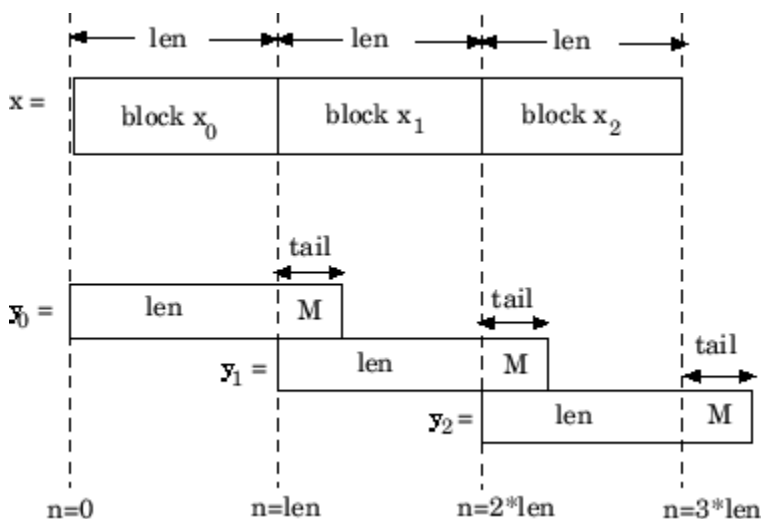
`Hd = dfilt.fftfir` returns a default, discrete-time, FFT, FIR filter, `Hd`, with the numerator `b=1` and block length, `len=100`. This filter passes the input through to the output unchanged.

---

**Note** When you use a `dfilt.fftfir` object to filter, the input signal length must be an integer multiple of the object's block length, `len`. The resulting number of FFT points = (filter length + the block length - 1). The filter is most efficient if the number of FFT points is a power of 2.

---

The `fftfir` uses an overlap-add block processing algorithm, which is represented as follows,



where  $len$  is the block length and  $M$  is the length of the numerator-1,  $(length(b) - 1)$ , which is also the number of states. The output of each convolution is a block that is longer than the input block by

a tail of  $(\text{length}(b) - 1)$  samples. These tails overlap the next block and are added to it. The states reported by `dfilt.fftfilter` are the tails of the final convolution.

## Examples

Create an FFT FIR discrete-time filter with coefficients from a 30<sup>th</sup> order lowpass equiripple design:

```
b = firpm(30,[0 .1 .2 .5]*2,[1 1 0 0]);  
Hd = dfilt.fftfilter(b);  
% To obtain frequency domain coefficients  
% used in filtering  
Coeffs = fftcoeffs(Hd);
```

## Version History

Introduced in R2011a

# dfilt.latticeallpass

Discrete-time, lattice allpass filter

## Syntax

```
hd = dfilt.latticeallpass(k)
hd = dfilt.latticeallpass
```

## Description

`hd = dfilt.latticeallpass(k)` returns a discrete-time, lattice allpass filter object `hd`, with lattice coefficients, `k`.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter
 

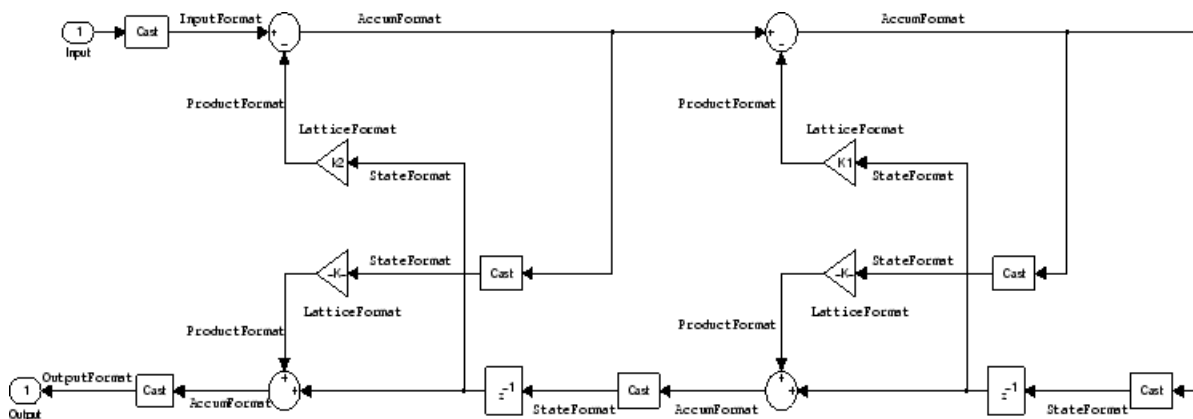
```
set(hd,'arithmetic','single');
```
- To change to fixed-point filtering, enter
 

```
set(hd,'arithmetic','fixed');
```

`hd = dfilt.latticeallpass` returns a default, discrete-time, lattice allpass filter object `hd`, with `k=[ ]`. This filter passes the input through to the output unchanged.

## Fixed-Point Filter Structure

The following figure shows the signal flow for the allpass lattice filter implemented by `dfilt.latticeallpass`. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.



## Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the word “format.” In this use, “format” means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFormat label refers to the word length and fraction length used to interpret the data input to the filter. The format properties InputWordLength and InputFracLength (as shown in the table) store the word length and the fraction length in bits. Or consider NumFormat, which refers to the word and fraction lengths (CoeffWordLength, NumFracLength) associated with representing filter numerator coefficients.

| Signal Flow Label | Corresponding Word Length Property | Corresponding Fraction Length Property | Related Properties |
|-------------------|------------------------------------|--|--------------------|
| AccumFormat       | AccumWordLength                    | AccumFracLength                        | AccumMode          |
| InputFormat       | InputWordLength                    | InputFracLength                        | None               |
| LatticeFormat     | CoeffWordLength                    | LatticeFracLength                      | CoeffAutoScale     |
| OutputFormat      | OutputWordLength                   | OutputFracLength                       | OutputMode         |
| ProductFormat     | ProductWordLength                  | ProductFracLength                      | ProductMode        |
| StateFormat       | StateWordLength                    | StateFracLength                        | States             |

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label ProductFormat, which always follows a coefficient multiplication element in the signal flow. The label indicates that coefficients leave the multiplication element with the word length and fraction length associated with product operations that include coefficients. From reviewing the table, you see that the ProductFormat refers to the properties ProductFracLength, ProductWordLength, and ProductMode that fully define the coefficient format after multiply (or product) operations.

## Properties

In this table you see the properties associated with the allpass lattice implementation of `dfilt` objects.

**Note** The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

```
get(hd)
```

where `hd` is a filter.



| Property Name     | Brief Description  |
|-------------------|--|
| AccumFracLength   | Specifies the fraction length used to interpret data output by the accumulator. This is a property of FIR filters and lattice filters. IIR filters have two similar properties —DenAccumFracLength and NumAccumFracLength — that let you set the precision for numerator and denominator operations separately.  |
| AccumMode         | Determines how the accumulator outputs stored values. Choose from full precision (FullPrecision), or whether to keep the most significant bits (KeepMSB) or least significant bits (KeepLSB) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set AccumMode to SpecifyPrecision.  |
| AccumWordLength   | Sets the word length used to store data in the accumulator/buffer.   |
| Arithmetic        | Defines the arithmetic the filter uses. Gives you the options double, single, and fixed. In short, this property defines the operating mode for your filter.   |
| CastBeforeSum     | Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.  |
| CoeffAutoScale    | Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to false enables you to change the LatticeFracLength property value to specify the precision used.   |
| CoeffWordLength   | Specifies the word length to apply to filter coefficients.   |
| FilterStructure   | Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering — gains, delays, sums, products, and input/output.  |
| InputFracLength   | Specifies the fraction length the filter uses to interpret input data.   |
| InputWordLength   | Specifies the word length applied to interpret input data.   |
| Lattice           | Any lattice structure coefficients. No default value.  |
| LatticeFracLength | Sets the fraction length applied to the lattice coefficients.  |
| OutputFracLength  | Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set OutputMode to SpecifyPrecision.   |
| OutputMode        | Sets the mode the filter uses to scale the filtered data for output. You have the following choices: <ul style="list-style-type: none"> <li>• <b>AvoidOverflow</b> — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow.</li> <li>• <b>BestPrecision</b> — directs the filter to set the output data word length and fraction length to maximize the precision in the output data.</li> <li>• <b>SpecifyPrecision</b> — lets you set the word and fraction lengths used by the output data from filtering.</li> </ul> |

| Property Name     | Brief Description   |
|-------------------|---|
| OutputWordLength  | Determines the word length used for the output data.  |
| OverflowMode      | Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either <code>saturate</code> (limit the output to the largest positive or negative representable value) or <code>wrap</code> (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow—they maintain full precision.  |
| ProductFracLength | For the output from a product operation, this sets the fraction length used to interpret the data. This property becomes writable (you can change the value) when you set <code>ProductMode</code> to <code>SpecifyPrecision</code> .   |
| ProductMode       | Determines how the filter handles the output of product operations. Choose from full precision ( <code>FullPrecision</code> ), or whether to keep the most significant bit ( <code>KeepMSB</code> ) or least significant bit ( <code>KeepLSB</code> ) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set <code>ProductMode</code> to <code>SpecifyPrecision</code> .  |
| ProductWordLength | Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set <code>ProductMode</code> to <code>SpecifyPrecision</code> .   |
| PersistentMemory  | Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. <code>False</code> is the default setting.  |
| RoundMode         | <p>Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths).</p> <ul style="list-style-type: none"> <li>• <code>ceil</code> - Round toward positive infinity.</li> <li>• <code>convergent</code> - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.</li> <li>• <code>fix</code> - Round toward zero.</li> <li>• <code>floor</code> - Round toward negative infinity.</li> <li>• <code>nearest</code> - Round toward nearest. Ties round toward positive infinity.</li> <li>• <code>round</code> - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.</li> </ul> <p>The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.</p> |
| Signed            | Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.   |

| Property Name   | Brief Description  |
|-----------------|--|
| StateFracLength | When you set StateAutoScale to false, you enable the StateFracLength property that lets you set the fraction length applied to interpret the filter states.  |
| States          | This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use fi objects, with the associated properties from those objects. For details, refer to filtstates in Signal Processing Toolbox documentation or in the Help system. |
| StateWordLength | Sets the word length used to represent the filter states.  |

## Examples

Specify a third-order lattice allpass filter structure for a dfilt object hd, with the following code:

```
k = [.66 .7 .44];
hd=dfilt.latticeallpass(k);
% convert to fixed-point arithmetic
hd.arithmetic = 'fixed';
```

## Version History

Introduced in R2011a

## dfilt.latticear

Discrete-time, lattice, autoregressive filter

### Syntax

```
hd = dfilt.latticear(k)
hd = dfilt.latticear
```

### Description

`hd = dfilt.latticear(k)` returns a discrete-time, lattice autoregressive filter object `hd`, with lattice coefficients, `k`.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter  

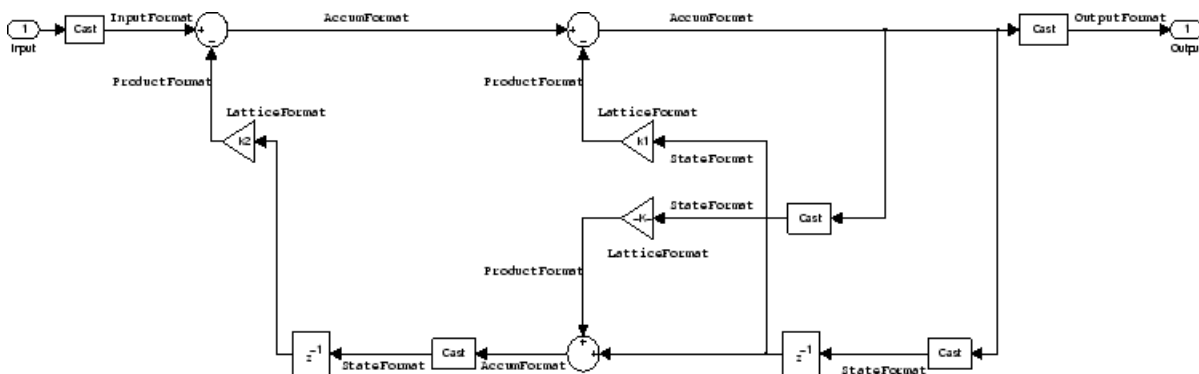
```
set(hd,'arithmetic','single');
```
- To change to fixed-point filtering, enter  

```
set(hd,'arithmetic','fixed');
```

`hd = dfilt.latticear` returns a default, discrete-time, lattice autoregressive filter object `hd`, with `k=[]`. This filter passes the input through to the output unchanged.

### Fixed-Point Filter Structure

The following figure shows the signal flow for the autoregressive lattice filter implemented by `dfilt.latticear`. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.



### Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following

table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the word “format.” In this use, “format” means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFormat label refers to the word length and fraction length used to interpret the data input to the filter. The format properties InputWordLength and InputFracLength (as shown in the table) store the word length and the fraction length in bits. Or consider NumFormat, which refers to the word and fraction lengths (CoeffWordLength, NumFracLength) associated with representing filter numerator coefficients.

| Signal Flow Label | Corresponding Word Length Property | Corresponding Fraction Length Property | Related Properties |
|-------------------|------------------------------------|--|--------------------|
| AccumFormat       | AccumWordLength                    | AccumFracLength                        | AccumMode          |
| InputFormat       | InputWordLength                    | InputFracLength                        | None               |
| LatticeFormat     | CoeffWordLength                    | LatticeFracLength                      | CoeffAutoScale     |
| OutputFormat      | OutputWordLength                   | OutputFracLength                       | OutputMode         |
| ProductFormat     | ProductWordLength                  | ProductFracLength                      | ProductMode        |
| StateFormat       | StateWordLength                    | StateFracLength                        | States             |

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label ProductFormat, which always follows a coefficient multiplication element in the signal flow. The label indicates that coefficients leave the multiplication element with the word length and fraction length associated with product operations that include coefficients. From reviewing the table, you see that the ProductFormat refers to the properties ProductFracLength, ProductWordLength, and ProductMode that fully define the coefficient format after multiply (or product) operations.

## Properties

In this table you see the properties associated with the autoregressive lattice implementation of `dfilt` objects.

---

**Note** The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

```
get(hd)
```

where `hd` is a filter.

---

| Property Name     | Brief Description  |
|-------------------|--|
| AccumFracLength   | Specifies the fraction length used to interpret data output by the accumulator. This is a property of FIR filters and lattice filters. IIR filters have two similar properties — <code>DenAccumFracLength</code> and <code>NumAccumFracLength</code> — that let you set the precision for numerator and denominator operations separately.   |
| AccumMode         | Determines how the accumulator outputs stored values. Choose from full precision ( <code>FullPrecision</code> ), or whether to keep the most significant bits ( <code>KeepMSB</code> ) or least significant bits ( <code>KeepLSB</code> ) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set <code>AccumMode</code> to <code>SpecifyPrecision</code> .  |
| AccumWordLength   | Sets the word length used to store data in the accumulator/buffer.   |
| Arithmetic        | Defines the arithmetic the filter uses. Gives you the options <code>double</code> , <code>single</code> , and <code>fixed</code> . In short, this property defines the operating mode for your filter.   |
| CastBeforeSum     | Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.  |
| CoeffAutoScale    | Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to <code>false</code> enables you to change the <code>LatticeFracLength</code> to specify the precision used.  |
| CoeffWordLength   | Specifies the word length to apply to filter coefficients.   |
| FilterStructure   | Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering—gains, delays, sums, products, and input/output.  |
| InputFracLength   | Specifies the fraction length the filter uses to interpret input data.   |
| InputWordLength   | Specifies the word length applied to interpret input data.   |
| Lattice           | Any lattice structure coefficients.  |
| LatticeFracLength | Sets the fraction length applied to the lattice coefficients.  |
| OutputFracLength  | Determines how the filter interprets the filter output data. You can change the value of <code>OutputFracLength</code> when you set <code>OutputMode</code> to <code>SpecifyPrecision</code> .   |
| OutputMode        | Sets the mode the filter uses to scale the filtered data for output. You have the following choices: <ul style="list-style-type: none"> <li>• <code>AvoidOverflow</code> — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow.</li> <li>• <code>BestPrecision</code> — directs the filter to set the output data word length and fraction length to maximize the precision in the output data.</li> <li>• <code>SpecifyPrecision</code> — lets you set the word and fraction lengths used by the output data from filtering.</li> </ul> |

| Property Name     | Brief Description   |
|-------------------|---|
| OutputWordLength  | Determines the word length used for the output data.  |
| OverflowMode      | Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either <code>saturate</code> (limit the output to the largest positive or negative representable value) or <code>wrap</code> (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.  |
| ProductFracLength | For the output from a product operation, this sets the fraction length used to interpret the data. This property becomes writable (you can change the value) when you set <code>ProductMode</code> to <code>SpecifyPrecision</code> .   |
| ProductMode       | Determines how the filter handles the output of product operations. Choose from full precision ( <code>FullPrecision</code> ), or whether to keep the most significant bit ( <code>KeepMSB</code> ) or least significant bit ( <code>KeepLSB</code> ) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set <code>ProductMode</code> to <code>SpecifyPrecision</code> .  |
| ProductWordLength | Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set <code>ProductMode</code> to <code>SpecifyPrecision</code> .   |
| PersistentMemory  | Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. <code>False</code> is the default setting.  |
| RoundMode         | <p>Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths).</p> <ul style="list-style-type: none"> <li>• <code>ceil</code> - Round toward positive infinity.</li> <li>• <code>convergent</code> - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.</li> <li>• <code>fix</code> - Round toward zero.</li> <li>• <code>floor</code> - Round toward negative infinity.</li> <li>• <code>nearest</code> - Round toward nearest. Ties round toward positive infinity.</li> <li>• <code>round</code> - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.</li> </ul> <p>The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.</p> |
| Signed            | Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.   |

| Property Name   | Brief Description  |
|-----------------|--|
| StateFracLength | When you set StateAutoScale to false, you enable the StateFracLength property that lets you set the fraction length applied to interpret the filter states.  |
| States          | This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use fi objects, with the associated properties from those objects. For details, refer to filtstates in Signal Processing Toolbox documentation or in the Help system. |
| StateWordLength | Sets the word length used to represent the filter states.  |

## Examples

Specify a third-order lattice autoregressive filter structure for a `dfilt` object, `hd`, with the following code that creates a fixed-point filter.

```
k = [.66 .7 .44];  
hd1=dfilt.latticear(k);  
hd1.arithmetic='fixed';  
specifyall(hd1);
```

## Version History

Introduced in R2011a



## dfilt.latticearma

Discrete-time, lattice, autoregressive, moving-average filter

### Syntax

```
hd = dfilt.latticearma(k)
hd dfilt.latticearma
```

### Description

`hd = dfilt.latticearma(k)` returns a discrete-time, lattice moving-average autoregressive filter object `hd`, with lattice coefficients, `k`.

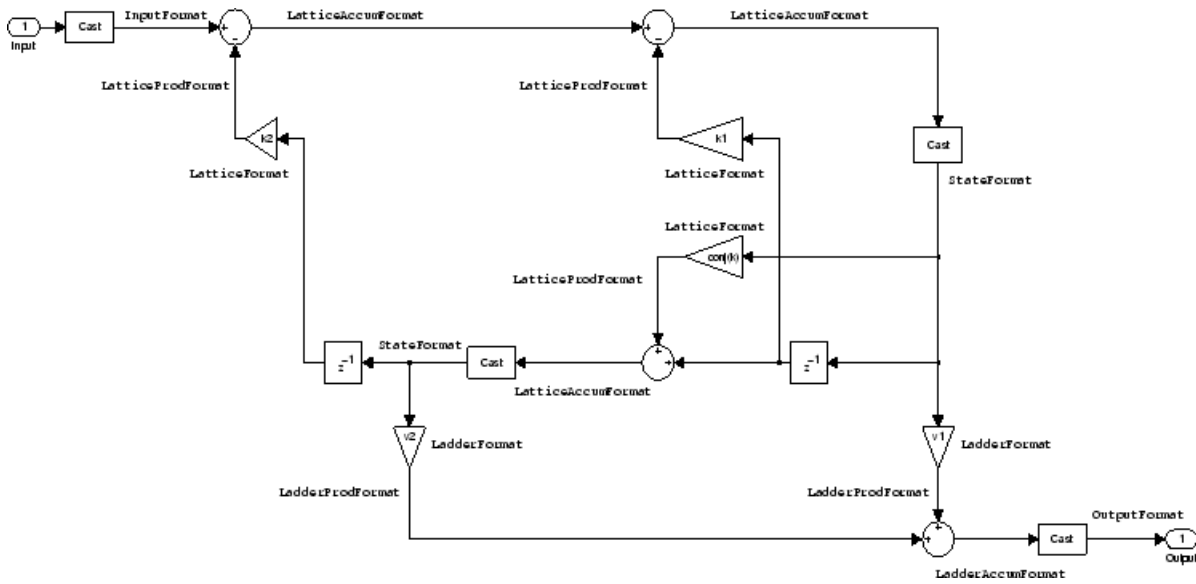
Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter  
`set(hd,'arithmetic','single');`
- To change to fixed-point filtering, enter  
`set(hd,'arithmetic','fixed');`

`hd dfilt.latticearma` returns a default, discrete-time, lattice moving-average, autoregressive filter object `hd`, with `k = [ ]`. This filter passes the input through to the output unchanged.

### Fixed-Point Filter Structure

The following figure shows the signal flow for the autoregressive lattice filter implemented by `dfilt.latticearma`. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.



**Notes About the Signal Flow Diagram**

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the word “format.” In this use, “format” means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFormat label refers to the word length and fraction length used to interpret the data input to the filter. The format properties InputWordLength and InputFracLength (as shown in the table) store the word length and the fraction length in bits. Or consider NumFormat, which refers to the word and fraction lengths (CoeffWordLength, NumFracLength) associated with representing filter numerator coefficients.

| Signal Flow Label  | Corresponding Word Length Property | Corresponding Fraction Length Property | Related Properties |
|--------------------|------------------------------------|--|--------------------|
| InputFormat        | InputWordLength                    | InputFracLength                        | None               |
| LadderAccumFormat  | AccumWordLength                    | LadderAccumFracLength                  | AccumMode          |
| LadderFormat       | CoeffWordLength                    | LadderFracLength                       | CoeffAutoScale     |
| LadderProdFormat   | ProductWordLength                  | LadderProdFracLength                   | ProductMode        |
| LatticeAccumFormat | AccumWordLength                    | LatticeAccumFracLength                 | AccumMode          |
| LatticeFormat      | CoeffWordLength                    | LatticeFracLength                      | CoeffAutoScale     |
| LatticeProdFormat  | ProductWordLength                  | LatticeProdFracLength                  | ProductMode        |
| OutputFormat       | OutputWordLength                   | OutputFracLength                       | OutputMode         |
| StateFormat        | StateWordLength                    | StateFracLength                        | States             |

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label `LatticeProdFormat`, which always follows a coefficient multiplication element in the signal flow. The label indicates that lattice coefficients leave the multiplication element with the word length and fraction length associated with product operations that include coefficients. From reviewing the table, you see that the `LatticeProdFormat` refers to the properties `ProductWordLength`, `LatticeProdFracLength`, and `ProductMode` that fully define the coefficient format after multiply (or product) operations.

## Properties

In this table you see the properties associated with the autoregressive moving-average lattice implementation of `dfilt` objects.

**Note** The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

```
get(hd)
```

where `hd` is a filter.

| Property Name                | Brief Description   |
|------------------------------|---|
| <code>AccumFracLength</code> | Specifies the fraction length used to interpret data output by the accumulator. This is a property of FIR filters and lattice filters. IIR filters have two similar properties — <code>DenAccumFracLength</code> and <code>NumAccumFracLength</code> — that let you set the precision for numerator and denominator operations separately.  |
| <code>AccumMode</code>       | Determines how the accumulator outputs stored values. Choose from full precision ( <code>FullPrecision</code> ), or whether to keep the most significant bits ( <code>KeepMSB</code> ) or least significant bits ( <code>KeepLSB</code> ) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set <code>AccumMode</code> to <code>SpecifyPrecision</code> . |
| <code>AccumWordLength</code> | Sets the word length used to store data in the accumulator/buffer.  |
| <code>Arithmetic</code>      | Defines the arithmetic the filter uses. Gives you the options <code>double</code> , <code>single</code> , and <code>fixed</code> . In short, this property defines the operating mode for your filter.  |
| <code>CastBeforeSum</code>   | Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.   |
| <code>CoeffAutoScale</code>  | Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to <code>false</code> enables you to change the <code>LatticeFracLength</code> property to specify the precision used.  |

| Property Name         | Brief Description  |
|-----------------------|--|
| CoeffWordLength       | Specifies the word length to apply to filter coefficients.   |
| FilterStructure       | Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering—gains, delays, sums, products, and input/output.  |
| InputFracLength       | Specifies the fraction length the filter uses to interpret input data.   |
| InputWordLength       | Specifies the word length applied to interpret input data.   |
| Ladder                | Stores the ladder coefficients for lattice ARMA ( <code>dfilt.latticearma</code> ) filters.  |
| LadderAccumFracLength | Sets the fraction length used to interpret the output from sum operations that include the ladder coefficients. You can change this property value after you set <code>AccumMode</code> to <code>SpecifyPrecision</code> .   |
| LadderFracLength      | Determines the precision used to represent the ladder coefficients in ARMA lattice filters.  |
| Lattice               | Stores the lattice structure coefficients.   |
| LatticeFracLength     | Sets the fraction length applied to the lattice coefficients.  |
| OutputFracLength      | Determines how the filter interprets the filter output data. You can change the value of <code>OutputFracLength</code> when you set <code>OutputMode</code> to <code>SpecifyPrecision</code> .   |
| OutputMode            | Sets the mode the filter uses to scale the filtered data for output. You have the following choices: <ul style="list-style-type: none"> <li>• <code>AvoidOverflow</code> — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow.</li> <li>• <code>BestPrecision</code> — directs the filter to set the output data word length and fraction length to maximize the precision in the output data.</li> <li>• <code>SpecifyPrecision</code> — lets you set the word and fraction lengths used by the output data from filtering.</li> </ul> |
| OutputWordLength      | Determines the word length used for the output data.   |
| OverflowMode          | Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either <code>saturate</code> (limit the output to the largest positive or negative representable value) or <code>wrap</code> (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow—they maintain full precision.   |
| ProductFracLength     | For the output from a product operation, this sets the fraction length used to interpret the data. This property becomes writable (you can change the value) when you set <code>ProductMode</code> to <code>SpecifyPrecision</code> .  |

| Property Name     | Brief Description   |
|-------------------|---|
| ProductMode       | Determines how the filter handles the output of product operations. Choose from full precision ( <code>FullPrecision</code> ), or whether to keep the most significant bit ( <code>KeepMSB</code> ) or least significant bit ( <code>KeepLSB</code> ) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set <code>ProductMode</code> to <code>SpecifyPrecision</code> .  |
| ProductWordLength | Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set <code>ProductMode</code> to <code>SpecifyPrecision</code> .   |
| PersistentMemory  | Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. <code>False</code> is the default setting.  |
| RoundMode         | <p>Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths).</p> <ul style="list-style-type: none"> <li>• <code>ceil</code> - Round toward positive infinity.</li> <li>• <code>convergent</code> - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.</li> <li>• <code>fix</code> - Round toward zero.</li> <li>• <code>floor</code> - Round toward negative infinity.</li> <li>• <code>nearest</code> - Round toward nearest. Ties round toward positive infinity.</li> <li>• <code>round</code> - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.</li> </ul> <p>The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.</p> |
| Signed            | Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.   |
| StateFracLength   | When you set <code>StateAutoScale</code> to <code>false</code> , you enable the <code>StateFracLength</code> property that lets you set the fraction length applied to interpret the filter states.   |
| States            | This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use <code>fi</code> objects, with the associated properties from those objects. For details, refer to <code>filtstates</code> in Signal Processing Toolbox documentation or in the Help system.  |
| StateWordLength   | Sets the word length used to represent the filter states.   |

# **Version History**

**Introduced in R2011a**

# dfilt.latticemamax

Discrete-time, lattice, moving-average filter with maximum phase

## Syntax

```
hd = dfilt.latticemamax(k)
hd = dfilt.latticemamax
```

## Description

`hd = dfilt.latticemamax(k)` returns a discrete-time, lattice, moving-average filter object `hd`, with lattice coefficients `k`.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter
 

```
set(hd,'arithmetic','single');
```
- To change to fixed-point filtering, enter
 

```
set(hd,'arithmetic','fixed');
```

---

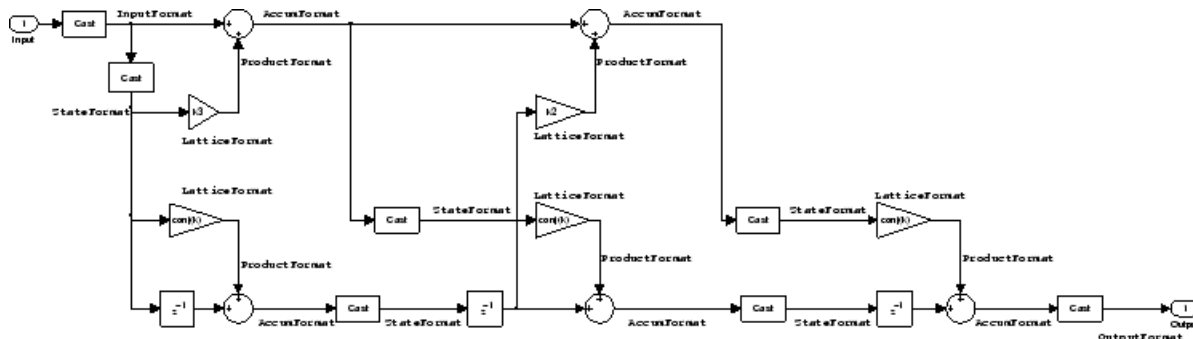
**Note** When the `k` coefficients define a maximum phase filter, the resulting filter in this structure is maximum phase. When your coefficients do not define a maximum phase filter, placing them in this structure does not produce a maximum phase filter.

---

`hd = dfilt.latticemamax` returns a default discrete-time, lattice, moving-average filter object `hd`, with `k = []`. This filter passes the input through to the output unchanged.

## Fixed-Point Filter Structure

The following figure shows the signal flow for the maximum phase implementation of a moving-average lattice filter implemented by `dfilt.latticemamax`. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.



## Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the word “format.” In this use, “format” means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFormat label refers to the word length and fraction length used to interpret the data input to the filter. The format properties InputWordLength and InputFracLength (as shown in the table) store the word length and the fraction length in bits. Or consider NumFormat, which refers to the word and fraction lengths (CoeffWordLength, NumFracLength) associated with representing filter numerator coefficients.

| Signal Flow Label | Corresponding Word Length Property | Corresponding Fraction Length Property | Related Properties |
|-------------------|------------------------------------|--|--------------------|
| AccumFormat       | AccumWordLength                    | AccumFracLength                        | AccumMode          |
| InputFormat       | InputWordLength                    | InputFracLength                        | None               |
| LatticeFormat     | CoeffWordLength                    | LatticeFracLength                      | CoeffAutoScale     |
| OutputFormat      | OutputWordLength                   | OutputFracLength                       | OutputMode         |
| ProductFormat     | ProductWordLength                  | ProductFracLength                      | ProductMode        |
| StateFormat       | StateWordLength                    | StateFracLength                        | States             |

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label ProductFormat, which always follows a coefficient multiplication element in the signal flow. The label indicates that coefficients leave the multiplication element with the word length and fraction length associated with product operations that include coefficients. From reviewing the table, you see that the ProductFormat refers to the properties ProductFracLength, ProductWordLength, and ProductMode that fully define the coefficient format after multiply (or product) operations.

## Properties

In this table you see the properties associated with the maximum phase, moving average lattice implementation of `dfilt` objects.

**Note** The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

```
get(hd)
```

where `hd` is a filter.



| Property Name     | Brief Description  |
|-------------------|--|
| AccumFracLength   | Specifies the fraction length used to interpret data output by the accumulator. This is a property of FIR filters and lattice filters. IIR filters have two similar properties — DenAccumFracLength and NumAccumFracLength — that let you set the precision for numerator and denominator operations separately.   |
| AccumMode         | Determines how the accumulator outputs stored values. Choose from full precision (FullPrecision), or whether to keep the most significant bits (KeepMSB) or least significant bits (KeepLSB) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set AccumMode to SpecifyPrecision.  |
| AccumWordLength   | Sets the word length used to store data in the accumulator/buffer.   |
| Arithmetic        | Defines the arithmetic the filter uses. Gives you the options double, single, and fixed. In short, this property defines the operating mode for your filter.   |
| CastBeforeSum     | Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.  |
| CoeffAutoScale    | Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to false enables you to change the LatticeFracLength property to specify the precision used.   |
| CoeffWordLength   | Specifies the word length to apply to filter coefficients.   |
| FilterStructure   | Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering—gains, delays, sums, products, and input/output.  |
| InputFracLength   | Specifies the fraction length the filter uses to interpret input data.   |
| InputWordLength   | Specifies the word length applied to interpret input data.   |
| Lattice           | Any lattice structure coefficients.  |
| LatticeFracLength | Sets the fraction length applied to the lattice coefficients.  |
| OutputFracLength  | Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set OutputMode to SpecifyPrecision.   |
| OutputMode        | Sets the mode the filter uses to scale the filtered data for output. You have the following choices: <ul style="list-style-type: none"> <li>• <b>AvoidOverflow</b> — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow.</li> <li>• <b>BestPrecision</b> — directs the filter to set the output data word length and fraction length to maximize the precision in the output data.</li> <li>• <b>SpecifyPrecision</b> — lets you set the word and fraction lengths used by the output data from filtering.</li> </ul> |

| Property Name     | Brief Description   |
|-------------------|---|
| OutputWordLength  | Determines the word length used for the output data.  |
| OverflowMode      | Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either <code>saturate</code> (limit the output to the largest positive or negative representable value) or <code>wrap</code> (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow—they maintain full precision.  |
| ProductFracLength | For the output from a product operation, this sets the fraction length used to interpret the data. This property becomes writable (you can change the value) when you set <code>ProductMode</code> to <code>SpecifyPrecision</code> .   |
| ProductMode       | Determines how the filter handles the output of product operations. Choose from full precision ( <code>FullPrecision</code> ), or whether to keep the most significant bit ( <code>KeepMSB</code> ) or least significant bit ( <code>KeepLSB</code> ) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set <code>ProductMode</code> to <code>SpecifyPrecision</code> .  |
| ProductWordLength | Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set <code>ProductMode</code> to <code>SpecifyPrecision</code> .   |
| PersistentMemory  | Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. <code>False</code> is the default setting.  |
| RoundMode         | <p>Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths).</p> <ul style="list-style-type: none"> <li>• <code>ceil</code> - Round toward positive infinity.</li> <li>• <code>convergent</code> - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.</li> <li>• <code>fix</code> - Round toward zero.</li> <li>• <code>floor</code> - Round toward negative infinity.</li> <li>• <code>nearest</code> - Round toward nearest. Ties round toward positive infinity.</li> <li>• <code>round</code> - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.</li> </ul> <p>The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.</p> |
| Signed            | Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.   |

| Property Name   | Brief Description  |
|-----------------|--|
| StateFracLength | When you set StateAutoScale to false, you enable the StateFracLength property that lets you set the fraction length applied to interpret the filter states.  |
| States          | This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use fi objects, with the associated properties from those objects. For details, refer to filtstates in Signal Processing Toolbox documentation or in the Help system. |
| StateWordLength | Sets the word length used to represent the filter states.  |

## Examples

Specify a fourth-order lattice, moving-average, maximum phase filter structure for a dfilt object, hd, with the following code:

```
k = [.66 .7 .44 .33];
hd = dfilt.latticemamax(k);
```

## Version History

**Introduced in R2011a**

## dfilt.latticemamin

Discrete-time, lattice, moving-average filter with minimum phase

### Syntax

```
hd = dfilt.latticemamin(k)
hd = dfilt.latticemamin
```

### Description

`hd = dfilt.latticemamin(k)` returns a discrete-time, lattice, moving-average, minimum phase, filter object `hd`, with lattice coefficients `k`.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter  

```
set(hd,'arithmetic','single');
```
- To change to fixed-point filtering, enter  

```
set(hd,'arithmetic','fixed');
```

---

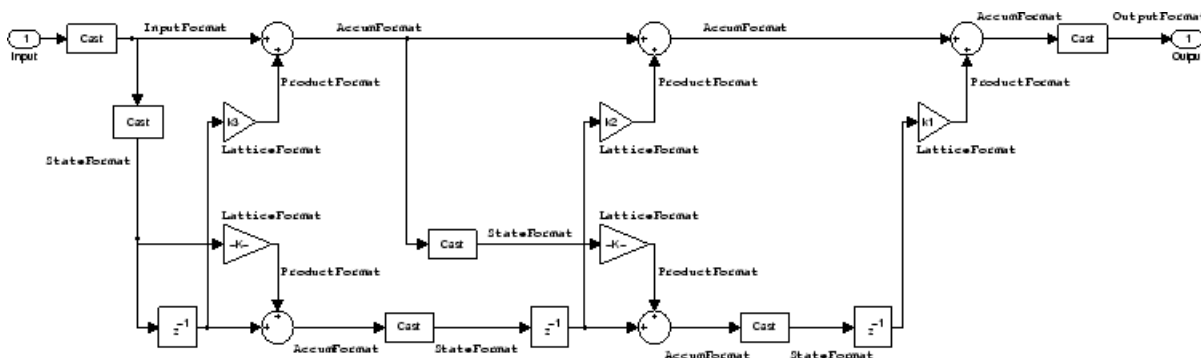
**Note** When the `k` coefficients define a minimum phase filter, the resulting filter in this structure is minimum phase. When your coefficients do not define a minimum phase filter, placing them in this structure does not produce a minimum phase filter.

---

`hd = dfilt.latticemamin` returns a default discrete-time, lattice, moving-average, minimum phase, filter object `hd`, with `k=[ ]`. This filter passes the input through to the output unchanged.

### Fixed-Point Filter Structure

The following figure shows the signal flow for the minimum phase implementation of a moving-average lattice filter implemented by `dfilt.latticemamin`. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.



## Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the word “format.” In this use, “format” means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFormat label refers to the word length and fraction length used to interpret the data input to the filter. The format properties InputWordLength and InputFracLength (as shown in the table) store the word length and the fraction length in bits. Or consider NumFormat, which refers to the word and fraction lengths (CoeffWordLength, NumFracLength) associated with representing filter numerator coefficients.

| Signal Flow Label | Corresponding Word Length Property | Corresponding Fraction Length Property | Related Properties |
|-------------------|------------------------------------|--|--------------------|
| AccumFormat       | AccumWordLength                    | AccumFracLength                        | AccumMode          |
| InputFormat       | InputWordLength                    | InputFracLength                        | None               |
| LatticeFormat     | CoeffWordLength                    | LatticeFracLength                      | CoeffAutoScale     |
| OutputFormat      | OutputWordLength                   | OutputFracLength                       | OutputMode         |
| ProductFormat     | ProductWordLength                  | ProductFracLength                      | ProductMode        |
| StateFormat       | StateWordLength                    | StateFracLength                        | States             |

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label ProductFormat, which always follows a coefficient multiplication element in the signal flow. The label indicates that coefficients leave the multiplication element with the word length and fraction length associated with product operations that include coefficients. From reviewing the table, you see that the ProductFormat refers to the properties ProductFracLength, ProductWordLength, and ProductMode that fully define the coefficient format after multiply (or product) operations.

## Properties

In this table you see the properties associated with the minimum phase, moving average lattice implementation of `dfilt` objects.

**Note** The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

```
get(hd)
```

where `hd` is a filter.

| Property Name     | Brief Description  |
|-------------------|--|
| AccumFracLength   | Specifies the fraction length used to interpret data output by the accumulator. This is a property of FIR filters and lattice filters. IIR filters have two similar properties — DenAccumFracLength and NumAccumFracLength — that let you set the precision for numerator and denominator operations separately.   |
| AccumMode         | Determines how the accumulator outputs stored values. Choose from full precision (FullPrecision), or whether to keep the most significant bits (KeepMSB) or least significant bits (KeepLSB) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set AccumMode to SpecifyPrecision.  |
| AccumWordLength   | Sets the word length used to store data in the accumulator/buffer.   |
| Arithmetic        | Defines the arithmetic the filter uses. Gives you the options double, single, and fixed. In short, this property defines the operating mode for your filter.   |
| CastBeforeSum     | Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.  |
| CoeffAutoScale    | Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to false enables you to change the LatticeFracLength property to specify the precision used.   |
| CoeffWordLength   | Specifies the word length to apply to filter coefficients.   |
| FilterStructure   | Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering — gains, delays, sums, products, and input/output.  |
| InputFracLength   | Specifies the fraction length the filter uses to interpret input data.   |
| InputWordLength   | Specifies the word length applied to interpret input data.   |
| Lattice           | Any lattice structure coefficients.  |
| LatticeFracLength | Sets the fraction length applied to the lattice coefficients.  |
| OutputFracLength  | Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set OutputMode to SpecifyPrecision.   |
| OutputMode        | Sets the mode the filter uses to scale the filtered data for output. You have the following choices: <ul style="list-style-type: none"> <li>• <b>AvoidOverflow</b> — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow.</li> <li>• <b>BestPrecision</b> — directs the filter to set the output data word length and fraction length to maximize the precision in the output data.</li> <li>• <b>SpecifyPrecision</b> — lets you set the word and fraction lengths used by the output data from filtering.</li> </ul> |

| Property Name     | Brief Description   |
|-------------------|---|
| OutputWordLength  | Determines the word length used for the output data.  |
| OverflowMode      | Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either <code>saturate</code> (limit the output to the largest positive or negative representable value) or <code>wrap</code> (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.  |
| ProductFracLength | For the output from a product operation, this sets the fraction length used to interpret the data. This property becomes writable (you can change the value) when you set <code>ProductMode</code> to <code>SpecifyPrecision</code> .   |
| ProductMode       | Determines how the filter handles the output of product operations. Choose from full precision ( <code>FullPrecision</code> ), or whether to keep the most significant bit ( <code>KeepMSB</code> ) or least significant bit ( <code>KeepLSB</code> ) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set <code>ProductMode</code> to <code>SpecifyPrecision</code> .  |
| ProductWordLength | Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set <code>ProductMode</code> to <code>SpecifyPrecision</code> .   |
| PersistentMemory  | Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. <code>False</code> is the default setting.  |
| RoundMode         | <p>Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths).</p> <ul style="list-style-type: none"> <li>• <code>ceil</code> - Round toward positive infinity.</li> <li>• <code>convergent</code> - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.</li> <li>• <code>fix</code> - Round toward zero.</li> <li>• <code>floor</code> - Round toward negative infinity.</li> <li>• <code>nearest</code> - Round toward nearest. Ties round toward positive infinity.</li> <li>• <code>round</code> - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.</li> </ul> <p>The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.</p> |
| Signed            | Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.   |

| Property Name   | Brief Description  |
|-----------------|--|
| StateFracLength | When you set StateAutoScale to false, you enable the StateFracLength property that lets you set the fraction length applied to interpret the filter states.  |
| States          | This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use <code>fi</code> objects, with the associated properties from those objects. For details, refer to <code>filtstates</code> in Signal Processing Toolbox documentation or in the Help system. |
| StateWordLength | Sets the word length used to represent the filter states.  |

## Examples

Specify a third-order lattice, moving-average, minimum phase, filter structure for a `dfilt` object, `hd`, with the following code:

```
k = [.66 .7 .44];  
hd = dfilt.latticemamin(k);
```

## Version History

**Introduced in R2011a**



## dfilt.parallel

Discrete-time, parallel structure filter

### Syntax

```
hd = dfilt.parallel(hd1,hd2,...)
```

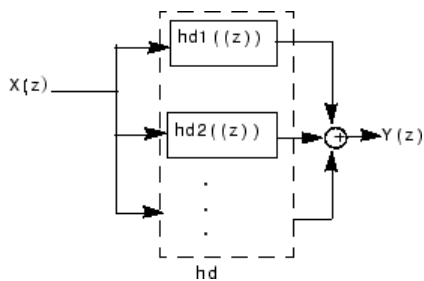
### Description

`hd = dfilt.parallel(hd1,hd2,...)` returns a discrete-time filter object `hd`, which is a structure of two or more `dfilt` filter objects, `hd1`, `hd2`, and so on arranged in parallel.

You can also use the standard notation to combine filters into a parallel structure.

```
parallel(hd1,hd2,...)
```

In this syntax, `hd1`, `hd2`, and so on can be a mix of `dfilt` objects and other filtering objects.



`hd1`, `hd2`, and so on can be fixed-point filters. All filters in the parallel structure must be the same arithmetic format — double, single, or fixed. `hd`, the filter returned, inherits the format of the individual filters.

## Version History

Introduced in R2011a

## dfilt.scalar

Discrete-time, scalar filter

### Syntax

```
dfilt.scalar(g)
dfilt.scalar
```

### Description

`dfilt.scalar(g)` returns a discrete-time, scalar filter object with gain `g`, where `g` is a scalar.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

```
set(hd,'arithmetic','single');
```

- To change to fixed-point filtering, enter

```
set(hd,'arithmetic','fixed');
```

`dfilt.scalar` returns a default, discrete-time scalar gain filter object `hd`, with gain 1.

### Properties

In this table you see the properties associated with the scalar implementation of `dfilt` objects.

---

**Note** The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

```
get(hd)
```

where `hd` is a filter.

---

| Property Name | Brief Description  |
|---------------|--|
| Arithmetic    | Defines the arithmetic the filter uses. Gives you the options <code>double</code> , <code>single</code> , and <code>fixed</code> . In short, this property defines the operating mode for your filter. |
| CastBeforeSum | Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.  |

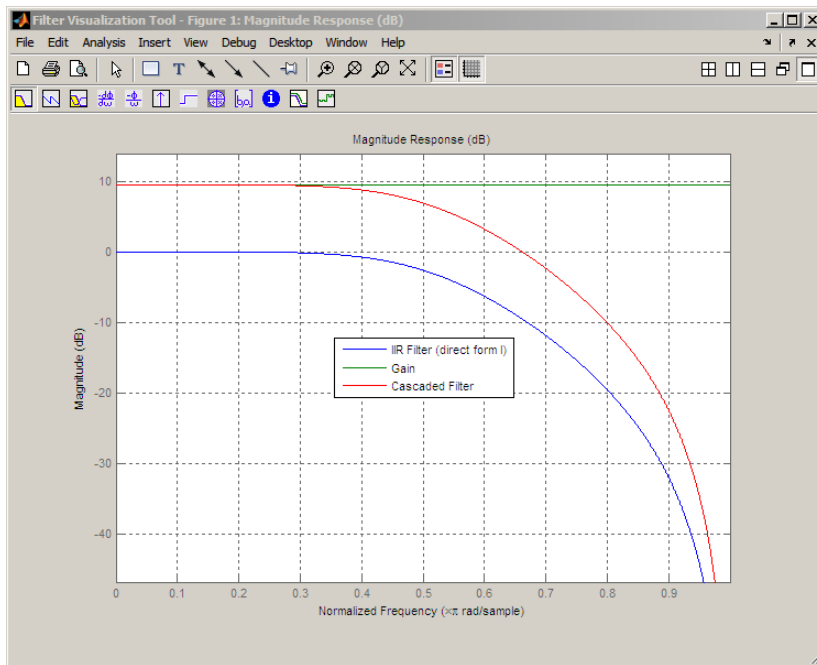
| Property Name    | Brief Description  |
|------------------|--|
| CoeffAutoScale   | Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to <code>false</code> enables you to change the <code>CoeffFracLength</code> property to specify the precision used.   |
| CoeffFracLength  | Set the fraction length the filter uses to interpret coefficients. <code>CoeffFracLength</code> is always available, but it is read-only until you set <code>CoeffAutoScale</code> to <code>false</code> .   |
| CoeffWordLength  | Specifies the word length to apply to filter coefficients.   |
| FilterStructure  | Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering — gains, delays, sums, products, and input/output.  |
| Gain             | Returns the gain for the scalar filter. Scalar filters do not alter the input data except by adding gain.  |
| InputFracLength  | Specifies the fraction length the filter uses to interpret input data.   |
| InputWordLength  | Specifies the word length applied to interpret input data.   |
| OutputFracLength | Determines how the filter interprets the filter output data. You can change the value of <code>OutputFracLength</code> when you set <code>OutputMode</code> to <code>SpecifyPrecision</code> .   |
| OutputMode       | Sets the mode the filter uses to scale the filtered data for output. You have the following choices: <ul style="list-style-type: none"> <li>• <code>AvoidOverflow</code> — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow.</li> <li>• <code>BestPrecision</code> — directs the filter to set the output data word length and fraction length to maximize the precision in the output data.</li> <li>• <code>SpecifyPrecision</code> — lets you set the word and fraction lengths used by the output data from filtering.</li> </ul> |
| OutputWordLength | Determines the word length used for the output data.   |
| OverflowMode     | Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either <code>saturate</code> (limit the output to the largest positive or negative representable value) or <code>wrap</code> (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.   |
| PersistentMemory | Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. <code>False</code> is the default setting.   |

| Property Name | Brief Description   |
|---------------|---|
| RoundMode     | <p>Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths).</p> <ul style="list-style-type: none"> <li>• <code>ceil</code> - Round toward positive infinity.</li> <li>• <code>convergent</code> - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.</li> <li>• <code>fix</code> - Round toward zero.</li> <li>• <code>floor</code> - Round toward negative infinity.</li> <li>• <code>nearest</code> - Round toward nearest. Ties round toward positive infinity.</li> <li>• <code>round</code> - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.</li> </ul> <p>The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.</p> |
| Signed        | Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.   |
| States        | This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use <code>fi</code> objects, with the associated properties from those objects. For details, refer to <code>filtstates</code> in Signal Processing Toolbox documentation or in the Help system.  |

## Examples

Create a direct-form I filter object `hd_filt` and a scalar object with a gain of 3 `hd_gain` and cascade them together.

```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
hd_filt = dfilt.df1(b,a);
hd_gain = dfilt.scalar(3);
hd_cascade=cascade(hd_gain,hd_filt);
fvtool_handle = fvtool(hd_filt,hd_gain,hd_cascade);
legend(fvtool_handle,'IIR Filter (direct form I)',...
'Gain','Cascaded Filter');
```



To view the stages of the cascaded filter, use

```
hd.Stage(1)
```

and

```
hd.Stage(2)
```

## Version History

Introduced in R2011a

## dfilt.wdfallpass

Wave digital allpass filter

### Syntax

```
hd = dfilt.wdfallpass(c)
```

### Description

`hd = dfilt.wdfallpass(c)` constructs an allpass wave digital filter structure given the allpass coefficients in vector `c`.

Vector `c` must have, one, two, or four elements (filter coefficients). Filters with three coefficients are not supported. When you use `c` with four coefficients, the first and third coefficients must be 0.

Given the coefficients in `c`, the transfer function for the wave digital allpass filter is defined by

$$H(z) = \frac{c(n) + c(n-1)z^{-1} + \dots + z^{-n}}{1 + c(1)z^{-1} + \dots + c(n)z^{-n}}$$

Internally, the allpass coefficients are converted to wave digital filters for filtering. Note that `dfilt.wdfallpass` allows only stable filters. Also note that the leading coefficient in the denominator, a 1, does not need to be included in vector `c`.

Use the constructor `dfilt.cascadewdfallpass` to cascade `wdfallpass` filters.

To compare these filters to other similar filters, `dfilt.wdfallpass` and `dfilt.cascadewdfallpass` filters have the same number of multipliers as the non-wave digital filters `dfilt.allpass` and `dfilt.cascadeallpass`. However, the wave digital filters use fewer states and they may require more adders in the filter structure.

Wave digital filters are usually used to create other filters. This toolbox uses them to implement halfband filters, which the first example in Examples demonstrates. They are most often building blocks for filters.

### Properties

In the next table, the row entries are the filter properties and a brief description of each property.

| Property Name       | Brief Description   |
|---------------------|---|
| AllpassCoefficients | Contains the coefficients for the allpass wave digital filter object  |
| FilterStructure     | Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering — gains, delays, sums, products, and input/output. |

| Property Name    | Brief Description  |
|------------------|--|
| PersistentMemory | Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. <code>False</code> is the default setting.   |
| States           | This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. They also provide linkage between the sections of a multisection filter, such as a cascade filter. For details, refer to <code>filtstates</code> in Signal Processing Toolbox documentation or in the Help system. |

## Filter Structure

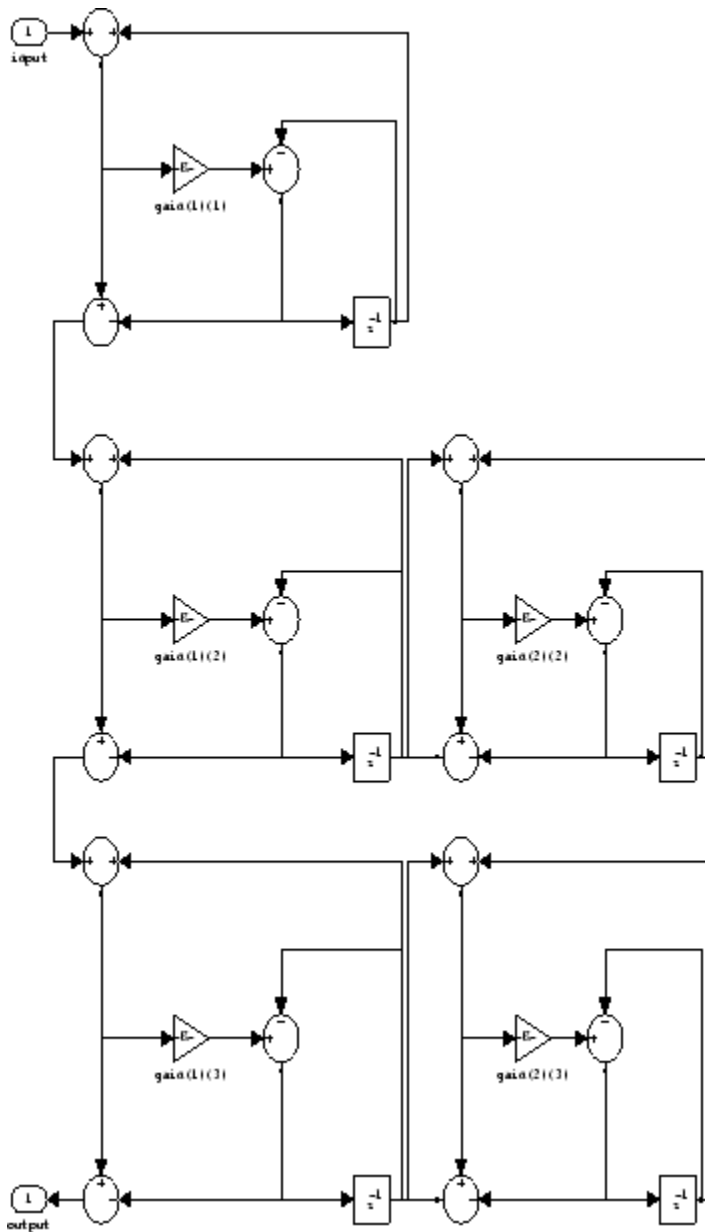
When you change the order of the wave digital filters in the cascade, the filter structure changes as well.

As shown in this example, `realizemdl` lets you see the filter structure used for your filter, if you have Simulink installed.

```
section11=0.8;
section12=[1.5,0.7];
section13=[1.8,0.9];
hd1=dfilt.cascadewdfallpass(section11,section12,section13);
```

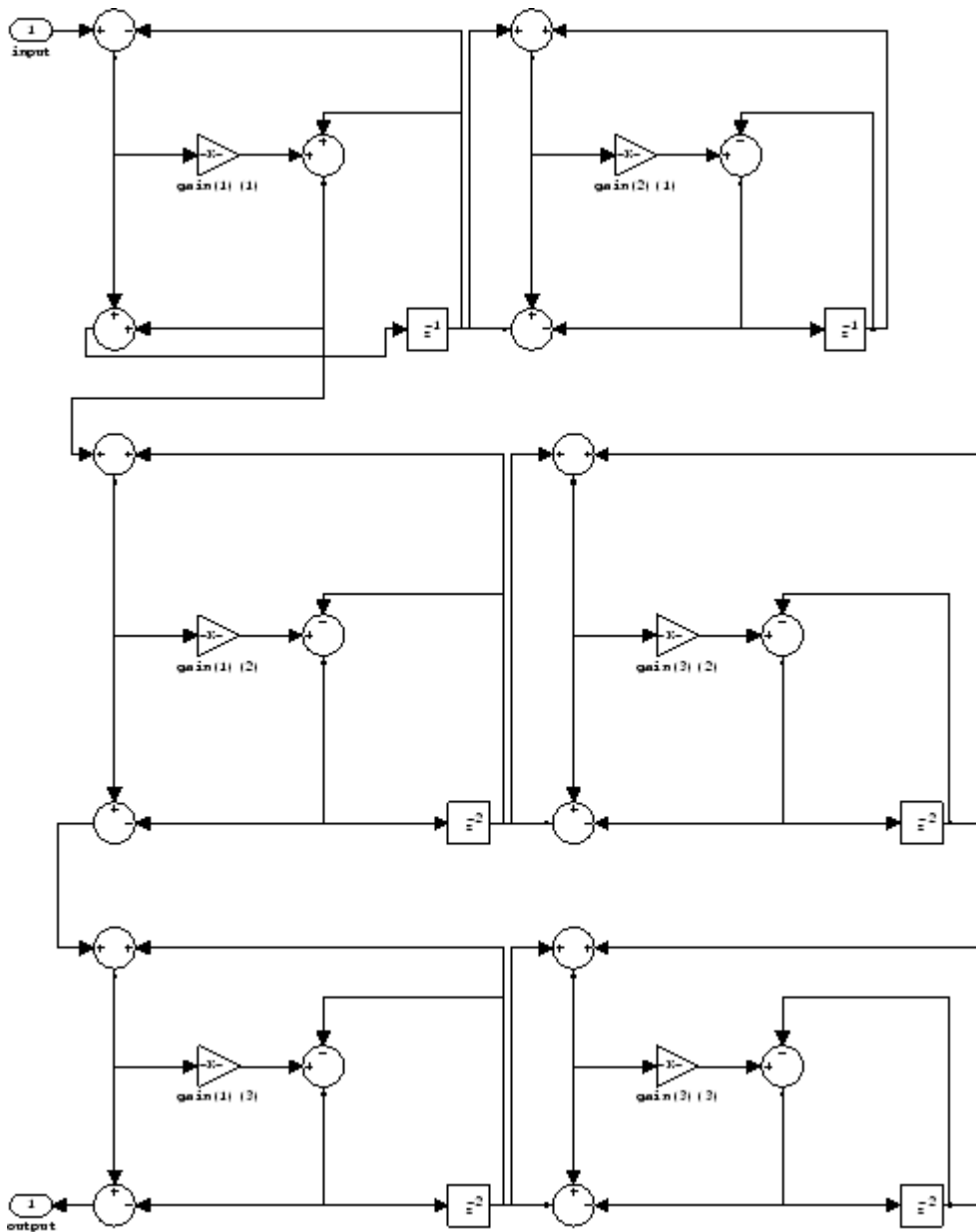
```
section21=[0.8,0.4];
section22=[0,1.5,0,0.7];
section23=[0,1.8,0,0.9];
hd2=dfilt.cascadewdfallpass(section21,section22,section23);
% If you have Simulink
realizemdl(hd2)
```

`hd1` has this filter structure with three sections.



The filter structure for `hd2` is somewhat different, with the different orders and interconnections between the three sections.



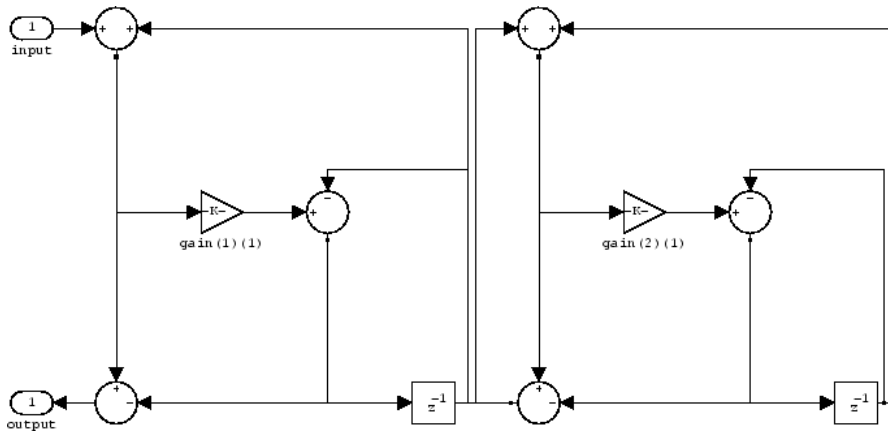


## Examples

Construct a second-order wave digital allpass filter with two coefficients. Note that to use `realizemdl`, you must have Simulink.

```
c = [1.5,0.7];
hd = dfilt.wdfallpass(c);
```

With Simulink installed, `realizemdl` returns this structure for `hd`.



## Version History

Introduced in R2011a

### See Also

`dsp.IIRHalfbandDecimator` | `dsp.CICInterpolator`

# disp

Display filter properties and values

## Syntax

```
disp(obj)
```

## Description

`disp(obj)` lists the property names and property values of the object `obj`. In all ways, it is the same as leaving the semicolon off an on the command line while creating the object, except that `disp` does not display the variable name.

## Examples

### Display Properties of Filter Object

Create a `dsp.FIRFilter` object.

```
filt = dsp.FIRFilter
filt =
    dsp.FIRFilter with properties:
        Structure: 'Direct form'
        NumeratorSource: 'Property'
        Numerator: [0.5000 0.5000]
        InitialConditions: 0
    Show all properties
```

Display its properties using the `disp` function.

```
disp(filt)
dsp.FIRFilter with properties:
    Structure: 'Direct form'
    NumeratorSource: 'Property'
    Numerator: [0.5000 0.5000]
    InitialConditions: 0
    Use get to show all properties
```

## Input Arguments

### **obj** — Input object

object

Input object, specified as one of the System objects or `fdesign` objects supported by DSP System Toolbox. For a list, see “Functions”.

## **Version History**

**Introduced in R2011a**

## **See Also**

`set`

## double

Cast fixed-point filter to use double-precision arithmetic

### Syntax

```
hd = double(h)
```

### Description

`hd = double(h)` returns a new filter `hd` that has the same structure and coefficients as `h`, but whose arithmetic property is set to `double` to use double-precision arithmetic for filtering. `double(h)` is not the same as the `refilter(h)` function:

- `hd`, the filter returned by `double` has the quantized coefficients of `h` represented in double-precision floating-point format
- The reference filter returned by `refilter` has double-precision, floating-point coefficients that have not been quantized.

You might find `double(h)` useful to isolate the effects of quantizing the coefficients of a filter by using `double` to create a filter `hd` that operates in double-precision but uses the quantized filter coefficients.

### Examples

#### Compare Fixed-point Output with Floating-Point Output

Construct a Lowpass Filter

```
h = dfilt.dffir(firgr(27,[0 .4 .6 1],...
[1 1 0 0]));
```

Set `h` to use fixed-point arithmetic to filter. Quantize the coefficients.

```
h.arithmetic = 'fixed';
```

Cast `h` to double-precision

```
hd = double(h);
```

Set up an input signal.

```
n = 0:99; x = sin(0.7*pi*n(:));
```

Fixed-point output.

```
y = filter(h,x);
```

Floating-point output.

```
yd = filter(hd,x);
```

Compare the Fixed-point output with Floating-point output

```
FixedFloatNormDiff=norm(yd-double(y),inf)
```

```
FixedFloatNormDiff = 2.1234e-05
```

## **Version History**

**Introduced in R2011a**

## dsp\_links

Identify whether blocks in model are current, deprecated, or obsolete

### Syntax

```
dsp_links
dsp_links('modelname')
```

### Description

`dsp_links` returns a structure with three elements that identify whether the DSP System Toolbox blocks in the current model are current, deprecated, or obsolete. Each element is one of the three block categories and contains a cell array of character vectors. Each character vector is the name of a library block in the current model.

`dsp_links('modelname')` returns the three-element structure for the specified model.

### Examples

#### Name of the First Current Block

Display block support information for the specified model, and then find the name of the first current block.

```
sys = 'cochlearimplantspeechprocessor';
```

Load the dspcochlear model

```
load_system(sys)
```

Run `dsp_links` on the model

```
links = dsp_links(sys)
```

```
links = struct with fields:
    obsolete: {}
    deprecated: {'cochlearimplantspeechprocessor/Info'}
    current: {1x23 cell}
```

Find the name of the first current block

```
links.current{1}
```

```
ans =
'cochlearimplantspeechprocessor/Filter bank signal processing/Subsystem3/IIR Filter Bank/Digital
```

## **More About**

### **Obsolete Blocks**

*Obsolete blocks* are blocks that the toolbox no longer supports. In some cases, these blocks no longer function properly.

### **Deprecated Blocks**

*Deprecated blocks* are blocks that the toolbox still supports but are likely to become obsolete in a future release. Refer to the block reference page for suggested replacements.

### **Current Blocks**

*Current blocks* are blocks that the toolbox supports and that represent the latest block functionality.

## **Version History**

**Introduced before R2006a**

### **See Also**

liblinks



# dsplib

Open top-level DSP System Toolbox library

## Syntax

```
dsplib
```

## Description

dsplib opens the top-level DSP System Toolbox block library model.

## Examples

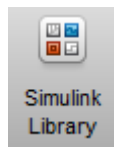
View and gain access to the DSP System Toolbox blocks:

```
dsplib
```

## Alternatives

To view and gain access to the DSP System Toolbox blocks using the Simulink library browser:

- Type `simulink` at the MATLAB command line, and then expand the DSP System Toolbox node in the library browser.
- 



Click the Simulink icon from the MATLAB Toolstrip.

## Version History

Introduced before R2006a

## **dspunfold**

Generates a multi-threaded MEX file from a MATLAB function

### **Syntax**

```
dspunfold file  
dspunfold options file
```

### **Description**

`dspunfold file` generates a multi-threaded MEX file from the entry-point MATLAB function specified by `file`, using the unfolding technology. Unfolding is a technique to improve throughput through parallelization. The multi-threaded MEX file leverages the multicore CPU architecture of the host computer and can improve speed significantly. In addition to the multi-threaded MEX file, the function generates a single-threaded MEX file, a self-diagnostic analyzer function, and the corresponding help files.

`dspunfold options file` generates a multi-threaded MEX file from the entry-point MATLAB function specified by `file`, using the function arguments specified by `options`.

---

**Note** This function requires a MATLAB Coder license.

---

### **Input Arguments**

**options** — Function parameters

option value pairs

| Option          | Values     | Description  | Examples  |
|-----------------|------------|--|---|
| -args arguments | Cell array | <p>Argument types for the entry-point MATLAB function, specified as a cell array.</p> <p>The cell array accepts numeric elements, the <code>coder.typeof</code> function, and the <code>coder.Constant</code> function.</p> <p>The generated multi-threaded MEX file is specialized to the size, class, and complexity of arguments.</p> | <p>The number of elements in the cell array must be the same as the number of arguments that the entry-point MATLAB function expects.</p> <ul style="list-style-type: none"> <li>• <code>dspunfold fcn -args {ones(10,1), 5}</code></li> </ul> <p><code>dspunfold</code> extracts the type (size, class, and complexity) information from the elements in the arguments cell array.</p> <p><code>fcn</code> is the entry-point MATLAB function.</p> <ul style="list-style-type: none"> <li>• <code>dspunfold fcn -args {coder.typeof(ones(10,1)), coder.typeof(5)}</code></li> </ul> <p><code>coder.typeof</code> is used to specify the types of the <code>fcn</code> arguments.</p> <ul style="list-style-type: none"> <li>• <code>dspunfold fcn -args {coder.Constant(ones(10,1)), coder.Constant(5)}</code></li> <li>• <code>dspunfold fcn -args {}</code></li> </ul> <p>By default, <code>arguments</code> is <code>{}</code>. An empty cell array <code>{}</code> indicates that <code>fcn</code> accepts no input arguments.</p> |

| Option    | Values           | Description  | Examples   |
|-----------|------------------|--|--|
| -o output | Character vector | Name of the output multi-threaded MEX file, specified as a character vector. If no output name is specified, the name of the generated multi-threaded MEX file is inherited from the input MATLAB function with an '_mt' suffix. <code>dspunfold</code> also adds a platform-specific extension to this name. In addition, <code>dspunfold</code> generates a single-threaded MEX file with an '_st' suffix, and a test bench file with an '_analyzer' suffix. | <ul style="list-style-type: none"><li>• No output name specified<br/><code>dspunfold fcn</code><br/>Files generated: <code>fcn_mt.mexw64</code>, <code>fcn_st.mexw64</code>, <code>fcn_analyzer.p</code></li><li>• output name specified<br/><code>dspunfold fcn -o foo</code><br/>Files generated: <code>foo.mexw64</code>, <code>foo_st.mexw64</code>, <code>foo_analyzer.p</code></li></ul> |

| Option         | Values   | Description  | Examples   |
|----------------|--|--|--|
| -s statelength | Scalar integer greater than or equal to zero<br><br>auto | <p>State length of the algorithm in the entry-point MATLAB function, specified as a scalar integer greater than or equal to zero, or auto. By default, the <code>statelength</code> is zero frames, indicating that the algorithm is stateless.</p> <p>If at least one entry of <code>frameinputs</code> is true, <code>statelength</code> is considered in samples.</p> <p>For information on frames and samples, see “Sample- and Frame-Based Concepts”</p> <p>-s auto triggers automatic state length detection. In this mode, you must provide numeric inputs to the <code>arguments</code> cell array. These inputs detect the state length of the algorithm. You can input <code>coder.Constant</code> but not <code>coder.typeof</code>. When automatic state length detection is invoked, it is recommended that you provide random inputs to the <code>arguments</code> array. See “Automatic State Length Detection” on page 5-365</p> | <ul style="list-style-type: none"> <li>• <code>dspunfold fcn -args {randn(10,1), randn(10,1), randn(10,1)} -s 3 -f [false, false, false]</code><br/><br/>State length is three frames.</li> <li>• <code>dspunfold fcn -args {randn(10,1), randn(10,1), randn(10,1)} -s 3 -f [true, false, false]</code><br/><br/>State length is three samples. State length is considered in samples, because at least one entry of the -f option is true.</li> <li>• <code>dspunfold fcn -args {randn(10,1), randn(10,1), randn(10,1)} -s auto</code><br/><br/>Automatic state length detection is invoked.</li> <li>• <code>dspunfold fcn -args {coder.typeof (randn(10,1)), coder.typeof(randn(10,1)), coder.typeof(randn(10,1))} -s auto</code> generates this error message: The input argument 1 is of type <code>coder.PrimitiveType</code> which is not supported when using -s auto</li> </ul> |

| Option                | Values   | Description   | Examples  |
|-----------------------|--|---|---|
| <p>-f frameinputs</p> | <p>scalar logical<br/>vector of logical values</p> | <p>Frame status of input arguments for the entry-point MATLAB function, specified as one of true or false.</p> <ul style="list-style-type: none"> <li>• true — Input is in frames and can be subdivided into samples without changing the system behavior.</li> <li>• false — Input cannot be subdivided into samples without changing the system behavior. For example, you cannot subdivide the coefficients of a filter without changing the characteristics of the filter.</li> </ul> <p>By default, frameinputs is false.</p> <p>frameinputs set to a scalar logical value sets the frame status of all the inputs simultaneously.</p> <p>To specify stateLength in samples, set at least one entry of frameinputs to true.</p> <p>If frameinputs is not specified, the unit of stateLength is frames.</p> | <ul style="list-style-type: none"> <li>• dspunfold fcn -args {randn(10,1), randn(10,1), randn(10,1)} -s 3 -f true</li> </ul> <p>All the inputs are marked as frames. State length is three samples.</p> <ul style="list-style-type: none"> <li>• dspunfold fcn -args {randn(10,1), randn(10,1), randn(10,1)} -s 3 -f [true, false, false]</li> </ul> <p>State length is three samples.</p> <ul style="list-style-type: none"> <li>• dspunfold fcn -args {randn(10,1), randn(10,1), randn(10,1)} -s 3</li> </ul> <p>The default value of frameinputs is false. State length is three frames.</p> |

| Option        | Values           | Description   | Examples  |
|---------------|------------------|---|---|
| -r repetition | Positive integer | Repetition factor used to generate the multi-threaded MEX file, specified as a positive integer. The default value of <code>repetition</code> is 1. See “Repetition Factor” on page 5-366.                                  | <code>dspunfold fcn -args {randn(10,2), randn(20,2), randn(30,3)} -r 2</code>   |
| -t threads    | Positive integer | Number of threads used by the multi-threaded MEX file, specified as a positive integer. The default value of <code>threads</code> is the number of physical CPU cores present on your machine. See “Threads” on page 5-366. | <code>dspunfold fcn -args {randn(10,1), randn(20,2), randn(30,3)} -t 4</code>   |
| -v verbose    | Scalar logical   | Option to show verbose output during code generation, specified as <code>true</code> or <code>false</code> . The default is <code>true</code> .   | <ul style="list-style-type: none"> <li><code>dspunfold fcn -args {randn(10,1), randn(20,2), randn(30,3)} -v true</code></li> <li><code>dspunfold fcn -args {randn(10,1), randn(20,2), randn(30,3)} -v false</code></li> </ul> |

### file — entry-point MATLAB function

character vector

Entry-point MATLAB function from which `dspunfold` generates the multi-threaded MEX file. The function must support code generation.

Example: `dspunfold fcn -args {randn(10,1), randn(10,2), randn(20,1)}`

`fcn` is the entry-point MATLAB function and `{randn(10,1), randn(10,2), randn(20,1)}` are its input arguments.

## Output Files

When you invoke `dspunfold` on an entry-point MATLAB function, `dspunfold` generates the following files.

| File                                      | Value       | Description   | Examples  |
|---|-------------|---|---|
| Multi-threaded MEX file                   | MEX file    | Multi-threaded MEX file generated from the entry-point MATLAB function. The MEX file inherits the output name. If no output name is specified, the name of this file is inherited from the MATLAB function with an '_mt' suffix. A platform-specific extension is also added to the name.   | <ul style="list-style-type: none"> <li>• <code>dspunfold fcn -o foo</code> generates <code>foo.mexw64</code></li> <li>• <code>dspunfold fcn</code> generates <code>fcn_mt.mexw64</code></li> </ul>    |
| Help file for the multi-threaded MEX file | MATLAB file | <p>MATLAB help file for the multi-threaded MEX file. The help file has the same name as the MEX file, but with an '.m' extension. To invoke the help file, type <code>help &lt;MEX filename&gt;</code> at the MATLAB command prompt.</p> <p>This help file displays information on how to invoke the MEX file, its syntax, latency, and types (size, class, and complexity) of the inputs to the MEX file. In addition, the help file documents the parameters used by <code>dspunfold</code> — <code>Threads</code>, <code>Repetition</code>, and <code>State length</code>. This information is useful when you are invoking the MEX file. The syntax to invoke the MEX file should be the same as the syntax shown in the help file.</p> | <ul style="list-style-type: none"> <li>• <code>help foo</code></li> <li>• <code>help fcn_mt</code></li> </ul>   |
| Single-threaded MEX file                  | MEX file    | Single-threaded MEX file generated from the entry-point MATLAB function. The MEX file inherits the output name with an '_st' suffix. If no output name is specified, the name of this file is inherited from the MATLAB function with an '_st' suffix. A platform-specific extension is also added to the name. Use this file as a benchmark to compare against the speed of the multi-threaded MEX file.   | <ul style="list-style-type: none"> <li>• <code>dspunfold fcn -o foo</code> generates <code>foo_st.mexw64</code></li> <li>• <code>dspunfold fcn</code> generates <code>fcn_st.mexw64</code></li> </ul> |



| File                                       | Value       | Description  | Examples  |
|--|-------------|--|---|
| Help file for the single-threaded MEX file | MATLAB file | <p>MATLAB help file for the single-threaded MEX file. The help file has the same name as the MEX file, but with an '.m' extension. To invoke the help file, type <code>help &lt;MEX filename&gt;</code> at the MATLAB command prompt.</p> <p>The help file displays information on how to invoke the MEX file, its syntax, and types (size, class, and complexity) of the inputs to the MEX file. The syntax to invoke the MEX file should be the same as the syntax shown in the help file.</p> | <ul style="list-style-type: none"><li>• <code>help foo_st</code></li><li>• <code>help fcn_st</code></li></ul> |

| File                              | Value        | Description  | Examples  |
|-----------------------------------|--------------|--|---|
| Self-diagnostic analyzer function | P-coded file | <p><code>report = function_analyzer (input 1, input 2, ... input n)</code> measures the difference in speed between the multi-threaded MEX file and the single-threaded MEX file. This file verifies that the output values match.</p> <p><code>report = function_analyzer('latency')</code> reports the latency of the multi-threaded MEX file introduced by unfolding.</p> <p><code>report</code> contains the following fields:</p> <ul style="list-style-type: none"> <li>• Latency — The value of the latency (in frames)</li> <li>• Speedup — The speedup difference between the multi-threaded MEX file and single-threaded MEX file. If you specified <code>latency</code> option, the value of this field is empty <code>[]</code>.</li> <li>• Pass — Logical value that shows if the outputs match between the generated multi-threaded MEX file and the single-threaded MEX file. If you specified <code>latency</code> option, the value of this field is empty <code>[]</code>.</li> </ul> <p>The first dimension of the analyzer inputs must be a multiple of the first dimension of the corresponding inputs, given to the <code>-args</code> option. The other dimensions must match exactly.</p> <p>The analyzer inherits the <code>output</code> name with an <code>'_analyzer'</code> suffix. If no <code>output</code> name is specified, the name of this file is inherited from the MATLAB function with an <code>'_analyzer'</code> suffix.</p> | <ul style="list-style-type: none"> <li>• Multiple frames with different values are specified along the first dimension</li> </ul> <p>Example 1: <code>report = foo_analyzer(randn(10*2,1), randn(20*2,2), randn(30*3,3))</code></p> <p>Example 2: <code>report = foo_analyzer([randn(10,1); randn(10,1)], [randn(20,1); randn(20,1)], [randn(30,1); randn(30,1)]; randn(30,1))</code></p> <ul style="list-style-type: none"> <li>• <code>report = foo_analyzer('latency')</code></li> </ul> |

| File  | Value       | Description   | Examples                       |
|---|-------------|---|--------------------------------|
| Help file for the self-diagnostic analyzer function | MATLAB file | <p>Help file for the self-diagnostic analyzer function. The help file has the same name as the MEX file, but with an '.m' extension. To invoke the help file, type <code>help &lt;function_analyzer&gt;</code> in MATLAB.</p> <p>The help file for the self-diagnostic analyzer function displays information on how to invoke the analyzer function, its syntax, and types (size, class, and complexity) of the inputs to the analyzer function. The syntax to invoke the analyzer function should be the same as the syntax shown in the help file.</p> | <code>help foo_analyzer</code> |

## Limitations

### General Limitations:

- On Windows and Linux, you must use a compiler that supports the Open Multiprocessing (OpenMP) application interface. See [https://www.mathworks.com/support/compilers/current\\_release/](https://www.mathworks.com/support/compilers/current_release/).
- If you have a macOS with an Xcode version 12.0 or later, using the `dspunfold` function is not supported.
- If the input MATLAB function has runtime errors, the errors are not caught when you run the multi-threaded MEX file. Before you use the `dspunfold` function, call `codegen` on the MATLAB function and make sure that the MEX file is generated successfully.
- If the generated code uses a large amount of memory to store the local variables, around 4 MB on Windows platform, the generated multi-threaded MEX file can have unexpected behavior. This limit varies with each platform. As a workaround, reduce the size of the input signals or restructure the MATLAB function to use less local memory.
- `dspunfold` does not support:
  - `varargin` and `varargout` inside the MATLAB function
  - Variable-size inputs and outputs
  - Input signals with an arbitrary frame length to System objects that use the `DecimationFactor` property. The input signal is considered to have an arbitrary frame length when its frame length is not a multiple of the decimation factor. When this is the case, the output of the object in the generated code is a variable-size signal, and `dspunfold` does not support variable-size output signals.

In the case of the `dsp.FarrowRateConverter` object, you can determine the decimation factor using the `getRateChangeFactors` function.

- P-coded entry-point MATLAB functions

- Cell arrays as inputs and outputs

### Analyzer Limitations:

The following limitations apply to the analyzer function generated by the `dspunfold` function. For more information on the analyzer function, see 'Self-Diagnostic Analyzer' in the 'More About' section of `dspunfold`.

- If multiple frames of the analyzer input are identical, the analyzer might throw false positive pass results. It is recommended that you provide at least two different frames for each input of the analyzer.
- If the algorithm in the entry-point MATLAB function chooses its state length based on the input values, the analyzer might provide different *pass* results for different input values. For an example, see the `FIR_Mean` function in "Why Does the Analyzer Choose the Wrong State Length?".
- If the input to the entry-point MATLAB function does affect the output immediately, the analyzer might throw false positive *pass* results. For an example, see the `Input_Output` function in "Why Does the Analyzer Choose a Zero State Length?".
- If the output results of the multi-threaded MEX file and single-threaded MEX file match statistically but do not match numerically, the analyzer does not pass. Consider the `FilterNoise` function that follows, which filters a random noise signal with an FIR filter. The function calls `randn` from within itself to generate random noise. Hence, the output results of the `FilterNoise` function match statistically but not match numerically.

```
function Output = FilterNoise(x)

persistent FIRFilter
if isempty(FIRFilter)
    FIRFilter = dsp.FIRFilter('Numerator',fir1(12,0.4));
end
Output = FIRFilter(x+randn(1000,1));

end
```

When you run the automatic state length detection tool run on `FilterNoise`, the tool detects an infinite state length. Because the tool cannot find a numerical match for a finite state length, it chooses an infinite state length.

```
dspunfold FilterNoise -args {randn(1000,1)} -s auto
```

```
Analyzing input MATLAB function FilterNoise
Creating single-threaded MEX file FilterNoise_st.mexw64
Searching for minimal state length (this might take a while)
Checking stateless ... Insufficient
Checking 1 ... Insufficient
Checking Infinite ... Sufficient
Checking 2 ... Insufficient
Minimal state length is Inf
Creating multi-threaded MEX file FilterNoise_mt.mexw64
Warning: The multi-threading was disabled due to performance considerations.
This happens when the state length is greater than or
equal to (Threads-1)*Repetition frames (3 frames in this case).
> In coder.internal.warning (line 8)
   In unfoldingEngine/BuildParallelSolution (line 25)
   In unfoldingEngine/generate (line 207)
   In dspunfold (line 234)
Creating analyzer file FilterNoise_analyzer
```

The algorithm does not need an infinite state. The state length of the FIR filter, hence the algorithm is 12.

Call `dspunfold` with state length set to 12.

```
dspunfold FilterNoise -args {randn(1000,1)} -s 12 -f true
```

```
Analyzing input MATLAB function FilterNoise
Creating single-threaded MEX file FilterNoise_st.mexw64
Creating multi-threaded MEX file FilterNoise_mt.mexw64
Creating analyzer file FilterNoise_analyzer
```

Run the analyzer function.

```
FilterNoise_analyzer(randn(1000*4,1))
```

```
Analyzing multi-threaded MEX file FilterNoise_mt.mexw64 ...
Latency = 8 frames
Speedup = 0.5x
Warning: The output results of the multi-threaded MEX file FilterNoise_mt.mexw64 do not
match the output results of the single-threaded MEX file FilterNoise_st.mexw64. Check that
you provided the correct state length value to the dspunfold function when you generated the
multi-threaded MEX file FilterNoise_mt.mexw64. For best practices and possible solutions to
this problem, see the 'Tips' section in the dspunfold function reference page.
> In coder.internal.warning (line 8)
   In FilterNoise_analyzer
   ans =
       Latency: 8
       Speedup: 0.4970
       Pass: 0
```

The analyzer looks for a numerical match and fails the verification, even though the generated multi-threaded MEX file is valid.

### Speedup Limitations:

- If the entry-point MATLAB function contains code with low complexity, MATLAB overhead or multi-threaded MEX overhead overshadow any performance gains. In such cases, do not use `dspunfold`.
- If the number of operations in the input MATLAB function is small compared to the size of the input or output data, the multi-threaded MEX file does not provide any speedup gain. Sometimes, it can result in a speedup loss, even if the repetition value is increased. In such cases, do not use `dspunfold`.

## More About

### State Length

State length of the algorithm.

Most of the time, the state length used by `dspunfold` matches the state length of the algorithm in the entry-point MATLAB function. If the algorithm is simple, state length is easy to determine. For example, the state length of an FIR filter is the number of taps in the filter - 1. In some scenarios, to optimize speedup, `dspunfold` chooses a state length that is different from the algorithm state length or the state length specified using the `-s` option. For example, when the state length is greater than  $(threads - 1) \times repetition$  frames, `dspunfold` considers the state length to be infinite. Also, multi-threading gets disabled due to performance considerations.

### Automatic State Length Detection

You can automatically detect the minimum state length for which the outputs of the multi-threaded MEX and single-threaded MEX match.

In complex algorithms, it is not easy to determine the state length analytically. In such scenarios, use the analyzer to compute the state length. When you set `-s` to `auto`, `dspunfold` invokes the analyzer. The analyzer computes the outputs for different state lengths and detects the minimum state length for which the outputs of the multi-threaded MEX file and single-threaded MEX file match. The analyzer uses the numeric value of the inputs given to `-args`. To detect the most efficient state length, provide random inputs to `-args`. In this mode, you cannot input `coder.typeof` to arguments. Due to the extra analysis this tool requires, the time to generate the MEX file increases.

When you use automatic state length detection on an algorithm with code paths that depend on the input values, use inputs that choose the code path with the longest state length. Also, the inputs must have an immediate effect on the output. If inputs choose a code path that triggers runtime errors, automatic state length detection stops, and so does the analyzer. Make sure that the MATLAB function supports code generation and does not have run-time errors for the inputs under test. Before invoking `dspunfold`, call `codegen` on the entry-point MATLAB function. In addition, simulate the entry-point MATLAB function to make sure it has no run-time errors.

### Threads

The `-t` option specifies the number of threads used by the multi-threaded MEX file.

Increasing this value can improve the multi-threaded MEX speedup, at the cost of a larger latency. Decreasing this value reduces the latency and potentially decreases the multi-threaded MEX speedup.

### Repetition Factor

*Repetition factor* is the number of consecutive frames processed by each thread in one processing step.

Increasing this value reduces the overhead per frame of data, potentially improving the speedup at the cost of larger latency. Decreasing this value reduces the latency, and potentially decreases the multi-threaded MEX speedup.

### Self-Diagnostic Analyzer

The self-diagnostic analyzer function is a help tool that is generated with the MEX file. This function measures the speedup gain of the multi-threaded MEX file compared to the single-threaded MEX file. The analyzer function also verifies that the outputs of the multi-threaded MEX file and single-threaded MEX file match.

If you specify an incorrect state length value, the outputs usually do not match. To check for the numerical match between the multi-threaded MEX file and the single-threaded MEX file, provide at least two different frames for each input argument of the analyzer. The frames are appended along the first dimension. The analyzer alternates between these frames while verifying that the outputs match. Failure to provide multiple frames for each input can decrease the effectiveness of the analyzer and can lead to false positive verification results. In other words, the analyzer might produce `pass = 1` results even when an incorrect state length value is specified. The analyzer alternates through a maximum of  $3 \times (2 \times \text{threads} \times \text{repetition})$  frames. If your algorithm requires more than  $3 \times (2 \times \text{threads} \times \text{repetition})$  frames to verify the results, then the analyzer cannot verify accurately.

## Tips

### General

- Do not display plots, scopes, or execute other user interface operations from within the multi-threaded MEX file. The generated MEX file can have unexpected behavior.
- Do not use `coder.extrinsic` inside the input MATLAB function. The generated MEX file can have unexpected behavior.

When the state length is less than or equal to  $(threads - 1) \times repetition$  frames:

- Do not use a random number inside the MATLAB function. The outputs of the single-threaded MEX file and the multi-threaded MEX file might not match. Also, the outputs of the consecutive executions of the multi-threaded MEX file might not match. The analyzer might not pass the numerical match verification.

It is recommended that you generate the random number outside the entry-point MATLAB function and pass it as an argument to the function.

- Do not use global or persistent variables anywhere other than in the entry-point MATLAB function. For example, avoid using persistent variables in subfunctions. The generated MEX file can produce inaccurate results. In general, global variables are not recommended.
- Do not access I/O resources from within the multi-threaded MEX file. The generated MEX file can have unexpected behavior. These resources include file writers and readers, UDP sockets, and audio players and recorders.
- Do not use functions with interactive inputs (for example, the keyboard) inside the multi-threaded MEX file. The generated MEX file can have unexpected behavior.

### Workflow

- To generate a valid multi-threaded MEX file with the required speedup and latency, follow the “Workflow for Generating a Multithreaded MEX File using dspunfold”.
- Before using `dspunfold`, call `codegen` on the entry-point MATLAB function and make sure that the function generates a MEX file successfully.
- After generating the multi-threaded MEX file using `dspunfold`, run the analyzer function. Make sure that the analyzer function passes. The exception to this rule is when the algorithm produces results that match statistically, but not numerically. In this exception, the analyzer function does not pass, even though the `dspunfold` function generates a valid multi-threaded MEX file. See 'Analyzer Limitations' for an example.
- For help on using the MEX file and analyzer, at the MATLAB command prompt, enter `help <mexfile name>` and `help <analyzer name>`.

### State Length

- If you choose a state length that is greater than or equal to the value of the exact state length, the analyzer passes. If the analyzer fails, increase the state length, regenerate the MEX file, and verify again.
- If the state length is greater than 0, the inputs marked as frames (through `-f` option) must all have the same dimensions.
- When generating the MEX file and running the analyzer, use inputs that invoke the same state length.

### Automatic State Length Detection

When you set `-s` to `auto`:

- If the algorithm in the entry-point MATLAB function chooses a code path based on the input values, use inputs that choose the code path with the longest state length.
- Provide random inputs to `-args`.
- Choose inputs that have an immediate effect on the output. See “Why Does the Analyzer Choose a Zero State Length?”.

### Analyzer

- Make sure the outputs of the multi-threaded MEX file and the single-threaded MEX file do not contain `NaN` or an `Inf`. The analyzer cannot do numeric checks and returns `pass` as `false`. The automatic state length detection tool detects infinite state length and displays a warning

---

**Warning** The output results of the multi-threaded MEX file do not match the output results of the single-threaded MEX file even for Infinite state length. A possible reason is that input MATLAB function generates different output results between consecutive runs even for the same input values.

---

- Provide multiple frames with different values for each input of the analyzer. To improve the analyzer effectiveness, append successive frames along the first dimension.
- Provide inputs to the analyzer that lead to efficient code coverage.

### Speedup

- To improve the speedup of the multi-threaded MEX file, specify the exact state length in samples. You can specify the state length in samples by setting at least one entry of `frameinputs` to `true`. The use of samples reduces the overhead and increases the speedup.
- To increase the speedup at the cost of larger latency, you can:
  - Increase the repetition factor. Use the `-r` option.
  - Increase the number of threads. Use the `-t` option.
- For each input that can be divided into samples without altering the algorithm behavior, set frame status to `true` using the `-f` option. The input is then considered in samples, which can increase the speedup of the generated multi-threaded MEX file.

## Algorithms

The multi-threaded MEX file buffers multiple-input signal frames into a buffer of  $2 \times \text{threads} \times \text{repetition}$  frames, where *threads* is the number of threads, and *repetition* is the repetition factor. The MEX file processes these frames simultaneously, using multiple cores. This process introduces some deterministic latency, where  $\text{latency} = 2 \times \text{threads} \times \text{repetition}$ . Latency is traded off with the speedup you might gain by increasing the number of threads or the repetition factor.

## Version History

Introduced in R2015b

### See Also

#### Topics

“Multithreaded MEX File Generation”



"How Is dspunfold Different from parfor?"

"Workflow for Generating a Multithreaded MEX File using dspunfold"

"Why Does the Analyzer Choose the Wrong State Length?"

"Why Does the Analyzer Choose a Zero State Length?"

"MATLAB Algorithm Acceleration" (MATLAB Coder)

## ellip

Elliptic filter using specification object

### Syntax

```
ellipFilter = ellip(designSpecs, SystemObject=true)
ellipFilter = ellip(designSpecs, designoption=value, ... SystemObject=true)
ellipFilter = design(designSpecs, 'ellip', SystemObject=true)
```

### Description

`ellipFilter = ellip(designSpecs, SystemObject=true)` designs an elliptical IIR digital filter using the specifications in the object `designSpecs`.

Depending on the filter specification object `designSpecs`, the `ellip` design method might not be valid. Use `designmethods` with the filter specification object to determine if an elliptic IIR filter design is possible.

```
designmethods(designSpecs, SystemObject=true)
```

`ellipFilter = ellip(designSpecs, designoption=value, ... SystemObject=true)` returns an elliptic IIR digital filter with one or more specified designed options and the corresponding values.

To view a list of available design options, run the `designoptions` function on the specification object. The function also lists the default design options the filter uses.

```
designoptions(designSpecs, 'ellip')
```

`ellipFilter = design(designSpecs, 'ellip', SystemObject=true)` is an alternative syntax for designing the elliptic IIR digital filter.

For complete help about using the `ellip` design method for a specification object, `designSpecs`, enter the following at the MATLAB command prompt.

```
help(designSpecs, 'ellip')
```

## Examples

### Design a Bandpass Filter

Design an elliptic digital filter with bandpass frequency response. The filter design procedure is:

- 1 Specify the filter design specifications using a `fdesign` function.
- 2 Pick a design method provided by the `designmethods` function.
- 3 To determine the available design options to choose from, use the `designoptions` function.
- 4 Design the filter using the `design` function.

Construct the default bandpass filter specification object using `fdesign.bandpass`.

```
designSpecs = fdesign.bandpass

designSpecs =
    bandpass with properties:
        Response: 'Bandpass'
        Specification: 'Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2'
        Description: {7x1 cell}
        NormalizedFrequency: 1
        Fstop1: 0.3500
        Fpass1: 0.4500
        Fpass2: 0.5500
        Fstop2: 0.6500
        Astop1: 60
        Apass: 1
        Astop2: 60
```

Determine the available design methods using the `designmethods` function. To design an elliptic digital filter, pick `ellip`.

```
designmethods(designSpecs,SystemObject=true)
```

Design Methods that support System objects for class `fdesign.bandpass` (Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2):

```
butter
cheby1
cheby2
ellip
equiripple
kaiserwin
```

While designing the filter, you can specify additional design options. View a list of the options using the `designoptions` function. This function also shows the default design options the filter uses.

```
designoptions(designSpecs,'ellip',SystemObject=true)
```

```
ans = struct with fields:
    FilterStructure: {'df1sos' 'df2sos' 'df1tsos' 'df2tsos' 'cascadeallpass' 'cascad
    SOSScaleNorm: 'ustring'
    SOSScaleOpts: 'fdopts.sosscaling'
    MatchExactly: {'passband' 'stopband' 'both'}
    DefaultFilterStructure: 'df2sos'
    DefaultMatchExactly: 'both'
    DefaultSOSScaleNorm: ''
    DefaultSOSScaleOpts: [1x1 fdopts.sosscaling]
```

Use the `design` function to design the filter. Pass `'ellip'` and the specifications given by the variable `designSpecs`, as input arguments. Specify the `'matchexactly'` design option to `'both'` to ensure the performance of the filter in both the passband and the stopband regions.

```
bandpassEllip = design(designSpecs,'ellip',matchexactly='both',...
    SystemObject=true)
```

```
bandpassEllip =
    dsp.SOSFilter with properties:
```

```

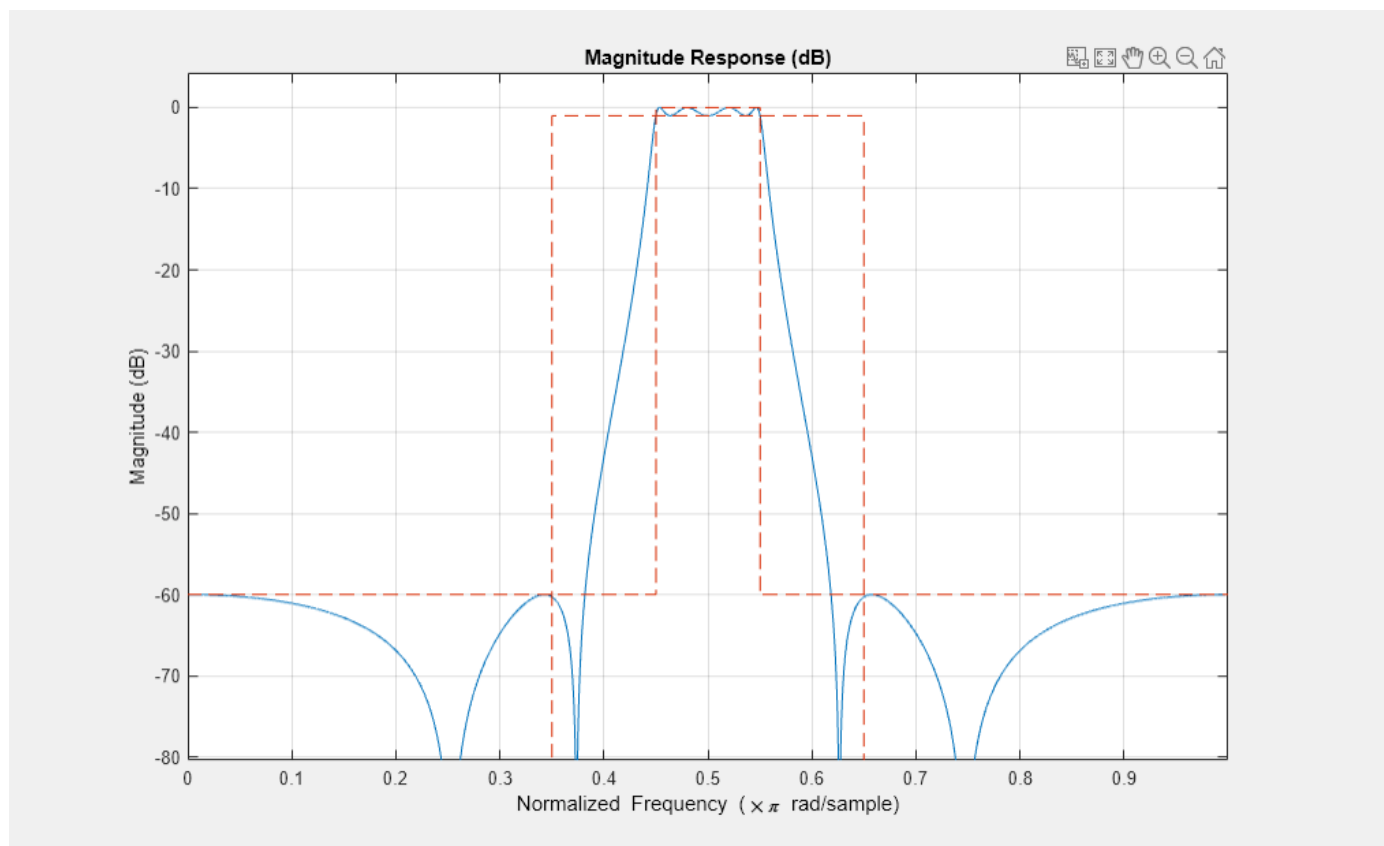
Structure: 'Direct form II'
CoefficientSource: 'Property'
Numerator: [4x3 double]
Denominator: [4x3 double]
HasScaleValues: true
ScaleValues: [0.4705 0.4705 0.0936 0.0936 1]

```

Show all properties

Use `fvtool` to view the frequency response of the designed filter.

```
fvtool(bandpassEllip)
```



### Design Lowpass Filter

Design an elliptic digital filter with lowpass frequency response.

Construct a lowpass filter specification object using `fdesign.lowpass`. Specify the filter order, passband edge frequency, stopband edge frequency, and the passband ripple of the filter.

```
designSpecs = fdesign.lowpass('n,fp,fst,ap',6,20,25,.8,80);
```

Determine the available design methods. To design an elliptic filter, pick `ellip`.

```
designmethods(designSpecs,SystemObject=true)
```

Design Methods that support System objects for class `fdesign.lowpass (N,Fp,Fst,Ap)`:

```
ellip
equiripple
```

Use the `design` function to design the filter. Pass `'ellip'` and the specifications given by the variable `designSpecs`, as input arguments.

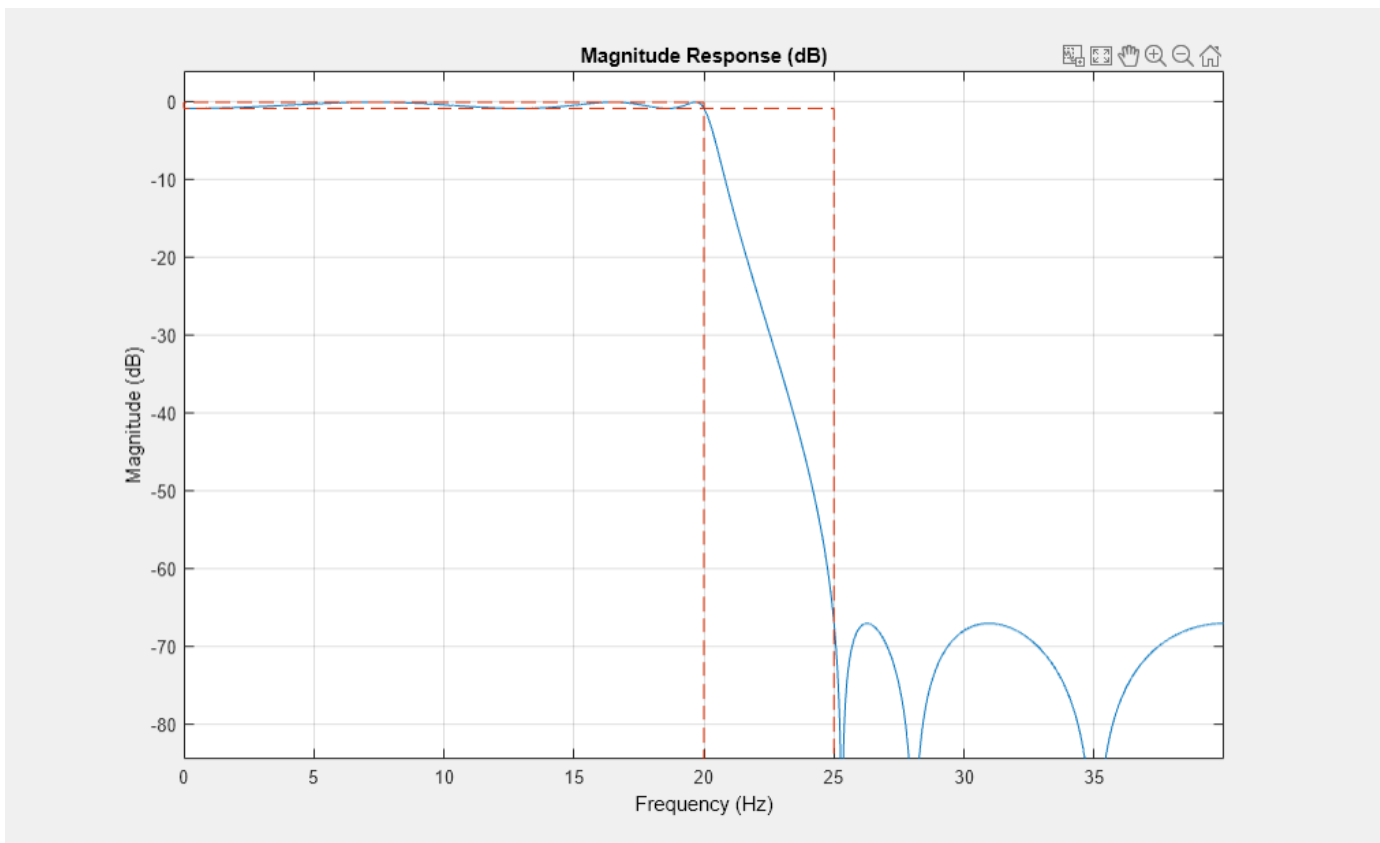
```
ellipLowpass = design(designSpecs,'ellip',SystemObject=true)
```

```
ellipLowpass =
  dsp.SOSFilter with properties:
    Structure: 'Direct form II'
    CoefficientSource: 'Property'
    Numerator: [3x3 double]
    Denominator: [3x3 double]
    HasScaleValues: true
    ScaleValues: [0.0968 0.4116 0.6792 1]
```

Show all properties

Use `fvtool` to visualize the lowpass frequency response.

```
fvtool(ellipLowpass)
```



## Input Arguments

### **designSpecs** — Filter specification object

object

Filter specification object, specified as one of the following:

- `fdesign.bandpass`
- `fdesign.bandstop`
- `fdesign.decimator`
- `fdesign.halfband`
- `fdesign.highpass`
- `fdesign.hilbert`
- `fdesign.interpolator`
- `fdesign.lowpass`
- `fdesign.nyquist`

## Output Arguments

### **ellipFilter** — Elliptical digital filter

System object

Elliptical digital filter, returned as a filter System object. The System object and the values of its properties depend on the input `designSpecs` object and the other design options specified to the function.

## Version History

**Introduced in R2011a**

### **R2022b: Designs `dsp.SOSFilter` object instead of `dsp.BiquadFilter` object**

*Behavior changed in R2022b*

The `ellip` design method now generates a `dsp.SOSFilter` object instead of the `dsp.BiquadFilter` object.

Initialize the default `fdesign.lowpass` filter specification object. Use the `design` function with the `SystemObject=true` flag to design a `dsp.SOSFilter` filter object.

```
designSpecs = fdesign.lowpass;  
filtObj = design(designSpecs, 'ellip', SystemObject=true)
```

```
filtObj =
```

```
    dsp.SOSFilter with properties:
```

```
        Structure: 'Direct form II'  
    CoefficientSource: 'Property'  
        Numerator: [3×3 double]  
        Denominator: [3×3 double]
```

```
HasScaleValues: true
ScaleValues: [0.8629 2.0523 0.0127 1]
```

Show all properties

To design a `dsp.BiquadFilter` object instead of the `dsp.SOSFilter` object, you can use the `UseLegacyBiquadFilter` flag and set it to `true`.

```
filtObj = design(designSpecs, 'ellip', SystemObject=true, ...
UseLegacyBiquadFilter=true)
```

```
filtObj =
```

`dsp.BiquadFilter` with properties:

```
Structure: 'Direct form II'
SOSMatrixSource: 'Property'
SOSMatrix: [3x6 double]
ScaleValues: [4x1 double]
InitialConditions: 0
OptimizeUnityScaleValues: true
```

Show all properties

## See Also

[butter](#) | [cheby1](#) | [cheby2](#) | [design](#) | [designmethods](#) | [designoptions](#) | [fdesign](#) | [equiripple](#) | [kaiserwin](#)

## euclidfactors

Euclid factors for multirate filter

### Compatibility

`mfilt` will be removed in a future release. See `dsp.CICDecimator`, `dsp.CICInterpolator`, `dsp.FIRDecimator`, `dsp.FIRInterpolator`, `dsp.FilterCascade`, `dsp.FarrowRateConverter`, `dsp.FIRRateConverter`, `dsp.IIRHalfbandDecimator`, or `dsp.IIRHalfbandInterpolator` instead.

### Syntax

```
[lo,mo] = euclidfactors(hm)
```

### Description

`[lo,mo] = euclidfactors(hm)` returns integer factors `lo` and `mo` such that  $(lo*L)-(mo*M) = -1$ . `L` and `M` are relatively prime and represent the interpolation and decimation factors of the multirate filter `hm`.

`euclidfactors` works with the multirate filter `mfilt.firsc`. You cannot return `lo` and `mo` for decimators or interpolators.

### Examples

Use an FIR fractional decimator, with `L = 5` and `M = 7`, to show what `euclidfactors` does.

Indeed,  $(lo*L)-(mo*M) = (4*5)-(3*7) = -1$ .

## Version History

Introduced in R2011a

### See Also

`polyphase`



# equiripple

Equiripple single-rate FIR filter from specification object

## Syntax

```
equiFilt = design(d,'equiripple','SystemObject',true)
equiFilt = design(d,'equiripple',designoption,value,...,'SystemObject',true)
```

## Description

`equiFilt = design(d,'equiripple','SystemObject',true)` designs an equiripple FIR digital filter using the specifications supplied in the object `d`. Equiripple filter designs minimize the maximum ripple in the passbands and stopbands.

When you use `equiripple` with Nyquist filter specification objects, you might encounter design cases where the filter design does not converge. Convergence errors occur mostly at large filter orders, or small transition widths, or large stopband attenuations. These specifications, alone or combined, can cause design failures. For more information, refer to `fdesign.nyquist` in the online Help system.

`equiFilt = design(d,'equiripple',designoption,value,...,'SystemObject',true)` returns an equiripple FIR filter where you specify design options as input arguments.

To determine the available design options, use `designopts` with the specification object and the design method as input arguments as shown.

```
designopts(d,'method')
```

For complete help about using `equiripple`, refer to the command line help system. For example, to get specific information about using `equiripple` with `d`, the specification object, enter the following at the MATLAB prompt.

```
help(d,'equiripple')
```

## Examples

### Design a Single-Rate Equiripple Filter

Design a single-rate equiripple filter from a halfband filter specification object. Notice the help command used to learn about the options for the specification object and method.

```
d = fdesign.halfband('tw,ast',0.1,80);
designmethods(d,'Systemobject',true)
```

Design Methods that support System objects for class `fdesign.halfband` (TW,Ast):

```
butter
ellip
```

```
iirlinphase
equiripple
kaiserwin
```

```
help(d,'equiripple')
```

DESIGN Design a Equiripple FIR filter.

HD = DESIGN(D, 'equiripple') designs a Equiripple filter specified by the FDESIGN object D, and returns the DFILT/MFILT object HD.

HD = DESIGN(D, ..., 'SystemObject', true) implements the filter, HD, using a System object instead of a DFILT/MFILT object.

HD = DESIGN(..., 'FilterStructure', STRUCTURE) returns a filter with the structure STRUCTURE. STRUCTURE is 'dffir' by default and can be any of the following:

```
'dffir'
'dffirt'
'dfsymfir'
'fftfir'
```

Some of the listed structures may not be supported by System object filters. Type `validstructures(D, 'equiripple', 'SystemObject', true)` to get a list of structures supported by System objects.

HD = DESIGN(..., 'MinPhase', MPHASE) designs a minimum-phase filter when MPHASE is TRUE. MPHASE is FALSE by default.

HD = DESIGN(..., 'StopbandShape', SHAPE) designs a filter whose stopband has the shape defined by SHAPE. SHAPE can be 'flat', '1/f', or 'linear'. SHAPE is 'flat' by default.

HD = DESIGN(..., 'StopbandDecay', DECAY) specifies the decay to use when 'StopbandShape' is not set to 'flat'. When the shape is '1/f' this specifies the power that 1/f is raised. When shaped is 'linear' this specifies the slope of the stopband in dB/rad/s.

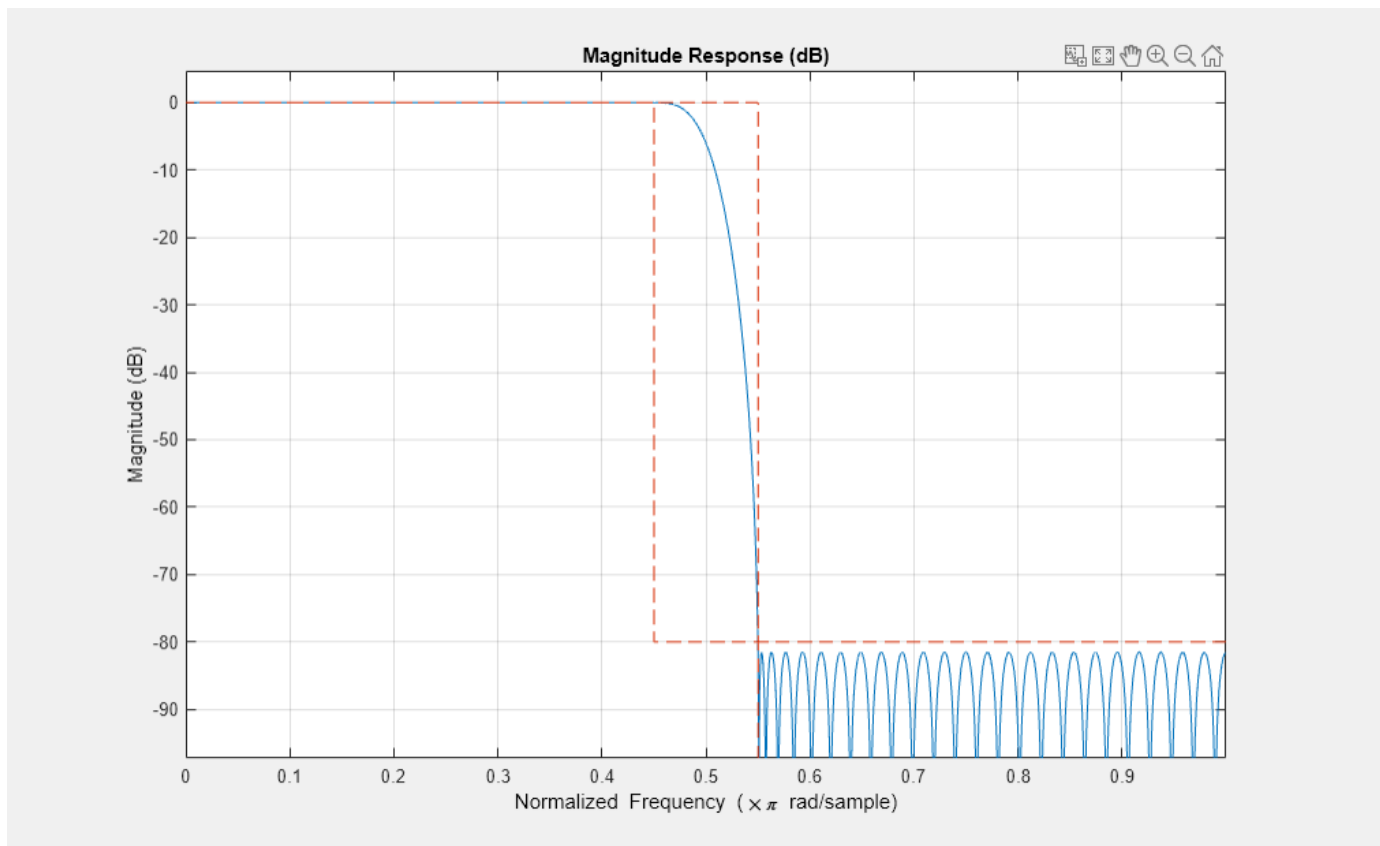
% Example #1 - Design a halfband lowpass equiripple filter with increased stopband attenuation

```
TW = 0.1; % Transition Width
Ast = 80; % Stopband Attenuation (dB)
h = fdesign.halfband('Type','Lowpass','TW,Ast',TW,Ast);
Hd = design(h, 'equiripple', 'StopbandShape','linear','StopbandDecay',50);
fvtool(Hd)
```

```
designopts(d,'equiripple')
```

```
ans = struct with fields:
  FilterStructure: 'dffir'
  MinPhase: 0
  StopbandShape: 'flat'
  StopbandDecay: 0
  SystemObject: 0
```

```
equiFilt = design(d,'equiripple','stopbandshape','flat','SystemObject',true);
fvtool(equiFilt);
```



The `fvtool` shows the equiripple nature of the filter.

### Design an Equiripple FIR Filter with a Direct-Form Transposed Structure

This example designs an equiripple filter with a direct-form transposed structure by specifying the 'FilterStructure' argument. To set the design options for the filter, use the `designopts` method and options object `opts`.

```
d = fdesign.lowpass('fp,fst,ap,ast');
opts = designopts(d,'equiripple');
opts.FilterStructure='dffirt';
opts.DensityFactor=20
```

```
opts = struct with fields:
  FilterStructure: 'dffirt'
  DensityFactor: 20
  MinPhase: 0
  MaxPhase: 0
  MinOrder: 'any'
  StopbandShape: 'flat'
  StopbandDecay: 0
  UniformGrid: 1
  SystemObject: 0
```

```
firFilt = design(d,'equiripple','SystemObject',true,opts)
```

```
firFilt =
```

```
  dsp.FIRFilter with properties:
```

```
    Structure: 'Direct form transposed'
```

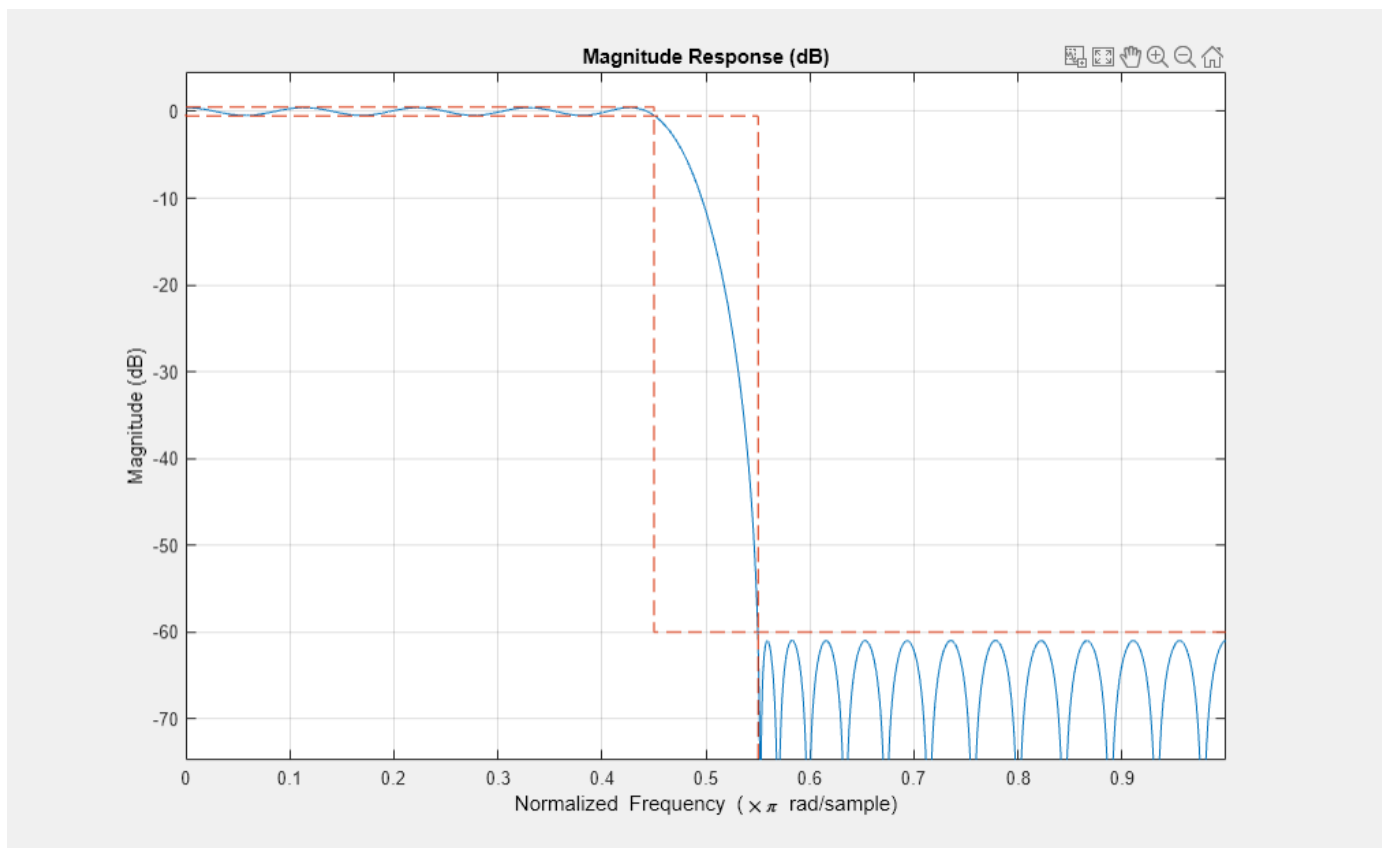
```
    NumeratorSource: 'Property'
```

```
    Numerator: [-0.0024 -0.0021 0.0068 0.0167 0.0111 -0.0062 -0.0084 0.0093 0.0130 -0.0101]
```

```
    InitialConditions: 0
```

```
Show all properties
```

```
fvtool(firFilt);
```



The MaxPhase design option for equiripple FIR filters is currently only available for lowpass, highpass, bandpass, and bandstop filters.

## Version History

Introduced in R2011a

## See Also

### Functions

fdesign.nyquist | firls | kaiserwin

## fcfwrite

Write file containing filter coefficients

### Syntax

```
fcfwrite(h)
fcfwrite(h,filename)
fcfwrite(...,'fmt')
```

### Description

`fcfwrite(h)` writes a filter coefficient ASCII file in a folder you choose, or your current MATLAB working folder. `h` can be a single filter object or a vector of filter objects. On execution, `fcfwrite` opens the **Export Filter Coefficients to .FCF File** dialog box to let you assign a file name for the output file. You can choose the destination folder within this dialog as well.

The default file name is `untitled.fcf`. When you have DSP System Toolbox software, you can use `fcfwrite(h)` to write filter coefficient files for multirate filters, adaptive filters, and discrete-time filters.

`fcfwrite(h,filename)` writes the filter coefficients and general information to a text file called `filename` in your present MATLAB working folder and opens the file in the MATLAB editor for you to review or modify.

If you do not include a file extension in `filename`, `fcfwrite` adds the extension `fcf` to `filename`.

`fcfwrite(...,'fmt')` writes the filter coefficients in the format specified by the input argument `fmt`. Valid `fmt` values are `hex` for hexadecimal, `dec` for decimal, or `bin` for binary representation of the filter coefficients.

### Examples

To demonstrate `fcfwrite`, create a fixed-point IIR filter at the command line, and then write the filter coefficients to a file named `iirfilter.fcf`.

```
d=fdesign.lowpass;
hd=design(d,'butter');
set(hd,'arithmetic','fixed');
fcfwrite(hd,'iirfilter.fcf');
```

Here is the output from `fcfwrite` as it appears in the MATLAB editor. Not shown here is the filename — `iirfilter.fcf` as specified and some comments at the top of the file.

```
%
%
% Coefficient Format: Decimal
%
% Discrete-Time IIR Filter (real)
% -----
% Filter Structure      : Direct-Form II, Second-Order
%                               Sections
```

```

% Number of Sections : 13
% Stable : Yes
% Linear Phase : No
% Arithmetic : fixed
% Numerator : s16,13 -> [-4 4)
% Denominator : s16,14 -> [-2 2)
% Scale Values : s16,14 -> [-2 2)
% Input : s16,15 -> [-1 1)
% Section Input : s16,8 -> [-128 128)
% Section Output : s16,10 -> [-32 32)
% Output : s16,10 -> [-32 32)
% State : s16,15 -> [-1 1)
% Numerator Prod : s32,28 -> [-8 8)
% Denominator Prod : s32,29 -> [-4 4)
% Numerator Accum : s40,28 -> [-2048 2048)
% Denominator Accum : s40,29 -> [-1024 1024)
% Round Mode : convergent
% Overflow Mode : wrap
% Cast Before Sum : true

```

SOS matrix:

```

1 2 1 1 -0.22222900390625 0.88262939453125
1 2 1 1 -0.19903564453125 0.68621826171875
1 2 1 1 -0.18060302734375 0.5303955078125
1 2 1 1 -0.1658935546875 0.40570068359375
1 2 1 1 -0.154052734375 0.305419921875
1 2 1 1 -0.14453125 0.22479248046875
1 2 1 1 -0.136962890625 0.16015625
1 2 1 1 -0.13092041015625 0.10906982421875
1 2 1 1 -0.126220703125 0.06939697265625
1 2 1 1 -0.12274169921875 0.0399169921875
1 2 1 1 -0.12030029296875 0.01947021484375
1 2 1 1 -0.118896484375 0.0074462890625
1 1 0 1 -0.0592041015625 0

```

Scale Values:

```

0.41510009765625
0.371826171875
0.33746337890625
0.3099365234375
0.287841796875
0.27008056640625
0.25579833984375
0.2445068359375
0.23577880859375
0.22930908203125
0.22479248046875
0.22216796875
0.47039794921875
1

```

To write two or more filters out to one file, provide the filters as a vector to `fcfwrite`:

```
fcfwrite([hd hd1 hd2])
```

## **Version History**

**Introduced in R2011a**



# filterDesigner

Open Filter Designer app

## Syntax

```
filterDesigner
```

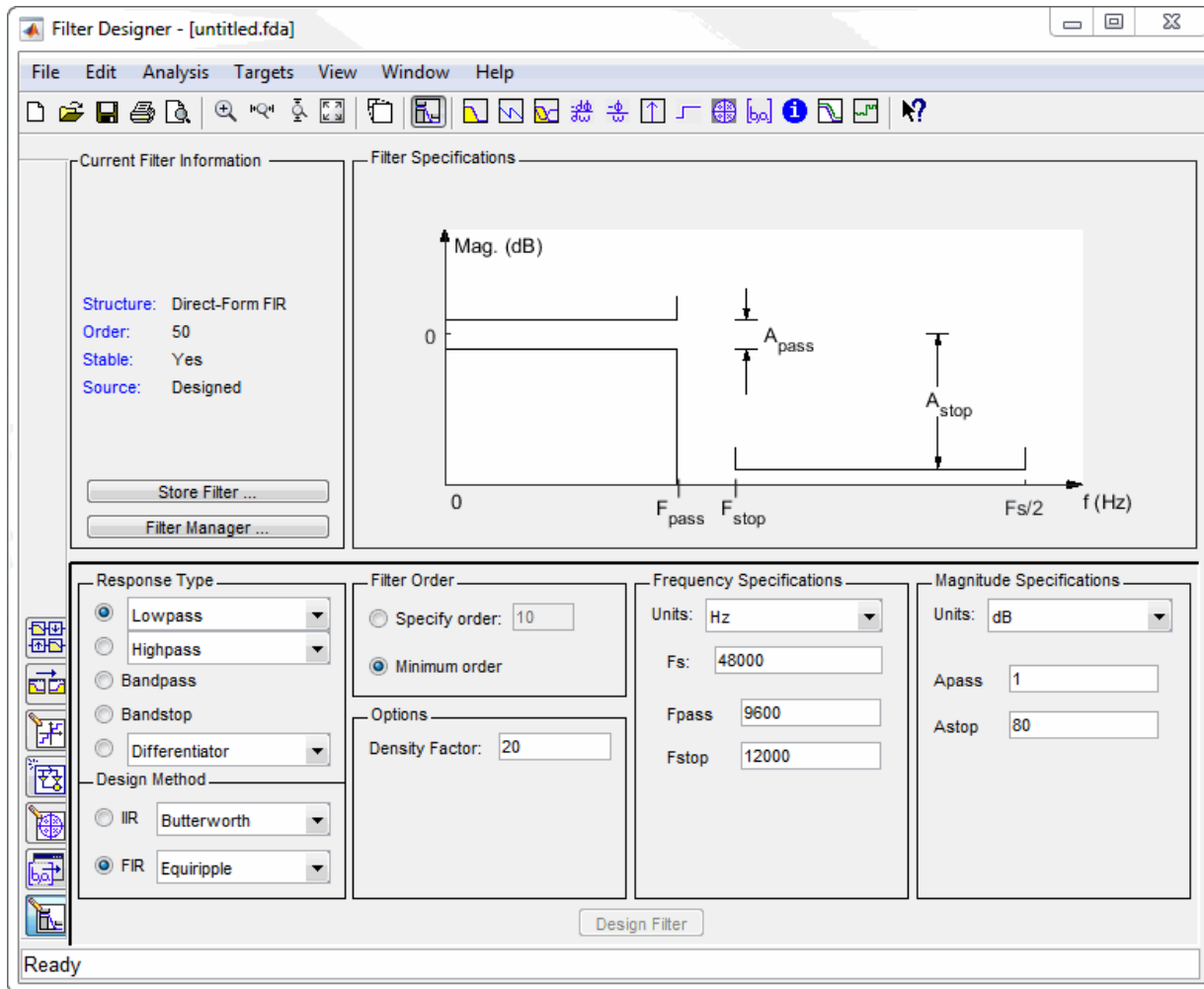
## Description

`filterDesigner` opens the Filter Designer app. Use this tool to:

- Design filters
- Quantize filters (with DSP System Toolbox software installed)
- Analyze filters
- Modify existing filter designs
- Create multirate filters (with DSP System Toolbox software installed)
- Realize Simulink models of quantized, direct-form, FIR filters (with DSP System Toolbox software installed)
- Perform digital frequency transformations of filters (with DSP System Toolbox software installed)

Refer to “Use Filter Designer with DSP System Toolbox Software” for more information about using the analysis, design, and quantization features of filter designer. For general information about using filter designer, refer to “Using Filter Designer”.

When you open the filter designer app and you have DSP System Toolbox software installed, filter designer incorporates features that are added by DSP System Toolbox software. With DSP System Toolbox software installed, filter designer lets you design and analyze quantized filters, as well as convert quantized filters to various filter structures, transform filters, design multirate filters, and realize models of filters.



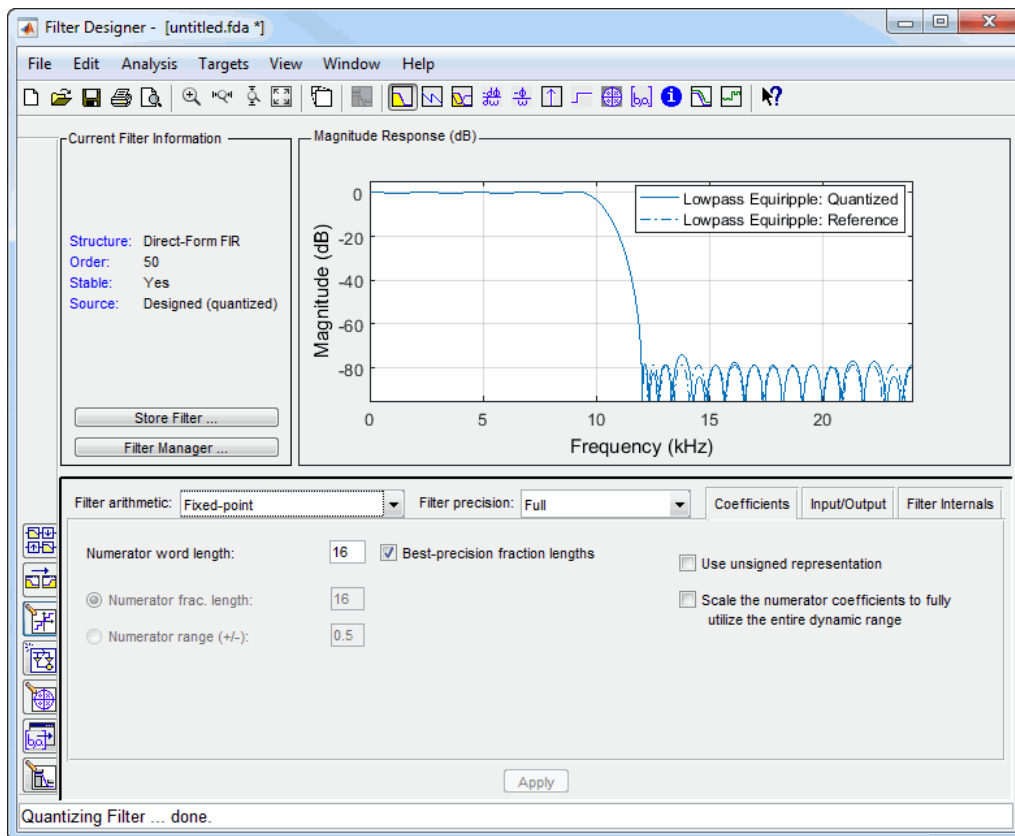
Use the buttons on the sidebar to configure the design area to use various tools in the filter designer app.

**Set Quantization Parameters** — provides access to the properties of the quantizers that compose a quantized filter. When you click **Set Quantization Parameters**, you see filter designer displaying the quantization options at the bottom of the dialog box (the design area), as shown in the figure.

**Transform Filter** — clicking this button opens the *Frequency Transformations* pane so you can use digital frequency transformations to change the magnitude response of your filter.

**Create a multirate filter** — clicking this button switches filter designer to multirate filter design mode so you can design interpolators, decimators, and fractional rate change filters.

**Realize Model** — starting from your quantized, direct-form, FIR filter, clicking this button creates a Simulink model of your filter structure in new model window.



Other options in the menu bar let you convert the filter structure to a new structure, change the order of second-order sections in a filter, or change the scaling applied to the filter, among many possibilities.

## Limitations

- The **Input/Output** section in the **Set quantization parameters** panel of the filterDesigner app supports only signed data types.
- The word length value specified by the **Input word length** and the **Output word length** settings in the **Input/Output** section of the **Set quantization parameters** panel must be 2 or more to support signed numeric types.

## Examples

- “Use Filter Designer with DSP System Toolbox Software”

## Version History

Introduced in R2011a

## See Also

**Apps**  
filterDesigner

**Tools**  
**FVTool**

# fdesign

Filter design specification object

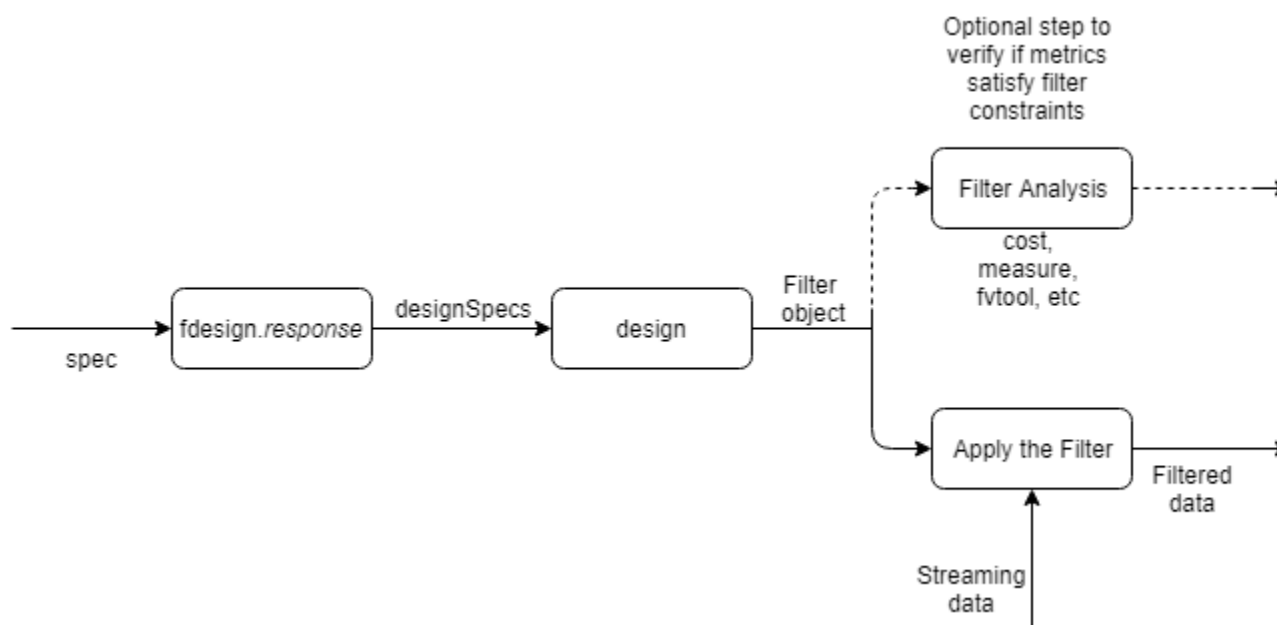
## Syntax

```
designSpecs = fdesign.response
designSpecs = fdesign.response(spec)
designSpecs = fdesign.response( __ ,Fs)
designSpecs = fdesign.response( __ ,magunits)
```

## Description

Use the `fdesign` function to create a filter design specification object that contains the specifications for a filter, such as passband ripple, stopband attenuation, and filter order. Then, use the `design` function to design the filter from the filter design specifications object. For an example, see “Design of Lowpass Decimator” on page 5-390.

Here is the workflow diagram that shows the simple procedure to design, analyze, and finally apply the filter on streaming data.



For more control options, see “Filter Design Procedure” on page 5-399. For a complete workflow, see “Design a Filter in Fdesign — Process Overview”.

`designSpecs = fdesign.response` returns a design specification object for the filter with a given *response*.

`designSpecs = fdesign.response(spec)` specifies the variables to use that define your filter design. The filter design parameters are applied to the filter design method you choose for your filter. The specification option you choose determines which design methods apply to the `fdesign` object.

`designSpecs = fdesign.response( ____, Fs)` specifies the sample rate in Hz to use in the filter specifications. The sample rate scalar must be the last input argument. If you specify a sample rate, all frequency specifications are in Hz.

`designSpecs = fdesign.response( ____, magunits)` specifies the units for any magnitude specification you provide in the input arguments.

## Examples

### Design of Lowpass Decimator

Design a 100-tap FIR lowpass decimator filter that reduces the sample rate of a signal from 60 kHz to 20 kHz. The passband of the filter extends up to 6 kHz. Specify a passband ripple of 0.01 dB and a stopband attenuation of 100 dB.

```

Fs = 60e3;
N = 99;
Fpass = 6e3;
Apass = 0.01;
Astop = 100;
M = Fs/20e3;

```

Setup the filter design specifications object using the `fdesign.decimator` function.

```

filtSpecs = fdesign.decimator(M, 'lowpass', 'N,Fp,Ap,Ast', N, Fpass, Apass, Astop, Fs);

```

Design the FIR lowpass decimator using the `design` function.

The resulting filter is a `dsp.FIRDecimator` System object™. For details on how to apply this filter to streaming data, refer to `dsp.FIRDecimator`.

```

decimFIR = design(filtSpecs, 'SystemObject', true)

```

```

decimFIR =
  dsp.FIRDecimator with properties:

```

```

  Main

```

```

  DecimationFactor: 3

```

```

  NumeratorSource: 'Property'

```

```

    Numerator: [-1.5100e-05 -2.2164e-05 -9.6058e-06 4.3636e-05 1.3739e-04 2.3440e-04 2.63

```

```

    Structure: 'Direct form'

```

```

  Show all properties

```

Use `info` to display information about the filter.

```

info(decimFIR)

```

```

ans = 10x56 char array

```

```

'Discrete-Time FIR Multirate Filter (real)

```

```

'-----

```

```

'Filter Structure : Direct-Form FIR Polyphase Decimator'

```

```

'Decimation Factor : 3

```

```

'Polyphase Length : 34

```

```

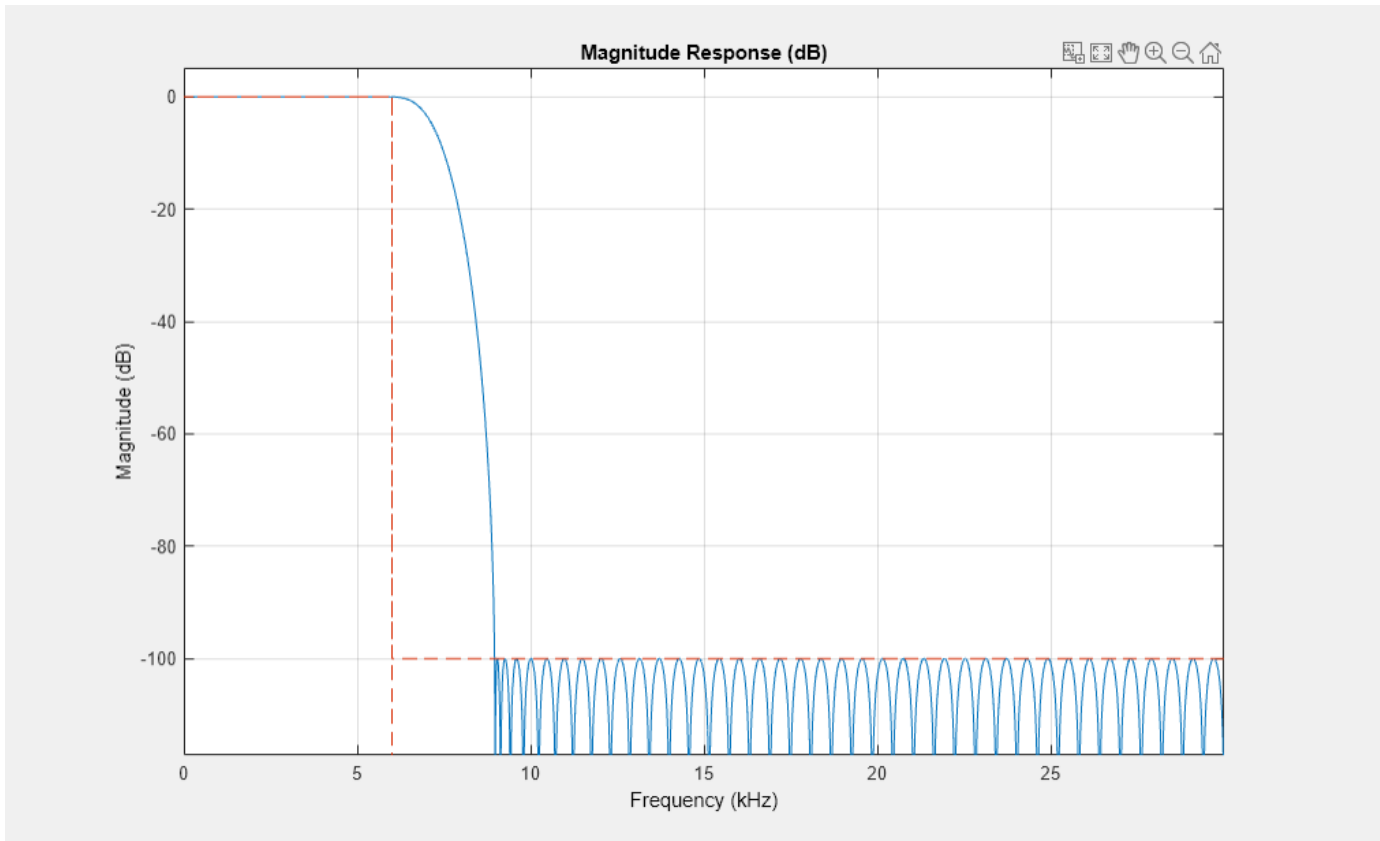
'Filter Length : 100

```

```
'Stable           : Yes           '
'Linear Phase     : Yes (Type 2)  '
'Arithmetic       : double       '
'
```

Visualize the magnitude response of the filter using `fvtool`.

```
fvtool(decimFIR, 'Fs',Fs)
```



### Design of Lowpass Filter

Design a lowpass filter to use on a signal sampled at 96 kHz. The passband of the filter extends up to 20 kHz. The stopband of the filter starts at 24 kHz. Specify a passband ripple of 0.01 dB and a stopband attenuation of 80 dB. Determine automatically the order required to meet the specifications.

Set up the filter design specifications object using the `fdesign.lowpass` function.

```
Fs = 96e3;
Fpass = 20e3;
Fstop = 24e3;
Apass = 0.01;
Astop = 80;
```

```
filtSpecs = fdesign.lowpass(Fpass,Fstop,Apass,Astop,Fs);
```

Determine the available design algorithms using the `designmethods` function.

```
designmethods(filtSpecs, 'SystemObject', true)
```

```
Design Methods that support System objects for class fdesign.lowpass (Fp,Fst,Ap,Ast):
```

```
butter
cheby1
cheby2
ellip
equiripple
ifir
kaiserwin
multistage
```

Using the design function, design an equiripple FIR filter and an elliptic IIR filter that meet the specifications.

```
lpFIR = design(filtSpecs, 'equiripple', 'SystemObject', true)
```

```
lpFIR =
  dsp.FIRFilter with properties:
      Structure: 'Direct form'
      NumeratorSource: 'Property'
      Numerator: [1.0908e-04 2.1016e-05 -2.3369e-04 -2.8798e-04 9.0899e-05 3.6470e-04 -5.32e-05]
      InitialConditions: 0
```

```
Show all properties
```

```
lpIIR = design(filtSpecs, 'ellip', 'SystemObject', true)
```

```
lpIIR =
  dsp.SOSFilter with properties:
      Structure: 'Direct form II'
      CoefficientSource: 'Property'
      Numerator: [5x3 double]
      Denominator: [5x3 double]
      HasScaleValues: true
      ScaleValues: [0.8017 0.7532 0.7030 1.2976 0.0087 1]
```

```
Show all properties
```

You can also measure the designs to verify that the filters satisfy the constraints.

```
FIRmeas = measure(lpFIR)
```

```
FIRmeas =
  Sample Rate      : 96 kHz
  Passband Edge    : 20 kHz
  3-dB Point       : 21.4297 kHz
  6-dB Point       : 21.8447 kHz
```



```

Stopband Edge      : 24 kHz
Passband Ripple    : 0.0092309 dB
Stopband Atten.    : 80.6014 dB
Transition Width    : 4 kHz

```

```
IIRmeas = measure(lpIIR)
```

```

IIRmeas =
Sample Rate        : 96 kHz
Passband Edge      : 20 kHz
3-dB Point         : 20.5524 kHz
6-dB Point         : 20.7138 kHz
Stopband Edge      : 24 kHz
Passband Ripple    : 0.01 dB
Stopband Atten.    : 80 dB
Transition Width    : 4 kHz

```

Estimate and display the computational cost of each filter. The equiripple FIR filter requires many more coefficients than the elliptic IIR filter.

```
FIRcost = cost(lpFIR)
```

```

FIRcost = struct with fields:
    NumCoefficients: 101
    NumStates: 100
    MultiplicationsPerInputSample: 101
    AdditionsPerInputSample: 100

```

```
IIRcost = cost(lpIIR)
```

```

IIRcost = struct with fields:
    NumCoefficients: 20
    NumStates: 10
    MultiplicationsPerInputSample: 20
    AdditionsPerInputSample: 20

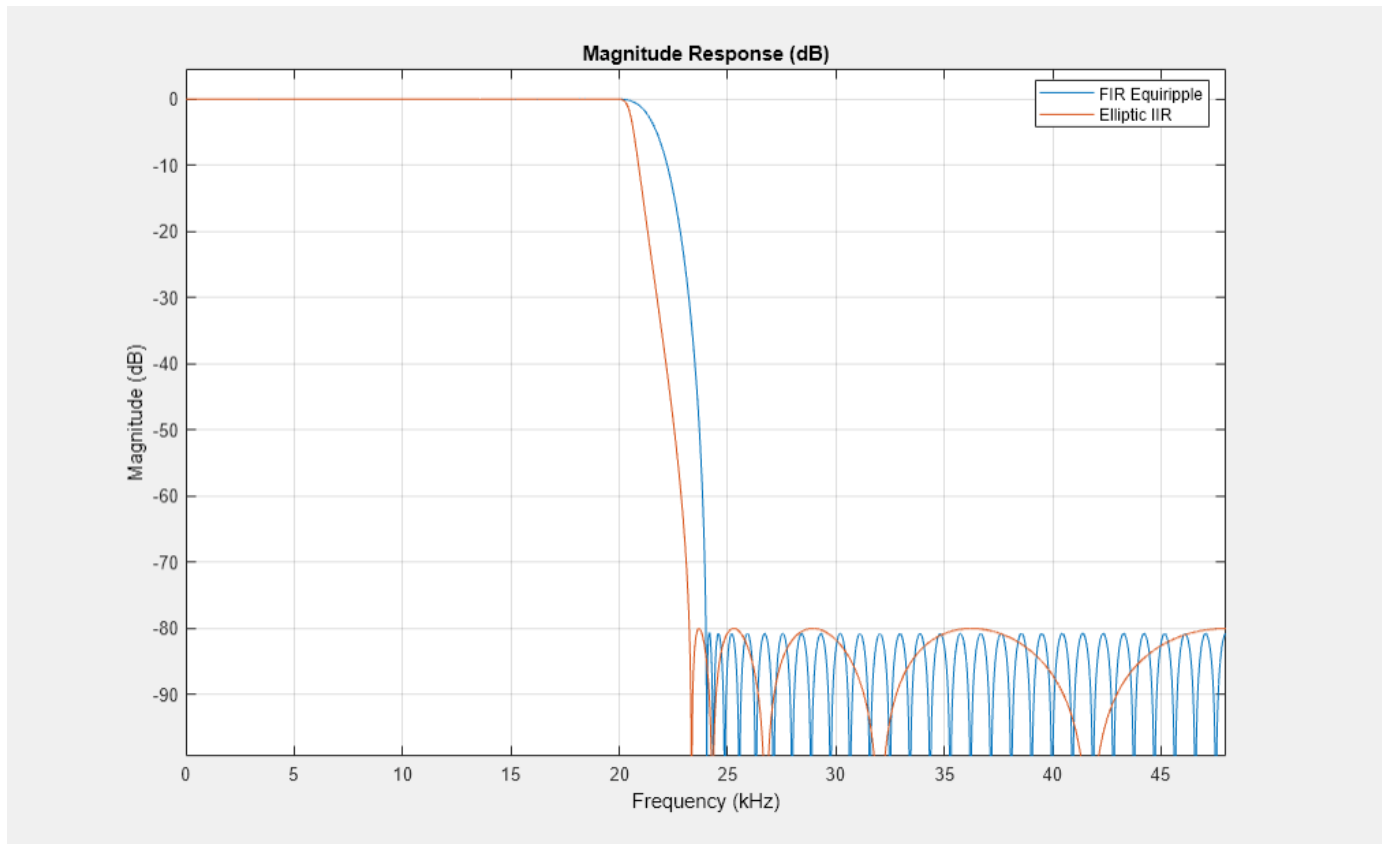
```

Use `fvtool` function to visualize the resulting designs and compare their properties.

```

hvft = fvtool(lpFIR,lpIIR,'Fs',Fs);
legend(hvft,'FIR Equiripple','Elliptic IIR')

```



### Lowpass Butterworth Filter Specification and Design

Design a lowpass Butterworth filter that has a passband edge frequency of  $0.4\pi$  rad/sample, a stopband frequency of  $0.5\pi$  rad/sample, a passband ripple of 1 dB, and a stopband attenuation of 80 dB.

Create a `lowpass` filter design specification object using the `fdesign.lowpass` function. Specify the design parameters.

```
lowpassSpecs = fdesign.lowpass(0.4,0.5,1,80);
```

To view a list of design methods available for the specification object, use the `designmethods` function. If multiple methods are available, pick one that best meets the design criteria. For this example, pick `'butter'`.

```
designmethods(lowpassSpecs, SystemObject=true)
```

Design Methods that support System objects for class `fdesign.lowpass (Fp,Fst,Ap,Ast)`:

```
butter
cheby1
cheby2
ellip
```

```

equiripple
ifir
kaiserwin
multistage

```

Furthermore, you can specify the design options used in designing the filter. To see a list of available options, run the `designoptions` function on `lowpassSpecs`. The design options are dependent on the design method you pick. The design method, in this case, `'butter'`, must be specified as an argument to the `designoptions` function.

```
designoptions(lowpassSpecs,"butter",Systemobject=true)
```

```

ans = struct with fields:
    FilterStructure: {'df1sos' 'df2sos' 'df1tsos' 'df2tsos' 'cascadeallpass' 'cascadeallpass'}
    SOSScaleNorm: 'ustring'
    SOSScaleOpts: 'fdopts.sosscaling'
    MatchExactly: {'passband' 'stopband'}
    DefaultFilterStructure: 'df2sos'
    DefaultMatchExactly: 'stopband'
    DefaultSOSScaleNorm: ''
    DefaultSOSScaleOpts: [1x1 fdopts.sosscaling]

```

The filter order necessary to meet a set of design constraints must also be rounded up to an integer value. This loosens some of the constraints, and as a consequence, some design specifications are met while others are exceeded. The `'MatchExactly'` option allows you to match the passband or stopband exactly while exceeding the specification for the other band. Design the filter so that it matches the passband exactly.

The resulting filter is a `dsp.SOSFilter` System object™. For details on how to apply this filter on streaming data, refer to `dsp.SOSFilter`.

```
IIRbutter = design(lowpassSpecs,"butter",MatchExactly="passband", ...
    SystemObject=true)
```

```

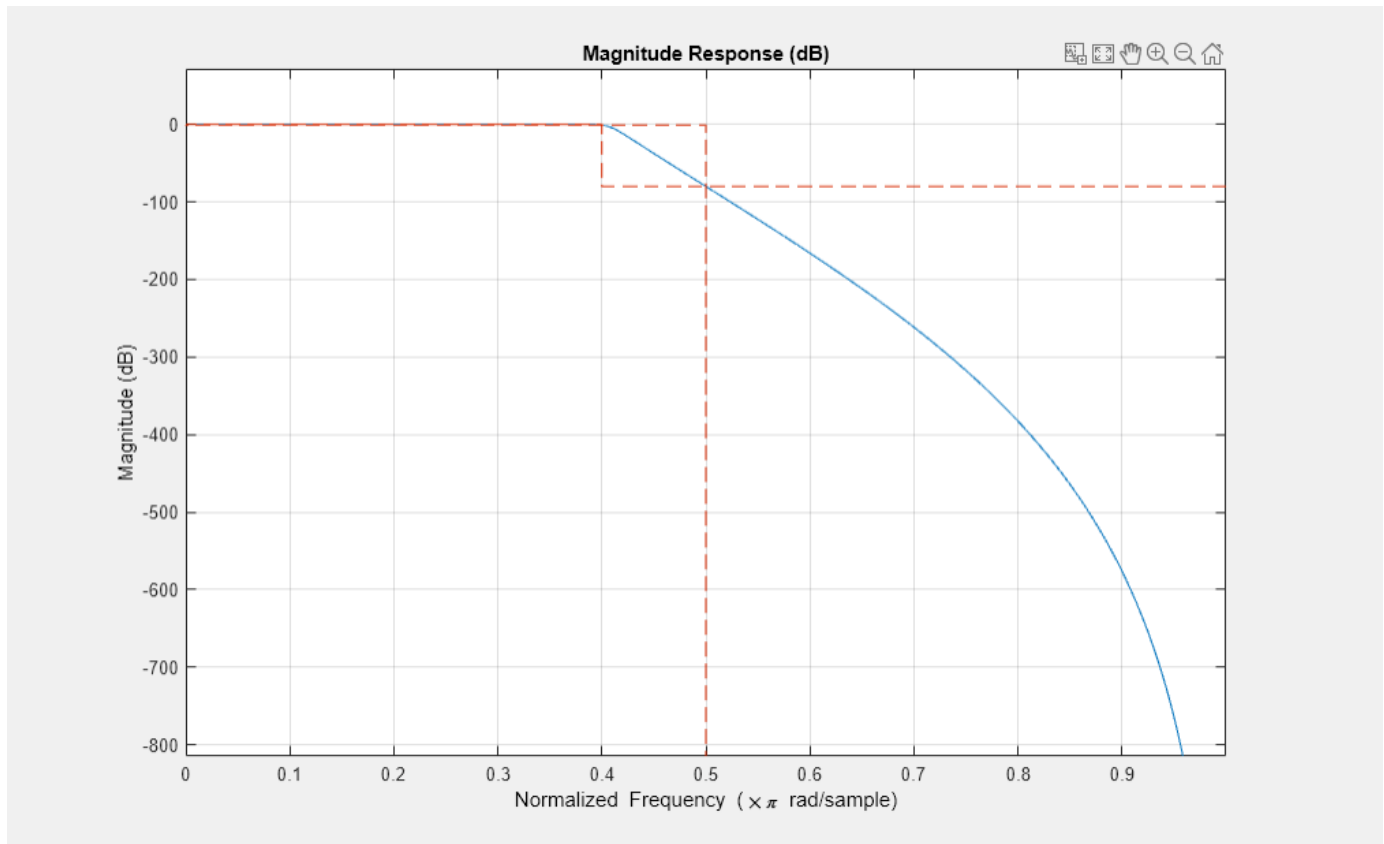
IIRbutter =
    dsp.SOSFilter with properties:
        Structure: 'Direct form II'
        CoefficientSource: 'Property'
        Numerator: [16x3 double]
        Denominator: [16x3 double]
        HasScaleValues: true
        ScaleValues: [0.3390 0.3104 0.2866 0.2667 0.2500 0.2359 0.2241 0.2142 0.2059 0.1990 0.1930 0.1870 0.1810 0.1750 0.1690 0.1630 0.1570 0.1510 0.1450 0.1390 0.1330 0.1270 0.1210 0.1150 0.1090 0.1030 0.0970 0.0910 0.0850 0.0790 0.0730 0.0670 0.0610 0.0550 0.0490 0.0430 0.0370 0.0310 0.0250 0.0190 0.0130 0.0070 0.0010]

```

```
Show all properties
```

Use `fvtool` to visualize the magnitude response of the filter.

```
fvtool(IIRbutter)
```



## Input Arguments

### response — Desired filter response

response entry

The table specifies the possible filter responses.

| <b>fdesign Response Method</b> | <b>Description</b>   |
|--------------------------------|--|
| arbgrpdelay                    | <code>fdesign.arbgrpdelay</code> creates an object to specify allpass arbitrary group delay filters.   |
| arbmag                         | <code>fdesign.arbmag</code> creates an object to specify IIR filters that have arbitrary magnitude responses defined by the input arguments.                 |
| arbmagnphase                   | <code>fdesign.arbmagnphase</code> creates an object to specify IIR filters that have arbitrary magnitude and phase responses defined by the input arguments. |
| bandpass                       | <code>fdesign.bandpass</code> creates an object to specify bandpass filters.   |
| bandstop                       | <code>fdesign.bandstop</code> creates an object to specify bandstop filters.   |
| ciccomp                        | <code>fdesign.ciccomp</code> creates an object to specify filters that compensate for the CIC decimator or interpolator response curves.                     |

| <b>fdesign Response Method</b> | <b>Description</b>  |
|--------------------------------|---|
| comb                           | <code>fdesign.comb</code> creates an object to specify a notching or peaking comb filter.           |
| decimator                      | <code>fdesign.decimator</code> creates an object to specify decimators.                             |
| differentiator                 | <code>fdesign.differentiator</code> creates an object to specify an FIR differentiator filter.      |
| fracdelay                      | <code>fdesign.fracdelay</code> creates an object to specify fractional delay filters.               |
| halfband                       | <code>fdesign.halfband</code> creates an object to specify halfband filters.                        |
| highpass                       | <code>fdesign.highpass</code> creates an object to specify highpass filters.                        |
| hilbert                        | <code>fdesign.hilbert</code> creates an object to specify an FIR Hilbert transformer.               |
| interpolator                   | <code>fdesign.interpolator</code> creates an object to specify interpolators.                       |
| isinchp                        | <code>fdesign.isinchp</code> creates an object to specify an inverse sinc highpass filter.          |
| isinclp                        | <code>fdesign.isinclp</code> creates an object to specify an inverse sinc lowpass filters.          |
| lowpass                        | <code>fdesign.lowpass</code> creates an object to specify lowpass filters.                          |
| notch                          | <code>fdesign.notch</code> creates an object to specify notch filters.                              |
| nyquist                        | <code>fdesign.nyquist</code> creates an object to specify Nyquist filters.                          |
| peak                           | <code>fdesign.peak</code> creates an object to specify peak filters.                                |
| polysrc                        | <code>fdesign.polysrc</code> creates an object to specify polynomial sample-rate converter filters. |
| rsrc                           | <code>fdesign.rsrc</code> creates an object to specify rational-factor sample-rate convertors.      |

Use the `doc fdesign.response` syntax at the MATLAB prompt to get help on a specific structure. For example, this command provides more information about the lowpass specification object:

```
doc fdesign.lowpass
```

Each response has a `Specification` property that defines the specifications to use to design your filter. You can use defaults or specify the `Specification` property when you construct the specifications object.

Using the `Specification` property, you can provide filter constraints such as the filter order or the passband attenuation to use when you construct your filter from the specification object.

### **spec — Filter design specifications**

character vector

Filter design specifications, specified as a character vector. The set of available specification options depends on the `fdesign.response` function. For more information, refer to the individual `fdesign.response` pages.

The filter design is based on the specifications provided by the `fdesign.response` object. For example, when you create a default lowpass filter design specification object, `fdesign.lowpass`, the

specification expression is set to 'Fp,Fst,Ap,Ast'. The filter design parameters — Fp (passband frequency), Fst (stopband frequency), Ap (passband ripple), and Ast (stopband attenuation) — are set to default values. The `design` function designs the filter based on these parameters.

Specifications that do not contain the filter order result in minimum-order designs when you invoke the `design` function:

```
d = fdesign.lowpass;           % Specification is 'Fp,Fst,Ap,Ast'
FIRReq = design(d,'equiripple','SystemObject',true);
length(FIRReq.Numerator)    % Returns 43. The filter order is 42
```

The specification option you choose determines which design methods are applicable. You can use the `setspecs` function to set all of the specifications simultaneously.

You can set filter specification values by passing them after the `Specification` argument, or by passing the values without the `Specification`.

Filter object constructors take the input arguments in the same order as `setspecs` and `Specification`.

When the first input to `fdesign.response` is not a valid `Specification` option, `fdesign` assumes that the input argument is a filter specification and applies it using the default `Specification` option. For example, 'Fp,Fst,Ap,Ast' is the default for a lowpass object.

### **Fs — Sample rate**

scalar

Sample rate to use in filter specifications, specified in Hz. The sample rate scalar must be the last input argument. If you specify a sample rate, all frequency specifications are in Hz.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **magunits — Units for magnitude specification**

'dB' (default) | 'linear' | 'squared'

Units for magnitude specification, specified as:

- 'dB' -- decibels
- 'linear' -- linear units
- 'squared' -- power units

When you omit the `magunits` argument, `fdesign` assumes that all magnitudes are in dB. Note that `fdesign` stores all magnitude specifications in dB. If you set `magunits` to an option other than 'dB', the function converts the unit to 'dB'.

## **Output Arguments**

### **designSpecs — Filter design specification object**

`fdesign.response` object

`fdesign` returns a filter design specification object. Every filter design specification object has these properties:

| Property Name       | Default Value                         | Description  |
|---------------------|---------------------------------------|--|
| Response            | Depends on the chosen type            | Defines the type of filter to design, such as an interpolator or bandpass filter. This is a read-only value.   |
| Specification       | Depends on the chosen type            | Defines the filter characteristics used to define the desired filter performance, such as the cutoff frequency $F_c$ or the filter order $N$ .   |
| Description         | Depends on the filter type you choose | Contains descriptions of the filter specifications used to define the object, and the filter specifications you use when you create a filter from the object. This is a read-only value.   |
| NormalizedFrequency | Logical true                          | Determines whether the filter calculation uses a normalized frequency from 0 to 1, or the frequency band from 0 to $F_s/2$ , the sampling frequency. Accepts either <code>true</code> or <code>false</code> without single quotation marks. Audio weighting filters do not support normalized frequency. |

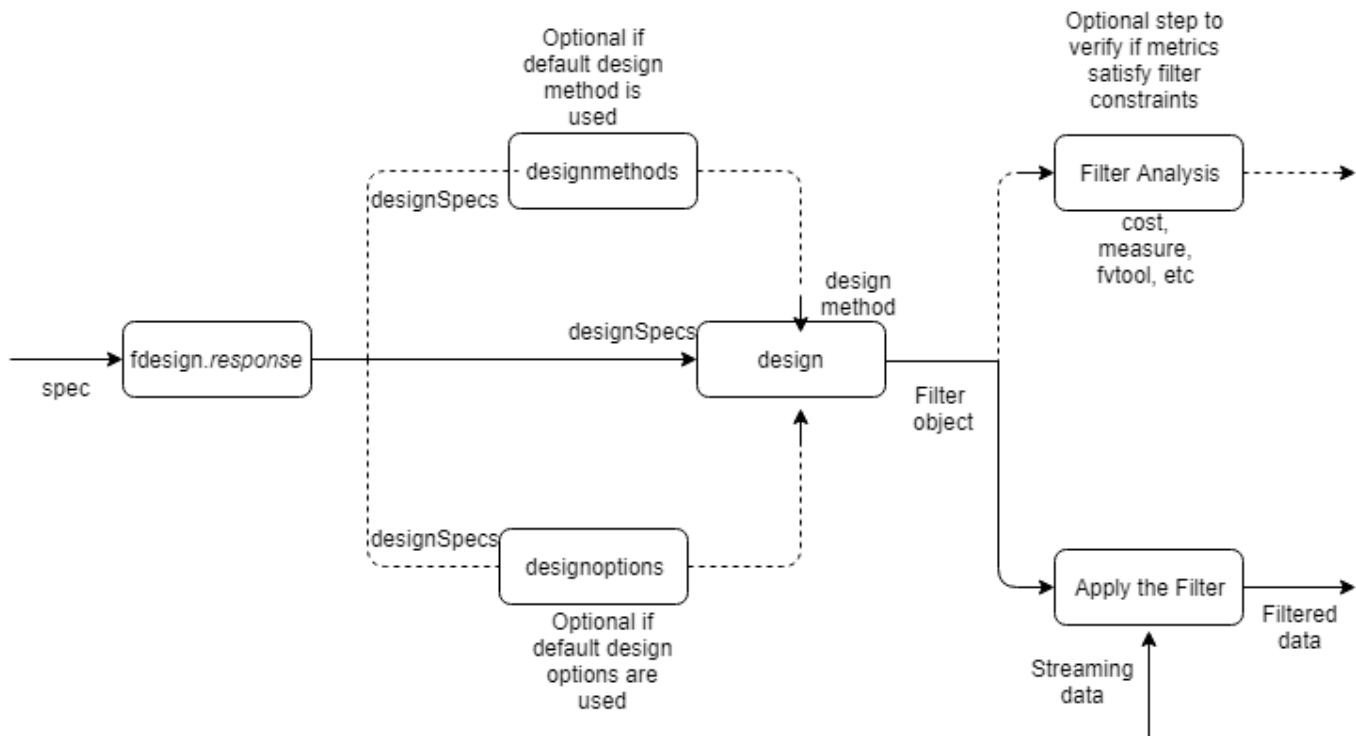
In addition to these properties, filter design specification objects can have other properties as well, depending on whether they design single-rate filters or multirate filters.

| Added Properties for Multirate Filters | Description   |
|--|---|
| DecimationFactor                       | Specifies the amount to decrease the sampling rate. Always a positive integer.  |
| InterpolationFactor                    | Specifies the amount to increase the sampling rate. Always a positive integer.  |
| PolyphaseLength                        | Polyphase length is the length of each polyphase subfilter that composes the decimator or interpolator or rate-change factor filters. Total filter length is the product of <code>pl</code> and the rate change factors. <code>pl</code> must be an even integer. |

## More About

### Filter Design Procedure

Here is the workflow diagram of the overall procedure for designing and analyzing the filter.



Here are the steps in detail:

- 1 Create an `fdesign.response` specification object to specify the design parameters.
- 2 Use `designmethods` to determine the filter design methods that work for your new filter specification object. If you choose to use the default design method, then this step is optional.
- 3 If you prefer to change the design options and would like to see a list of available options, run the `designoptions` function on the specification object. The output also shows the design options the filter uses by default.
- 4 Use `design` to design the filter from the filter specification object. Specify the design method (determined from step 2) as an input. If the design options must change from the default values, specify them as name-value pairs following the design method.

If you call the `design` function without any output arguments, FVTool is launched and shows the magnitude response of the designed filter.

Alternatively, use the `fvtool` function.

- 5 Further analysis, such as viewing the frequency response of the filter, computing the cost of implementing the filter, and measuring the filter response characteristics, can be done using one of the supported “Analysis Methods for Filter System Objects” on page 3-2.
- 6 Once you analyze the filter and determine that the filter satisfies the design constraints, you can apply the filter object to streaming input data. For details on how to pass data to the filter object, refer to the corresponding filter System object reference pages.

For a detailed example on the design and analysis, see “Lowpass Butterworth Filter Specification and Design” on page 5-394.



## **Version History**

Introduced in R2009a

### **See Also**

#### **Apps**

**Filter Designer**

#### **Functions**

`designmethods` | `designoptions` | `design` | `setspecs` | `filterBuilder`

#### **Topics**

“Design a Filter in Fdesign — Process Overview”

## fdesign.arbgrpdelay

Arbitrary group delay filter specification object

### Syntax

```
specObj = fdesign.arbgrpdelay
specObj = fdesign.arbgrpdelay(spec)
specObj = fdesign.arbgrpdelay(spec,value1,...,valueN)
specObj = fdesign.arbgrpdelay(N,F,Gd)
specObj = fdesign.arbgrpdelay( ____,Fs)
```

### Description

Arbitrary group delay filters are allpass filters that you can use to correct the phase distortion introduced by other filters. The `fdesign.arbgrpdelay` function uses an iterative least  $p$ -th norm optimization procedure to minimize the phase response error [1] on page 5-410.

The `fdesign.arbgrpdelay` function returns a filter design specification object containing filter specifications, such as filter order, number of bands, frequency vector, and group delay response. You then use the `design` function to design the filter from the filter design specifications object.

For more control options, see “Filter Design Procedure” on page 5-399. For a complete workflow, see “Design a Filter in Fdesign — Process Overview”.

`specObj = fdesign.arbgrpdelay` creates an allpass arbitrary group delay filter design specification object with the filter order of 10, frequency vector of [0 0.1 1], and group delay response vector of [2 3 1].

`specObj = fdesign.arbgrpdelay(spec)` creates a filter design specification object with the specification property `spec`. For more information on the specifications that the function supports, see `spec` on page 5-0 .

`specObj = fdesign.arbgrpdelay(spec,value1,...,valueN)` creates a filter design specification object with the specification values `value1,...,valueN`. Set the specification options in the expression `spec`. After the expression, specify a value for each option.

`specObj = fdesign.arbgrpdelay(N,F,Gd)` creates a filter design specification object with the filter order, frequency vector, and group delay response vector specified in `N`, `F`, and `Gd`, respectively. For more information on the filter order, frequency vector, and group delay vector inputs, see `spec` on page 5-0 .

`specObj = fdesign.arbgrpdelay( ____,Fs)` specifies the sample rate in Hz. If you specify a sample rate, the group delay response values in the filter design specification object are in seconds. If you do not specify a sample rate, the function normalizes all frequencies and the group delay response values are in samples.

### Examples

## Design Arbitrary Group Delay Filter

Create a signal consisting of two discrete-time windowed sinusoids (wave packets) with disjoint time support to illustrate frequency dispersion. One discrete-time sinusoid has a frequency of  $\pi/2$  radians/sample and the other has a frequency of  $\pi/4$  radians/sample. There are nine periods of the sinusoid with the higher frequency that precede five periods of the signal with the lower frequency.

Create the signal.

```
x = zeros(300,1);
x(1:36) = cos(pi/2*(0:35)).*hamming(36)';
x(40:40+39) = cos(pi/4*(0:39)).*hamming(40)';
```

Create an arbitrary group delay filter that delays the wave packet with the higher frequency by approximately 100 samples.

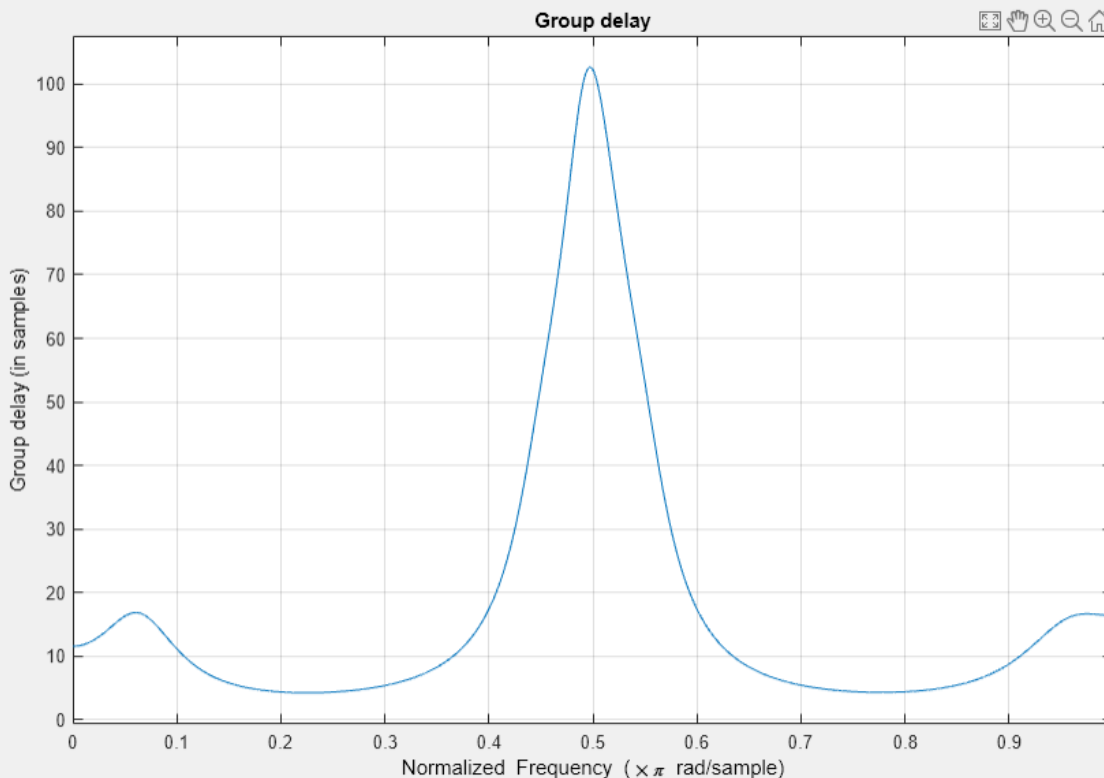
```
N = 18;
f = 0:.1:1;
gd = ones(size(f));
```

Delay  $\pi/2$  radians/sample by 100 samples.

```
gd(6) = 100;
d = fdesign.arbgrpdelay(N,f,gd);
Hd = design(d,'iirlpnorm',MaxPoleRadius=0.9,SystemObject=true);
```

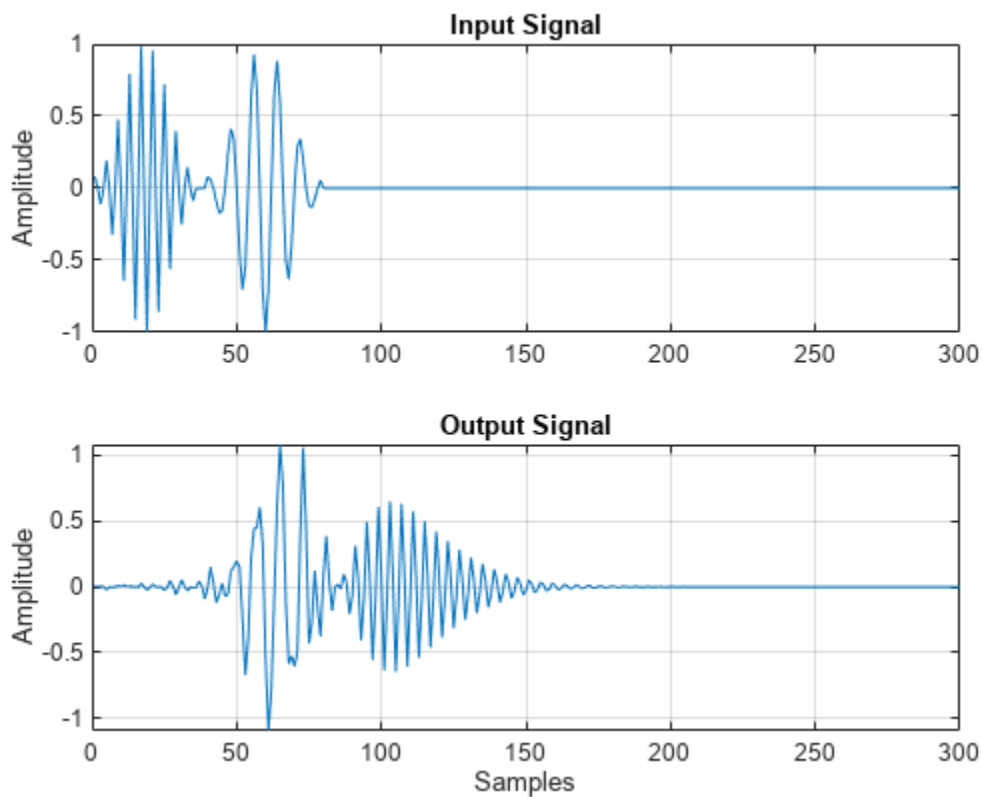
Visualize the group delay.

```
fvtool(Hd,analysis='grpdelay');
```



Filter the input signal with the arbitrary group delay filter and plot the frequency dispersion. The high-frequency wave packet, which initially preceded the low-frequency wave packet, now occurs later because of the nonconstant group delay.

```
y = Hd(x);
subplot(211)
plot(x);
title('Input Signal');
grid on;
ylabel('Amplitude');
subplot(212);
plot(y);
title('Output Signal');
grid on;
xlabel('Samples');
ylabel('Amplitude');
```



### Design an Allpass Filter With an Arbitrary Group Delay

Design an allpass filter with an arbitrary group delay using the `fdesign.arbgrpdelay` and the `design` functions. Pass `iirlpnorm` as the design method.

```
N = 10;
f = [0 0.02 0.04 0.06 0.08 0.1 0.25 0.5 0.75 1];
g = [5 5 5 5 5 5 4 3 2 1];
```

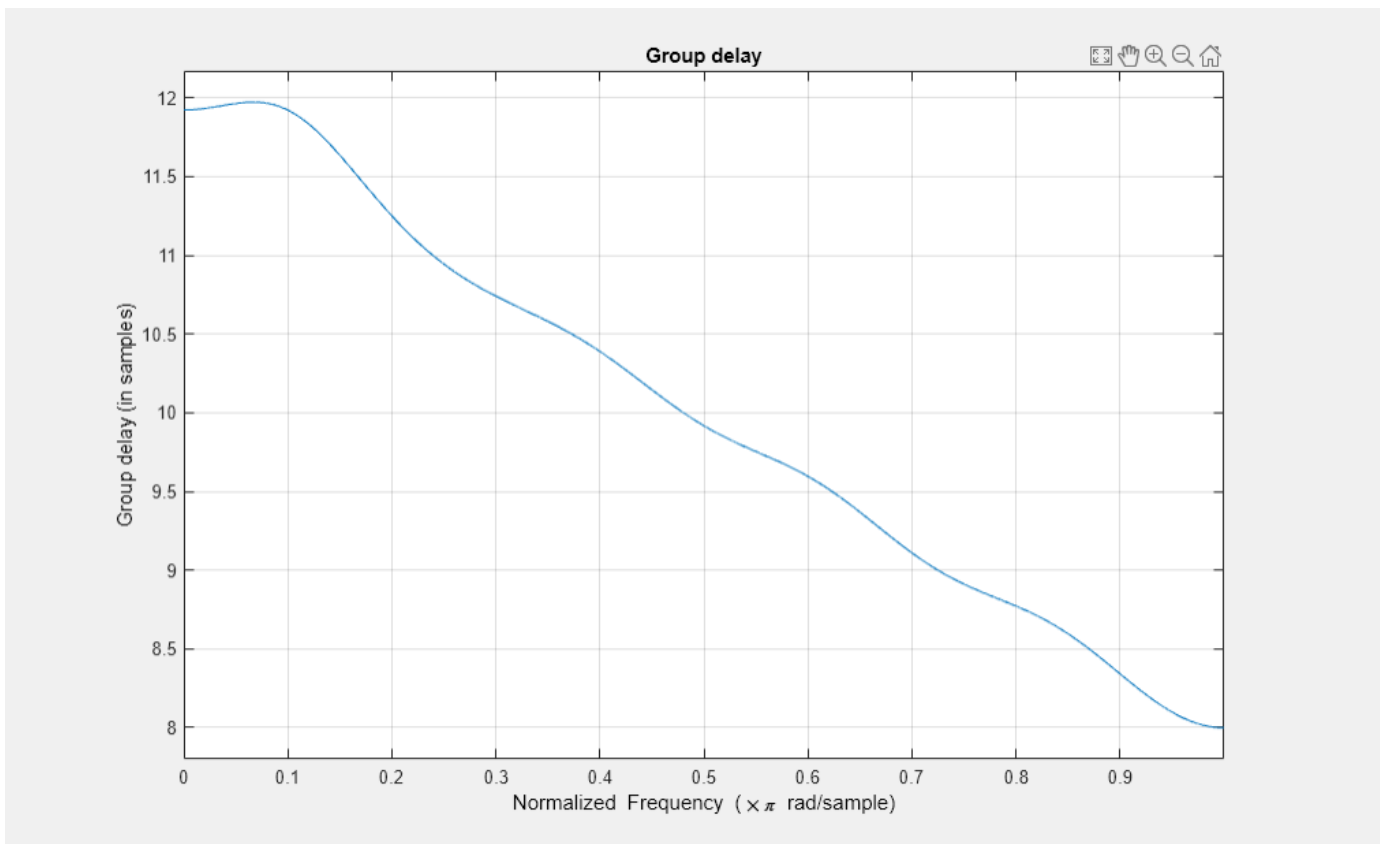
```
w = [2 2 2 2 2 2 1 1 1 1];
hgd = fdesign.arbgrpdelay(N,f,g);
Hgd = design(hgd,'iirlpnorm',Weights=w,MaxPoleRadius=0.95,...
    SystemObject=true)
```

```
Hgd =
    dsp.SOSFilter with properties:

        Structure: 'Direct form II'
    CoefficientSource: 'Property'
        Numerator: [5x3 double]
        Denominator: [5x3 double]
    HasScaleValues: true
    ScaleValues: [0.2274 1 1 1 1 1.3513]
```

Show all properties

```
fvtool(Hgd,Analysis='grpdelay') ;
```



### Multiband Delay Equalization

Perform group delay equalization of a bandstop Chebyshev filter operating with a sample rate of 1 kHz.

```

Fs = 1e3;
Hcheby2 = design(fdesign.bandstop('N,Fst1,Fst2,Ast',10,150,400,60,Fs),'cheby2',...
    SystemObject=true);
f1 = 0.0:0.5:150; % Hz
g1 = grpdelay(Hcheby2,f1,Fs).'/Fs; % seconds
f2 = 400:0.5:500; % Hz
g2 = grpdelay(Hcheby2,f2,Fs).'/Fs; % seconds
maxg = max([g1 g2]);

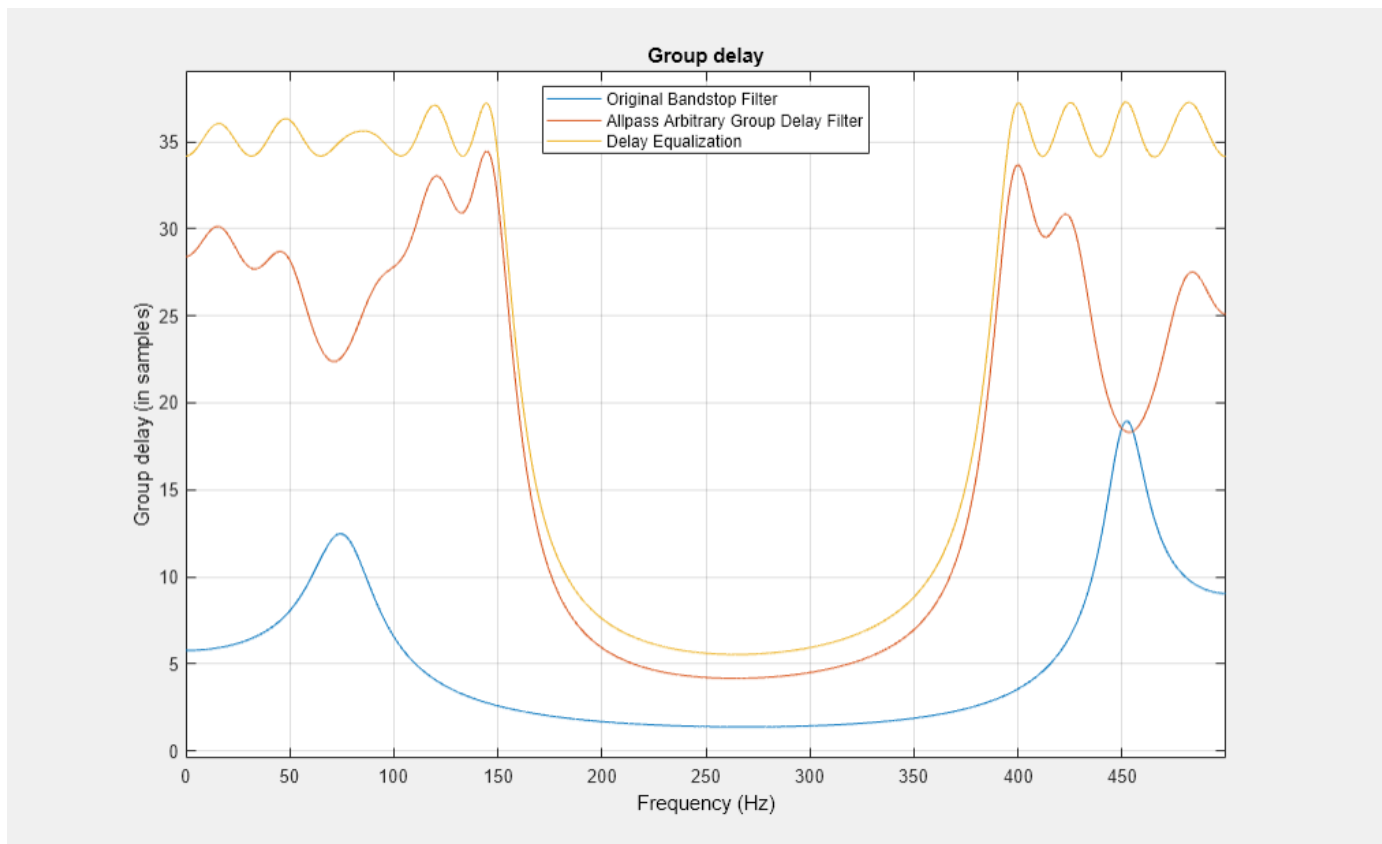
```

Design an arbitrary group delay allpass filter to equalize the group delay of the bandstop filter. Use an 18 order multiband design and specify two bands.

```

hgd = fdesign.arbgrpdelay('N,B,F,Gd',18,2,f1,maxg-g1,f2,maxg-g2,Fs);
Hgd = design(hgd,'iirlpnm',MaxPoleRadius=0.95,SystemObject=true);
Hcascade = cascade(Hcheby2,Hgd);
hft = fvtool(Hcheby2,Hgd,Hcascade,Analysis='grpdelay',Fs=Fs);
legend(hft,'Original Bandstop Filter','Allpass Arbitrary Group Delay Filter',...
    'Delay Equalization','Location','North');

```



## Input Arguments

### spec — Filter specification

'N, F, Gd' (default) | 'N, B, F, Gd'

Filter specification, specified as one of these character vectors:

- 'N,F,Gd'
- 'N,B,F,Gd'

This table describes the options available in each expression.

| Specification Option | Description                 |
|----------------------|-----------------------------|
| N                    | Filter order (must be even) |
| F                    | Frequency vector            |
| Gd                   | Group delay response vector |
| B                    | Number of frequency bands   |

The design methods that you can use to design the filter depend on the specification expression. You can obtain these methods using the `designmethods` function. This table lists the specification expressions supported by the `fdesign.arbgrpdelay` function and the corresponding design method.

| Specification Expression | Supported Design Method | Filter Description              |
|--------------------------|-------------------------|---------------------------------|
| 'N,F,Gd'                 | <code>iirlpnorm</code>  | Least P-norm optimal IIR filter |
| 'N,B,F,Gd'               | <code>iirlpnorm</code>  | Least P-norm optimal IIR filter |

To design the filter, call the `design` function with one of the design methods as an input. For more details on the procedure, see “Filter Design Procedure” on page 5-399. For an example, see “Design an Allpass Filter With an Arbitrary Group Delay” on page 5-404.

### value1, ..., valueN — Specification values

comma-separated list of values

Specification values, specified as a comma-separated list of values. Specify a value for each option in `spec` in the same order as the options in the specification expression.

Example: `specObj = fdesign.arbgrpdelay('N,F,Gd',N,F,Gd)`

The arguments below describe more details for each option in the expression.

#### N — Filter order

10 (default) | even positive integer

Filter order, specified as an even positive integer. The numerator and denominator orders are both equal to N. For more information on why the numerator and denominator filter orders are equal and even in the `fdesign.arbgrpdelay` function, see “Allpass Systems” on page 5-409.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

#### F — Frequency vector

[0 0.1 1] (default) | row vector

Frequency vector for the group delay specifications, specified as a row vector. The elements of the frequency vector must increase monotonically.

If you do not specify a sample rate  $F_s$ , the function normalizes the frequencies. For a single-band design, the first element of the normalized frequency vector must be 0 and the last element must be 1. These correspond to 0 and  $\pi$  radians/sample, respectively. For multiband designs, the union of the frequency vectors must range between [0, 1].

If you specify a sample rate  $F_s$ , the first element of the frequency vector in a single-band design must be 0. The last element must be the Nyquist frequency  $F_s/2$ . For multiband designs, the union of the frequency vectors must range between  $[0, F_s/2]$ .

Data Types: `double`

### **Gd — Group delay vector**

`[2 3 1]` (default) | row vector

Group delay vector, specified as a row vector with nonnegative elements and equal in length to the frequency vector  $F$ . The elements of  $Gd$  specify the nonnegative group delay at the corresponding element of the frequency vector  $F$ .

If you do not specify a sample rate  $F_s$  in Hz, the group delay values are in samples. If you specify a sample rate, the group delay values are in seconds.

Data Types: `double`

### **B — Number of frequency bands**

1 (default) | positive integer

Number of frequency bands, specified as a positive integer. To use this argument, you must specify a frequency vector  $F$  and a group delay vector  $Gd$  for each band. The union of the frequency vectors must range between  $[0, 1]$  in normalized frequency or  $[0, F_s/2]$  when you specify a sample rate. The elements in the union of the frequency bands must be monotonically increasing.

Example:

```
filtorder = 14;
freqband1 = [0 0.1 0.4]; grpdelay1 = [1 2 3];
freqband2 = [0.5 0.8 1]; grpdelay2 = [3 2 1];
D = fdesign.arbgrpdelay('N,B,F,Gd',filtorder,2,freqband1,grpdelay1,freqband2,grpdelay2);
```

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **Fs — Sample rate**

positive scalar

Sample rate, specified as a positive scalar. Specify this argument after all the other input arguments. Specifying a sample rate forces the group delay units to be in seconds. If you specify a sample rate, the first element of the frequency vector must be 0 and the last element must be the Nyquist frequency  $F_s/2$ .

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

## **Output Arguments**

### **specObj — Allpass arbitrary group delay filter specification object**

`arbgrpdelay` object

Allpass arbitrary group delay filter design specification object, returned as a `arbgrpdelay` object. The object properties depend on the `spec` input argument.

The filter design specification object can contain one or more of these properties:

- `Specification`
- `NormalizedFrequency`



- FilterOrder
- Frequencies
- GroupDelay
- NBands -- Appears if the Specification property is set to 'N,B,F,Gd'.

Use the `normalizefreq` function to change the `NormalizedFrequency` property after constructing the specification object.

## More About

### Group Delay in Discrete-Time Filter Design

The frequency response of a rational discrete-time filter is:

$$H(e^{j\omega}) = \frac{B(e^{j\omega})}{A(e^{j\omega})}$$

The argument of the frequency response as a function of the angle,  $\omega$ , is referred to as the *phase response*.

The negative of the first derivative of the argument with respect to  $\omega$  is the group delay.

$$\tau(\omega) = -\frac{d}{d\omega} \text{Arg}(H(e^{j\omega}))$$

Systems with nonlinear phase responses have nonconstant group delay, which causes the frequency components of the signal to disperse. You may not want this phase distortion even if the magnitude distortion introduced by the filter produces the desired effect. See “Design Arbitrary Group Delay Filter” on page 5-402 for an illustration of frequency dispersion resulting from nonconstant group delay.

In these cases, you can cascade the frequency-selective filter with an allpass filter that compensates for the group delay. This process is often referred to as *delay equalization*.

### Allpass Systems

The general form of an allpass system function with a real-valued impulse response is:

$$H_{ap}(z) = \prod_{k=1}^M \frac{z^{-1} - d_k}{1 - d_k z^{-1}} \prod_{k=1}^N \frac{(z^{-1} - c_k)(z^{-1} - c_k^*)}{(1 - c_k z^{-1})(1 - c_k^* z^{-1})}$$

where the  $d_k$  denote the real-valued poles and the  $c_k$  denote the complex-valued poles, which occur in conjugate pairs.

This equation shows that allpass systems with real-valued impulse responses have  $2N+M$  zeros and poles. The poles and zeros occur in pairs with reciprocal magnitudes. The filter order is always the same for the numerator and denominator.

Because the `fdesign.arbgrpdelay` function uses robust second-order section (biquad) filter structures to implement the allpass arbitrary group delay filter, the filter order must be even.

## Tips

If your arbitrary group delay design produces the Poorly conditioned Hessian matrix error, perform one of the following or both.

- Set the `MaxPoleRadius` IIR lp norm design option to a number less than 1 when you design your filter.

```
design(d,'iirlpnorm','MaxPoleRadius',0.95)
```

For more information on the `MaxPoleRadius` design option, see the “Design Arbitrary Group Delay Filter” on page 5-402 and “Multiband Delay Equalization” on page 5-405 examples.

- Reduce the order of your filter design.

## Algorithms

The `fdesign.arbgrpdelay` function uses a least  $p$ -th norm iterative optimization described in [1] on page 5-410.

## Version History

Introduced in R2011b

## References

[1] Antoniou, A. *Digital Signal Processing: Signals, Systems, and Filters.*, New York:McGraw-Hill, 2006, pp. 719-771.

## See Also

`fdesign` | `design` | `iirgrpdelay` | `designmethods` | `designoptions`

## Topics

“Design a Filter in Fdesign — Process Overview”

## fdesign.arbmag

Arbitrary response magnitude filter specification object

### Syntax

```
D= fdesign.arbmag
D= fdesign.arbmag(SPEC)
D = fdesign.arbmag(SPEC,specvalue1,specvalue2,...)
D = fdesign.arbmag(specvalue1,specvalue2,specvalue3)
D = fdesign.arbmag(...,Fs)
```

### Description

D= fdesign.arbmag constructs an arbitrary magnitude filter specification object D.

D= fdesign.arbmag(SPEC) initializes the Specification property to SPEC. The input argument SPEC must be one of the entries shown in the following table. Specification entries are not case sensitive.

---

**Note** Specification entries marked with an asterisk require the DSP System Toolbox software.

---

- 'N,F,A' — Single band design (default)
- 'F,A,R' — Single band minimum order design \*
- 'N,B,F,A' — Multiband design
- 'N,B,F,A,C' — Constrained multiband design \*
- 'B,F,A,R' — Multiband minimum order design \*
- 'Nb,Na,F,A' — Single band design \*
- 'Nb,Na,B,F,A' — Multiband design \*

The SPEC entries are defined as follows:

- A — Amplitude vector. Values in A define the filter amplitude at frequency points you specify in f, the frequency vector. If you use A, you must use F as well. Amplitude values must be real. For complex values designs, use fdesign.arbmagnphase.
- B — Number of bands in the multiband filter
- C — Constrained band flag. This enables you to constrain the passband ripple in your multiband design. You cannot constrain the passband ripple in all bands simultaneously.
- F — Frequency vector. Frequency values in specified in F indicate locations where you provide specific filter response amplitudes. When you provide F, you must also provide A.
- N — Filter order for FIR filters and the numerator and denominator orders for IIR filters.
- Nb — Numerator order for IIR filters
- Na — Denominator order for IIR filter designs
- R — Ripple

By default, this method assumes that all frequency specifications are supplied in normalized frequency.

### Specifying Frequency and Amplitude Vectors

F and A are the input arguments you use to define the filter response desired. Each frequency value you specify in F must have a corresponding response value in A. The following table shows how F and A are related.

Define the frequency vector F as [0 0.25 0.3 0.4 0.5 0.6 0.7 0.75 1.0]

Define the response vector A as [1 1 0 0 0 0 0 1 1]

These specifications connect F and A as shown here:

| F (Normalized Frequency) | A (Response Desired at F) |
|--------------------------|---------------------------|
| 0                        | 1                         |
| 0.25                     | 1                         |
| 0.3                      | 0                         |
| 0.4                      | 0                         |
| 0.5                      | 0                         |
| 0.6                      | 0                         |
| 0.7                      | 0                         |
| 0.75                     | 1                         |
| 1.0                      | 1                         |

Different specifications can have different design methods available. Use `designmethods` to get a list of design methods available for a given specification and filter specification object.

Use `designopts` to get a list of design options available for a filter specification object and a given design method. Enter `help(D,METHOD)` to get detailed help on the available design options for a given design method.

`D = fdesign.arbmag(SPEC,specvalue1,specvalue2,...)` initializes the specifications with `specvalue1`, `specvalue2`. Use `get(D,'Description')` for descriptions of the various specifications `specvalue1`, `specvalue2`, ... `specvalueN`.

`D = fdesign.arbmag(specvalue1,specvalue2,specvalue3)` uses the default specification 'N,F,A', setting the filter order, filter frequency vector, and the amplitude vector to the values `specvalue1`, `specvalue2`, and `specvalue3`.

`D = fdesign.arbmag(...,Fs)` specifies the sampling frequency in Hz. All other frequency specifications are also assumed to be in Hz when you specify `Fs`.

## Examples

### Design Multiband Arbitrary-Magnitude Filter

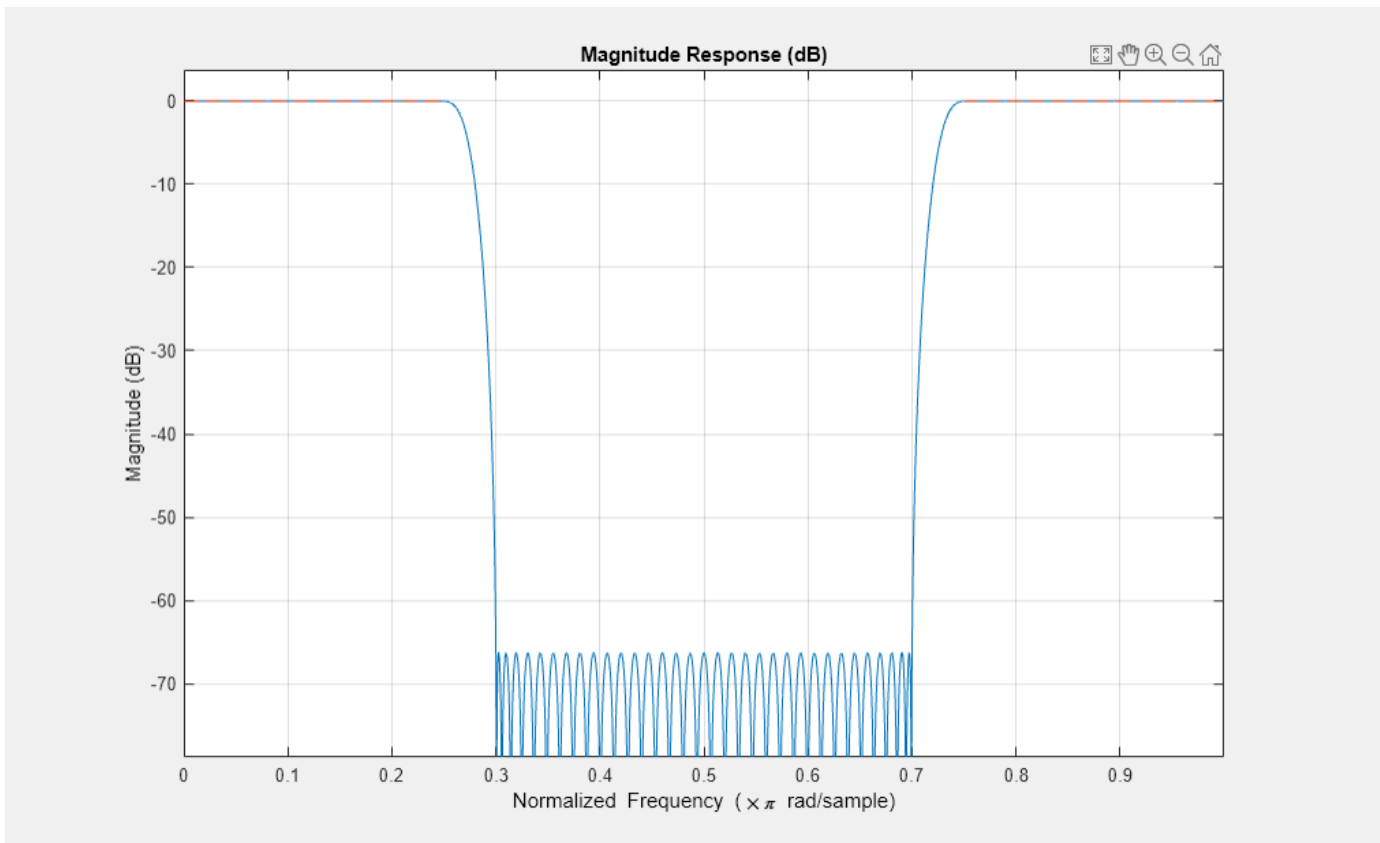
Use `fdesign.arbmag` to design a three-band filter.

- Define the frequency vector  $F = [0 \ 0.25 \ 0.3 \ 0.4 \ 0.5 \ 0.6 \ 0.7 \ 0.75 \ 1.0]$ .
- Define the response vector  $A = [1 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 1]$ .

```

N = 150;
B = 3;
F = [0 .25 .3 .4 .5 .6 .7 .75 1];
A = [1 1 0 0 0 0 0 1 1];
A1 = A(1:2);
A2 = A(3:7);
A3 = A(8:end);
F1 = F(1:2);
F2 = F(3:7);
F3 = F(8:end);
d = fdesign.arbmag('N,B,F,A',N,B,F1,A1,F2,A2,F3,A3);
Hd = design(d);
fvtool(Hd)

```



A response with two passbands -- one roughly between 0 and 0.25 and the second between 0.75 and 1 -- results from the mapping between  $F$  and  $A$ .

### Design Single-Band Arbitrary-Magnitude Filter

Use `fdesign.arbmag` to design a single band equiripple filter.

Specify 100 frequency points.

```

n = 120;
f = linspace(0,1,100);

as = ones(1,100) - f*0.2;
absorb = [ones(1,30), 1-0.6*bohmanwin(10)', ones(1,5), ...
          1-0.5*bohmanwin(8)', ones(1,47)];
a = as.*absorb;

```

```

d = fdesign.arbmag('N,F,A',n,f,a);
hd1 = design(d,'equiripple');

```

Design a minimum-phase equiripple filter. Visualize the poles and zeros of the two filters.

```

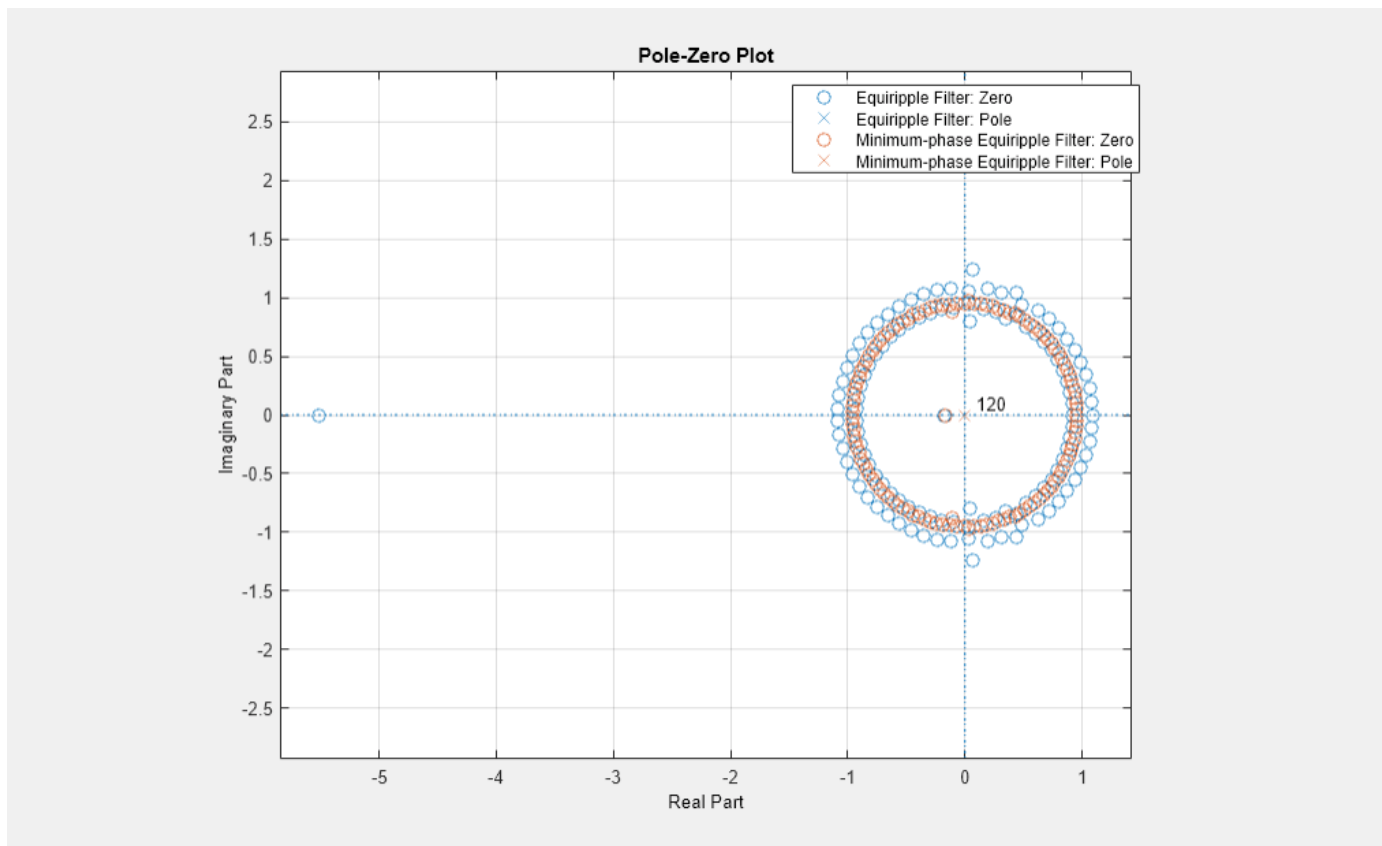
hd2 = design(d,'equiripple','MinPhase',true);

```

```

hfvt = fvtool(hd1,hd2,'Analysis','polezero');
legend(hfvt,'Equiripple Filter','Minimum-phase Equiripple Filter')

```



### Design Multiband Minimum-Order Arbitrary-Magnitude Filter

Use `fdesign.arbmag` to design a multiband minimum order filter.

Place the notches at  $0.25\pi$  and  $0.55\pi$  rad/sample.

```

d = fdesign.arbmag('B,F,A,R');
d.NBands = 5;

```

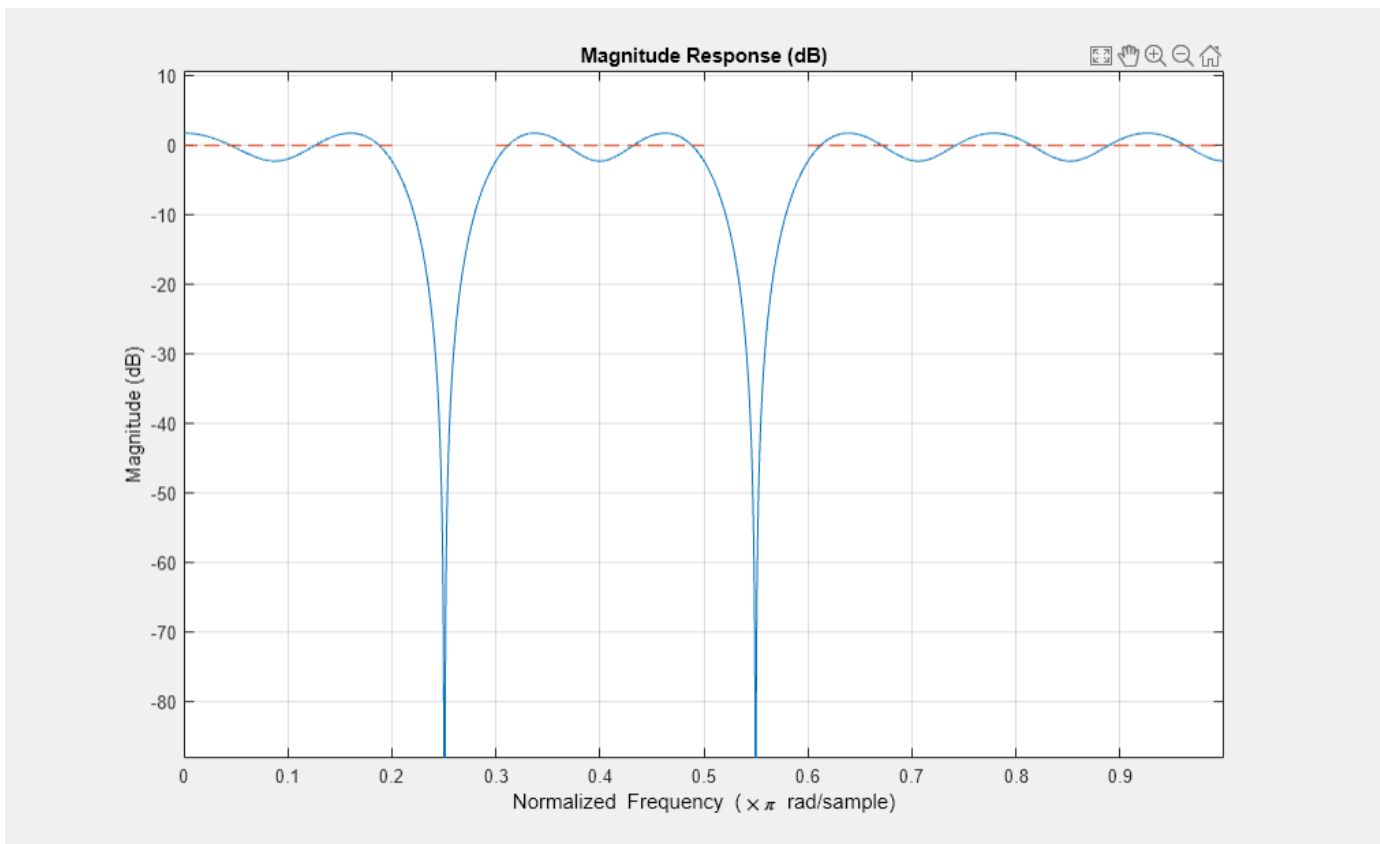
```

d.B1Frequencies = [0 0.2];
d.B1Amplitudes = [1 1];
d.B1Ripple = 0.25;
d.B2Frequencies = 0.25;
d.B2Amplitudes = 0;
d.B3Frequencies = [0.3 0.5];
d.B3Amplitudes = [1 1];
d.B3Ripple = 0.25;
d.B4Frequencies = 0.55;
d.B4Amplitudes = 0;
d.B5Frequencies = [0.6 1];
d.B5Amplitudes = [1 1];
d.B5Ripple = 0.25;
Hd = design(d,'equiripple');

```

Visualize the frequency response of the resulting filter.

```
fvtool(Hd)
```



### Design Multiband Constrained Arbitrary-Magnitude Filter

Use `fdesign.arbmag` to design a multiband constrained FIR filter.

Force the frequency response at  $0.15\pi$  rad/sample to 0 dB.

```
d = fdesign.arbmag('N,B,F,A,C',82,2);  
d.B1Frequencies = [0 0.06 0.1];  
d.B1Amplitudes = [0 0 0];  
d.B2Frequencies = [0.15 1];  
d.B2Amplitudes = [1 1];
```

Design a filter with no constraints.

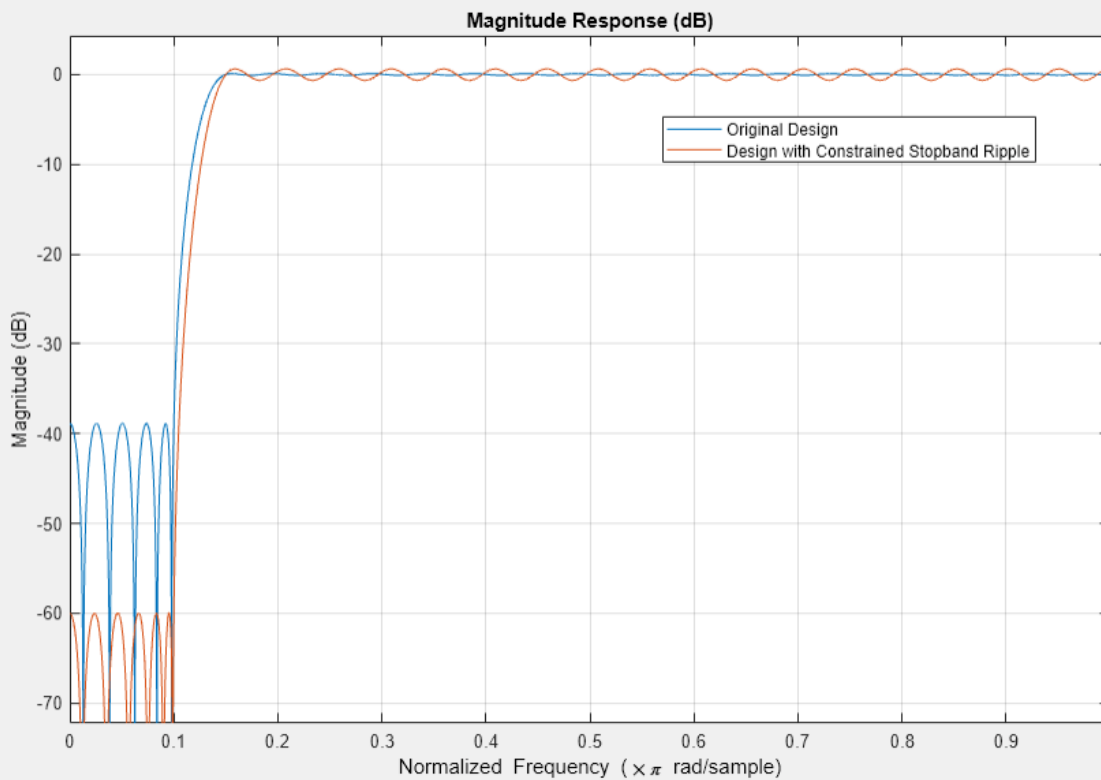
```
Hd1 = design(d, 'equiripple', 'B2ForcedFrequencyPoints', 0.15);
```

Add a constraint to the first band to increase attenuation.

```
d.B1Constrained = true;  
d.B1Ripple = 0.001;  
Hd2 = design(d, 'equiripple', 'B2ForcedFrequencyPoints', 0.15);
```

Visualize the frequency response.

```
hfvt = fvtool(Hd1,Hd2);  
legend(hfvt, 'Original Design', 'Design with Constrained Stopband Ripple')
```



## Version History

Introduced in R2009a



## **See Also**

design | designmethods | fdesign

## fdesign.arbmagnphase

Arbitrary response magnitude and phase filter specification object

### Syntax

```
d = fdesign.arbmagnphase
d = fdesign.arbmagnphase(specification)
d = fdesign.arbmagnphase(specification,specvalue1,specvalue2,...)
d = fdesign.arbmagnphase(specvalue1,specvalue2,specvalue3)
d = fdesign.arbmagnphase(...,fs)
```

### Description

`d = fdesign.arbmagnphase` constructs an arbitrary magnitude filter specification object `d`.

`d = fdesign.arbmagnphase(specification)` initializes the `Specification` property for specifications object `d` to `specification`. The input argument `specification` must be one of the choices shown in the following table. Specification options are not case sensitive.

| Specification             | Description of Resulting Filter  |
|---------------------------|--|
| <code>n, f, h</code>      | Single band design (default). FIR and IIR ( <code>n</code> is the order for both numerator and denominator). |
| <code>n, b, f, h</code>   | FIR multiband design where <code>b</code> defines the number of bands.                                       |
| <code>nb, na, f, h</code> | IIR single band design.  |

The following table describes the specification arguments.

| Argument        | Description   |
|-----------------|---|
| <code>b</code>  | Number of bands in the multiband filter.  |
| <code>f</code>  | Frequency vector. Frequency values specified in <code>f</code> indicate locations where you provide specific filter response amplitudes. When you provide <code>f</code> you must also provide <code>h</code> which contains the response values. |
| <code>h</code>  | Complex frequency response values.  |
| <code>n</code>  | Filter order for FIR filters and the numerator and denominator orders for IIR filters (when not specified by <code>nb</code> and <code>na</code> ).   |
| <code>nb</code> | Numerator order for IIR filters.  |
| <code>na</code> | Denominator order for IIR filter designs.   |

By default, this method assumes that all frequency specifications are supplied in normalized frequency.

### Specifying `f` and `h`

`f` and `h` are the input arguments you use to define the filter response desired. Each frequency value you specify in `f` must have a corresponding response value in `h`. This example creates a filter with

two passbands ( $b = 4$ ) and shows how  $f$  and  $h$  are related. This example is for illustration only. It is not an actual filter.

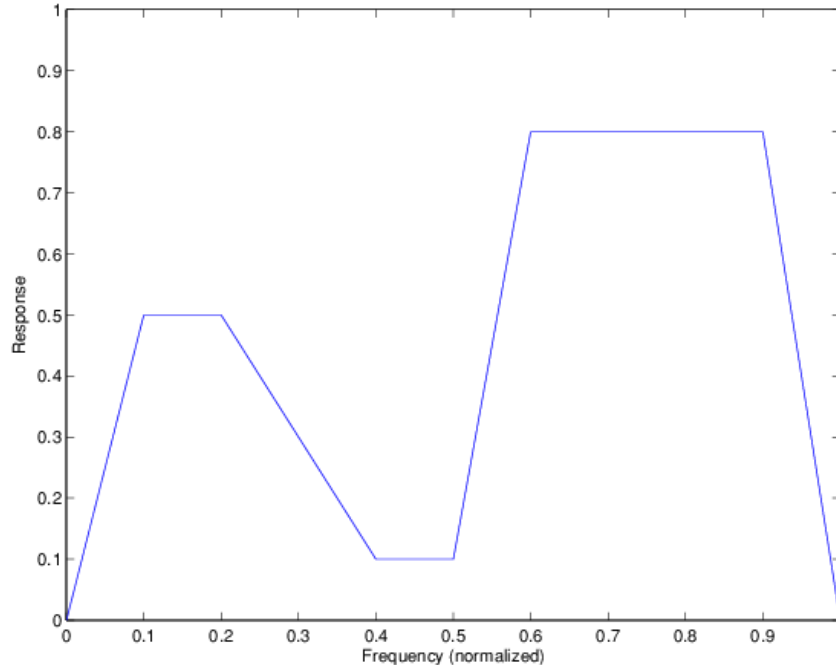
Define the frequency vector  $f$  as `[0 0.1 0.2 0.4 0.5 0.6 0.9 1.0]`

Define the response vector  $h$  as `[0 0.5 0.5 0.1 0.1 0.8 0.8 0]`

These specifications connect  $f$  and  $h$  as shown in the following table.

| <b>f (Normalized Frequency)</b> | <b>h (Response Desired at f)</b> |
|---------------------------------|----------------------------------|
| 0                               | 0                                |
| 0.1                             | 0.5                              |
| 0.2                             | 0.5                              |
| 0.4                             | 0.1                              |
| 0.5                             | 0.1                              |
| 0.6                             | 0.8                              |
| 0.9                             | 0.8                              |
| 1.0                             | 0.0                              |

A response with two passbands—one roughly between 0.1 and 0.2 and the second between 0.6 and 0.9—results from the mapping between  $f$  and  $h$ . Plotting  $f$  and  $h$  yields the following figure that resembles a filter with two passbands.



The second example in Examples shows this plot in more detail with a complex filter response for  $h$ . In the example,  $h$  uses complex values for the response.

Different specification types often have different design methods available. Use `designmethods(d)` to get a list of design methods available for a given specification option and specifications object.

`d = fdesign.arbmagnphase(specification,specvalue1,specvalue2,...)` initializes the filter specification object with `specvalue1`, `specvalue2`, and so on. Use `get(d,'description')` for descriptions of the various specifications `specvalue1`, `specvalue2`, ...`specn`.

`d = fdesign.arbmagnphase(specvalue1,specvalue2,specvalue3)` uses the default specification option `n`, `f`, `h`, setting the filter order, filter frequency vector, and the complex frequency response vector to the values `specvalue1`, `specvalue2`, and `specvalue3`.

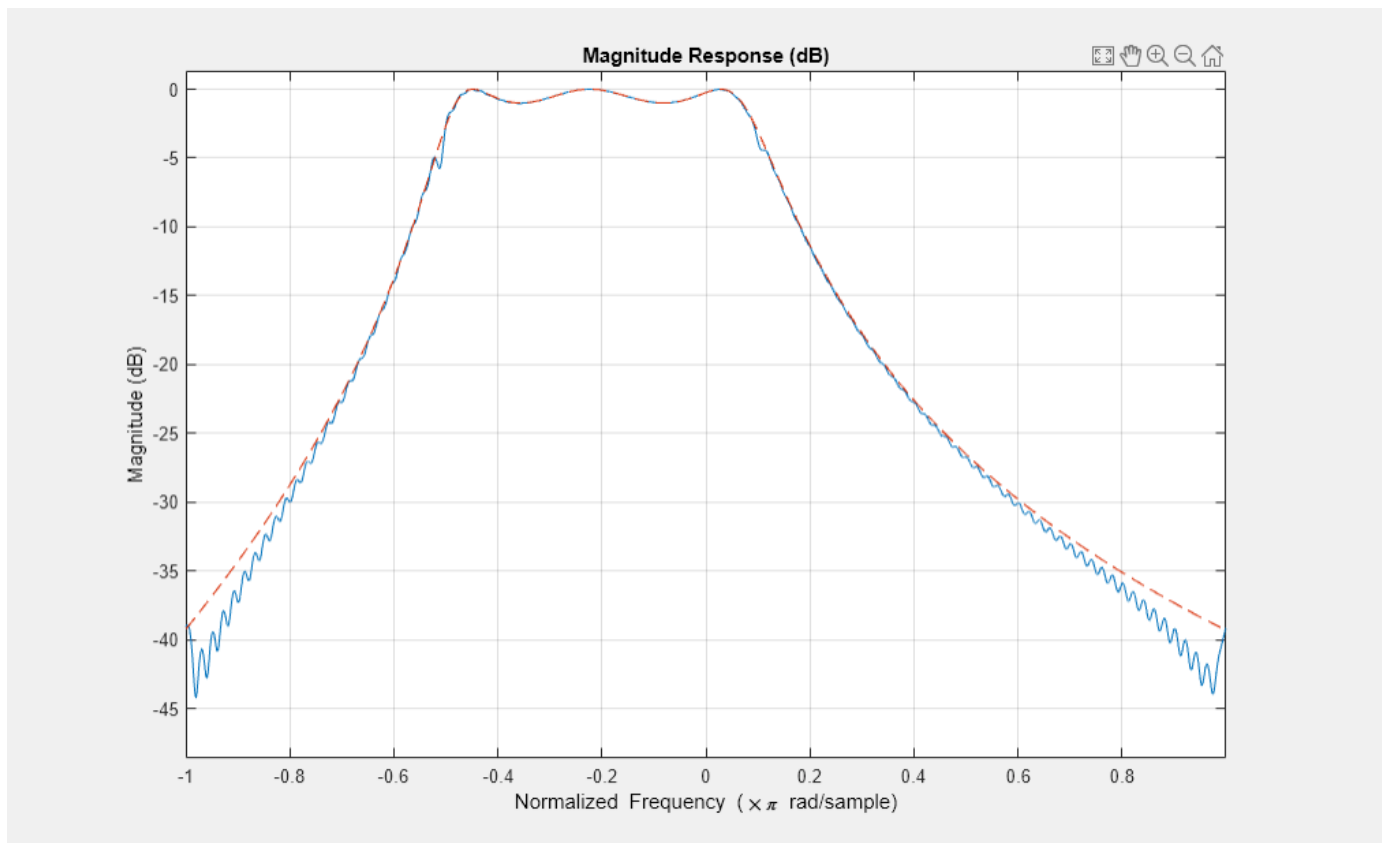
`d = fdesign.arbmagnphase(...,fs)` specifies the sampling frequency in Hz. All other frequency specifications are also assumed to be in Hz when you specify `fs`.

## Examples

### Construct a Complex Analog Filter

Use `fdesign.arbmagnphase` to model a complex analog filter.

```
d = fdesign.arbmagnphase('n,f,h',100); % N=100, f and h set to defaults.
design(d,'freqsamp','SystemObject',true);
```



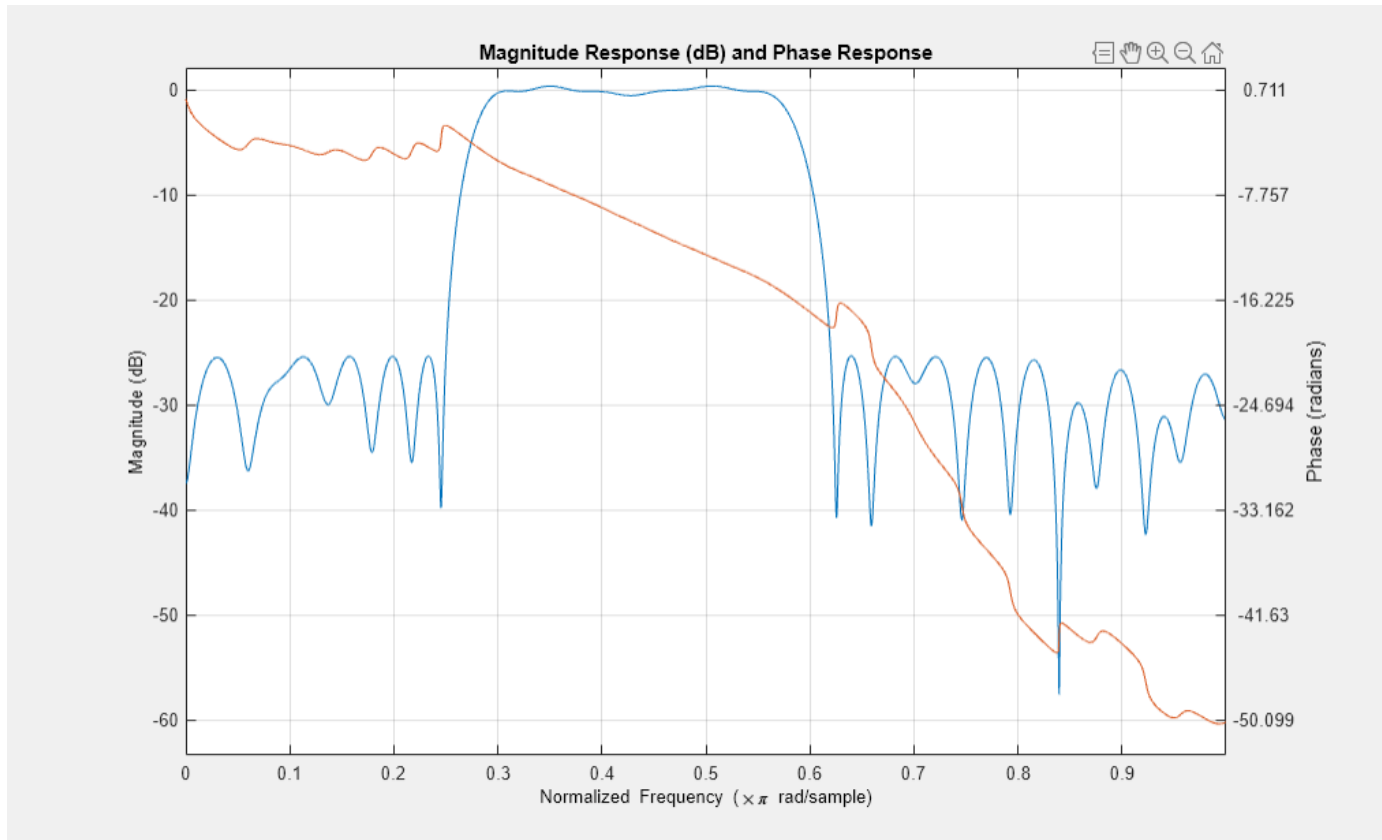
For a more complex example, design a bandpass filter with low group delay by specifying the desired delay and using `f` and `h` to define the filter bands.

```
n = 50; % Group delay of a linear phase filter would be 25.
gd = 12; % Set the desired group delay for the filter.
```

```

f1=linspace(0,.25,30); % Define the first stopband frequencies.
f2=linspace(.3,.56,40); % Define the passband frequencies.
f3=linspace(.62,1,30); % Define the second stopband frequencies.
h1 = zeros(size(f1)); % Specify the filter response at the freqs in f1.
h2 = exp(-1j*pi*gd*f2); % Specify the filter response at the freqs in f2.
h3 = zeros(size(f3)); % Specify the response at the freqs in f3.
d=fdesign.arbmagnphase('n,b,f,h',50,3,f1,h1,f2,h2,f3,h3);
D = design(d,'equiripple','SystemObject',true);
fvtool(D,'Analysis','freq');

```



## Version History

Introduced in R2011a

## See Also

fdesign | design | designmethods | setspecs

## fdesign.audioweighting

(To be removed) Audio weighting filter specification object

### Compatibility

---

**Note** The `fdesign.audioweighting` filter specification object will be removed from DSP System Toolbox in a future release. Existing instances of the object continue to run. For new code, use the `weightingFilter` System object instead. For more information on how to update your existing code, see “Compatibility Considerations” on page 5-426.

---

### Syntax

```
HAwf = fdesign.audioweighting
HAwf = fdesign.audioweighting(spec)
HAwf = fdesign.audioweighting(spec,specvalue1,specvalue2)
HAwf = fdesign.audioweighting(specvalue1,specvalue2)
HAwf = fdesign.audioweighting(...,Fs)
```

### Description

Supported audio weighting filter types are: A weighting, C weighting, C-message, ITU-T 0.41, and ITU-R 468-4 weighting.

`HAwf = fdesign.audioweighting` constructs an audio weighting filter specification object `HAwf` with a weighting type of A and a filter class of 1. Use the `design` method and set the 'SystemObject' flag to `true`, to instantiate a System object based on the specifications in `HAwf`. Use `designmethods` to find valid filter design methods. Because the standards for audio weighting filters are in Hz, normalized frequency specifications are not supported for `fdesign.audioweighting` objects. The default sampling frequency for A weighting, C weighting, C-message, and ITU-T 0.41 filters is 48 kHz. The default sampling frequency for the ITU-R 468-4 filter is 80 kHz. If you invoke the `normalizefreq` method, a warning is issued when you instantiate the System object and the default sampling frequencies in Hz are used.

`HAwf = fdesign.audioweighting(spec)` returns an audio weighting filter specification object using default values for the specification in `spec`. The following are valid entries for `spec`. The entries are not case sensitive.

- 'WT,Class' (default `spec`)

The 'WT,Class' specification is valid for A weighting and C weighting filters of class 1 or 2.

The weighting type is specified by the character vector: 'A' or 'C'. The class is the scalar 1 or 2.

The default values for 'WT,Class' are 'A',1.

- 'WT'

The 'WT' specification is valid for C-message (default), ITU-T 0.41, and ITU-R 468-4 weighting filters.

The weighting type is specified by the character vector: 'Cmessage', 'ITUT041', or 'ITUR4684'.

*HAWf* = fdesign.audioweighting(*spec*,*specvalue1*,*specvalue2*) constructs an audio weighting filter specification object *HAWf* and sets its specifications at construction time.

*HAWf* = fdesign.audioweighting(*specvalue1*,*specvalue2*) constructs an audio weighting filter specification object *HAWf* with the specification 'WT,Class' using the values you provide. The valid weighting types are 'A' or 'C'.

*HAWf* = fdesign.audioweighting(...,*Fs*) specifies the sampling frequency in Hz. The sampling frequency is a scalar trailing all other input arguments.

## Input Arguments

### Parameter Name/Value Pairs

#### WT

Weighting type

The weighting type defines the frequency response of the filter. The valid weighting types are: A weighting, C weighting, C-message, ITU-T 0.41, and ITU-R 468-4 weighting. The weighting types are described in “More About” on page 5-425.

#### Class

Filter Class

Filter class is only applicable for A weighting and C weighting filters. The filter class describes the frequency-dependent tolerances specified in the relevant standards [1], [2]. There are two possible class values: 1 and 2. Class 1 weighting filters have stricter tolerances than class 2 filters. The filter class value does not affect the design. The class value is only used to provide a specification mask in **FVTool** for the analysis of the filter design.

**Default:** 1

## Examples

### Compare Class 1 A and ITU-R 468-4 Weighting Filters

Compare class 1 A weighting and ITU-R 468-4 filters between 0.1 and 12 kHz. Sampling frequency is 44.1 kHz

```
HawfA = fdesign.audioweighting('WT,Class','A',1,44.1e3);
HawfITUR = fdesign.audioweighting('WT','ITUR4684',44.1e3);
```

```
Afilter = design(HawfA,'SystemObject',true);
ITURfilter = design(HawfITUR,'SystemObject',true);
```

```
hfvt = fvtool(Afilter,ITURfilter);
```

```
axis([0.1 12 -80 20]);
legend(hfvt, 'A-weighting', 'ITU-R 468-4');
```

### Compare C-message and ITU-T 0.41 Weighting Filters

```
hCmessage = fdesign.audioweighting('WT','Cmessage',24e3);
hITUT = fdesign.audioweighting('WT','ITUT041',24e3);
dCmessage = design(hCmessage,'SystemObject',true);
dITUT = design(hITUT,'SystemObject',true);
hfvt = fvtool(dCmessage,dITUT);
legend(hfvt,'C-Message Weighting','ITU-T 0.41 Weighting');
axis([0.1 10 -50 5]);
```

### Construct an ITU-R 468-4 Weighting Filter

Construct an ITU-R 468-4 filter using all available design methods.

```
HAwf = fdesign.audioweighting('WT','ITUR4684');
ValidDesigns = designmethods(HAwf);
% returns iirlpnorm,equiripple,freqsamp in cell array
D = design(HAwf,'all','SystemObject',true); % returns all designs
hfvt = fvtool(D{1},D{2},D{3});
legend(hfvt,'Least P-norm IIR','FIR Equiripple',...,
'FIR Frequency Sampling')
```

### Design and Use Audio Weighting Filter

Design a class 2 A-weighted filter.

```
fs = 48e3;
audioWeightingFilterDesign = fdesign.audioweighting('A',2,fs);
```

Convert the design to a System object for use.

```
audioWeightingFilter = design(audioWeightingFilterDesign,...
'SystemObject',true);
```

Create a spectrumAnalyzer object to visualize original and filtered signals.

```
windowLength = 2048;
scope = spectrumAnalyzer( ...
    'SampleRate',fs, ...
    'PlotAsTwoSidedSpectrum',false, ...
    'SpectralAverages',50, ...
    'FrequencyScale','Log', ...
    'FrequencyResolutionMethod','WindowLength', ...
    'WindowLength',windowLength, ...
    'Title','A-Weighted Filtering', ...
    'ShowLegend',true, ...
    'ChannelNames',{'Original signal','Filtered signal'});
```

Apply the A-weighting filter to white noise.



```
tic
while toc < 10
    x = rand(windowLength,1) - 0.5;
    y = audioWeightingFilter(x);
    scope([x,y])
end
```

## More About

### A weighting

The specifications for the A weighting filter are found in ANSI standard S1.42-2001. The A weighting filter is based on the 40-phon Fletcher-Munson equal loudness contour [3]. One phon is equal to one dB sound pressure level (SPL) at one kHz. The Fletcher-Munson equal loudness contours are designed to account for frequency and level dependent differences in the perceived loudness of tonal stimuli. The 40-phon contour reflects the fact that the human auditory system is more sensitive to frequencies around 1-2 kHz than lower and higher frequencies. The filter roll off is more pronounced at lower frequencies and more modest at higher frequencies. While A weighting is based on the equal loudness contour for low-level (40-phon) tonal stimuli, it is commonly used in the United States for assessing potential health risks associated with noise exposure to narrowband and broadband stimuli at high levels.

### C weighting

The specifications for the C weighting filter are found in ANSI standard S1.42-2001. The C weighting filter approximates the 100-phon Fletcher-Munson equal loudness contour for tonal stimuli. The C weighting magnitude response is essentially flat with 3-dB frequencies at 31.5 Hz and 8000 Hz. While C weighting is not as common as A weighting, sound level meters frequently have a C weighting filter option.

### C-message

The specifications for the C-message weighting filter are found in Bell System Technical Reference, PUB 41009. C-message weighting filters are designed for measuring the impact of noise on telecommunications circuits used in speech transmission [6]. C-message weighting filters are commonly used in North America, while the ITU-T 0.41 filter is more commonly used outside of North America.

### ITU-R 468-4

The specifications for the ITU-R 468-4 weighting filter are found in the International Telecommunication Union Recommendation ITU-R BS.468-4. ITU-R 468-4 is an equal loudness contour weighting filter. Unlike the A weighting filter, the ITU-R 468-4 filter describes subjective loudness judgements for broadband stimuli [4]. A common criticism of the A weighting filter is that it underestimates the loudness judgement of real-world stimuli particularly in the frequency band from about 1-9 kHz. A comparison of A weighting and ITU-R 468-4 weighting curves shows that the ITU-R 468-4 curve applies more gain between 1 and 10 kHz with a peak difference of approximately 12 dB around 6-7 kHz.

### ITU-T 0.41

The specifications for the ITU-T 0.41 filter are found in the ITU-T Recommendation 0.41. ITU-T 0.41 weighting filters are designed for measuring the impact of noise on telecommunications circuits used

in speech transmission [5]. ITU-T 0.41 weighting filters are commonly used outside of North America, while the C-message weighting filter is more common in North America.

## Version History

Introduced in R2011a

### R2022a: `fdesign.audioweighting` will be removed

Warns starting in R2022a

The `fdesign.audioweighting` filter specification object will be removed in a future release. Existing instances of the object continue to run. For new code, use the `weightingFilter` System object instead.

### Update Code

This table shows how the object is typically used and explains how to update the existing code to use the `weightingFilter` object. The `weightingFilter` object requires Audio Toolbox.

| Discouraged Usage   | Recommended Replacement  |
|---|--|
| <p><b>A-weighting filter</b></p> <p>Design a class 2, A-weighting filter for a sample rate of 44.1 kHz.</p> <pre>Fs = 44100; d = fdesign.audioweighting('WT,Class','A',2,Fs); wf = design(d,'Systemobject',true);</pre> <p>Visualize the filter response and make sure it fits within the Class 2 mask.</p> <pre>fvtool(wf)</pre> | <p><b>A-weighting filter</b></p> <p>Design a class 2, A-weighting filter for a sample rate of 44.1 kHz.</p> <pre>Fs = 44100; wf = weightingFilter('A-weighting',Fs);</pre> <p>Visualize the filter response and make sure it fits within the Class 2 mask.</p> <pre>visualize(wf,'class 2');</pre> |
| <p><b>C-weighting filter</b></p> <p>Design a class 2, C-weighting filter for a sample rate of 44.1 kHz.</p> <pre>Fs = 44100; d = fdesign.audioweighting('WT,Class','C',2,Fs); wf = design(d,'Systemobject',true);</pre> <p>Visualize the filter response and make sure it fits within the Class 2 mask.</p> <pre>fvtool(wf)</pre> | <p><b>C-weighting filter</b></p> <p>Design a class 2, C-weighting filter for a sample rate of 44.1 kHz.</p> <pre>Fs = 44100; wf = weightingFilter('C-weighting',Fs);</pre> <p>Visualize the filter response and make sure it fits within the Class 2 mask.</p> <pre>visualize(wf,'class 2');</pre> |

## References

- [1] *American National Standard Design Response of Weighting Networks for Acoustical Measurements*, ANSI S1.42-2001, Acoustical Society of America, New York, NY, 2001.

- [2] *Electroacoustics Sound Level Meters Part 1: Specifications*, IEC 61672-1, First Edition 2002-05.
- [3] Fletcher, H. and W.A. Munson. "Loudness, its definition, measurement and calculation." *Journal of the Acoustical Society of America*, Vol. 5, 1933, pp. 82-108.
- [4] *Measurement of Audio-Frequency Noise Voltage Level in Sound Broadcasting*, International Telecommunication Union Recommendation ITU-R BS.468-4, 1986.
- [5] *Psophometer for Use on Telephone-Type Circuits*, ITU-T Recommendation 0.41.
- [6] *Transmission Parameters Affecting Voiceband Data Transmission-Measuring Techniques*, Bell System Technical Reference, PUB 41009, 1972.

## **See Also**

### **Objects**

weightingFilter

### **Tools**

FVTool

### **Topics**

"Audio Weighting Filters" (Audio Toolbox)

"Design a Filter in Fdesign — Process Overview"

## fdesign.bandpass

Bandpass filter design specification object

### Syntax

```
bandpassSpecs = fdesign.bandpass
bandpassSpecs = fdesign.bandpass(spec,value1,...,valueN)
bandpassSpecs = fdesign.bandpass( ___,Fs)
bandpassSpecs = fdesign.bandpass( ___,magunits)
```

### Description

The `fdesign.bandpass` function returns a bandpass filter design specification object that contains specifications for a filter such as passband frequency, stopband frequency, passband ripple, and filter order. Use the `design` function to design the filter from the filter design specifications object.

For more control options, see “Filter Design Procedure” on page 5-399. For a complete workflow, see “Design a Filter in Fdesign — Process Overview”.

`bandpassSpecs = fdesign.bandpass` constructs a bandpass filter design specifications object with the following default values:

- First stopband frequency set to 0.35.
- First passband frequency set to 0.45.
- Second passband frequency set to 0.55.
- Second stopband frequency set to 0.65.
- First stopband attenuation set to 60 dB.
- Passband ripple set to 1dB.
- Second stopband attenuation set to 60 dB.

`bandpassSpecs = fdesign.bandpass(spec,value1,...,valueN)` constructs a bandpass filter specification object with a particular filter order, stopband frequency, passband frequency, and other specification options. Indicate the options you want to specify in the expression `spec`. After the expression, specify a value for each option. If you do not specify values after the `spec` argument, the function assumes the default values.

`bandpassSpecs = fdesign.bandpass( ___,Fs)` provides the sample rate in Hz of the signal to be filtered. `Fs` must be specified as a scalar trailing the other numerical values provided. In this case, all frequencies in the specifications are in Hz as well.

The design specification

```
fdesign.bandpass('Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2',.4,.5,.6,.7,60,1,80)
```

designs the same filter as

```
fdesign.bandstop('Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2',1600,2000,2400,2800,60,1,80,8000)
```

`bandpassSpecs = fdesign.bandpass( ___,magunits)` provides the units for the specified magnitude. `magunits` can be one of the following: 'linear', 'dB', or 'squared'. If this argument

is omitted, the object assumes the units of magnitude specification to be 'dB'. The magnitude specifications are always converted and stored in decibels regardless of how they were specified. If Fs is provided, magunits must follow Fs in the input argument list.

## Examples

### Design Equiripple FIR Bandpass Filter

Design a constrained-band FIR equiripple filter of order 100 with a passband of [1, 1.4] kHz. Both stopband attenuation values are constrained to 60 dB. The sample rate is 10 kHz.

Create a `bandpass` filter design specification object using the `fdesign.bandpass` function and specify these design parameters.

```
bandpassSpecs = fdesign.bandpass('N,Fst1,Fp1,Fp2,Fst2,C',100,800,1e3,1.4e3,1.6e3,1e4);
```

Constrain the two stopbands with a stopband attenuation of 60 dB.

```
bandpassSpecs.Stopband1Constrained = true;
bandpassSpecs.Astop1 = 60;
bandpassSpecs.Stopband2Constrained = true;
bandpassSpecs.Astop2 = 60;
```

Design the bandpass filter using the `design` function. The resulting filter is a `dsp.FIRFilter` System object™. For details on how to apply this filter on streaming data, refer to `dsp.FIRFilter`.

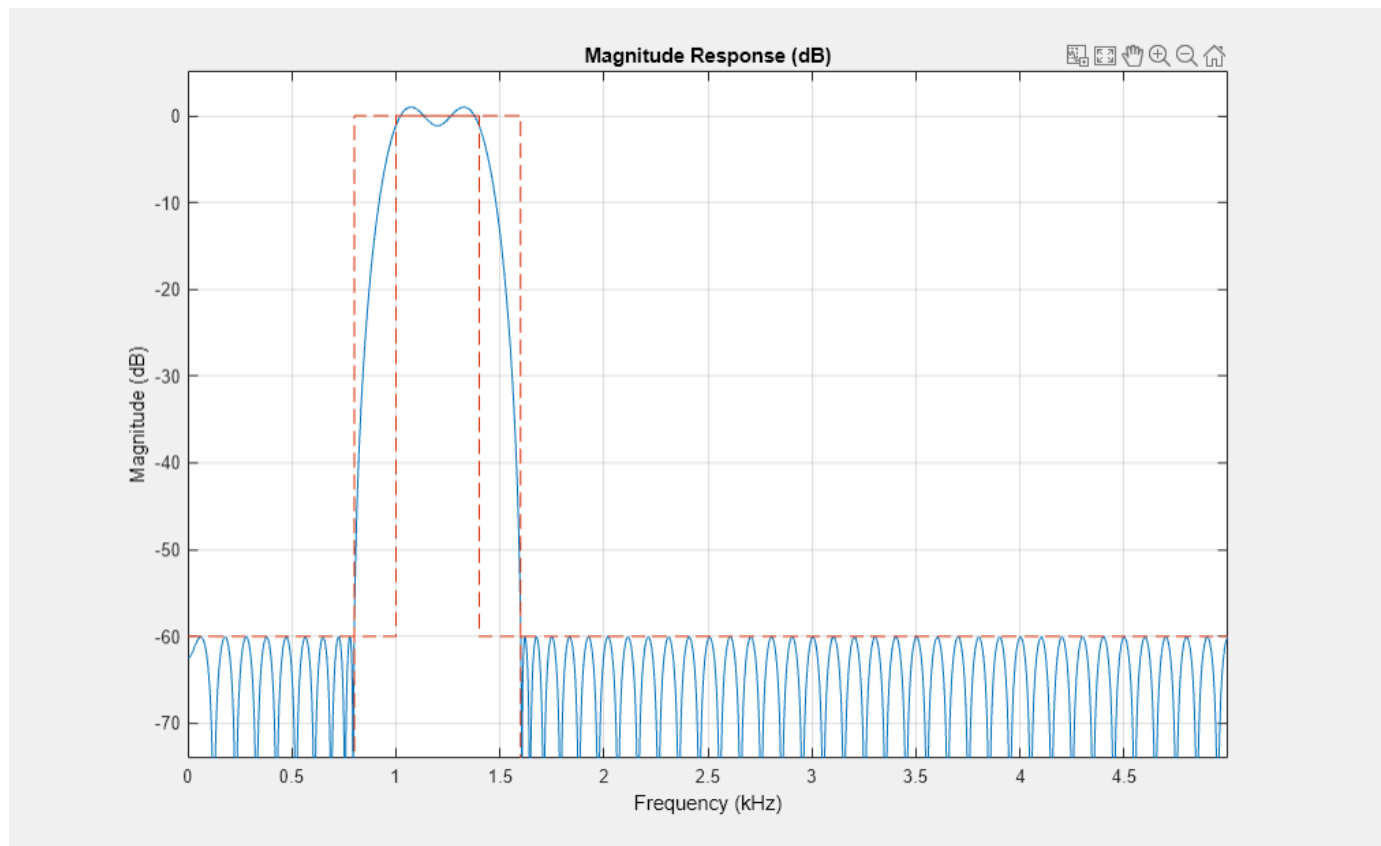
```
bandpassFilt = design(bandpassSpecs,'Systemobject',true)
```

```
bandpassFilt =
  dsp.FIRFilter with properties:
      Structure: 'Direct form'
      NumeratorSource: 'Property'
      Numerator: [5.5055e-04 5.4751e-05 -2.2052e-05 6.5244e-05 3.6129e-04 5.7237e-04 1.982
      InitialConditions: 0
```

Show all properties

Visualize the frequency response of the designed filter using `fvtool`.

```
fvtool(bandpassFilt)
```



Measure the frequency response characteristics of the filter using `measure`. The passband ripple is slightly over 2 dB. Because the design constrains both stopbands, you cannot constrain the passband ripple.

```
measure(bandpassFilt)
```

```
ans =
Sample Rate           : 10 kHz
First Stopband Edge   : 800 Hz
First 6-dB Point      : 946.7621 Hz
First 3-dB Point      : 975.1807 Hz
First Passband Edge   : 1 kHz
Second Passband Edge  : 1.4 kHz
Second 3-dB Point     : 1.4248 kHz
Second 6-dB Point     : 1.4533 kHz
Second Stopband Edge  : 1.6 kHz
First Stopband Atten. : 60.0614 dB
Passband Ripple       : 2.1443 dB
Second Stopband Atten.: 60.0399 dB
First Transition Width : 200 Hz
Second Transition Width : 200 Hz
```

## Design Butterworth IIR Bandpass Filter

Design a Butterworth IIR bandpass filter. The filter design procedure is:

- 1 Specify the filter design specifications using a `fdesign` function.
- 2 Pick a design method provided by the `designmethods` function.
- 3 To determine the available design options to choose from, use the `designoptions` function.
- 4 Design the filter using the `design` function.

Construct a default bandpass filter design specification object using `fdesign.bandpass`.

```
bandpassSpecs = fdesign.bandpass

bandpassSpecs =
    bandpass with properties:
        Response: 'Bandpass'
        Specification: 'Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2'
        Description: {7x1 cell}
        NormalizedFrequency: 1
            Fstop1: 0.3500
            Fpass1: 0.4500
            Fpass2: 0.5500
            Fstop2: 0.6500
            Astop1: 60
            Apass: 1
            Astop2: 60
```

Determine the available `designmethods` using the `designmethods` function. To design a Butterworth filter, pick `butter`.

```
designmethods(bandpassSpecs, 'Systemobject', true)
```

Design Methods that support System objects for class `fdesign.bandpass` (Fst1,Fp1,Fp2,Fst2,Ast1,Ap

```
butter
cheby1
cheby2
ellip
equiripple
kaiserwin
```

When designing the filter, you can specify additional design options. View a list of options using the `designoptions` function. The function also shows the default design options the filter uses.

```
designoptions(bandpassSpecs, 'butter')
```

```
ans = struct with fields:
    FilterStructure: {'df1sos' 'df2sos' 'df1tsos' 'df2tsos' 'cascadeallpass' 'cascad
    SOSScaleNorm: 'ustring'
    SOSScaleOpts: 'fdopts.sosscaling'
    MatchExactly: {'passband' 'stopband'}
    SystemObject: 'bool'
    DefaultFilterStructure: 'df2sos'
    DefaultMatchExactly: 'stopband'
```

```

DefaultSOSScaleNorm: ''
DefaultSOSScaleOpts: [1x1 fdopts.sosscaling]
DefaultSystemObject: 0

```

Use the `design` function to design the filter. Pass `'butter'` and the specifications given by variable `bandpassSpecs`, as input arguments. Specify the `'matchexactly'` design option to `'passband'`.

```
bpFilter = design(bandpassSpecs, 'butter', 'matchexactly', 'passband', 'SystemObject', true)
```

```

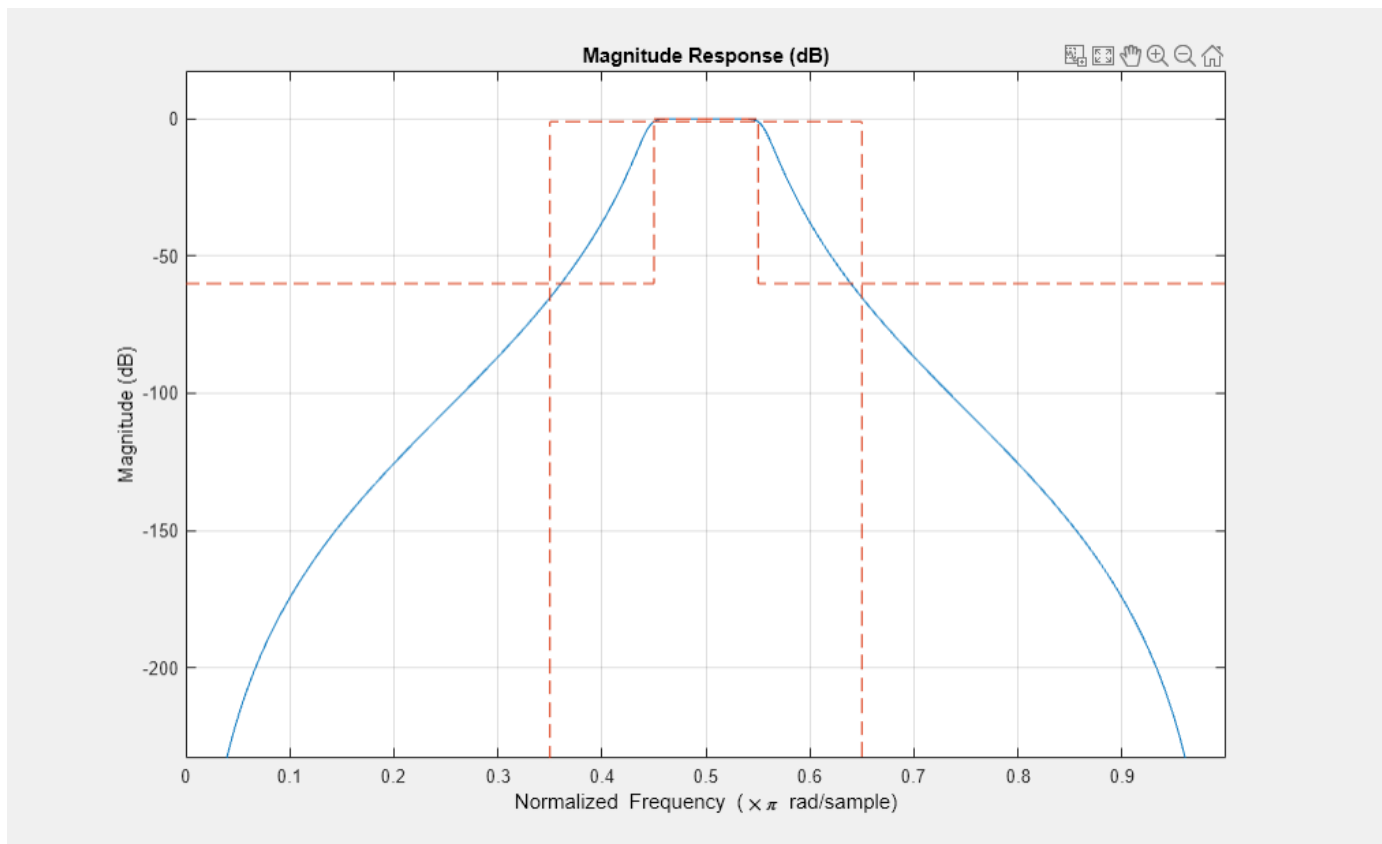
bpFilter =
  dsp.SOSFilter with properties:
    Structure: 'Direct form II'
    CoefficientSource: 'Property'
    Numerator: [7x3 double]
    Denominator: [7x3 double]
    HasScaleValues: true
    ScaleValues: [0.1657 0.1657 0.1561 0.1561 0.1504 0.1504 0.1485 1]

```

Show all properties

Visualize the frequency response of the designed filter.

```
fvtool(bpFilter)
```





## Bandpass Filtering of Sinusoids

Bandpass filter a discrete-time sine wave signal which consists of three sinusoids at frequencies, 1 kHz, 10 kHz, and 15 kHz.

Design an FIR Equiripple bandpass filter by first creating a bandpass filter design specifications object, and then designing a filter using these specifications.

### Design Bandpass Filter

Create a bandpass filter design specifications object using `fdesign.bandpass`.

```
bandpassSpecs = fdesign.bandpass('Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2', ...
    1/4,3/8,5/8,6/8,60,1,60);
```

List the available design methods for this object.

```
designmethods(bandpassSpecs)
```

```
Design Methods for class fdesign.bandpass (Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2):
```

```
butter
cheby1
cheby2
ellip
equiripple
kaiserwin
```

To design an Equiripple filter, pick 'equiripple'.

```
bpFilter = design(bandpassSpecs, 'equiripple', 'Systemobject', true)
```

```
bpFilter =
```

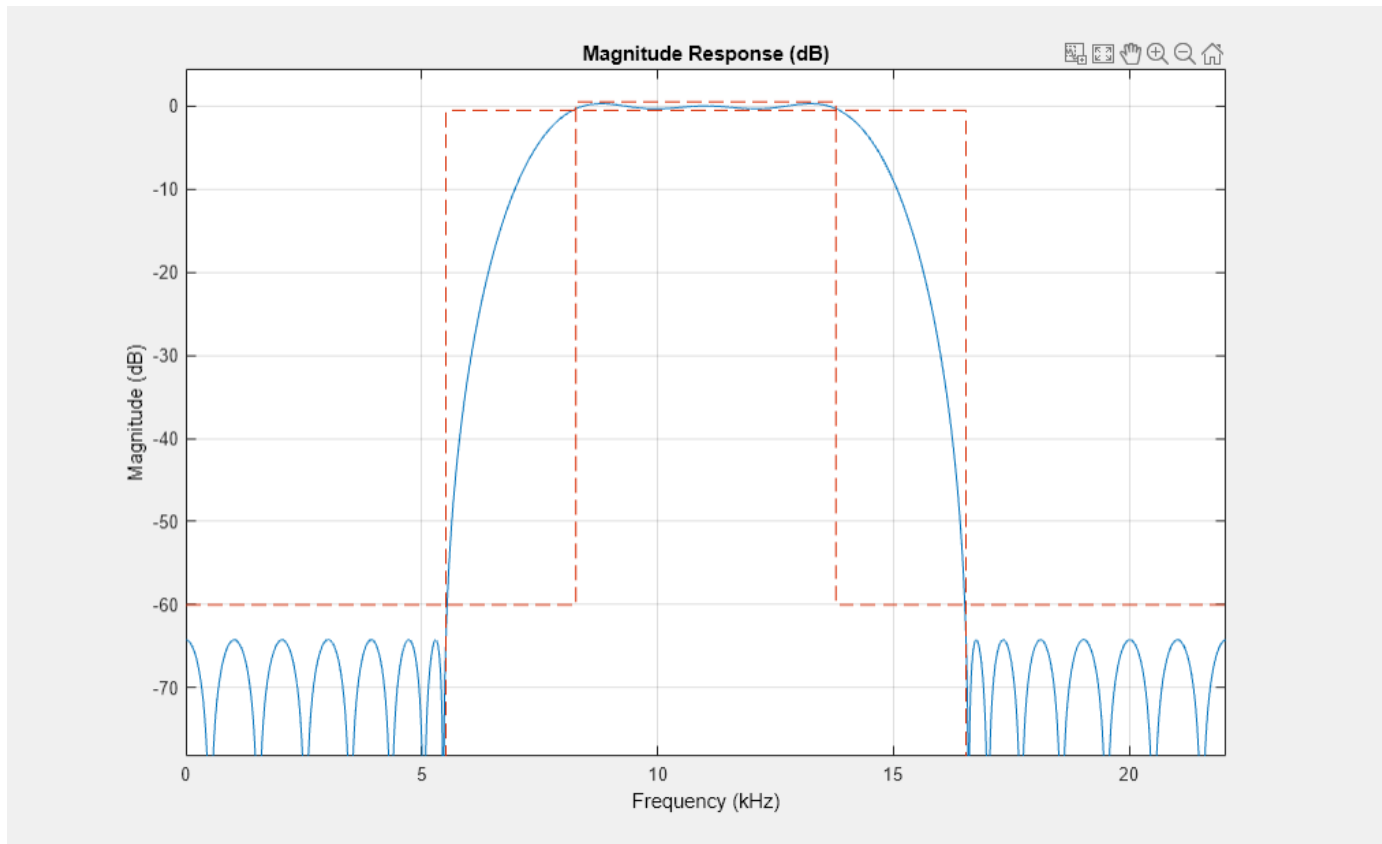
```
  dsp.FIRFilter with properties:
```

```
      Structure: 'Direct form'
      NumeratorSource: 'Property'
      Numerator: [-0.0043 -3.0812e-15 0.0136 3.7820e-15 -0.0180 -4.2321e-15 7.1634e-04 4.0
      InitialConditions: 0
```

```
  Show all properties
```

Visualize the frequency response of the designed filter.

```
fvtool(bpFilter, 'Fs', 44100)
```



### Create Sinusoidal Signal

Create a signal that is a sum of three sinusoids with frequencies at 1 kHz, 10 kHz, and 15 kHz. Initialize Spectrum Analyzer to view the original signal and the filtered signal.

```
Sine1 = dsp.SineWave('Frequency',1e3,'SampleRate',44.1e3,'SamplesPerFrame',4000);
Sine2 = dsp.SineWave('Frequency',10e3,'SampleRate',44.1e3,'SamplesPerFrame',4000);
Sine3 = dsp.SineWave('Frequency',15e3,'SampleRate',44.1e3,'SamplesPerFrame',4000);
```

```
SpecAna = spectrumAnalyzer('PlotAsTwoSidedSpectrum',false, ...
    'SampleRate',Sine1.SampleRate, ...
    'ShowLegend',true, ...
    'YLimits',[-240,45]);
```

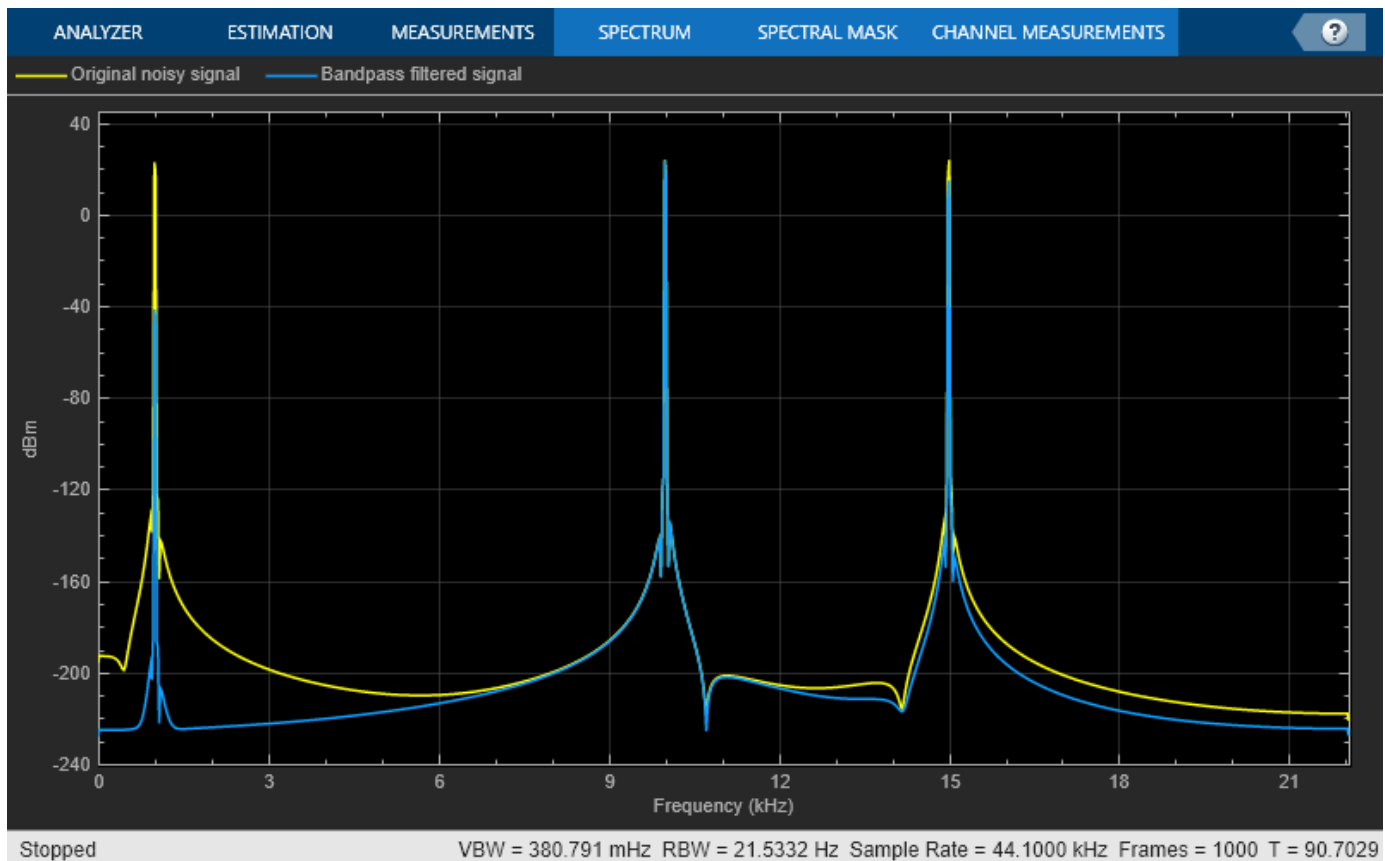
```
SpecAna.ChannelNames = {'Original noisy signal','Bandpass filtered signal'};
```

### Filter Sinusoidal Signal

Filter the sinusoidal signal using the bandpass filter that has been designed. View the original signal and the filtered signal in the Spectrum Analyzer. The tone at 1 kHz is filtered out and attenuated. The tone at 10 kHz is unaffected, and the tone at 15 kHz is mildly attenuated because it appears in the transition band of the filter.

```
for i = 1 : 1000
    x = Sine1()+Sine2()+Sine3();
    y = bpFilter(x);
    SpecAna(x,y);
```

```
end
release(SpecAna)
```



## Input Arguments

### spec — Specification

'Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2' (default) | 'N,F3dB1,F3dB2' | 'N,F3dB1,F3dB2,Ap' | ...

Specification expression, specified as one of these character vectors:

- 'Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2' (default)
- 'N,F3dB1,F3dB2'
- 'N,F3dB1,F3dB2,Ap'
- 'N,F3dB1,F3dB2,Ast'
- 'N,F3dB1,F3dB2,Ast1,Ap,Ast2'
- 'N,F3dB1,F3dB2,BWp'
- 'N,F3dB1,F3dB2,BWst'
- 'N,Fc1,Fc2'
- 'N,Fc1,Fc2,Ast1,Ap,Ast2'
- 'N,Fp1,Fp2,Ap'

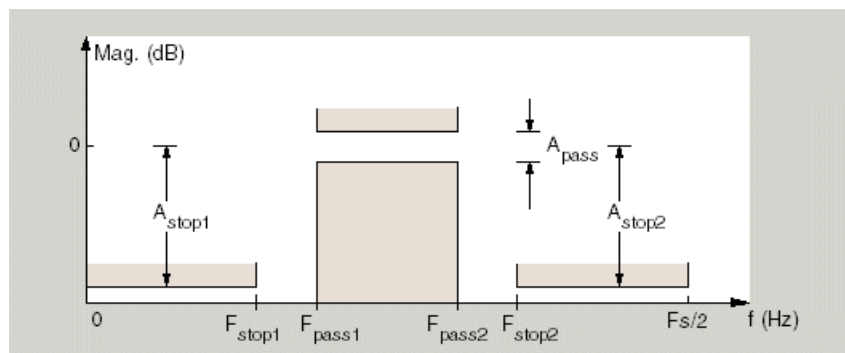
- 'N,Fp1,Fp2,Ast1,Ap,Ast2'
- 'N,Fst1,Fp1,Fp2,Fst2'
- 'N,Fst1,Fp1,Fp2,Fst2,C'
- 'N,Fst1,Fp1,Fp2,Fst2,Ap'
- 'N,Fst1,Fst2,Ast'
- 'Nb,Na,Fst1,Fp1,Fp2,Fst2'

This table describes each option that can appear in the expression.

| Specification option | Description  |
|----------------------|--|
| Ap                   | Amount of ripple allowed in passband, specified as <i>Apass</i> in dB.   |
| Ast                  | Stopband attenuation (dB), specified using <i>Astop</i> .  |
| Ast1                 | Attenuation in the first stopband (dB), specified using <i>Astop1</i> .  |
| Ast2                 | Attenuation in the second stopband (dB), specified using <i>Astop2</i> .   |
| BWp                  | Bandwidth of the filter passband, specified as <i>BWpass</i> in normalized frequency units.  |
| BWst                 | Frequency width between the two stopband frequencies, specified as <i>BWstop</i> in normalized frequency units.                              |
| F3dB1                | Frequency of the 3 dB point below the passband value for the first cutoff, specified in normalized frequency units. Applies to IIR filters.  |
| F3dB2                | Frequency of the 3 dB point below the passband value for the second cutoff, specified in normalized frequency units. Applies to IIR filters. |
| Fc1                  | First cutoff frequency (normalized frequency units), specified using <i>Fcutoff1</i> . Applies to FIR filters.                               |
| Fc2                  | Second cutoff frequency (normalized frequency units), specified using <i>Fcutoff1</i> . Applies to FIR filters.                              |
| Fp1                  | Frequency at the edge of the start of the passband, specified as <i>Fpass1</i> in normalized frequency units.                                |
| Fp2                  | Frequency at the edge of the end of the passband, specified as <i>Fpass2</i> in normalized frequency units.                                  |
| Fst1                 | Frequency at the edge of the end of the first stopband, specified as <i>Fstop1</i> in normalized frequency units.                            |

| Specification option | Description  |
|----------------------|--|
| Fst2                 | Frequency at the edge of the start of the second stop band, specified as Fstop2 in normalized frequency units.   |
| N                    | Filter order for FIR filters. Or both the numerator and denominator orders for IIR filters when Na and Nb are not provided. Specified using FilterOrder.                             |
| Nb                   | Numerator order for IIR filters, specified using the NumOrder property.  |
| Na                   | Denominator order for IIR filters, specified using the DenOrder property.  |
| C                    | Constrained band flag. This enables you to specify passband ripple or stopband attenuation for fixed-order designs in one or two of the three bands.<br><br>For more details, see c. |

Graphically, the filter specifications look similar to those shown in this figure.



Regions between specification values like Fst1 and Fp1 are transition regions where the filter response is not explicitly defined.

The design methods available for designing the filter depend on the specification expression. You can obtain these methods using the `designmethods` function. This table lists each specification expression supported by `fdesign.bandpass` and the available corresponding design methods.

| Specification expression               | Supported design methods                             |
|--|--|
| 'Fst1, Fp1, Fp2, Fst2, Ast1, Ap, Ast2' | butter, cheby1, cheby2, ellip, equiripple, kaiserwin |
| 'N, F3dB1, F3dB2'                      | butter   |
| 'N, F3dB1, F3dB2, Ap'                  | cheby1   |
| 'N, F3dB1, F3dB2, Ast'                 | cheby2, ellip  |
| 'N, F3dB1, F3dB2, Ast1, Ap, Ast2'      | ellip  |
| 'N, F3dB1, F3dB2, Bwp'                 | cheby1   |

| Specification expression  | Supported design methods      |
|---------------------------|-------------------------------|
| 'N,F3dB1,F3dB2,BWst'      | cheby2                        |
| 'N,Fc1,Fc2'               | window                        |
| 'N,Fc1,Fc2,Ast1,Ap,Ast2'  | fircls                        |
| 'N,Fp1,Fp2,Ap'            | cheby1                        |
| 'N,Fp1,Fp2,Ast1,Ap,Ast2'  | ellip                         |
| 'N,Fst1,Fp1,Fp2,Fst2'     | iirlpnorm, equiripple, fircls |
| 'N,Fst1,Fp1,Fp2,Fst2,C'   | equiripple                    |
| 'N,Fst1,Fp1,Fp2,Fst2,Ap'  | ellip                         |
| 'N,Fst1,Fst2,Ast'         | cheby2                        |
| 'Nb,Na,Fst1,Fp1,Fp2,Fst2' | iirlpnorm                     |

To design the filter, call the `design` function with one of these design methods as an input. You can choose the type of filter response by passing 'FIR' or 'IIR' to the `design` function. For more details, see `design`. Enter `help(bandpassSpecs, 'method')` at the MATLAB command line to obtain detailed help on the design options for a given design method.

#### **value1, ..., valueN — Specification values**

comma-separated list of values

Specification values, specified as a comma-separated list of values. Specify a value for each option in `spec` in the same order that the options appear in the expression.

Example: `bandpassSpecs = fdesign.bandpass('N,Fc1,Fc2,Ast1,Ap,Ast2',n,fc1,fc2,ast1,ap,ast2)`

The input arguments below provide more details for each option in the expression.

#### **n — Filter order**

positive integer

Filter order for FIR filters, specified as a positive integer. In the case of an IIR filter design, if `nb` and `na` are not provided, this value is interpreted as both the numerator order and the denominator order.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

#### **nb — Numerator order for IIR filters**

nonnegative integer

Numerator order for IIR filters, specified as a nonnegative integer.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

#### **na — Denominator order for IIR filters**

positive integer

Denominator order for IIR filters, specified as a positive integer.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

#### **c — Constrained band flag**

logical

This enables you to specify passband ripple or stopband attenuation for fixed-order designs in one or two of the three bands.

In the specification 'N,Fst1,Fp1,Fp2,Fst2,C', you cannot specify constraints for all three bands (two stopbands and one passband) simultaneously. You can specify constraints in any one or two bands.

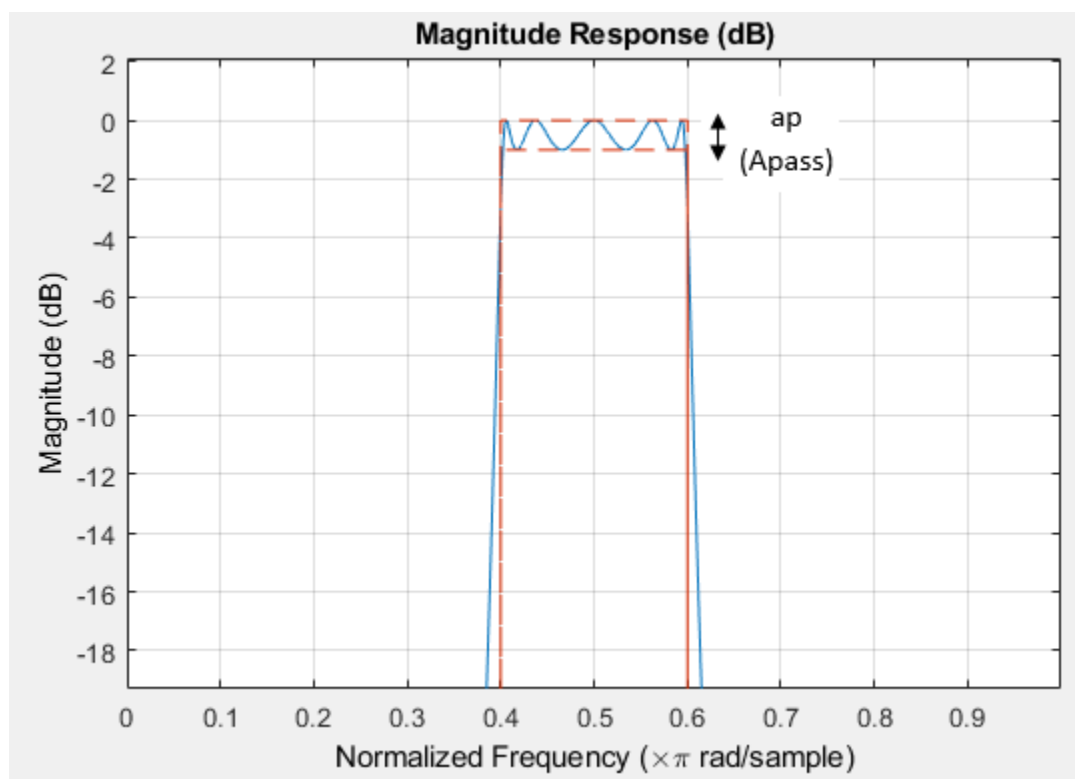
Consider the following bandpass design specification where both the stopbands are constrained to the default value 60 dB.

```
Example: spec =
fdesign.bandpass('N,Fst1,Fp1,Fp2,Fst2,C',100,800,1e3,1.4e3,1.6e3,1e4);
spec.Stopband1Constrained=true; spec.Stopband2Constrained=true;
```

### ap — Passband ripple

positive scalar

Passband ripple, specified as a positive scalar in dB. If `magunits` is 'linear' or 'squared', the passband ripple is converted and stored in dB by the function regardless of how it has been specified.

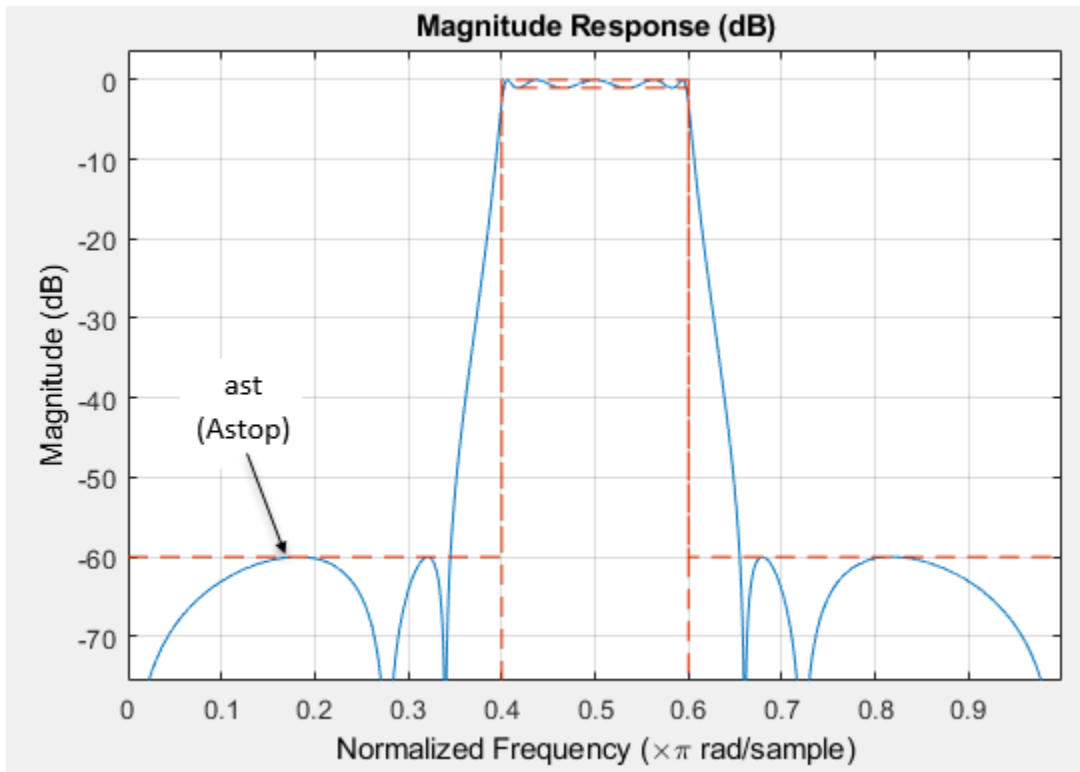


Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### ast — Stopband attenuation

positive scalar

Stopband attenuation, specified as a positive scalar in dB. If `magunits` is 'linear' or 'squared', the stopband attenuation is converted and stored in dB by the function regardless of how it has been specified.



Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### **ast1 — First stopband attenuation**

positive scalar

Attenuation in the first stopband, specified as a positive scalar in dB. If `magunits` is 'linear' or 'squared', the first stopband attenuation is converted and stored in dB by the function regardless of how it has been specified.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### **ast2 — Second stopband attenuation**

positive scalar

Attenuation in the second stopband, specified as a positive scalar in dB. If `magunits` is 'linear' or 'squared', the second stopband attenuation is converted and stored in dB by the function regardless of how it has been specified.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

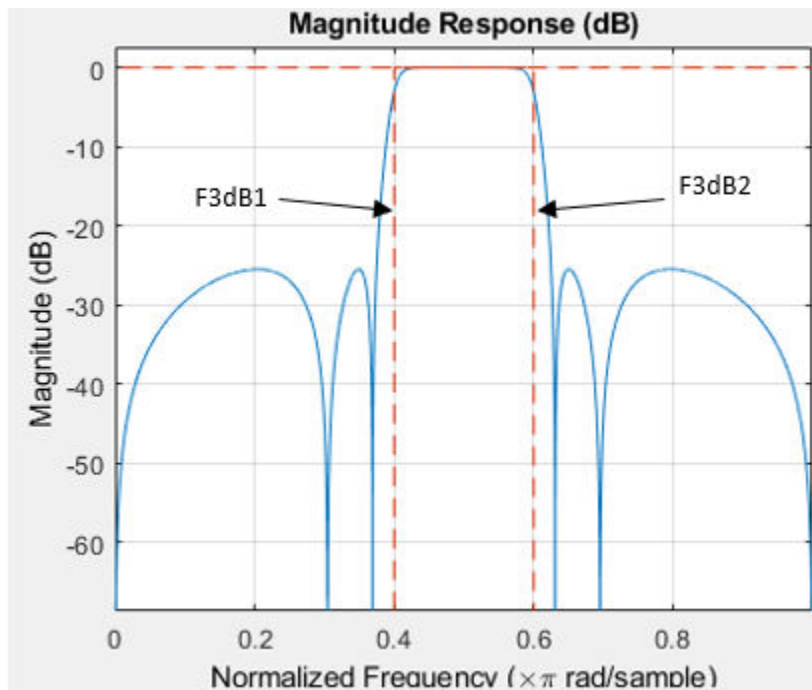
### **F3dB1 — First 3 dB frequency**

positive scalar

First 3 dB frequency, specified as positive scalar in normalized frequency units.

This is the frequency of the 3 dB point below the passband value for the first cutoff. This input argument applies to IIR filters only.





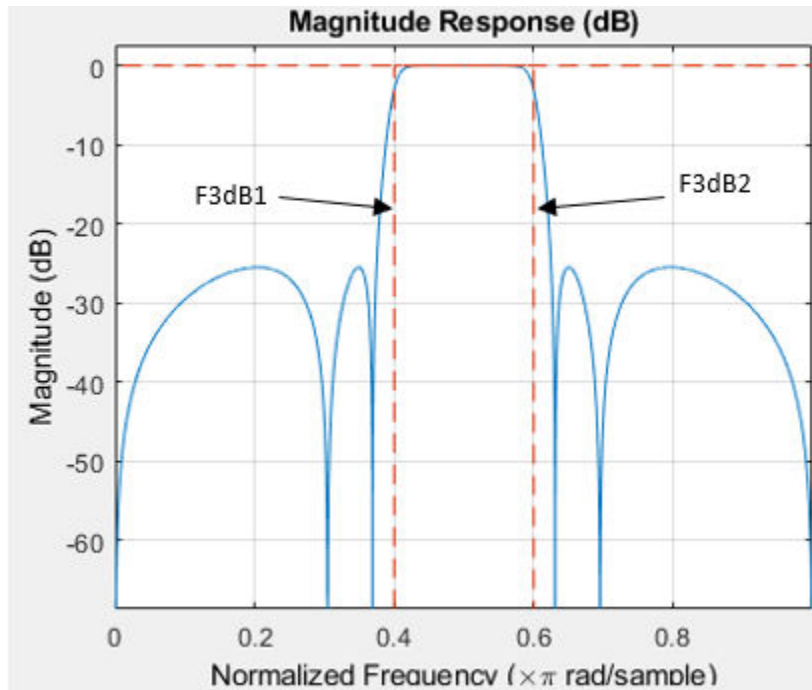
Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### **F3dB2 — Second 3 dB frequency**

positive scalar

Second 3 dB frequency, specified as positive scalar in normalized frequency units.

This is the frequency of the 3 dB point below the passband value for the second cutoff. This input argument applies to IIR filters only.



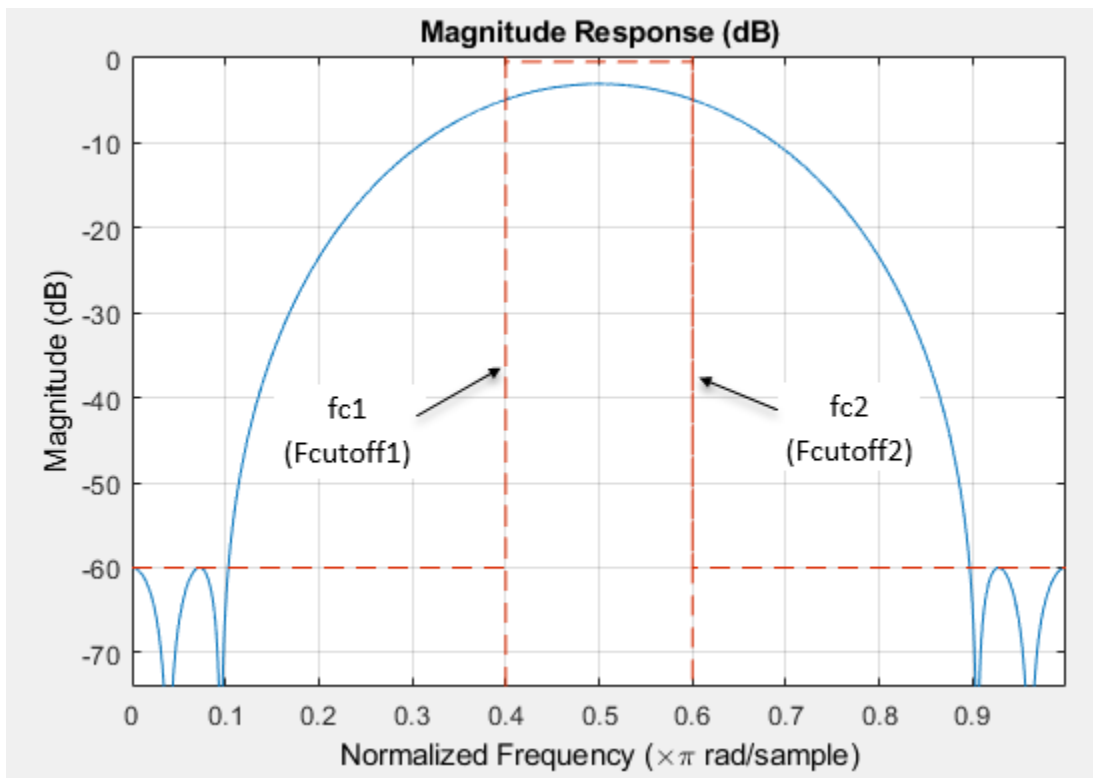
Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**fc1 – First cutoff frequency**

positive scalar

First cutoff frequency, specified as positive scalar in normalized frequency units.

This input argument applies to FIR filters only.



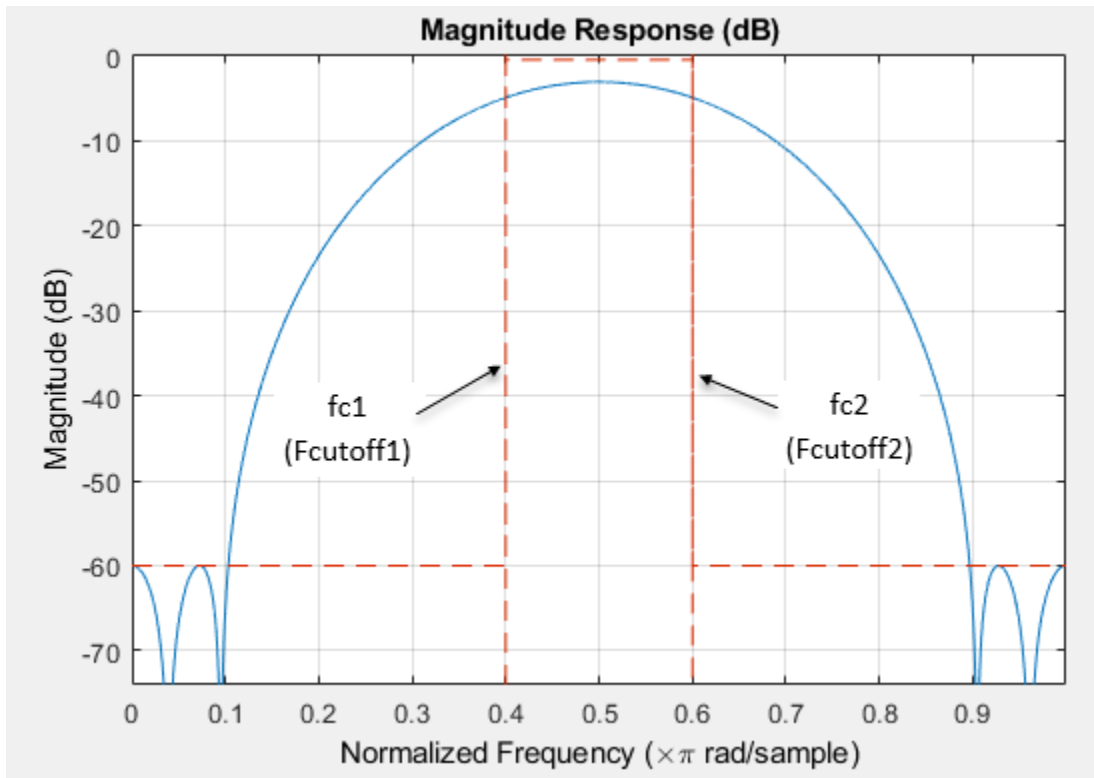
Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### **fc2 — Second cutoff frequency**

positive scalar

Second cutoff frequency, specified as positive scalar in normalized frequency units.

This input argument applies to FIR filters only.



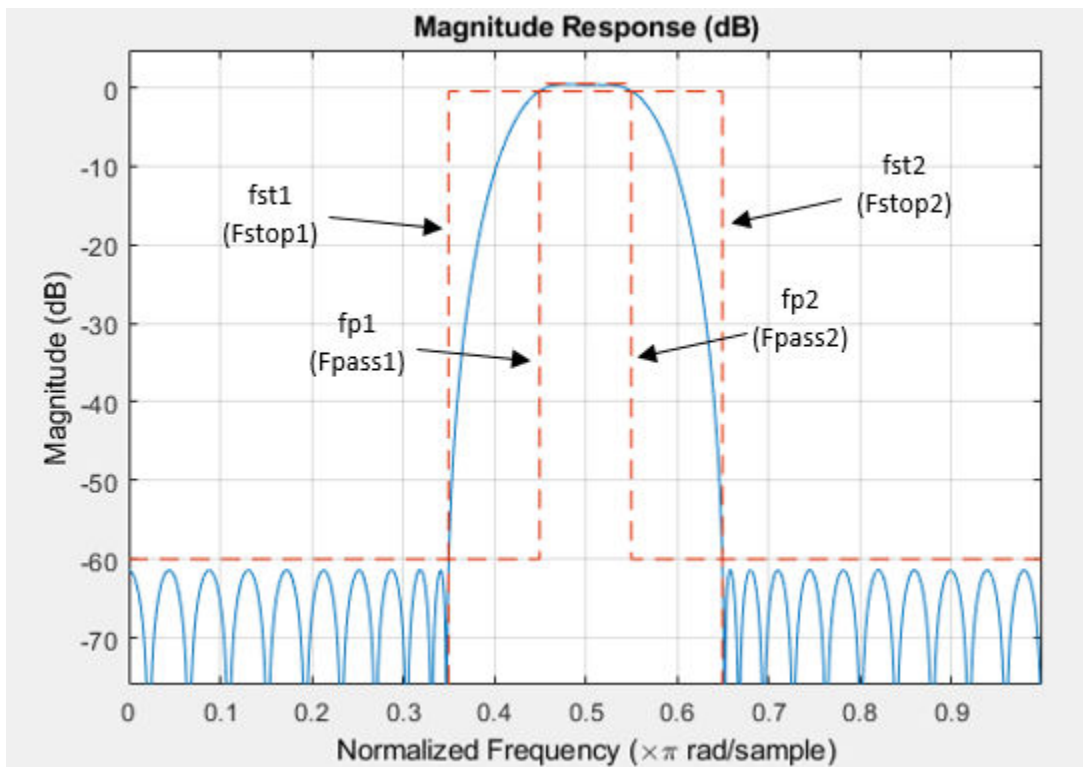
Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### **fst1 – First stopband frequency**

positive scalar

First stopband frequency, specified as positive scalar in normalized frequency units.

This is the frequency at the edge of the end of the first stopband.



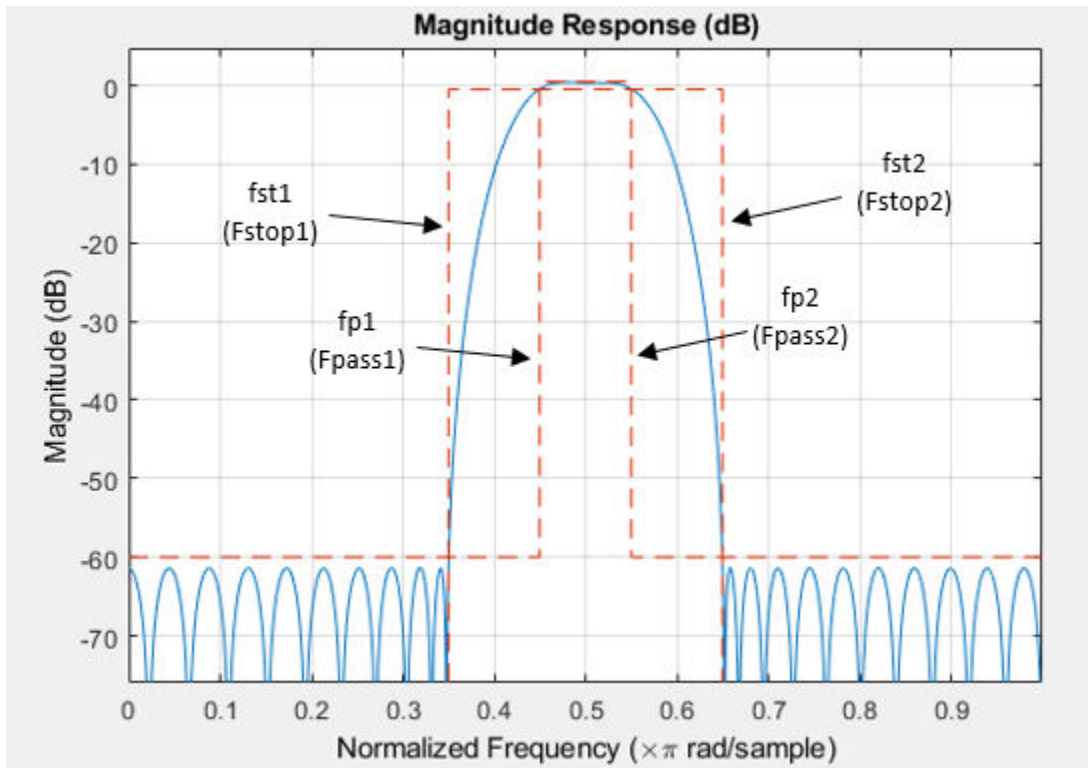
Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### **fst2 – Second stopband frequency**

positive scalar

Second stopband frequency, specified as a positive scalar in normalized frequency units.

This is the frequency at the edge of the start of the second stopband.



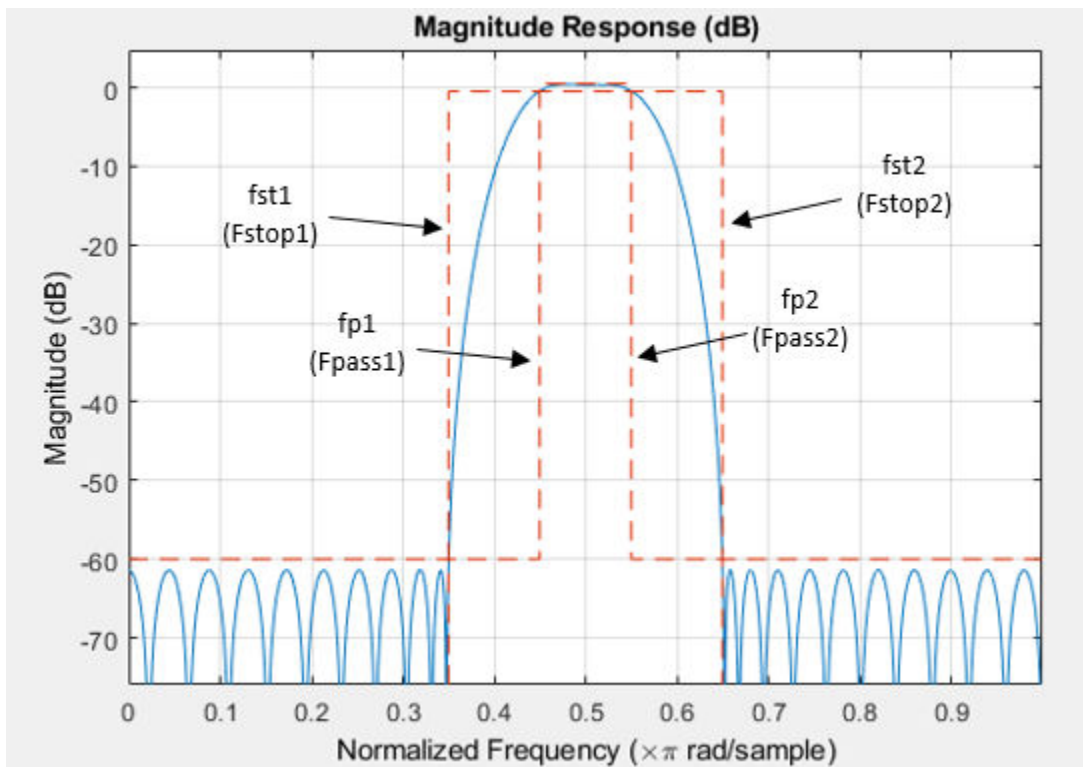
Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### **fp1 – First passband frequency**

positive scalar

First passband frequency, specified as positive scalar in normalized frequency units.

This is the frequency at the edge of the start of the first passband.



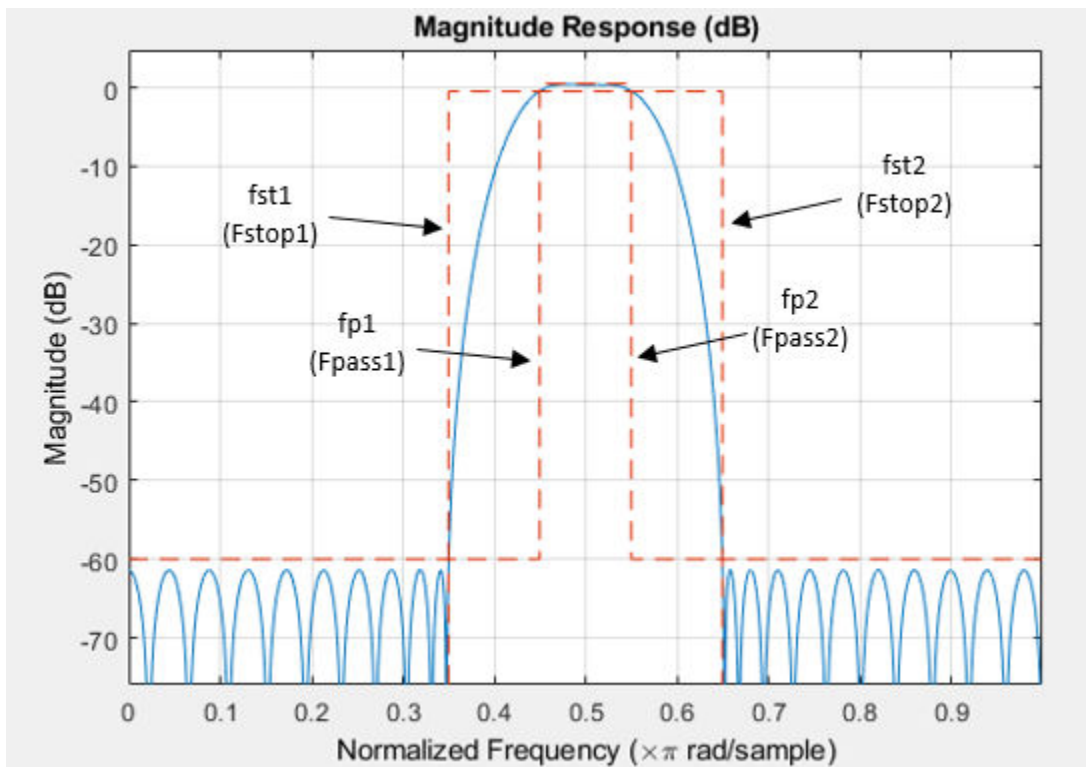
Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### **fp2 — Second passband frequency**

positive scalar

Second passband frequency, specified as positive scalar in normalized frequency units.

This is the frequency at the edge of the end of the passband.



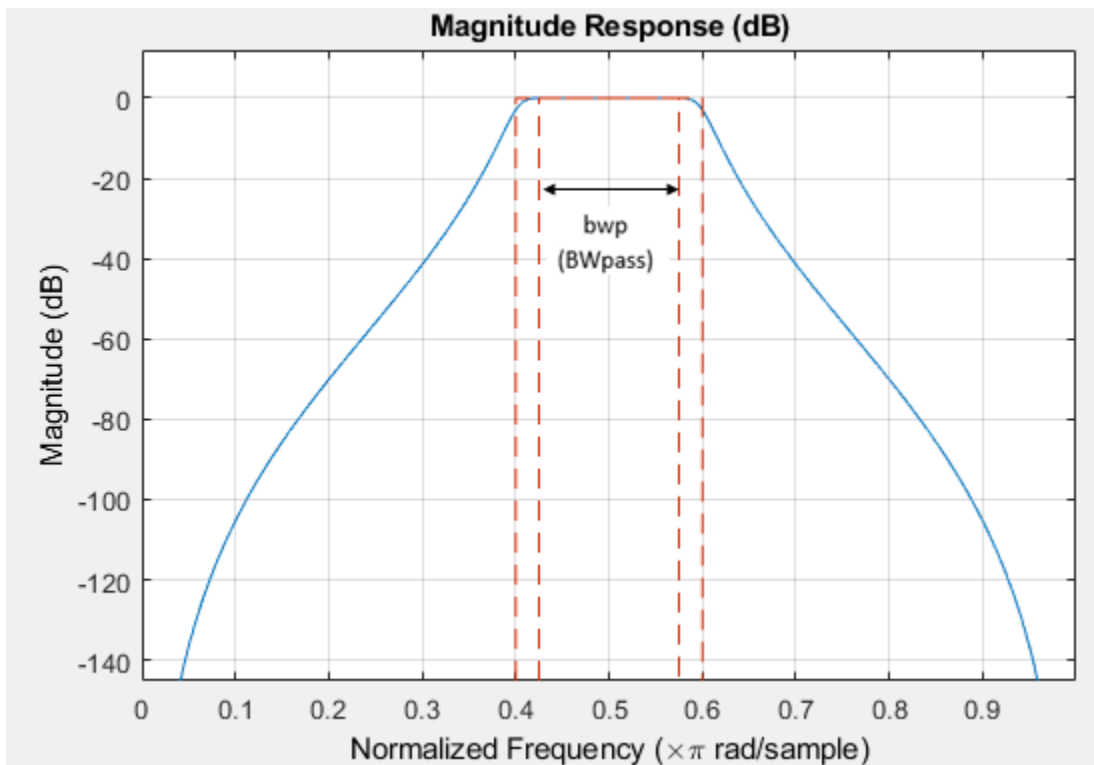
Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### **bwp** — Passband frequency width

positive scalar

Bandwidth of the filter passband in normalized frequency units, specified as a positive scalar less than  $F_{3dB2} - F_{3dB1}$ .



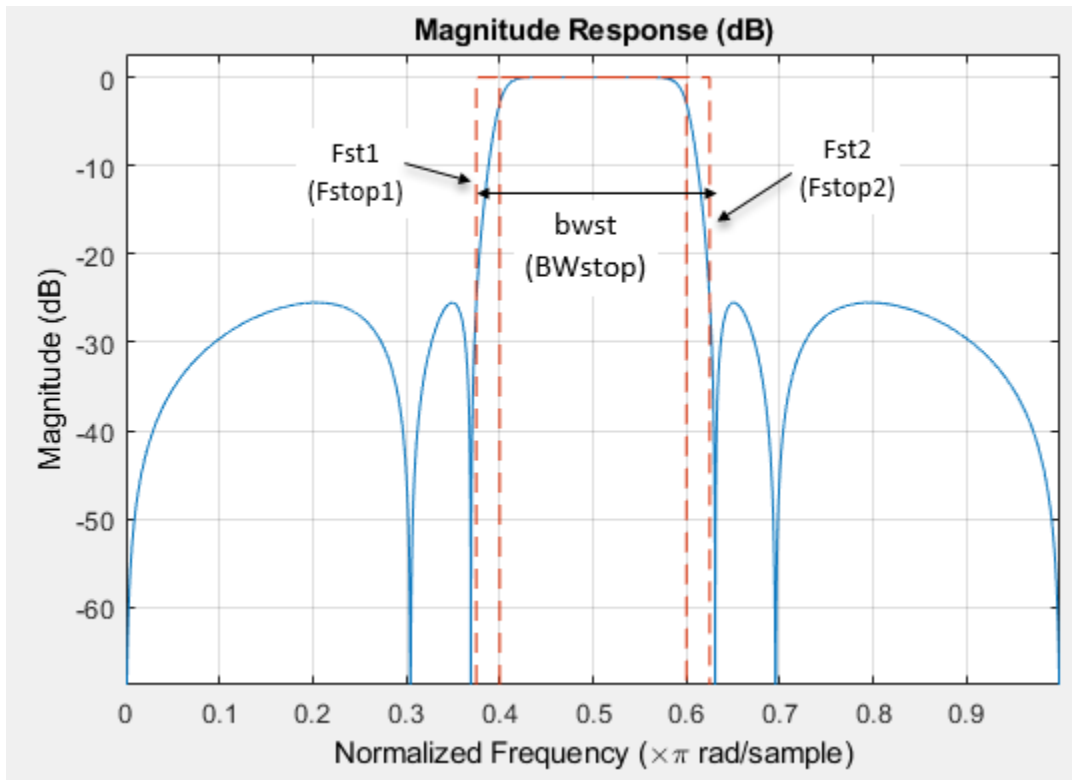


Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**bwst — Frequency width between stopband frequencies**

positive scalar

Frequency width between the two stopband frequencies, specified as a positive scalar in normalized frequency units.



Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### **Fs — Sample rate**

scalar

Sample rate of the signal to be filtered, specified as a scalar in Hz. Specify the sample rate as a scalar trailing the other numerical values provided. When  $F_s$  is provided,  $F_s$  is assumed to be in Hz, as are all other frequency values. Note that you do not have to change the specification string.

The following design has the specification string set to ' $F_{st1}, F_{p1}, F_{p2}, F_{st2}, A_{st1}, A_p, A_{st2}$ ', and sample rate set to 8000 Hz.

```
bandpassSpecs =
fdesign.bandpass('Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2',1600,2000,2400,2800,60,1,80
,8000); filt = design(bandpassSpecs,'Systemobject',true);
```

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### **magunits — Magnitude units**

'dB' (default) | 'linear' | 'squared'

Magnitude specification units, specified as 'dB', 'linear', or 'squared'. If this argument is omitted, the object assumes the units of magnitude to be 'dB'. Note that the magnitude specifications are always converted and stored in dB regardless of how they were specified. If  $F_s$  is one of the input arguments,  $magunits$  must be specified after  $F_s$  in the input argument list.

## Output Arguments

### **bandpassSpecs** — Bandpass filter design specification object

bandpass object

Bandpass filter design specification object, returned as a bandpass object. The fields of the object depend on the spec input character vector.

Consider an example where the spec argument is set to 'N, Fc1, Fc2', and the corresponding values are set to 10, 0.6, and 0.8, respectively. The bandpass filter design specification object is populated with the following fields:

```
>> bandpassSpecs = fdesign.bandpass('N, Fc1, Fc2', 10, 0.6, 0.8)
```

```
bandpassSpecs =
```

[bandpass](#) with properties:

```
        Response: 'Bandpass'  
    Specification: 'N, Fc1, Fc2'  
      Description: {3×1 cell}  
NormalizedFrequency: 1  
      FilterOrder: 10  
      Fcutoff1: 0.6000  
      Fcutoff2: 0.8000
```

## Version History

Introduced in R2009a

### See Also

fdesign | fdesign.bandstop | fdesign.highpass | fdesign.lowpass

### Topics

“Design a Filter in Fdesign — Process Overview”

## fdesign.bandstop

Bandstop filter design specification object

### Syntax

```
bandstopSpecs = fdesign.bandstop
bandstopSpecs = fdesign.bandstop(spec,value1,...,valueN)
bandstopSpecs = fdesign.bandstop( ____,Fs)
bandstopSpecs = fdesign.bandstop( ____,magunits)
```

### Description

The `fdesign.bandstop` function returns a `bandstop` filter design specification object that contains the specifications for a filter, such as passband frequency, stopband frequency, passband ripple, and filter order. Then, use the `design` function to design the filter from the filter design specifications object.

For more control options, see “Filter Design Procedure” on page 5-399. For a complete workflow, see “Design a Filter in Fdesign — Process Overview”.

`bandstopSpecs = fdesign.bandstop` constructs a bandstop filter design specifications object with the following default values:

- First passband frequency set to 0.35.
- First stopband frequency set to 0.45.
- Second stopband frequency set to 0.55.
- Second passband frequency set to 0.65.
- First passband ripple 1 dB.
- Stopband attenuation set to 60 dB.
- Second passband ripple set to 1 dB.

`bandstopSpecs = fdesign.bandstop(spec,value1,...,valueN)` constructs a bandstop filter design specifications object with a particular filter order, passband frequencies, stopband frequencies, and other specification options. Indicate the options you want to specify in the expression `spec`. After the expression, specify a value for each option. If you do not specify values after the `spec` argument, the function assumes the default values.

`bandstopSpecs = fdesign.bandstop( ____,Fs)` provides the sample rate of the signal to be filtered, in Hz. `Fs` must be specified as a scalar trailing the other numerical values provided. In this case, all frequencies in the specifications are in Hz as well.

The design specification

```
fdesign.bandstop('Fp1,Fst1,Fst2,Fp2,Ap1,Ast,Ap2',.4,.5,.6,.7,1,80,.5)
```

designs the same filter as

```
fdesign.bandstop('Fp1,Fst1,Fst2,Fp2,Ap1,Ast,Ap2',1600,2000,2400,2800,1,80,0.5,8000)
```

`bandstopSpecs = fdesign.bandstop( ____,magunits)` provides the units for any magnitude specification given. `magunits` can be one of the following: 'linear', 'dB', or 'squared'. If this

argument is omitted, 'dB' is assumed. The magnitude specifications are always converted and stored in dB regardless of how they were specified. If Fs is provided, magunits must follow Fs in the input argument list.

## Examples

### Design Equiripple FIR Bandstop Filter

Design a constrained-band FIR equiripple filter of order 60 with a stopband of [12.8 22.4] kHz. Both passband ripple values are constrained to 1 dB. The sample rate is 64 kHz.

Create a `bandstop` filter design specification object using the `fdesign.bandstop` function and specify these design parameters.

```
bandstopSpecs = fdesign.bandstop('N,Fp1,Fst1,Fst2,Fp2,C',60,9.6e3,12.8e3,22.4e3,25.6e3,64000);
```

Constrain the two passbands with a passband ripple of 1 dB.

```
bandstopSpecs.Passband1Constrained = true;
bandstopSpecs.Apass1 = 1;
bandstopSpecs.Passband2Constrained = true;
bandstopSpecs.Apass2 = 1;
```

Design the bandstop filter using the `design` function. The resulting filter is a `dsp.Filter System` object™. For details on how to apply this filter on streaming data, refer to `dsp.FIRFilter`.

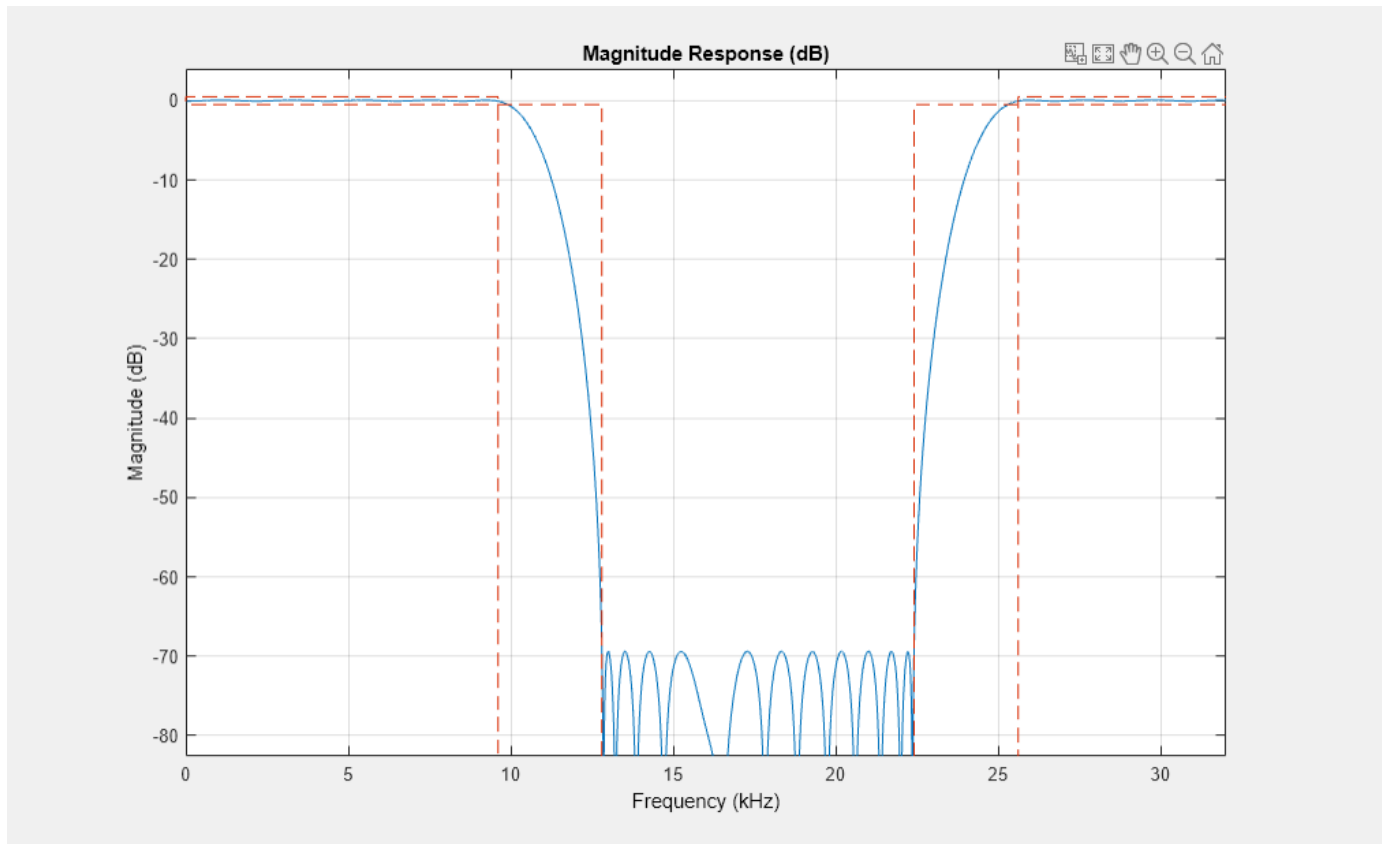
```
bandstopFilt = design(bandstopSpecs,'Systemobject',true)
```

```
bandstopFilt =
  dsp.FIRFilter with properties:
    Structure: 'Direct form'
    NumeratorSource: 'Property'
    Numerator: [-3.6116e-04 -0.0027 -3.1395e-04 -0.0033 0.0030 0.0030 -8.4856e-04 0.0017
    InitialConditions: 0
```

Show all properties

Visualize the frequency response of the designed filter using `fvtool`.

```
fvtool(bandstopFilt)
```



Measure the frequency response characteristics of the filter using `measure`.

```
measure(bandstopFilt)
```

```
ans =
Sample Rate           : 64 kHz
First Passband Edge   : 9.6 kHz
First 3-dB Point      : 10.5255 kHz
First 6-dB Point      : 10.9058 kHz
First Stopband Edge   : 12.8 kHz
Second Stopband Edge  : 22.4 kHz
Second 6-dB Point     : 24.2866 kHz
Second 3-dB Point     : 24.6685 kHz
Second Passband Edge  : 25.6 kHz
First Passband Ripple : 0.11754 dB
Stopband Atten.       : 69.3934 dB
Second Passband Ripple : 0.11761 dB
First Transition Width : 3.2 kHz
Second Transition Width : 3.2 kHz
```

### Design Minimum Order Elliptic Bandstop Filter

Design a minimum order elliptic bandstop filter. The filter design procedure is:

- 1 Specify the filter design specifications using a `fdesign` function.
- 2 Pick a design method provided by the `designmethods` function.
- 3 To determine the available design options to choose from, use the `designoptions` function.
- 4 Design the filter using the `design` function.

Construct `fdesign.bandstop` in the default state and input the design specifications to the function.

```
bandstopSpecs = fdesign.bandstop(.3,.4,.6,.7,.5,60,1)
```

```
bandstopSpecs =
    bandstop with properties:
        Response: 'Bandstop'
        Specification: 'Fp1,Fst1,Fst2,Fp2,Ap1,Ast,Ap2'
        Description: {7x1 cell}
        NormalizedFrequency: 1
            Fpass1: 0.3000
            Fstop1: 0.4000
            Fstop2: 0.6000
            Fpass2: 0.7000
            Apass1: 0.5000
            Astop: 60
            Apass2: 1
```

Determine the available `designmethods` using the `designmethods` function. To design an elliptic filter, pick `ellip`.

```
designmethods(bandstopSpecs,'Systemobject',true)
```

```
Design Methods that support System objects for class fdesign.bandstop (Fp1,Fst1,Fst2,Fp2,Ap1,Ast
```

```
butter
cheby1
cheby2
ellip
equiripple
kaiserwin
```

When designing the filter, you can specify additional design options. View a list of options using the `designoptions` function. The function also shows the default design options the filter uses.

```
designoptions(bandstopSpecs,'ellip')
```

```
ans = struct with fields:
    FilterStructure: {'df1sos' 'df2sos' 'df1tsos' 'df2tsos' 'cascadeallpass' 'cascad
    SOSScaleNorm: 'ustring'
    SOSScaleOpts: 'fdopts.sosscaling'
    MatchExactly: {'passband' 'stopband' 'both'}
    SystemObject: 'bool'
    DefaultFilterStructure: 'df2sos'
    DefaultMatchExactly: 'both'
    DefaultSOSScaleNorm: ''
    DefaultSOSScaleOpts: [1x1 fdopts.sosscaling]
    DefaultSystemObject: 0
```

Use the `design` function to design the filter. Pass `'ellip'` and the specifications given by the variable `'bandstopSpecs'`, as input arguments.

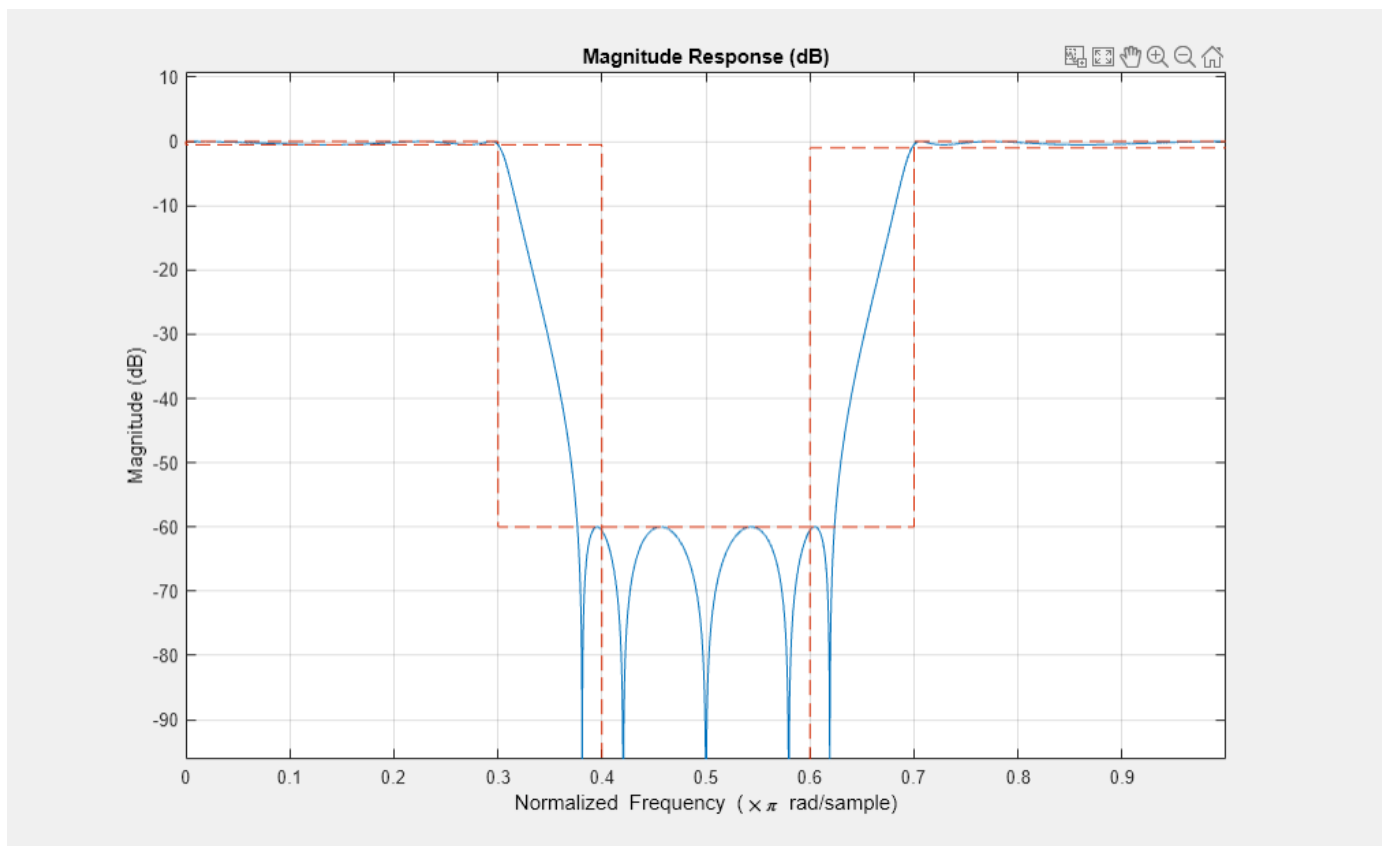
```
bsFilter = design(bandstopSpecs, 'ellip', 'Systemobject', true)
```

```
bsFilter =  
  dsp.SOSFilter with properties:  
    Structure: 'Direct form II'  
  CoefficientSource: 'Property'  
    Numerator: [5x3 double]  
  Denominator: [5x3 double]  
  HasScaleValues: true  
  ScaleValues: [0.5324 0.5324 0.6221 0.6221 0.8855 1]
```

Show all properties

Visualize the frequency response of the designed filter.

```
fvtool(bsFilter)
```





## Bandstop Filtering of Sinusoids

Construct a bandstop filter to reject the discrete frequency band between  $3\pi/8$  and  $5\pi/8$  rad/sample. With a sampling frequency of 48 kHz, these values translate to a frequency range of [9 15] kHz. Apply the filter to a discrete-time signal consisting of the superposition of three discrete-time sinusoids.

The filter is designed by first creating a bandstop filter design specifications object, and then passing the object as an input to the design function.

### Design Bandstop Filter

Create a bandstop filter design specifications object using `fdesign.bandstop`.

```
bandstopSpecs = fdesign.bandstop(1/4,3/8,5/8,6/8,1,60,1)
```

```
bandstopSpecs =
  bandstop with properties:

      Response: 'Bandstop'
  Specification: 'Fp1,Fst1,Fst2,Fp2,Ap1,Ast,Ap2'
  Description: {7x1 cell}
  NormalizedFrequency: 1
      Fpass1: 0.2500
      Fstop1: 0.3750
      Fstop2: 0.6250
      Fpass2: 0.7500
      Apass1: 1
      Astop: 60
      Apass2: 1
```

List the available design methods for this object.

```
designmethods(bandstopSpecs)
```

```
Design Methods for class fdesign.bandstop (Fp1,Fst1,Fst2,Fp2,Ap1,Ast,Ap2):
```

```
butter
cheby1
cheby2
ellip
equiripple
kaiserwin
```

To design an equiripple filter, pick 'equiripple'.

```
bsFilter = design(bandstopSpecs,'equiripple','Systemobject',true)
```

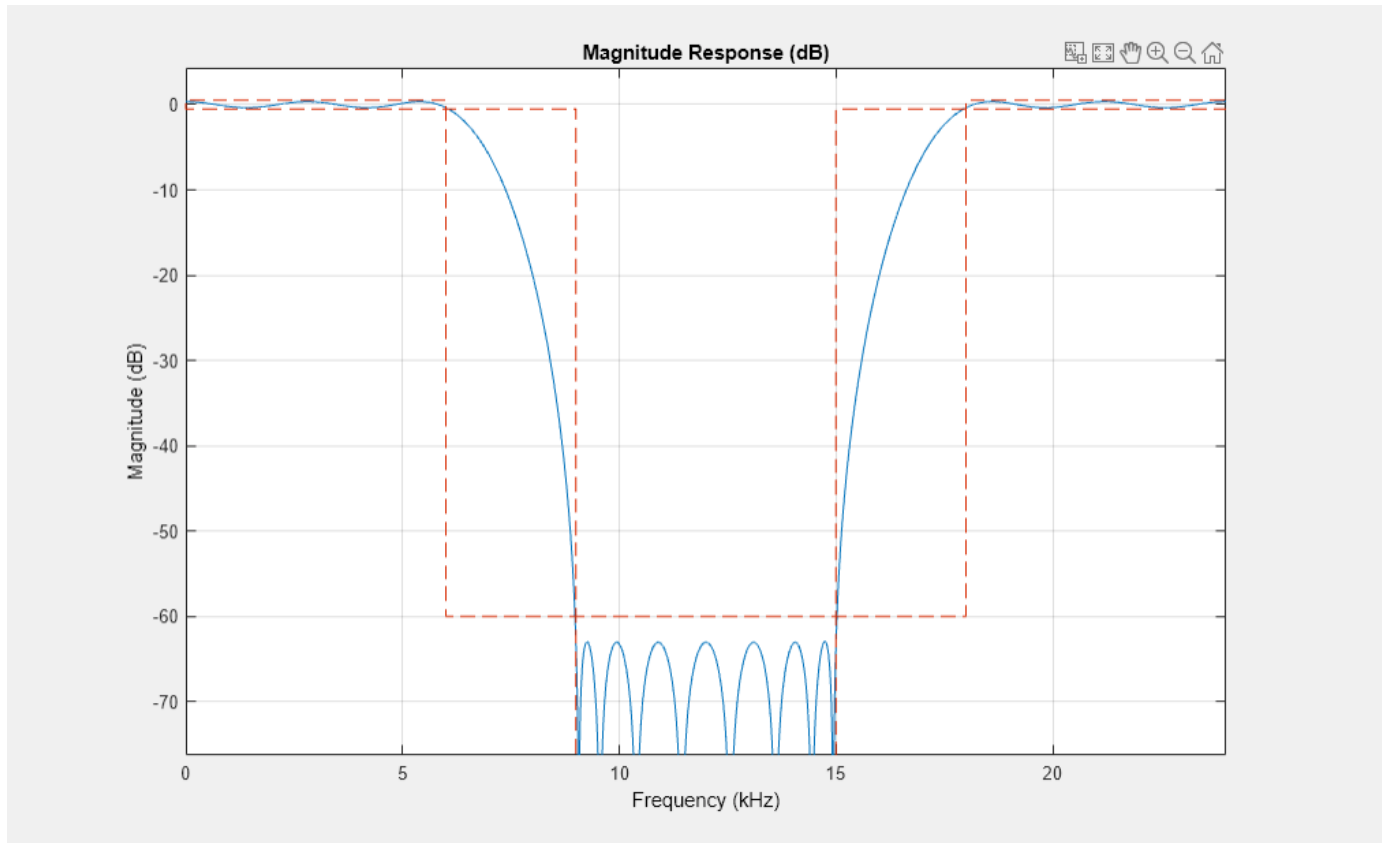
```
bsFilter =
  dsp.FIRFilter with properties:

      Structure: 'Direct form'
  NumeratorSource: 'Property'
      Numerator: [0.0054 -1.9744e-15 0.0202 -3.1206e-15 0.0064 -4.0688e-15 -0.0306 -3.6307e-15]
  InitialConditions: 0

  Show all properties
```

Visualize the frequency response of the designed filter.

```
fvtool(bsFilter, 'Fs', 48000)
```



### Create Sinusoidal Signal

Create a signal that is a sum of three sinusoids with frequencies at 1 kHz, 12 kHz, and 16 kHz. Initialize Spectrum Analyzer to view the original signal and the filtered signal.

```
Sine1 = dsp.SineWave('Frequency', 1e3, 'SampleRate', 44.1e3, 'SamplesPerFrame', 4000);
Sine2 = dsp.SineWave('Frequency', 12e3, 'SampleRate', 44.1e3, 'SamplesPerFrame', 4000);
Sine3 = dsp.SineWave('Frequency', 16e3, 'SampleRate', 44.1e3, 'SamplesPerFrame', 4000);
```

```
SpecAna = spectrumAnalyzer('PlotAsTwoSidedSpectrum', false, ...
    'SampleRate', Sine1.SampleRate, ...
    'ShowLegend', true, ...
    'YLimits', [-240, 45]);
```

```
SpecAna.ChannelNames = {'Original noisy signal', 'Filtered signal'};
```

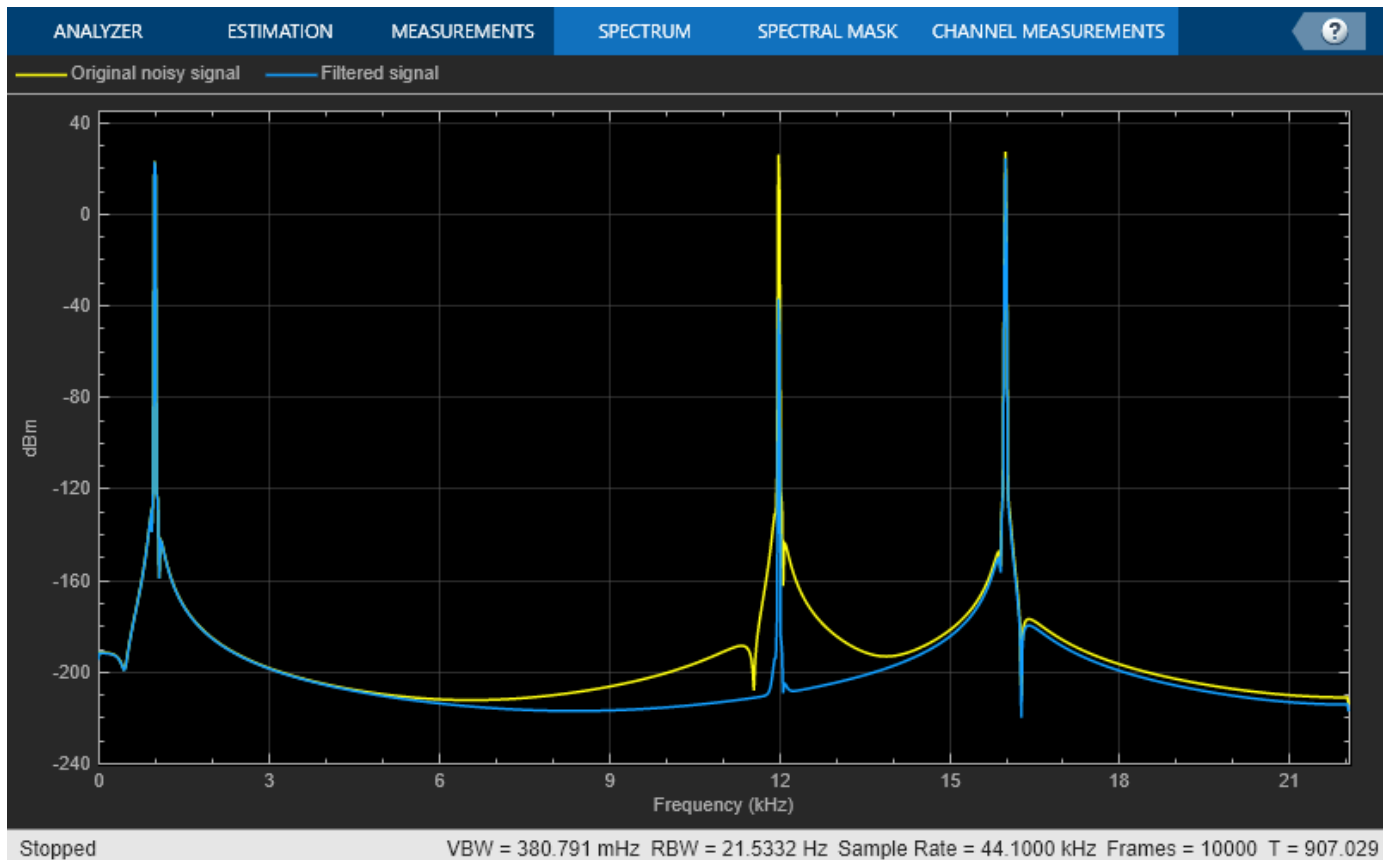
### Filter Sinusoidal Signal

Filter the sinusoidal signal using the bandstop filter that has been designed. View the original signal and the filtered signal in the Spectrum Analyzer. The tone at 1 kHz is unaffected. The tone at 12 kHz is filtered out and attenuated, and the tone at 16 kHz is mildly attenuated because it appears in the transition band of the filter.

```

for i = 1 : 10000
    x = Sine1()+Sine2()+Sine3();
    y = bsFilter(x);
    SpecAna(x,y);
end
release(SpecAna)

```



## Input Arguments

### spec — Specification

'Fp1,Fst1,Fst2,Fp2,Ap1,Ast,Ap2' (default) | 'N,F3dB1,F3dB2' | 'N,F3dB1,F3dB2,Ap' | ...

Specification expression, specified as one of these character vectors:

- 'Fp1,Fst1,Fst2,Fp2,Ap1,Ast,Ap2' (default)
- 'N,F3dB1,F3dB2'
- 'N,F3dB1,F3dB2,Ap' \*
- 'N,F3dB1,F3dB2,Ap,Ast' \*
- 'N,F3dB1,F3dB2,Ast' \*
- 'N,F3dB1,F3dB2,BWp' \*
- 'N,F3dB1,F3dB2,BWst' \*
- 'N,Fc1,Fc2'

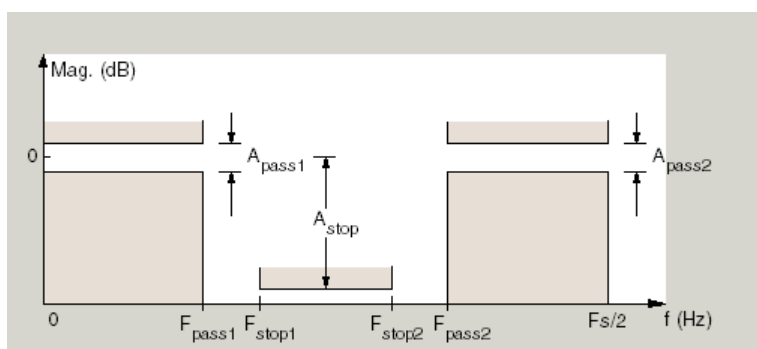
- 'N, Fc1, Fc2, Ap1, Ast, Ap2'
- 'N, Fp1, Fp2, Ap'
- 'N, Fp1, Fp2, Ap, Ast'
- 'N, Fp1, Fst1, Fst2, Fp2'
- 'N, Fp1, Fst1, Fst2, Fp2, C' \*
- 'N, Fp1, Fst1, Fst2, Fp2, Ap' \*
- 'N, Fst1, Fst2, Ast'
- 'Nb, Na, Fp1, Fst1, Fst2, Fp2' \*

This table describes each option that can appear in the expression.

| Specification option | Description  |
|----------------------|--|
| Ap                   | Amount of ripple allowed in passband, specified as <i>Apass</i> in dB.   |
| Ap1                  | Amount of ripple allowed in the first passband, specified as <i>Apass1</i> in dB.  |
| Ap2                  | Amount of ripple allowed in the second passband, specified as <i>Apass2</i> in dB.   |
| Ast                  | Stopband attenuation (dB), specified using <i>Astop</i> .  |
| BWp                  | Bandwidth of the filter passband, specified as <i>BWpass</i> in normalized frequency units.  |
| BWst                 | Bandwidth of the filter stopband, specified as <i>BWstop</i> in normalized frequency units.  |
| F3dB1                | Frequency of the 3 dB point below the passband value for the first cutoff, specified in normalized frequency units. Applies to IIR filters.  |
| F3dB2                | Frequency of the 3 dB point below the passband value for the second cutoff, specified in normalized frequency units. Applies to IIR filters. |
| Fc1                  | First cutoff frequency (normalized frequency units), specified using <i>Fcutoff1</i> . Applies to FIR filters.                               |
| Fc2                  | Second cutoff frequency (normalized frequency units), specified using <i>Fcutoff1</i> . Applies to FIR filters.                              |
| Fp1                  | Frequency at the start of the pass band, specified as <i>Fpass1</i> in normalized frequency units.   |
| Fp2                  | Frequency at the end of the pass band, specified as <i>Fpass2</i> in normalized frequency units.   |
| Fst1                 | Frequency at the end of the first stop band, specified as <i>Fstop1</i> in normalized frequency units.                                       |

| Specification option | Description   |
|----------------------|---|
| Fst2                 | Frequency at the start of the second stop band, specified as Fstop2 in normalized frequency units.  |
| N                    | Filter order for FIR filters. Or both the numerator and denominator orders for IIR filters when Na and Nb are not provided. Specified using FilterOrder.  |
| Nb                   | Numerator order for IIR filters, specified using the DenOrder property.   |
| Na                   | Denominator order for IIR filters, specified using the NumOrder property.   |
| C                    | <p>Constrained band flag. This enables you to specify passband ripple or stopband attenuation for fixed-order designs in one or two of the three bands.</p> <p>In the specification 'N,Fp1,Fst1,Fst2,Fp2,C', you cannot specify constraints in both passbands and the stopband simultaneously. You can specify constraints in any one or two bands.</p> |

Graphically, the filter specifications look similar to those shown in the following figure.



Regions between specification values like Fp1 and Fst1 are transition regions where the filter response is not explicitly defined.

The design methods available for designing the filter depend on the specification expression. You can obtain these methods using the `designmethods` function. The table lists each specification expression supported by `fdesign.bandstop` and the corresponding design methods available.

| Specification expression        | Supported design methods                             |
|---------------------------------|--|
| 'Fp1,Fst1,Fst2,Fp2,Ap1,Ast,Ap2' | butter, cheby1, cheby2, ellip, equiripple, kaiserwin |
| 'N,F3dB1,F3dB2'                 | butter   |
| 'N,F3dB1,F3dB2,Ap'              | cheby1   |

| Specification expression       | Supported design methods      |
|--------------------------------|-------------------------------|
| 'N, F3dB1, F3dB2, Ap, Ast'     | ellip                         |
| 'N, F3dB1, F3dB2, Ast'         | cheby2, ellip                 |
| 'N, F3dB1, F3dB2, BWp'         | cheby1                        |
| 'N, F3dB1, F3dB2, BWst'        | cheby2                        |
| 'N, Fc1, Fc2'                  | window                        |
| 'N, Fc1, Fc2, Ap1, Ast, Ap2'   | fircls                        |
| 'N, Fp1, Fp2, Ap'              | cheby1                        |
| 'N, Fp1, Fp2, Ap, Ast'         | ellip                         |
| 'N, Fp1, Fst1, Fst2, Fp2'      | iirlpnorm, equiripple, fircls |
| 'N, Fp1, Fst1, Fst2, Fp2, C'   | equiripple                    |
| 'N, Fp1, Fst1, Fst2, Fp2, Ap'  | ellip                         |
| 'N, Fst1, Fst2, Ast'           | cheby2                        |
| 'Nb, Na, Fp1, Fst1, Fst2, Fp2' | iirlpnorm                     |

To design the filter, call the `design` function with one of these design methods as an input. You can choose the type of filter response by passing 'FIR' or 'IIR' to the `design` function. For more details, see `design`. Enter `help(bandstopSpecs, 'method')` at the MATLAB command line to obtain detailed help on the design options for a given design method, 'method'.

For more details on the procedure, see “Filter Design Procedure” on page 5-399. For an example, see “Design Notch Filter” on page 5-543.

### **value1, ..., valueN — Specification values**

comma-separated list of values

Specification values, specified as a comma-separated list of values. Specify a value for each option in `spec` in the same order that the options appear in the expression.

Example: `bandstopSpecs = fdesign.bandstop('N, Fp1, Fst1, Fst2, Fp2, C', n, fp1, fst1, fst2, fp2, c)`

The arguments below describe more details for each option in the expression.

#### **n — Filter order**

positive integer

Filter order for FIR filters, specified as a positive integer. In the case of IIR filter design, if `nb` and `na` are not provided, this value is interpreted as both the numerator order and the denominator order.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

#### **nb — Numerator order for IIR filters**

nonnegative integer

Numerator order for IIR filters, specified as a nonnegative integer.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

#### **na — Denominator order for IIR filters**

positive integer

Denominator order for IIR filters, specified as a positive integer.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **c** — Constrained band flag

logical

This enables you to specify passband ripple or stopband attenuation for fixed-order designs in one or two of the three bands.

In the specification `'N,Fp1,Fst1,Fst2,Fp2,C'`, you cannot specify constraints for all three bands (two passbands and one stopband) simultaneously. You can specify constraints in any one or two bands.

Consider the following bandstop design specification where both the passbands are constrained to the default value, 1 dB.

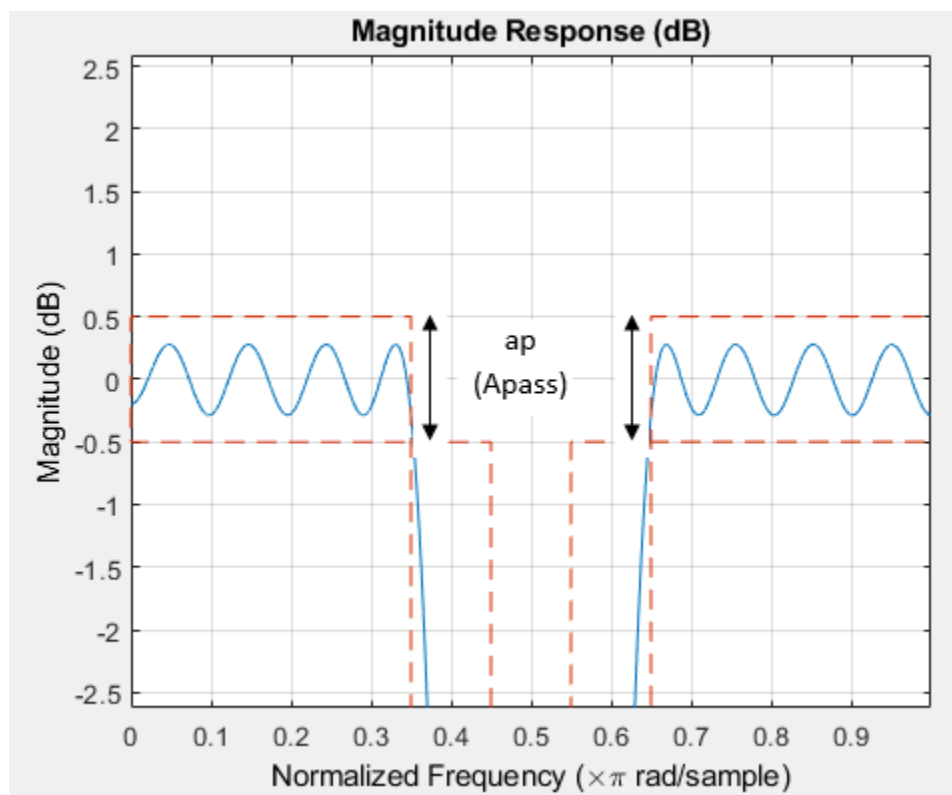
```
Example: spec =
fdesign.bandstop('N,Fp1,Fst1,Fst2,Fp2,C',10,0.35,0.45,0.55,0.65);
spec.Passband1Constrained=true; spec.Passband2Constrained=true;
```

### **ap** — Passband ripple

positive scalar

Passband ripple, specified as a positive scalar in dB. If `magunits` is `'linear'` or `'squared'`, the passband ripple is converted and stored in dB by the function regardless of how it has been specified.

The specified `ap` value applies to both the first passband and the second passband.



Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### ap1 — First passband ripple in dB

positive scalar

Amount of ripple allowed in the first passband, specified as a positive scalar in dB. If `magunits` is 'linear' or 'squared', the first passband ripple is converted and stored in dB by the function regardless of how it has been specified.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### ap2 — Second passband ripple in dB

positive scalar

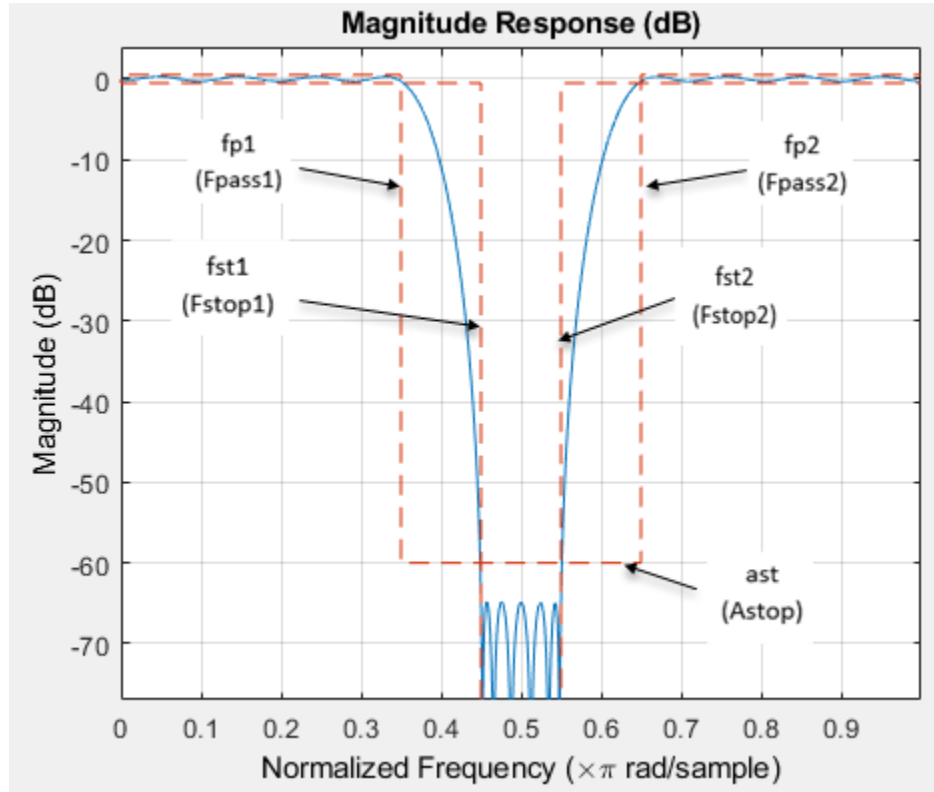
Amount of ripple allowed in the second passband, specified as a positive scalar in dB. If `magunits` is 'linear' or 'squared', the second passband ripple is converted and stored in dB by the function regardless of how it has been specified.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### ast — Stopband attenuation in dB

positive scalar

Stopband attenuation, specified as a positive scalar in dB. If `magunits` is 'linear' or 'squared', the stopband attenuation is converted and stored in dB by the function regardless of how it has been specified.



Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64



**bwp — Passband frequency width**

positive scalar

Bandwidth of the filter passband in normalized frequency units, specified as a positive scalar.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**bwst — Stopband frequency width**

positive scalar

Bandwidth of the filter stopband in normalized frequency units, specified as a positive scalar less than  $F_{3dB2} - F_{3dB1}$ .

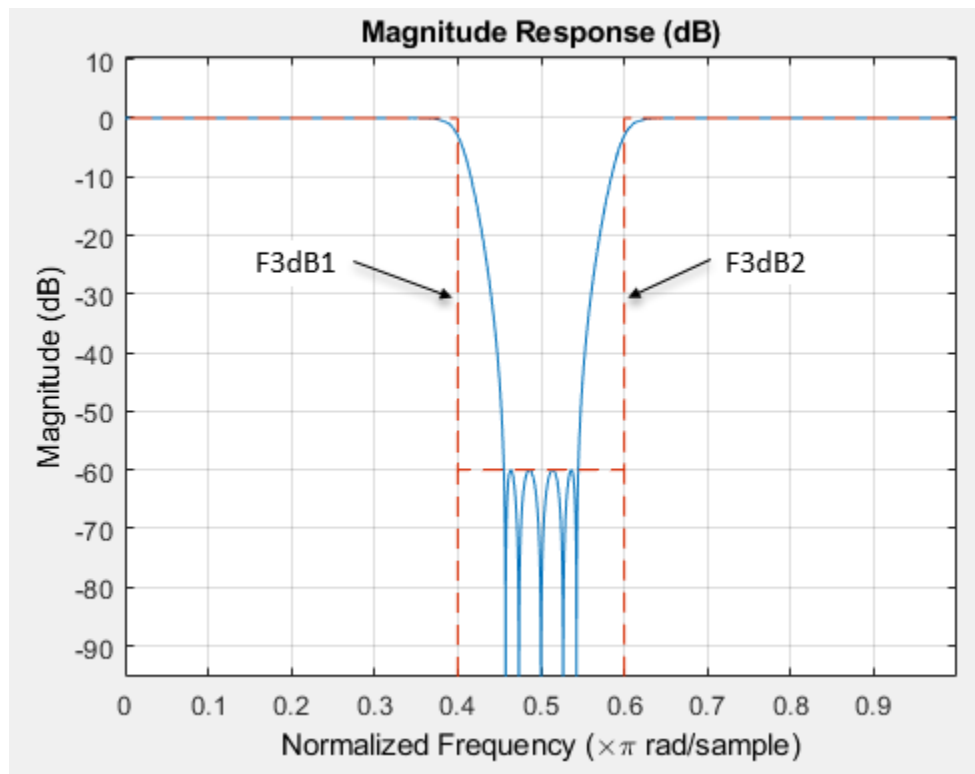
Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**F3dB1 — First 3 dB frequency**

positive scalar

First 3 dB frequency, specified as positive scalar in normalized frequency units.

This is the frequency of the 3 dB point below the passband value for the first cutoff. Applies to IIR filters only.



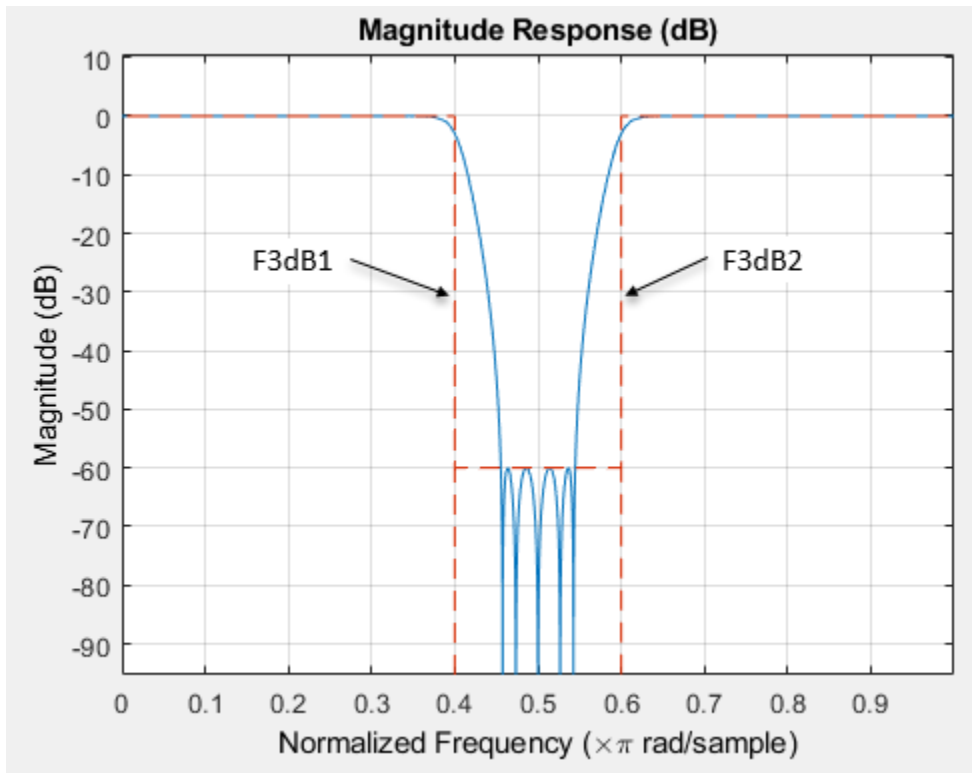
Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**F3dB2 — Second 3 dB frequency**

positive scalar

Second 3 dB frequency, specified as positive scalar in normalized frequency units.

This is the frequency of the 3 dB point below the passband value for the second cutoff. Applies to IIR filters only.



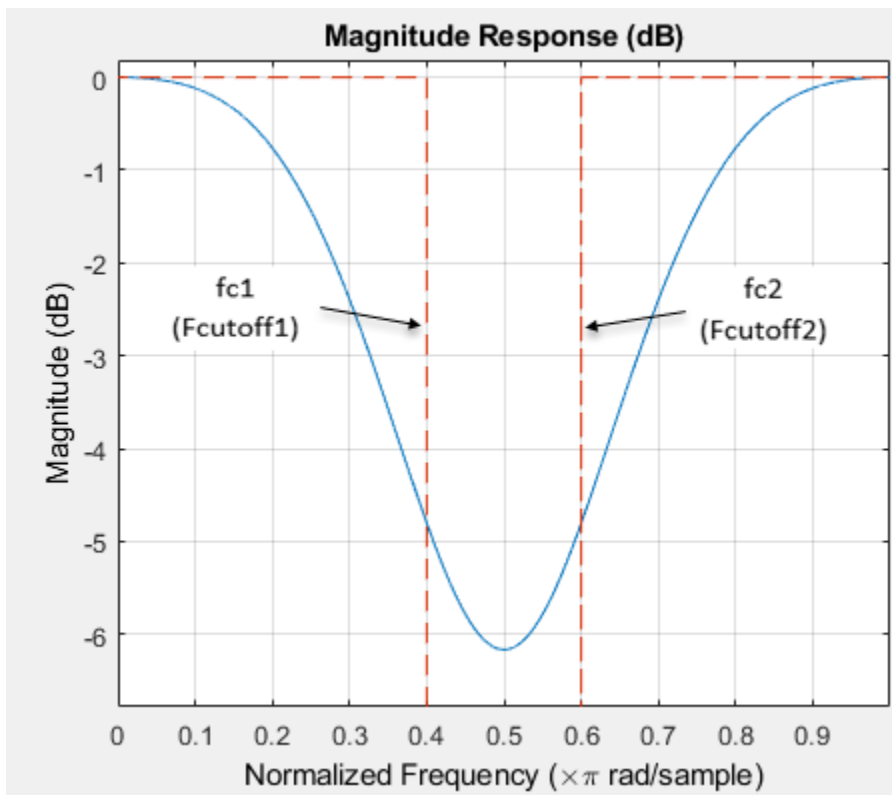
Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**fc1 – First cutoff frequency**

positive scalar

First cutoff frequency, specified as positive scalar in normalized frequency units.

Applies to FIR filters only.



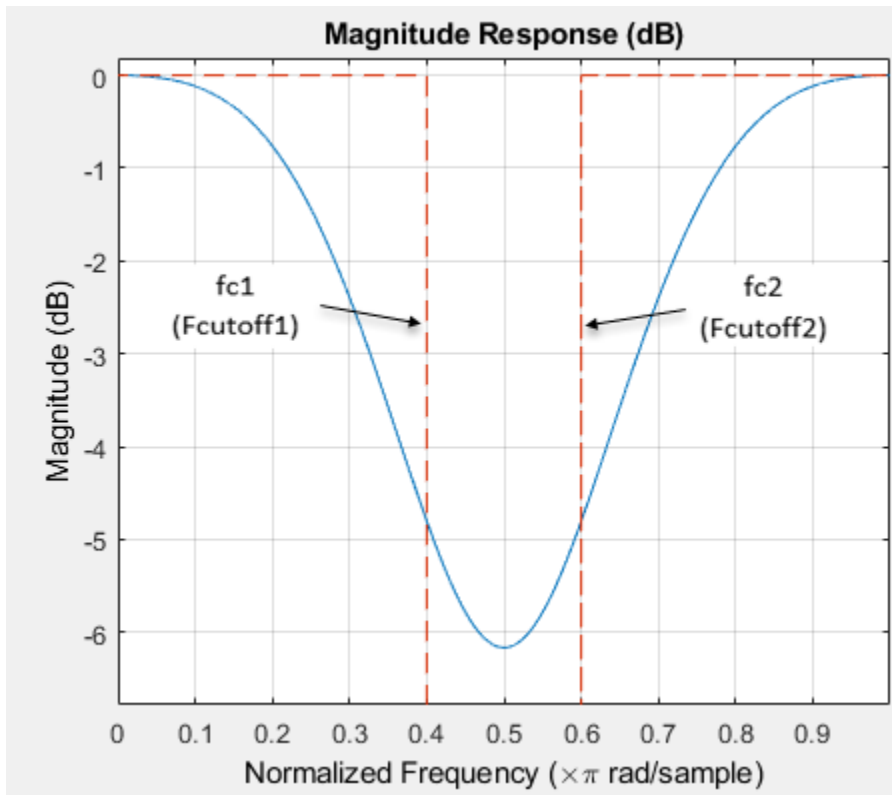
Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**fc2 — Second cutoff frequency**

positive scalar

Second cutoff frequency, specified as positive scalar in normalized frequency units.

Applies to FIR filters only.



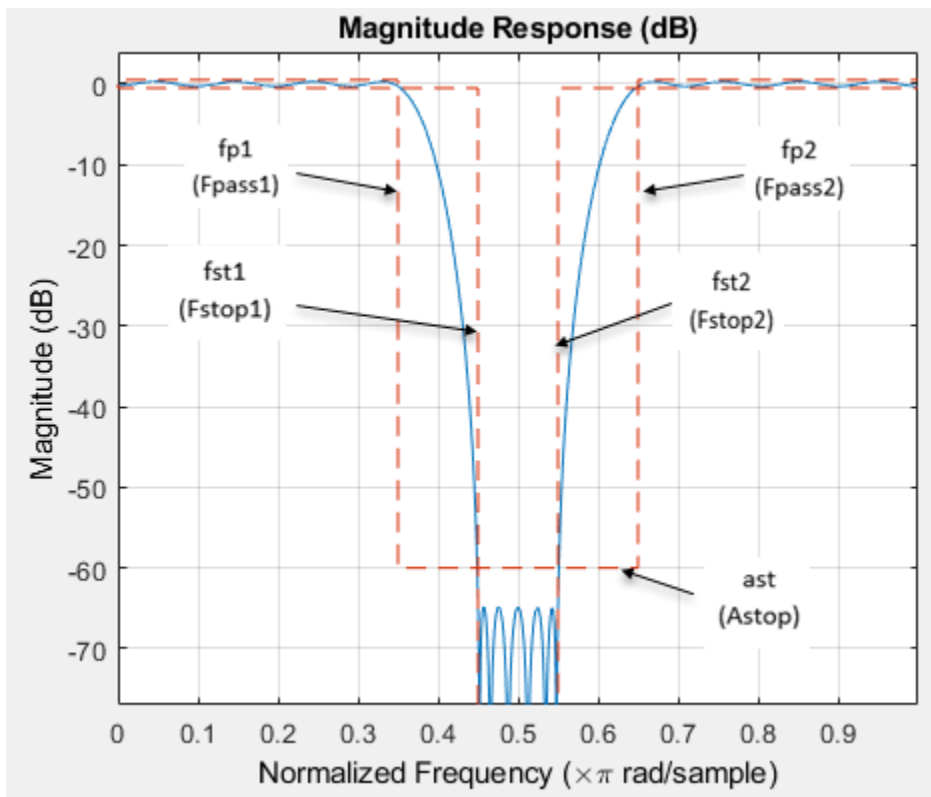
Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**fst1 — First stopband frequency**

positive scalar

First stopband frequency, specified as positive scalar in normalized frequency units.

This is the frequency at the start of the stopband.



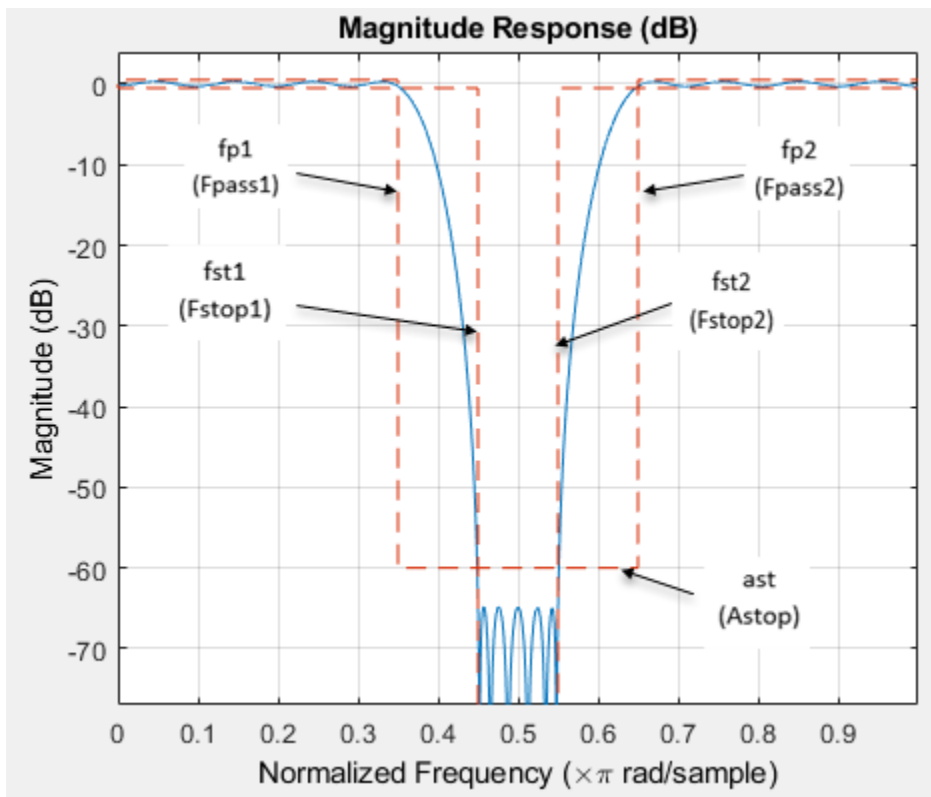
Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### **fst2 — Second stopband frequency**

positive scalar

Second stopband frequency, specified as positive scalar in normalized frequency units.

This is the frequency at the end of the stopband.



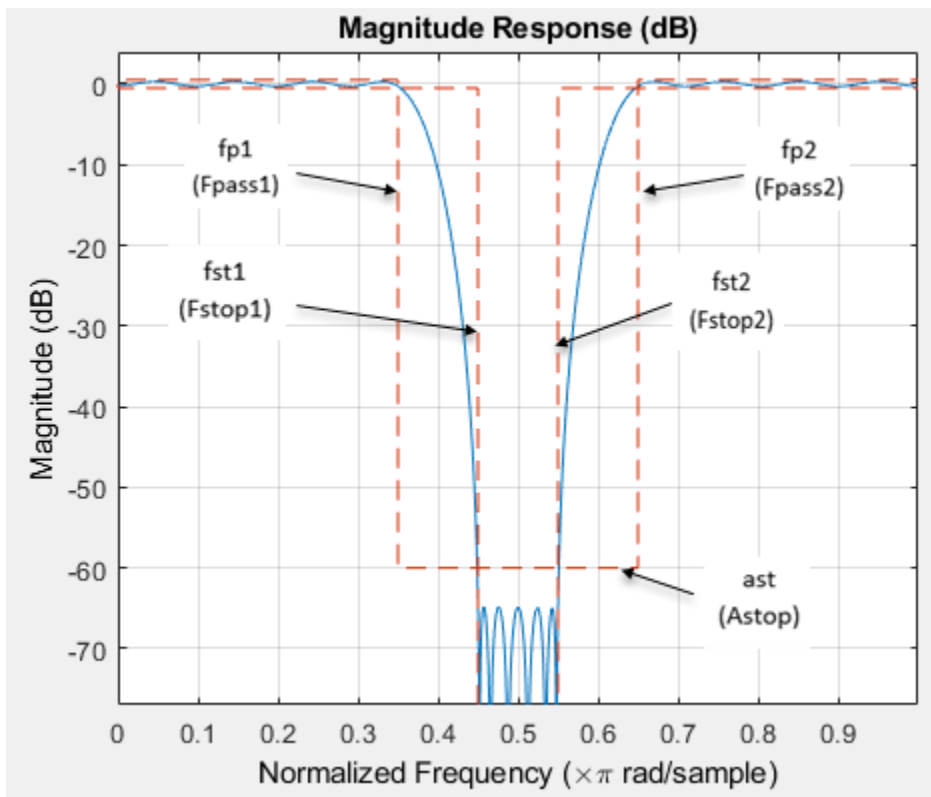
Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**fp1 — First passband frequency**

positive scalar

First passband frequency, specified as positive scalar in normalized frequency units.

This is the frequency at the end of the first passband.



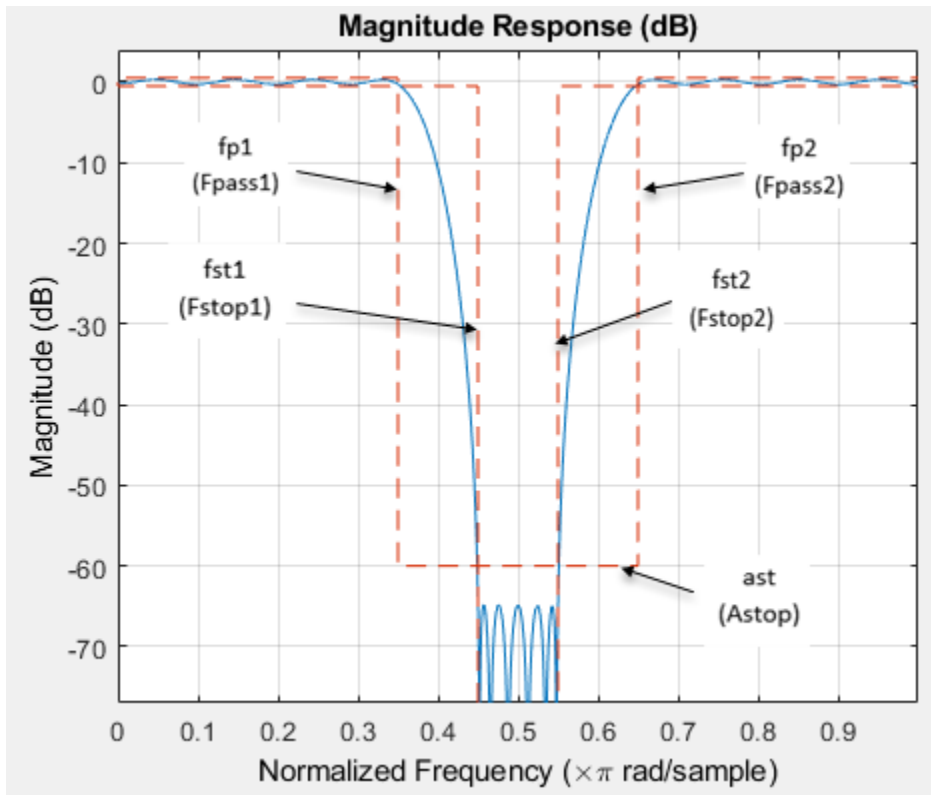
Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### **fp2 — Second passband frequency**

positive scalar

Second passband frequency, specified as positive scalar in normalized frequency units.

This is the frequency at the start of the second passband.



Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### **Fs — Sample rate**

scalar

Sample rate of the signal to be filtered, specified as a scalar in Hz. Specify the sample rate as a scalar trailing the other numerical values provided. When `Fs` is provided, `Fs` is assumed to be in Hz, as are all other frequency values provided. Note that you do not have to change the specification string.

The following design has the specification string set to `'Fp1,Fst1,Fst2,Fp2,Ap1,Ast,Ap2'`, and sample rate set to 8000 Hz.

```
bandstopSpecs =
fdesign.bandstop('Fp1,Fst1,Fst2,Fp2,Ap1,Ast,Ap2',1600,2000,2400,2800,1,80,.5,
8000); filt = design(bandstopSpecs,'Systemobject',true);
```

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### **magunits — Magnitude units**

'dB' (default) | 'linear' | 'squared'

Magnitude specification units, specified as `'dB'`, `'linear'`, or `'squared'`. If this argument is omitted, `'dB'` is assumed. Note that the magnitude specifications are always converted and stored in dB regardless of how they were specified. If `Fs` is one of the input arguments, `magunits` must be specified after `Fs` in the input argument list.



## Output Arguments

### bandstopSpecs — Bandstop filter design specification object

bandstop object

Bandstop filter design specification object, returned as a bandstop object. The fields of the object depend on the spec input character vector.

Consider an example where the spec argument is set to 'N,Fc1,Fc2', and the corresponding values are set to 10, 0.6, and 0.8, respectively. The bandstop filter design specification object is populated with the following fields:

```
>> bandstopSpecs = fdesign.bandstop('N,Fc1,Fc2',10,0.6,0.8)
```

```
bandstopSpecs =
```

```
bandstop with properties:
```

```
    Response: 'Bandstop'  
 Specification: 'N,Fc1,Fc2'  
 Description: {3×1 cell}  
 NormalizedFrequency: 1  
    FilterOrder: 10  
    Fcutoff1: 0.6000  
    Fcutoff2: 0.8000
```

## Version History

Introduced in R2009a

### See Also

fdesign | fdesign.bandpass | fdesign.highpass | fdesign.lowpass

### Topics

“Design a Filter in Fdesign — Process Overview”

## fdesign.ciccomp

CIC compensator filter specification object

### Syntax

```
d= fdesign.ciccomp
d= fdesign.ciccomp(d,nsections,rcic)
d= fdesign.ciccomp(...,spec)
h = fdesign.ciccomp(...,spec,specvalue1,specvalue2,...)
```

### Description

`d= fdesign.ciccomp` constructs a CIC compensator specifications object `d`, applying default values for the properties `Fpass`, `Fstop`, `Apass`, and `Astop`. In this syntax, the filter has two sections and the differential delay is 1.

Using `fdesign.ciccomp` with `design` method creates a System object, if the 'SystemObject' flag is set to `true`.

`d= fdesign.ciccomp(d,nsections,rcic)` constructs a CIC compensator specifications object with the filter differential delay set to `d`, the number of sections in the filter set to `nsections`, and the CIC rate change factor set to `rcic`. The default values of these parameters are: the differential delay equal to 1, the number of sections equal to 2, and the CIC rate change factor equal to 1.

If the CIC rate change factor is equal to 1, the filter passband response is an inverse sinc that is an approximation to the true inverse passband response of the CIC filter.

If you specify a CIC rate change factor not equal to 1, the filter passband response is an inverse Dirichlet sinc that matches exactly the true inverse passband response of the CIC filter.

`d= fdesign.ciccomp(...,spec)` constructs a CIC Compensator specifications object and sets its `Specification` property to `spec`. Entries in the `spec` represent various filter response features, such as the filter order, that govern the filter design. Valid entries for `spec` are shown in the list below. The entries are not case sensitive.

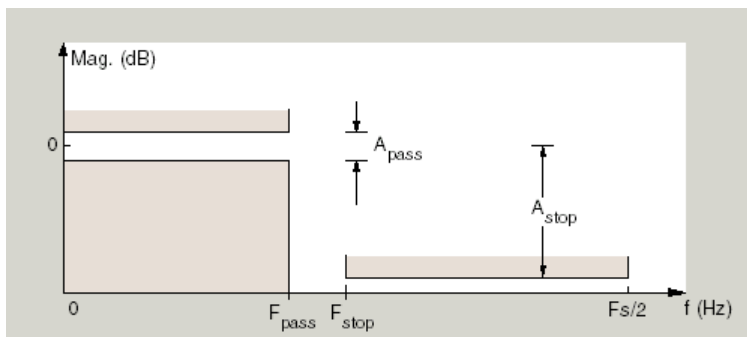
- 'fp,fst,ap,ast' (default spec)
- 'n,fc,ap,ast'
- 'n,fp,ap,ast'
- 'n,fp,fst'
- 'n,fst,ap,ast'

The filter specifications are defined as follows:

- `ap` — amount of ripple allowed in the pass band in decibels (the default units). Also called `Apass`.
- `ast` — attenuation in the stop band in decibels (the default units). Also called `Astop`.
- `fc` — cutoff frequency for the point 6 dB point below the passband value. Specified in normalized frequency units.

- $f_p$  — frequency at the end of the pass band. Specified in normalized frequency units. Also called  $F_{pass}$ .
- $f_{st}$  — frequency at the start of the stop band. Specified in normalized frequency units. Also called  $F_{stop}$ .
- $n$  — filter order.

In graphic form, the filter specifications look like this:



Regions between specification values like  $f_p$  and  $f_{st}$  are transition regions where the filter response is not explicitly defined.

The filter design methods that apply to a CIC compensator specifications object change depending on the `Specification`. Use `designmethods` to determine which design method applies to an object and its specification.

`h = fdesign.ciccomp(...,spec,specvalue1,specvalue2,...)` constructs an object and sets the specifications in the order they are specified in the `spec` input when you construct the object.

## Designing CIC Compensators

Typically, when they develop filters, designers want flat passbands and transition regions that are as narrow as possible. CIC filters present a  $(\sin x/x)$  profile in the passband and relatively wide transitions.

To compensate for this fall off in the passband, and to try to reduce the width of the transition region, you can use a CIC compensator filter that demonstrates an  $(x/\sin x)$  profile in the passband. `fdesign.ciccomp` is specifically tailored to designing CIC compensators.

You can design a compensator for CIC filter using differential delay,  $d$ , number of sections, `numberofsections`, and the usable passband frequency,  $F_{pass}$ .

By taking the number of sections, passband, and differential delay from your CIC filter and using them in the definition of the CIC compensator, the resulting compensator filter effectively corrects for the passband droop of the CIC filter, and narrows the transition region.

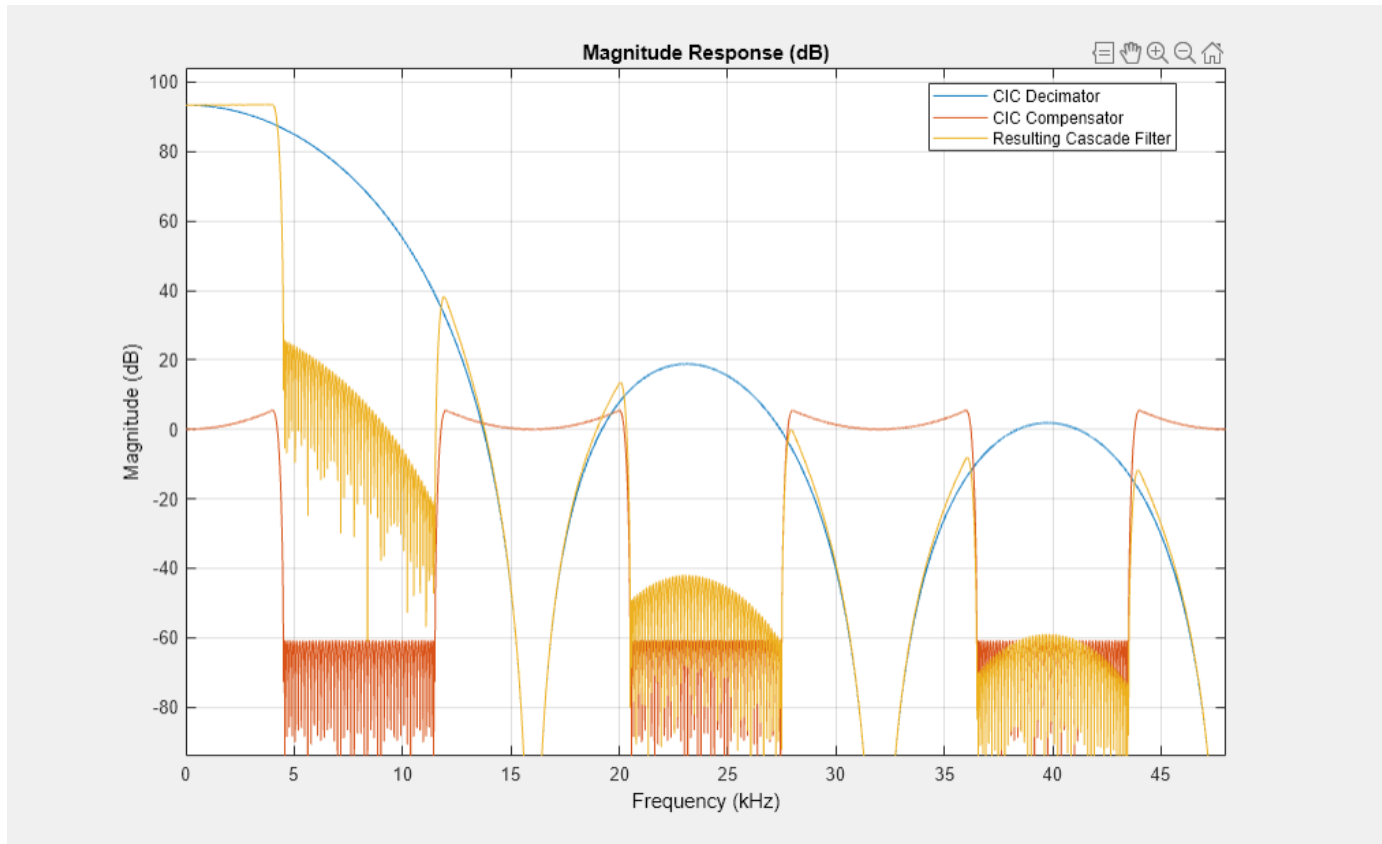
As a demonstration of this concept, this example creates a CIC decimator and its compensator.

```
fs = 96e3; % Input sampling frequency.
fpass = 4e3; % Frequency band of interest.
m = 6; % Decimation factor.
hcic = design(fdesign.decimator(m,'cic',1,fpass,60,fs),'SystemObject',true);
```

```

hd = design(fdesign.ciccomp(hcic.DifferentialDelay, ...
    hcic.NumSections, fpass, 4.5e3, .1, 60, fs/m), 'SystemObject', true);
hvft = fvtool(hcic, hd, ...
cascade(hcic, hd), 'ShowReference', 'off', 'Fs', [96e3 96e3/m 96e3]);
legend(hvft, 'CIC Decimator', 'CIC Compensator', 'Resulting Cascade Filter');

```



Here is a plot of a CIC filter and a compensator for that filter.

## Examples

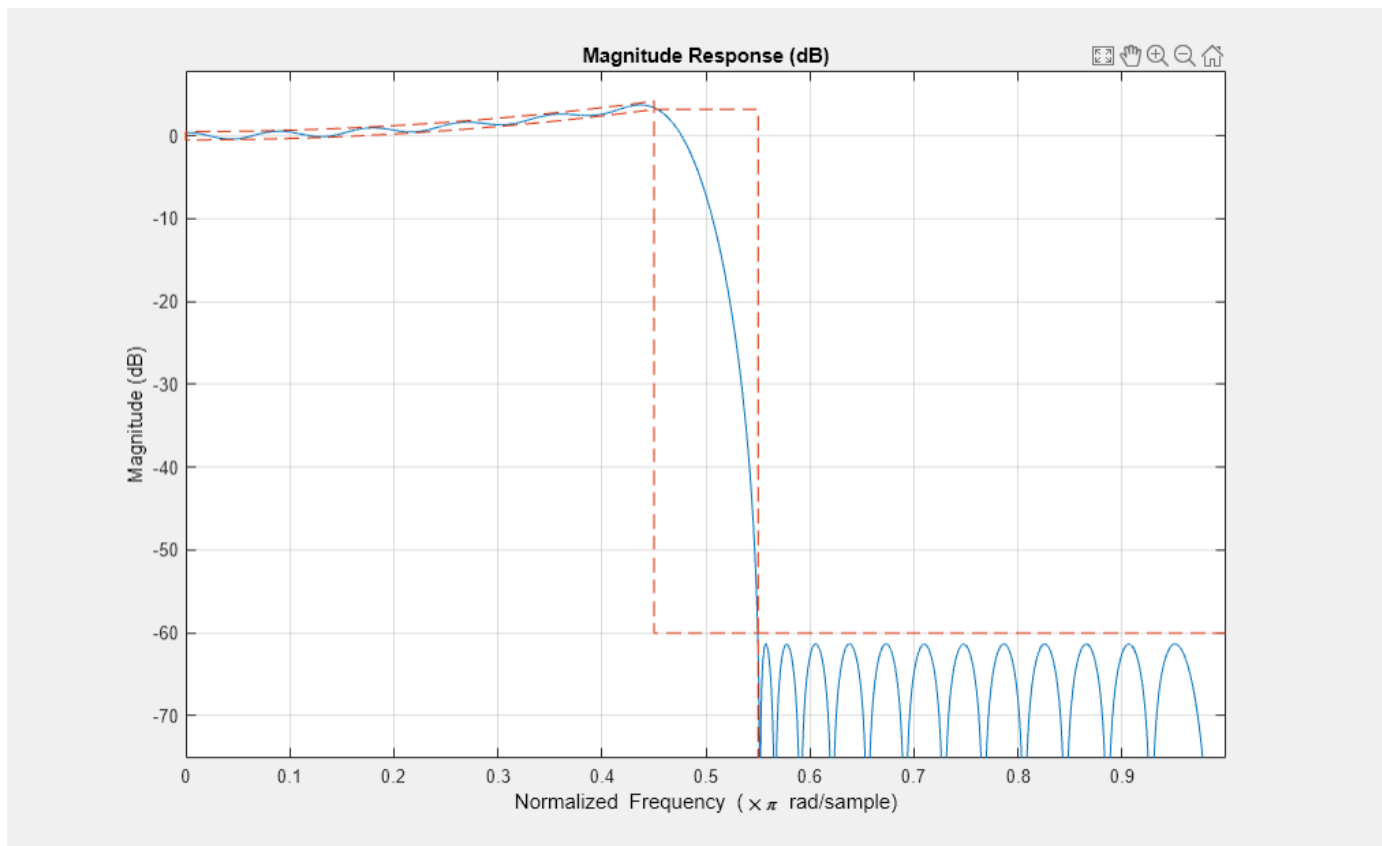
### Design a CIC Compensator Using fdesign Object

Designed to compensate for the rolloff inherent in CIC filters, CIC compensators can improve the performance of your CIC design. This example designs a compensator  $d$  with five sections and a differential delay equal to one. The plot displayed after the code shows the increasing gain in the passband that is characteristic of CIC compensators, to overcome the droop in the CIC filter passband. Ideally, cascading the CIC compensator with the CIC filter results in a lowpass filter with flat passband response and narrow transition region.

```

h = fdesign.ciccomp;
set(h, 'NumberOfSections', 5, 'DifferentialDelay', 1);
cicComp = design(h, 'equiripple', 'SystemObject', true);
fvtool(cicComp)

```



This compensator would work for a decimator or interpolator that had differential delay of 1 and 5 sections.

## Version History

Introduced in R2011a

### See Also

[fdesign.decimator](#) | [fdesign.interpolator](#)

## fdesign.comb

IIR comb filter specification object

### Syntax

```
d=fdesign.comb
d=fdesign.comb(combtype)
d=fdesign(combtype,specstring)
d=fdesign(combtype,specstring,specvalue1,specvalue2,...)
d=fdesign.comb(...,Fs)
```

### Description

`fdesign.comb` specifies a peaking or notching comb filter. Comb filters amplify or attenuate a set of harmonically related frequencies.

`d=fdesign.comb` creates a notching comb filter specification object and applies default values for the filter order ( $N=10$ ) and quality factor ( $Q=16$ ).

`d=fdesign.comb(combtype)` creates a comb filter specification object of the specified type and applies default values for the filter order and quality factor. The valid entries for `combtype` are shown in the following table. The entries are not case-sensitive.

| Argument | Description  |
|----------|--|
| notch    | creates a comb filter that attenuates a set of harmonically related frequencies. |
| peak     | creates a comb filter that amplifies a set of harmonically related frequencies.  |

`d=fdesign(combtype,specstring)` creates a comb filter specification object of type `combtype` and sets its `Specification` property to `specstring` with default values. The entries in `specstring` determine the number of peaks or notches in the comb filter as well as their bandwidth and slope. Valid entries for `specstring` are shown below. The entries are not case-sensitive.

- 'N,Q' (default)
- ''N,BW'
- 'L,BW,GWB,Nsh'

The following table describes the arguments in `specstring`.

| Argument | Description   |
|----------|---|
| BW       | Bandwidth of the notch or peak. By default the bandwidth is calculated at the point -3 dB down from the center frequency of the peak or notch. For example, setting <code>BW=0.01</code> specifies that the -3 dB point will be +/- 0.005 (in normalized frequency) from the center of the notch or peak. |

| Argument | Description  |
|----------|--|
| GWB      | Gain at which the bandwidth is measured. This allows the user to specify the bandwidth of the notch or peak at a gain different from the -3 dB default.  |
| L        | Upsampling factor for a shelving filter of order Nsh. L determines the number of peaks or notches, which are equally spaced over the normalized frequency interval [-1,1].   |
| N        | Filter order. Specifies a filter with N+1 numerator and denominator coefficients. The filter will have N peaks or notches equally spaced over the interval [-1,1].   |
| Nsh      | Shelving filter order. Nsh represents a positive integer that determines the sharpness of the peaks or notches. The greater the value of the shelving filter order, the steeper the slope of the peak or notch. This results in a filter of order L*Nsh. |
| Q        | Peak or notch quality factor. Q represents the ratio of the lowest center frequency peak or notch (not including DC) to the bandwidth calculated at the -3 dB point.   |

`d=fdesign(combtype,specstring,specvalue1,specvalue2,...)` creates an IIR comb filter specification object of type `combtype` and sets its `Specification` property to the values in `specvalue1,specvalue2,...`

`d=fdesign.comb(...,Fs)` creates an IIR comb filter specification object using the sampling frequency, `Fs`, of the signal to be filtered. The function assumes that `Fs` is in Hertz and must be specified as a scalar trailing all other provided values.

## Examples

### Construct a Peaking and Notching Combining Filter

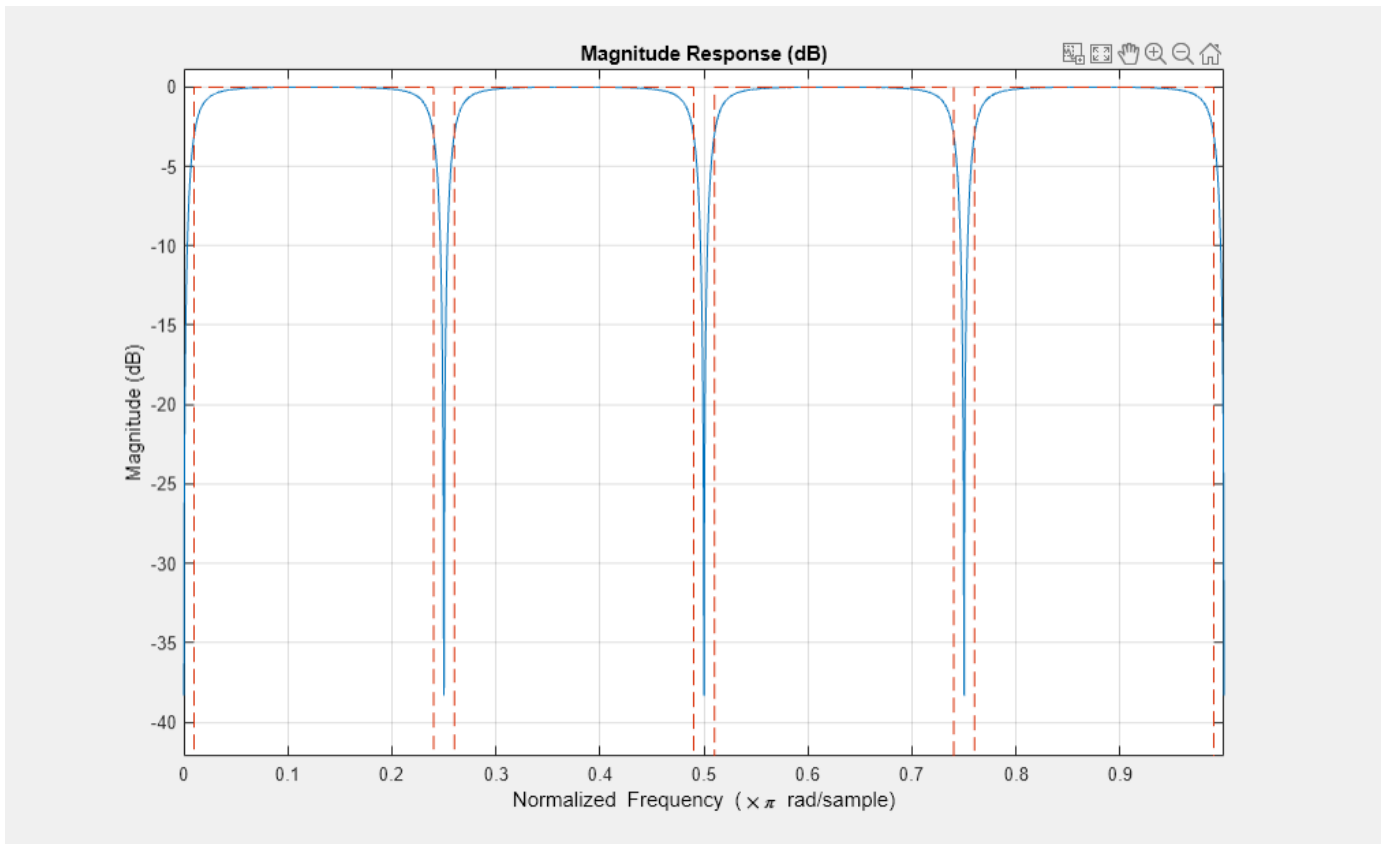
These examples demonstrate how to create IIR comb filter specification objects.

First, create a default specification object.

```
d = fdesign.comb; %#ok
```

In the next example, create a notching filter of order 8 with a bandwidth of 0.02 (normalized frequency) referenced to the -3 dB point.

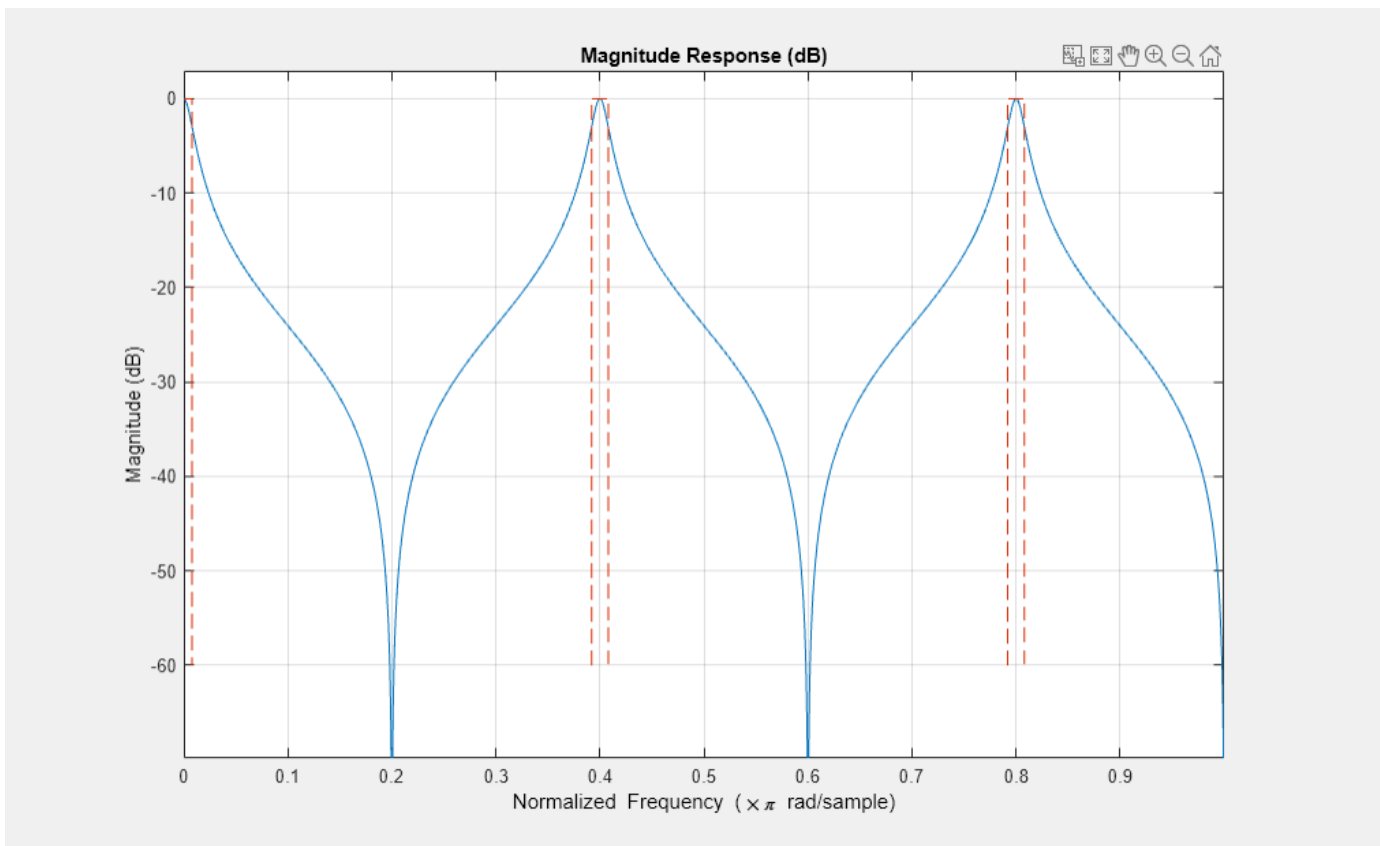
```
d = fdesign.comb('notch','N,BW',8,0.02);
Hd = design(d,'SystemObject',true);
fvtool(Hd);
```



Next, create a peaking comb filter with 5 peaks and a quality factor of 25.

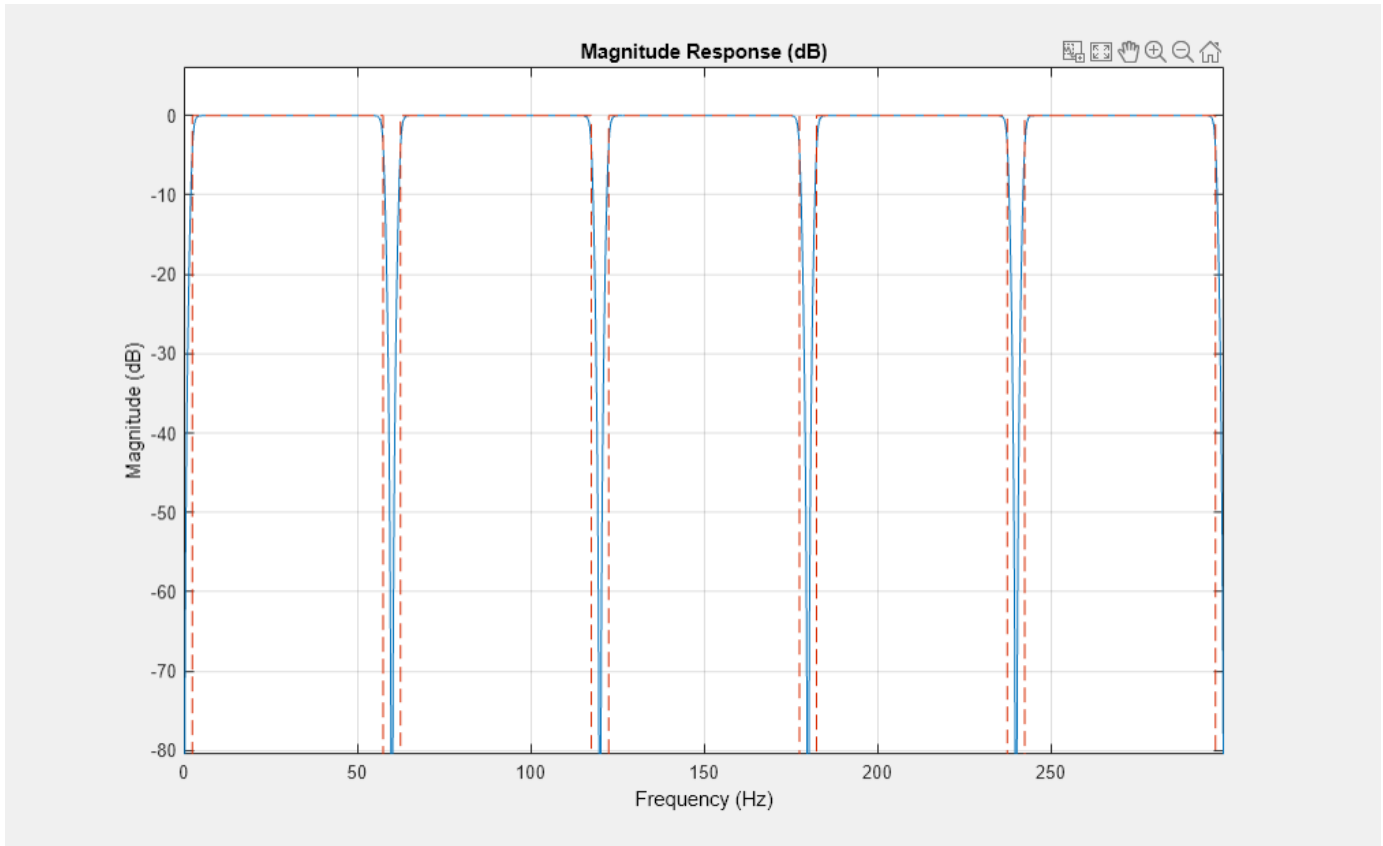
```
d = fdesign.comb('peak', 'N,Q', 5, 25);  
Hd = design(d, 'SystemObject', true);  
fvtool(Hd);
```





In the next example, create a notching filter to remove interference at 60 Hz and its harmonics. The following code creates a filter with 10 notches and a notch bandwidth of 5 Hz referenced to the -4dB level. The filter has a shelving filter order of 4 and a sampling frequency of 600 Hz. Because the notches are equidistantly spaced in the interval [-300, 300] Hz, they occur at multiples of 60 Hz.

```
d = fdesign.comb('notch','L,BW,GBW,Nsh',10,5,-4,4,600);
Hd=design(d,'SystemObject',true);
fvtool(Hd);
```



## Version History

Introduced in R2011a

# fdesign.decimator

Decimator filter specification object

---

## Note

---

**Note** The 'Raised Cosine' and 'Square Root Raised Cosine' response methods in the `fdesign.decimator` object will be removed in a future release. Use `rcosdesign` and `comm.RaisedCosineReceiveFilter` instead.

---



---

**Note** Support for designing an `mfilt.firdecim` object using the `fdesign.decimator` and the design functions will be removed in a future release. Use the `design` function with the `SystemObject=true` flag to design an FIR decimator System object™.

---



---

**Note** Support for multistage filter design using the `fdesign.decimator` object will be removed in a future release. Use the `designMultistageDecimator` function instead.

---

For more information, see “Compatibility Considerations”.

---

## Syntax

```
decimSpecs = fdesign.decimator(M)
decimSpecs = fdesign.decimator(M, RESPONSE)
decimSpecs = fdesign.decimator(M, CICRESPONSE, D)
decimSpecs = fdesign.decimator(M, RESPONSE, SPEC)
decimSpecs = fdesign.decimator(...,SPEC,specvalue1,specvalue2,...)
decimSpecs = fdesign.decimator(...,Fs)
decimSpecs = fdesign.decimator(...,MAGUNITS)
```

## Description

`decimSpecs = fdesign.decimator(M)` constructs a decimator filter specification object `decimSpecs` with the `DecimationFactor` property equal to the positive integer `M` and the `Response` property set to 'Nyquist'. The default values for the transition width and stopband attenuation in the Nyquist design are  $0.1\pi$  radians/sample and 80 dB. If `M` is unspecified, `M` defaults to 2.

`decimSpecs = fdesign.decimator(M, RESPONSE)` constructs a decimator specification object with the decimation factor `M` and the 'Response' property.

`decimSpecs = fdesign.decimator(M, CICRESPONSE, D)` constructs a CIC or CIC compensator decimator specification object with the decimation factor, `M`, 'Response' property equal to 'CIC' or 'CICCOMP', and `D` equal to the differential delay on page 4-252. The differential delay `D` must precede any specification option.

Because you are designing multirate filters, the specification options available are not the same as the specifications for designing single-rate filters. The decimation factor  $M$  is not included in the specification options. Different filter responses support different specifications. The following table lists the supported response types and specification options. The options are not case sensitive.

| Design Method                   | Valid Specification Options   |
|---------------------------------|---|
| 'Arbitrary Magnitude'           | See <code>fdesign.arbmag</code> for a description of the specification entries. <ul style="list-style-type: none"> <li>'N,F,A' (default option)</li> <li>'N,B,F,A'</li> </ul>   |
| 'Arbitrary Magnitude and Phase' | See <code>fdesign.arbmagnphase</code> for a description of the specification entries. <ul style="list-style-type: none"> <li>'N,F,H' (default option)</li> <li>'N,B,F,H'</li> </ul>   |
| 'Bandpass'                      | See <code>fdesign.bandpass</code> for a description of the specification entries. <ul style="list-style-type: none"> <li>'Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2' (default option)</li> <li>'N,Fc1,Fc2'</li> <li>'N,Fst1,Fp1,Fp2,Fst2'</li> </ul>   |
| 'Bandstop'                      | See <code>fdesign.bandstop</code> for a description of the specification entries. <ul style="list-style-type: none"> <li>'N,Fc1,Fc2'</li> <li>'N,Fp1,Fst1,Fst2,Fp2'</li> <li>'Fp1,Fst1,Fst2,Fp2,Ap1,Ast,Ap2' (default option)</li> </ul>  |
| 'CIC'                           | 'Fp,Ast' — Only valid specification. Fp is the passband frequency and Ast is the stopband attenuation in decibels.<br><br>To specify a CIC decimator, include the differential delay after 'CIC' and before the filter specification option: 'Fp,Ast'. For example:<br><code>d = fdesign.decimator(2,'cic',4,'Fp,Ast',0.4,40);</code>   |
| 'CIC Compensator'               | See <code>fdesign.ciccomp</code> for a description of the specification entries. <ul style="list-style-type: none"> <li>'Fp,Fst,Ap,Ast' (default option)</li> <li>'N,Fc,Ap,Ast'</li> <li>'N,Fp,Ap,Ast'</li> <li>'N,Fp,Fst'</li> <li>'N,Fst,Ap,Ast'</li> </ul><br>To specify a CIC compensator decimator, include the differential delay after 'CICCOMP' and before the filter specification. For example:<br><code>d = fdesign.decimator(2,'ciccomp',4);</code> |
| 'Differentiator'                | 'N' — filter order  |

| Design Method           | Valid Specification Options   |
|-------------------------|---|
| 'Gaussian'              | <p>'Nsym, BT' — Nsym is the filter order in symbols and BT is the bandwidth-symbol time product.</p> <p>The specification must be preceded by an integer-valued SamplesPerSymbol.</p>   |
| 'Halfband'              | <p>See <code>fdesign.halfband</code> for a description of the specification entries.</p> <ul style="list-style-type: none"> <li>• 'TW, Ast' (default option)</li> <li>• 'N, TW'</li> <li>• 'N'</li> <li>• 'N, Ast'</li> </ul> <p>If you use the quasi-linear IIR design method, <code>iirlinphase</code>, with a halfband specification, the interpolation factor must be 2.</p>                |
| 'Highpass'              | <p>See <code>fdesign.highpass</code> for a description of the specification entries.</p> <ul style="list-style-type: none"> <li>• 'Fst, Fp, Ast, Ap' (default option)</li> <li>• 'N, F3db'</li> <li>• 'N, Fc'</li> <li>• 'N, Fc, Ast, Ap'</li> <li>• 'N, Fp, Ast, Ap'</li> <li>• 'N, Fst, Ast, Ap'</li> <li>• 'N, Fst, Fp'</li> <li>• 'N, Fst, Ast, Ap'</li> <li>• 'N, Fst, Fp, Ast'</li> </ul> |
| 'Hilbert'               | <p>See <code>fdesign.hilbert</code> for a description of the specification entries.</p> <ul style="list-style-type: none"> <li>• 'N, TW' (default option)</li> <li>• 'TW, Ap'</li> </ul>  |
| 'Inverse-sinc Lowpass'  | <p>See <code>fdesign.isinclp</code> for a description of the specification entries.</p> <ul style="list-style-type: none"> <li>• 'Fp, Fst, Ap, Ast' (default option)</li> <li>• 'N, Fc, Ap, Ast'</li> <li>• 'N, Fp, Fst'</li> <li>• 'N, Fst, Ap, Ast'</li> </ul>  |
| 'Inverse-sinc Highpass' | <p>See <code>fdesign.isinchp</code> for a description of the specification entries.</p> <ul style="list-style-type: none"> <li>• 'Fst, Fp, Ast, Ap' (default option)</li> <li>• 'N, Fc, Ast, Ap'</li> <li>• 'N, Fst, Fp'</li> <li>• 'N, Fst, Ast, Ap'</li> </ul>  |

| Design Method | Valid Specification Options  |
|---------------|--|
| 'Lowpass'     | See <code>fdesign.lowpass</code> for a description of the specification entries. <ul style="list-style-type: none"> <li>'Fp,Fst,Ap,Ast' (default option)</li> <li>'N,F3dB'</li> <li>'N,Fc'</li> <li>'N,Fc,Ap,Ast'</li> <li>'N,Fp,Ap,Ast'</li> <li>'N,Fp,Fst'</li> <li>'N,Fp,Fst,Ap'</li> <li>'N,Fp,Fst,Ast'</li> <li>'N,Fst,Ap,Ast'</li> </ul>       |
| 'Nyquist'     | See <code>fdesign.nyquist</code> for a description of the specification entries. For all Nyquist specifications, you must specify the <i>L</i> th band. This typically corresponds to the <code>DecimationFactor</code> . <ul style="list-style-type: none"> <li>'TW,Ast' (default option)</li> <li>'N'</li> <li>'N,Ast'</li> <li>'N,Ast'</li> </ul> |

`decimSpecs = fdesign.decimator(M, RESPONSE, SPEC)` constructs object `decimSpecs` and sets the `Specification` property to `SPEC` for the response type, `RESPONSE`. Entries in the `SPEC` represent various filter response features, such as the filter order, that govern the filter design. Valid entries for `SPEC` depend on the `RESPONSE` type.

Because you are designing multirate filters, the specification options available are not the same as the specifications for designing single-rate filters with such response types as `fdesign.lowpass`. The options are not case sensitive.

The decimation factor `M` is not in the specification options.

`decimSpecs = fdesign.decimator(...,SPEC,specvalue1,specvalue2,...)` constructs an object `decimSpecs` and sets its specifications at construction time.

`decimSpecs = fdesign.decimator(...,Fs)` provides the sampling frequency of the signal to be filtered. `Fs` must be specified as a scalar trailing the other numerical values provided. `Fs` is assumed to be in Hz as are all other frequency values provided.

`decimSpecs = fdesign.decimator(...,MAGUNITS)` specifies the units for any magnitude specification you provide in the input arguments. `MAGUNITS` can be one of

- 'linear' — specify the magnitude in linear units.
- 'dB' — specify the magnitude in dB (decibels).
- 'squared' — specify the magnitude in power units.

When you omit the `MAGUNITS` argument, `fdesign` assumes that all magnitudes are in decibels. Note that `fdesign` stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

## Examples

### Construct a Decimator Using fdesign Object

These examples show how to construct decimating filter specification objects.

First, create a default specifications object without using input arguments except for the decimation factor  $m$ .

```
d = fdesign.decimator(2, 'Nyquist', 2, 0.1, 80) %#ok % Set tw=0.1, and ast=80.
```

```
d =
  decimator with properties:
      MultirateType: 'Decimator'
      Response: 'Nyquist'
  DecimationFactor: 2
      Specification: 'TW,Ast'
      Description: {2x1 cell}
      Band: 2
  NormalizedFrequency: 1
      TransitionWidth: 0.1000
      Astop: 80
```

Now create an object by passing a specification type option 'fst1,fp1,fp2,fst2,ast1,ap,ast2' and a design - the resulting object uses default values for the filter specifications. You must provide the design input argument, bandpass in this example, when you include a specification.

```
d = fdesign.decimator(8, 'bandpass', ...
  'fst1,fp1,fp2,fst2,ast1,ap,ast2'); %#ok
```

Create another decimating filter specification object, passing the specification values to the object rather than accepting the default values for fp,fst,ap,ast.

```
d = fdesign.decimator(3, 'lowpass', .45, 0.55, .1, 60); %#ok
```

Now pass the filter specifications that correspond to the specifications - n,fc,ap,ast.

```
d = fdesign.decimator(3, 'ciccomp', 1, 2, 'n,fc,ap,ast', ...
  20, 0.45, .05, 50);
```

Now design a decimator using the equiripple design method.

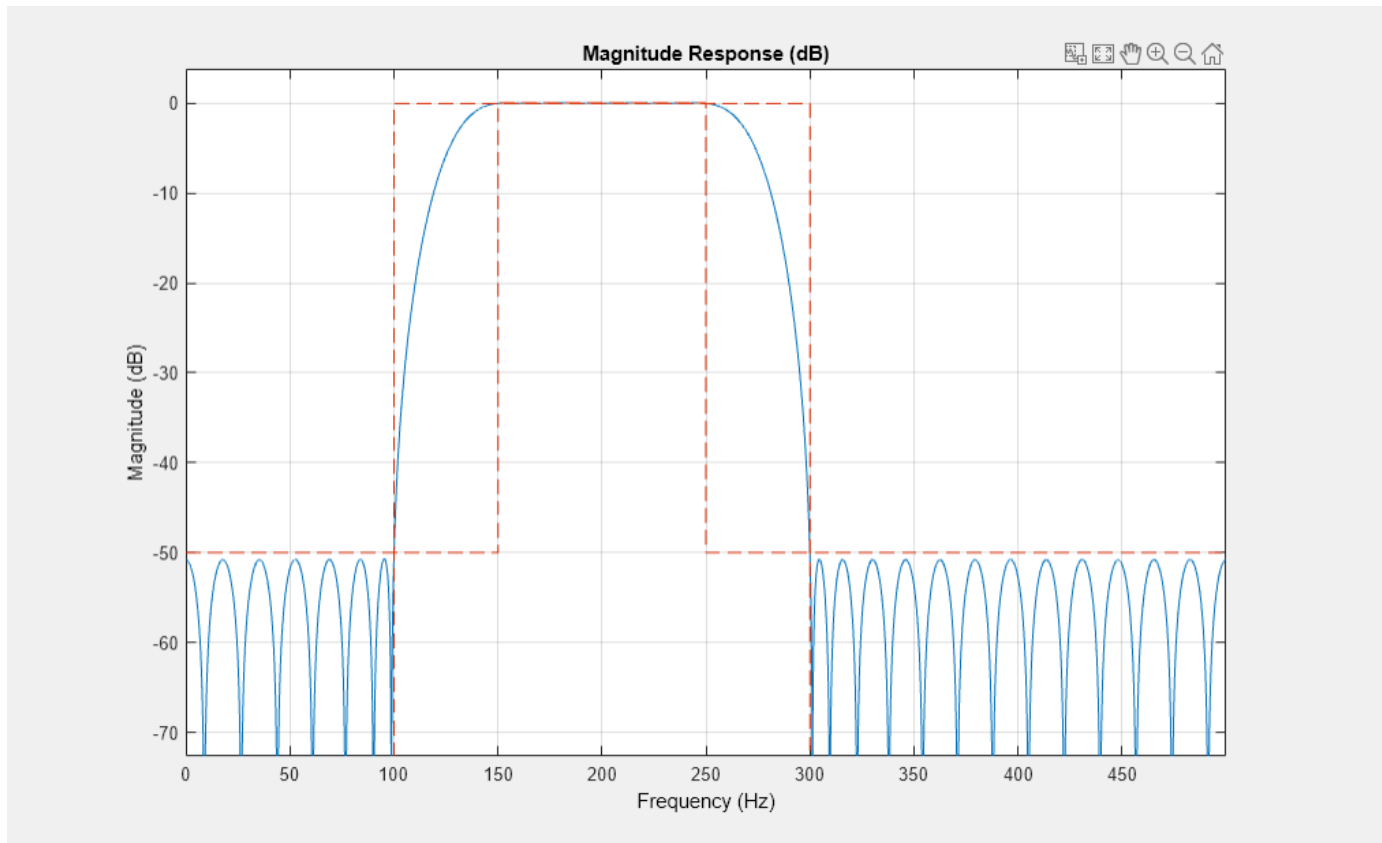
```
equiDecimator = design(d, 'equiripple', SystemObject=true);
```

Pass a new specification type for the filter, specifying the filter order. Note that the inputs must include the differential delay  $dd$  with the CIC input argument to design a CIC specification object.

```
m = 5;
dd = 2;
d = fdesign.decimator(m, 'cic', dd, 'fp,ast', 0.55, 55); %#ok
```

In this example, you specify a sampling frequency as the last input argument. Here is it 1000 Hz. Design an equiripple filter and plot the magnitude response:

```
d = fdesign.decimator(8, 'bandpass', 'fst1,fp1,fp2,fst2,ast1,ap,ast2',...
100,150,250,300,50,.05,50,1000);
fvtool(design(d, 'equiripple', SystemObject=true))
```



## Version History

Introduced in R2011a

**R2022a: 'Raised Cosine' and 'Square Root Raised Cosine' response methods will be removed**

*Warns starting in R2022a*

The 'Raised Cosine' and 'Square Root Raised Cosine' response methods in the `fdesign.decimator` object will be removed in a future release. Use `rcosdesign` and `comm.RaisedCosineReceiveFilter` instead.

**R2022b: Support for multistage design in `fdesign.decimator` will be removed**

*Warns starting in R2022b*

Support for multistage filter design using the `fdesign.decimator` object will be removed in a future release. Use the `designMultistageDecimator` function instead.

### Update Code

This table shows how to update your code.



| Discouraged Usage  | Recommended Replacement  |
|--|--|
| Using the design function and passing the 'multistage' flag as an input.<br><br><pre>D = 9; % Decimation factor TW = 2205; % Transition width Astop = 80; % Stopband attenuation Fs = 44100; % Input sample rate filtSpec = fdesign.decimator(D,'nyquist',D,TW,Astop,Fs) filtObj = design(filtSpec,'multistage',SystemObject=true)</pre> | When you set the cost method to 'design', the function designs an optimal number of stages so that the overall cost of implementation is low.<br><br><pre>filtObj = designMultistageDecimator(D,Fs,TW,Astop,... CostMethod='design')</pre> |
| Using the multistage function directly.<br><br><pre>filtObj = multistage(filtSpec,SystemObject=true)</pre>   |  |

### R2022b: Support for designing `mfilt.firdecim` will be removed

Warns starting in R2022b

Support for designing an `mfilt.firdecim` object using the `fdesign.decimator` and the `design` functions will be removed in a future release.

To design an FIR decimator using the `fdesign.decimator` filter specification object, use the `design` function with the `SystemObject=true` flag. Setting this flag to true ensures that the output object is a filter `System` object instead of an `mfilt.firdecim` object.

```
fdesignObj = fdesign.decimator;
FIRDecimObj = design(fdesignObj,SystemObject=true)
```

```
FIRDecimObj =
```

```
    dsp.FIRDecimator with properties:
```

```
    DecimationFactor: 2
    NumeratorSource: 'Property'
        Numerator: [-1.5162e-05 0 4.1707e-05 0 ... ]
        Structure: 'Direct form'
```

```
    Show all properties
```

### See Also

[fdesign](#) | [fdesign.arbmagnphase](#) | [fdesign.interpolator](#) | [fdesign.rsrc](#)

## fdesign.differentiator

Differentiator filter specification object

### Syntax

```
D = fdesign.differentiator
D = fdesign.differentiator(SPEC)
D = fdesign.differentiator(SPEC,specvalue1,specvalue2, ...)
D = fdesign.differentiator(specvalue1)
D = fdesign.differentiator(...,Fs)
D = fdesign.differentiator(...,MAGUNITS)
```

### Description

`D = fdesign.differentiator` constructs a default differentiator filter designer `D` with the filter order set to 31.

`D = fdesign.differentiator(SPEC)` initializes the filter designer `Specification` property to `SPEC`. You provide one of the following filter entries as input to replace `SPEC`. These entries are not case sensitive.

---

**Note** Specifications marked with an asterisk require the DSP System Toolbox software.

---

- 'N' — Full band differentiator (default)
- 'N,Fp,Fst' — Partial band differentiator
- 'N,Fp,Fst,Ap' — Partial band differentiator \*
- 'N,Fp,Fst,Ast' — Partial band differentiator \*
- 'Ap' — Minimum order full band differentiator \*
- 'Fp,Fst,Ap,Ast' — Minimum order partial band differentiator \*

The filter specifications are defined as follows:

- `Ap` — amount of ripple allowed in the pass band in decibels (the default units). Also called `Apass`.
- `Ast` — attenuation in the stop band in decibels (the default units). Also called `Astop`.
- `Fp` — frequency at the start of the pass band. Specified in normalized frequency units. Also called `Fpass`.
- `Fst` — frequency at the end of the stop band. Specified in normalized frequency units. Also called `Fstop`.
- `N` — filter order.

By default, `fdesign.differentiator` assumes that all frequency specifications are provided in normalized frequency units. Also, decibels is the default for all magnitude specifications.

Use `designopts` to determine the design options for a given design method. Enter `help(D,METHOD)` at the MATLAB command line to obtain detailed help on the design options for a given design method, `METHOD`.

`D = fdesign.differentiator(SPEC,specvalue1,specvalue2, ...)` initializes the filter designer specifications in `SPEC` with `specvalue1`, `specvalue2`, and so on. To get a description of the specifications `specvalue1`, `specvalue2`, and more, enter

```
get(d,'description')
```

at the Command prompt.

`D = fdesign.differentiator(specvalue1)` assumes the default specification `N`, setting the filter order to the value you provide.

`D = fdesign.differentiator(...,Fs)` adds the argument `Fs`, specified in Hz to define the sampling frequency to use. In this case, all frequencies in the specifications are in Hz as well.

`D = fdesign.differentiator(...,MAGUNITS)` specifies the units for any magnitude specification you provide in the input arguments. `MAGUNITS` can be one of

- 'linear' — specify the magnitude in linear units
- 'dB' — specify the magnitude in dB (decibels)
- 'squared' — specify the magnitude in power units

When you omit the `MAGUNITS` argument, `fdesign` assumes that all magnitudes are in decibels. Note that `fdesign` stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

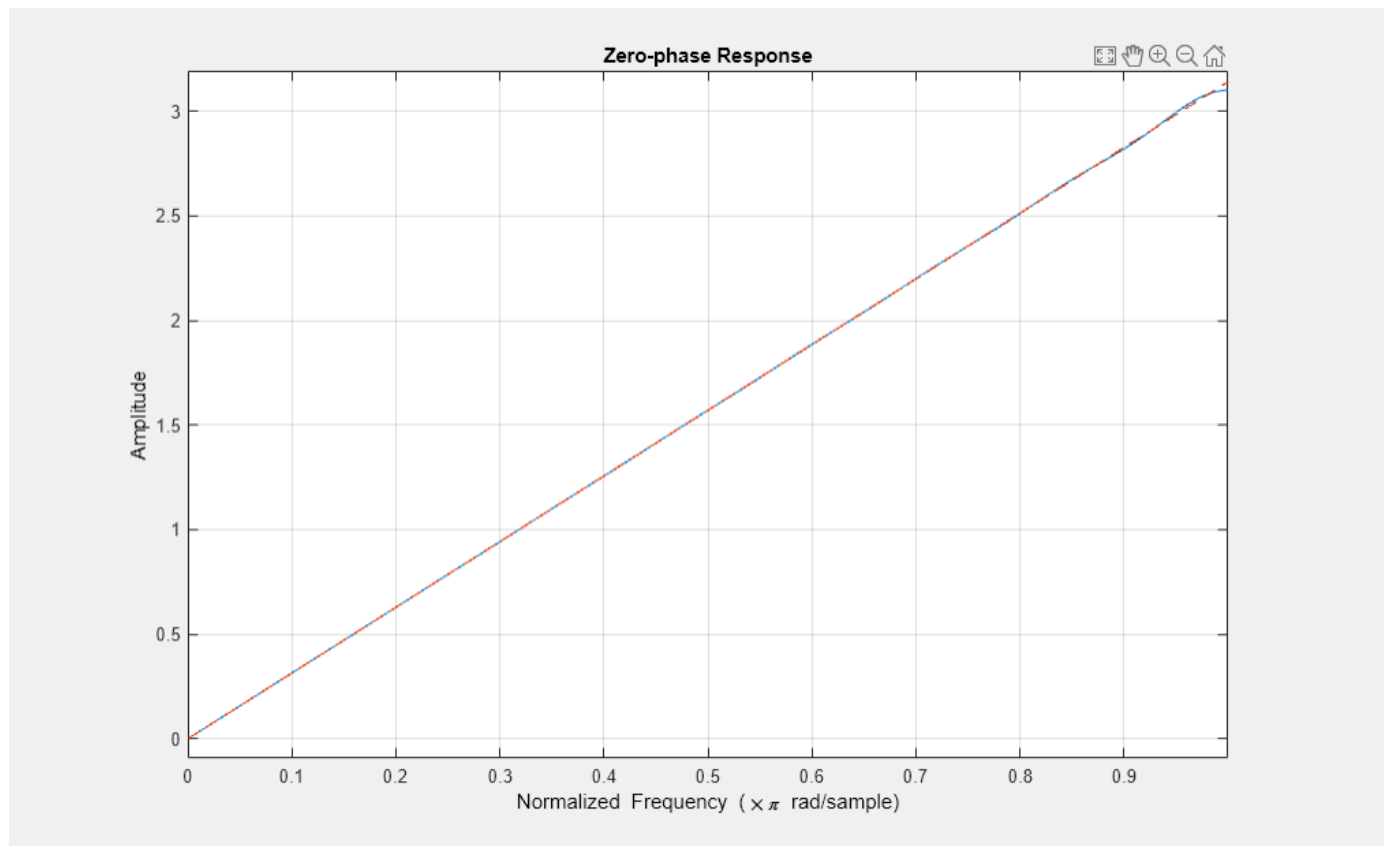
## Examples

### FIR Differentiator Filters

Design a 33rd-order FIR differentiator using least squares. Plot the zero-phase response of the filter.

```
d = fdesign.differentiator(33);
hd = design(d,'firls','SystemObject',true);

zerophase(hd)
```



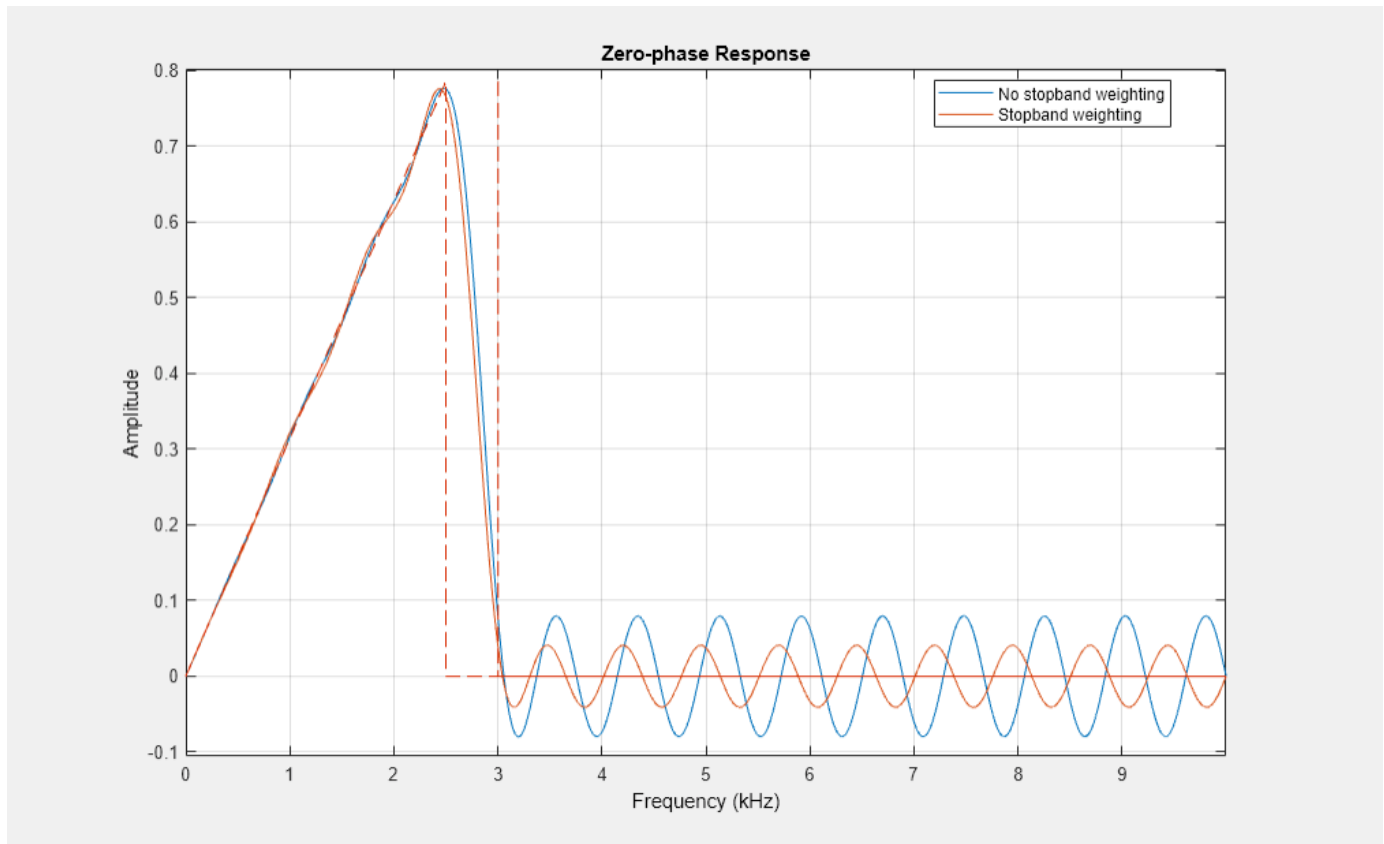
Design a 54th-order narrowband equiripple differentiator. Differentiate the lowest 25% of the frequencies in the Nyquist range and filter the higher frequencies. Specify a sample rate of 20 kHz, a passband frequency of 2.5 kHz, and a stopband frequency of 3 kHz.

```
Fs = 20000;
```

```
d = fdesign.differentiator('N,Fp,Fst',54,2500,3000,Fs);
Hd = design(d,'equiripple','SystemObject',true);
```

Redesign the filter, but this time weight the stopband to increase the attenuation.

```
Hd1 = design(d,'equiripple','Wstop',4,'SystemObject',true);
hfv1 = fvtool(Hd,Hd1,'MagnitudeDisplay','zero-phase',...
    'FrequencyRange',[0, Fs/2]);
legend(hfv1,'No stopband weighting','Stopband weighting');
```



## Version History

Introduced in R2009a

### See Also

design | fdesign

## fdesign.fracdelay

Fractional delay filter specification object

### Syntax

```
d = fdesign.fracdelay(delta)
d = fdesign.fracdelay(delta, 'N')
d = fdesign.fracdelay(delta, 'N', n)
d = fdesign.fracdelay(delta, n)
d = fdesign.fracdelay(..., fs)
```

### Description

`d = fdesign.fracdelay(delta)` constructs a default fractional delay filter designer `d` with the filter order set to 3 and the delay value set to `delta`. The fractional delay `delta` must be between 0 and 1 samples.

`d = fdesign.fracdelay(delta, 'N')` initializes the filter designer specification to `N`, where `N` specifies the fractional delay filter order and defaults to filter order of 3.

Use `designmethods(d)` to get a list of the design methods available for a specification.

`d = fdesign.fracdelay(delta, 'N', n)` initializes the filter designer to `N` and sets the filter order to `n`.

`d = fdesign.fracdelay(delta, n)` assumes the default specification `N`, filter order, and sets the filter order to the value you provide in input `n`.

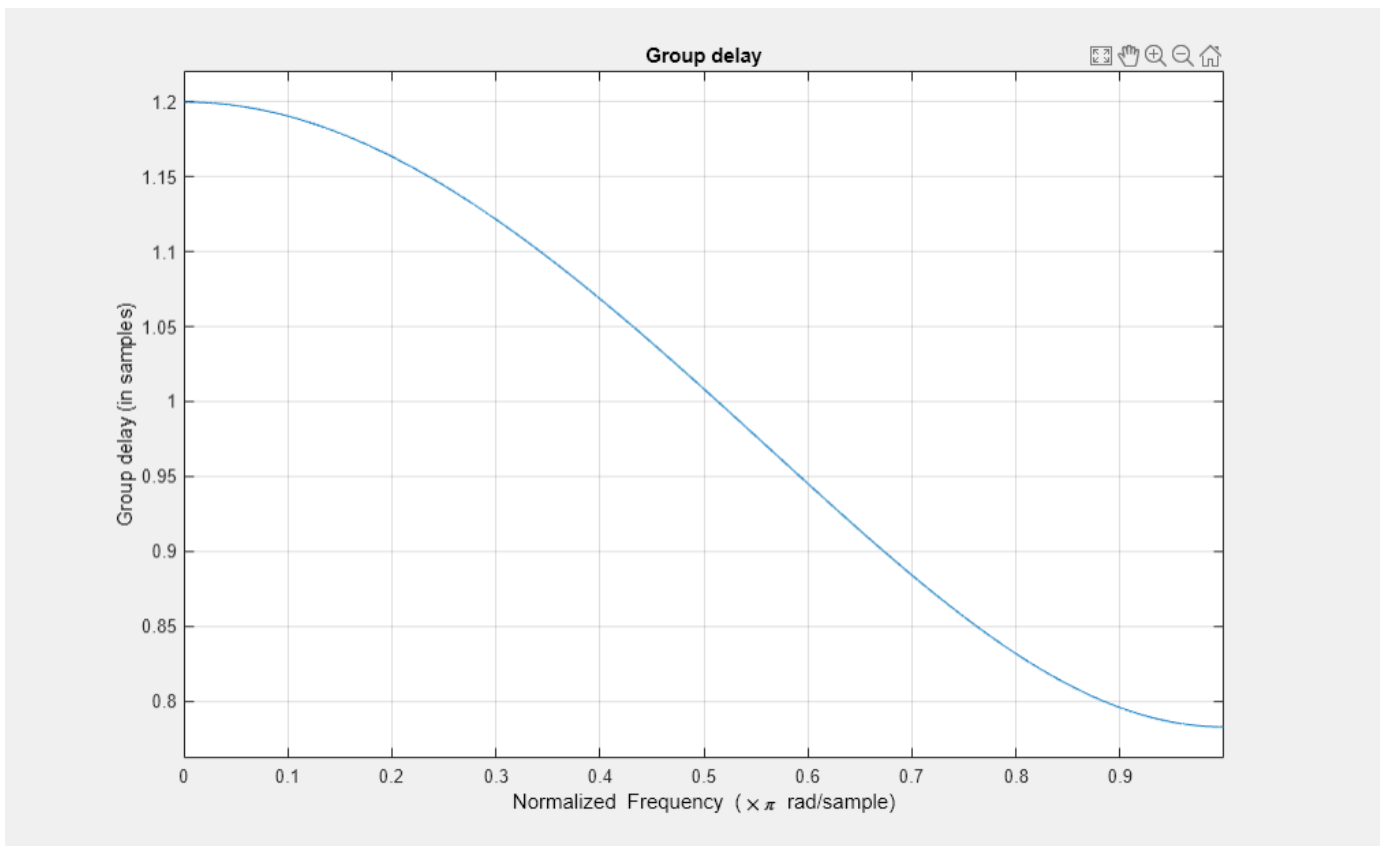
`d = fdesign.fracdelay(..., fs)` adds the argument `fs`, specified in units of Hertz (Hz) to define the sampling frequency. In this case, specify the fractional delay `delta` to be between 0 and  $1/fs$ .

### Examples

#### Create a Fractional Delay Filter Using fdesign Object

Design a second-order fractional delay filter of 0.2 samples using the Lagrange method. Implement the filter using a Farrow fractional delay (`fd`) structure.

```
d = fdesign.fracdelay(0.2, 'N', 2);
secondOrderFrac = design(d, 'lagrange', 'filterstructure', 'farrowfd');
fvtool(secondOrderFrac, 'analysis', 'grpdelay')
```



Design a cubic fractional delay filter with a sampling frequency of 8 kHz and fractional delay of 50 microseconds using the Lagrange method.

```
d = fdesign.fracdelay(50e-6, 'N', 3, 8000);  
cubicFrac = design(d, 'lagrange', 'FilterStructure', 'farrowfd');
```

## Version History

Introduced in R2011a

### See Also

[design](#) | [designopts](#) | [fdesign](#) | [setspecs](#)

## fdesign.halfband

Halfband filter specification object

### Syntax

```
d = fdesign.halfband
d = fdesign.halfband('type',type)
d = fdesign.halfband(spec)
d = fdesign.halfband(spec,specvalue1,specvalue2,...)
d = fdesign.halfband(specvalue1,specvalue2)
d = fdesign.halfband(...,fs)
d = fdesign.halfband(...,magunits)
```

### Description

`d = fdesign.halfband` constructs a halfband filter specification object `d`, applying default values for the properties `tw` and `ast`.

Using `fdesign.halfband` along with `design` method generates a `System` object, if the `'SystemObject'` flag in the `design` method is set to `true`.

`d = fdesign.halfband('type',type)` initializes the filter designer `'Type'` property with `type`. `'type'` must be either `lowpass` or `highpass` and is not case sensitive.

`d = fdesign.halfband(spec)` constructs object `d` and sets its `'Specification'` to `spec`. Entries in `spec` represent various filter response features, such as the filter order, that govern the filter design. Valid entries for `spec` are shown below. These options are not case sensitive.

- `tw,ast` (default `spec`)
- `n,tw`
- `n`
- `n,ast`

where,

- `ast` — attenuation in the stop band in decibels (the default units).
- `n` — filter order.
- `tw` — width of the transition region between the pass and stop bands. Specified in normalized frequency units.

By default, all frequency specifications are assumed to be in normalized frequency units. Moreover, all magnitude specifications are assumed to be in dB. Different specification types may have different design methods available.

The filter design methods that apply to a halfband filter specification object change depending on the `Specification` choice. Use `designmethods` to determine which design method applies to an object and its specification choice. Different filter design methods also have options that you can specify. Use `designopts` with the design method to see the available options. For example:



```
f=fdesign.halfband('N,TW');
designmethods(f)
```

`d = fdesign.halfband(spec,specvalue1,specvalue2,...)` constructs an object `d` and sets its specifications at construction time.

`d = fdesign.halfband(specvalue1,specvalue2)` constructs an object `d` assuming the default Specification property `tw,ast`, using the values you provide for the input arguments `specvalue1` and `specvalue2` for `tw` and `ast`.

`d = fdesign.halfband(...,fs)` adds the argument `fs`, specified in Hz to define the sampling frequency to use. In this case, all frequencies in the specifications are in Hz as well.

`d = fdesign.halfband(...,magunits)` specifies the units for any magnitude specification you provide in the input arguments. `magunits` can be one of

- `linear` — specify the magnitude in linear units
- `dB` — specify the magnitude in dB (decibels)
- `squared` — specify the magnitude in power units

When you omit the `magunits` argument, `fdesign` assumes that all magnitudes are in decibels. Note that `fdesign` stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

## Examples

### Create Halfband Filters Using fdesign Object

Create a default halfband filter specifications object.

```
d=fdesign.halfband;
```

Create another halfband filter object, passing the specification values to the object rather than accepting the default values for `n` and `ast`.

```
d2 = fdesign.halfband('n,ast', 42, 80);
```

For another example, pass the filter values that correspond to the default Specification - `n,ast`.

```
d3 = fdesign.halfband(.01, 80);
```

This example designs an equiripple FIR filter, starting by passing a new specification type and specification values to `fdesign.halfband`.

```
hs = fdesign.halfband('n,ast',80,70);
hd =design(hs,'equiripple','SystemObject',true);
```

In this example, pass the specifications for the filter, and then design a least-squares FIR filter from the object, using `firls` as the design method.

```
hs = fdesign.halfband('n,tw', 42, .04);
hd2 = design(hs,'firls','SystemObject',true);
```

Create two equiripple halfband filters with and without a nonnegative zero phase response:

```
f=fdesign.halfband('N,TW',12,0.2);
```

Equiripple halfband filter with nonnegative zero phase response

```
Hd1 = design(f,'equiripple','ZeroPhase',true,'SystemObject',true);
```

Equiripple halfband filter with zero phase false 'zerophase',false is the default

```
Hd2=design(f,'equiripple','ZeroPhase',false,'SystemObject',true);
```

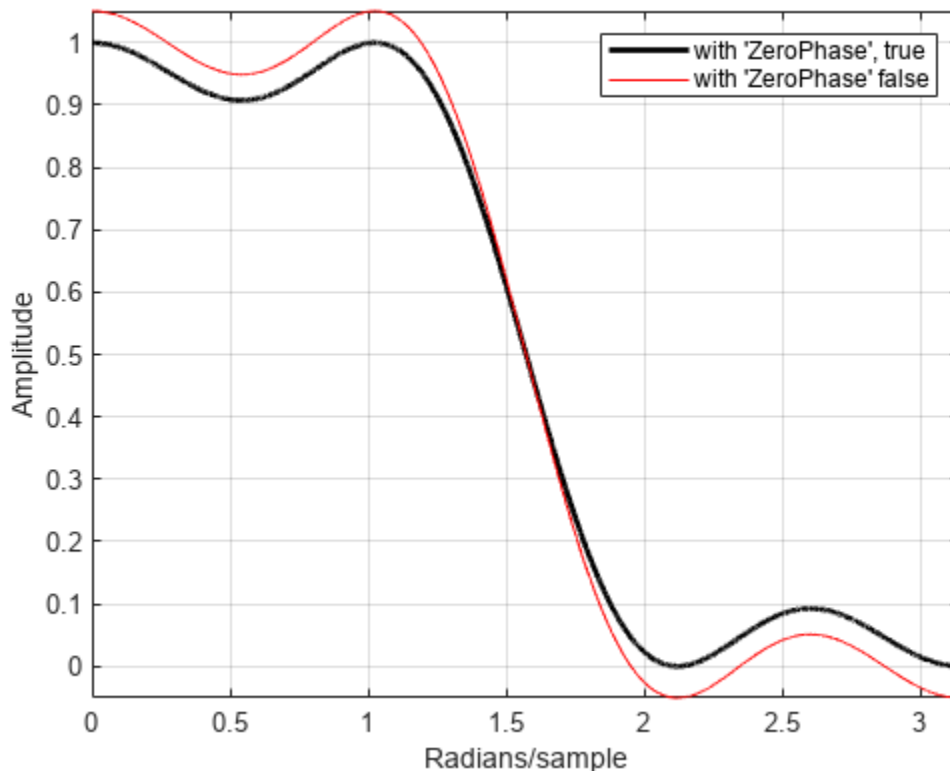
Obtain real-valued amplitudes (not magnitudes)

```
[Hr_zerophase,~]=zerophase(Hd1);
```

```
[Hr,W]=zerophase(Hd2);
```

Plot and compare response

```
plot(W,Hr_zerophase,'k','linewidth',2);
xlabel('Radians/sample'); ylabel('Amplitude');
hold on;
plot(W,Hr,'r');
axis tight; grid on;
legend('with ''ZeroPhase'' true','with ''ZeroPhase'' false');
```



Note that the amplitude of the zero phase response (black line) is nonnegative for all frequencies. The 'ZeroPhase' option is valid only for equiripple halfband designs with the 'N,TW' specification. You cannot specify 'MinPhase' and 'ZeroPhase' to be simultaneously 'true'.

## **Version History**

**Introduced in R2011a**

### **See Also**

fdesign | fdesign.decimator | design | fdesign.interpolator | fdesign.nyquist |  
setspecs | zerophase

## fdesign.highpass

Highpass filter specification object

### Syntax

```
D = fdesign.highpass
D = fdesign.highpass(SPEC)
D = fdesign.highpass(SPEC,specvalue1,specvalue2,...)
D = fdesign.highpass(specvalue1,specvalue2,specvalue3,
specvalue4)
D = fdesign.highpass(...,Fs)
D = fdesign.highpass(...,MAGUNITS)
```

### Description

`D = fdesign.highpass` constructs a highpass filter specification object `D`, applying default values for the specification, 'Fst,Fp,Ast,Ap'.

`D = fdesign.highpass(SPEC)` constructs object `D` and sets the `Specification` property to `SPEC`. Entries in the `SPEC` represent various filter response features, such as the filter order, that govern the filter design. Valid entries for `SPEC` are shown below. These entries are not case sensitive.

---

**Note** Specification entries marked with an asterisk require the DSP System Toolbox software.

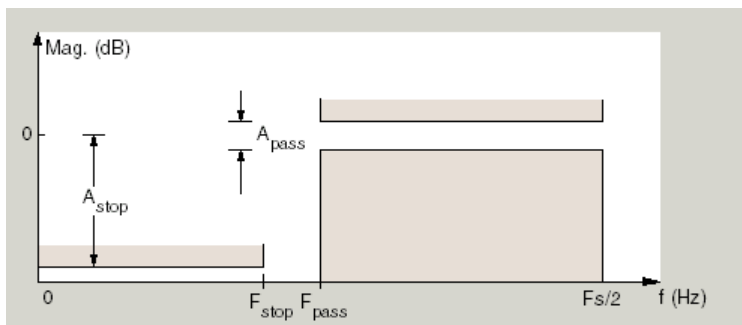
---

- 'Fst,Fp,Ast,Ap' (default spec)
- 'N,F3db'
- 'N,F3db,Ap' \*
- 'N,F3db,Ast' \*
- 'N,F3db,Ast,Ap' \*
- 'N,F3db,Fp' \*
- 'N,Fc'
- 'N,Fc,Ast,Ap'
- 'N,Fp,Ap'
- 'N,Fp,Ast,Ap'
- 'N,Fst,Ast'
- 'N,Fst,Ast,Ap'
- 'N,Fst,F3db' \*
- 'N,Fst,Fp'
- 'N,Fst,Fp,Ap' \*
- 'N,Fst,Fp,Ast' \*
- 'Nb,Na,Fst,Fp' \*

The filter specifications are defined as follows:

- $A_p$  — amount of ripple allowed in the pass band in decibels (the default units). Also called  $A_{pass}$ .
- $A_{st}$  — attenuation in the stop band in decibels (the default units). Also called  $A_{stop}$ .
- $F_{3db}$  — cutoff frequency for the point 3 dB point below the passband value. Specified in normalized frequency units.
- $F_c$  — cutoff frequency for the point 6 dB point below the passband value. Specified in normalized frequency units.
- $F_p$  — frequency at the start of the pass band. Specified in normalized frequency units. Also called  $F_{pass}$ .
- $F_{st}$  — frequency at the end of the stop band. Specified in normalized frequency units. Also called  $F_{stop}$ .
- $N$  — filter order.
- $N_a$  and  $N_b$  are the order of the denominator and numerator.

Graphically, the filter specifications look similar to those shown in the following figure.



Regions between specification values like  $F_{st}$  and  $F_p$  are transition regions where the filter response is not explicitly defined.

The filter design methods that apply to a highpass filter specification object change depending on the `Specification`. Use `designmethods` to determine which design method applies to an object and its specification.

Use `designopts` to determine which design options are valid for a given design method. For detailed information on design options for a given design method, `METHOD`, enter `help(D,METHOD)` at the MATLAB command line.

`D = fdesign.highpass(SPEC,specvalue1,specvalue2,...)` constructs an object `d` and sets its specification values at construction time.

`D = fdesign.highpass(specvalue1,specvalue2,specvalue3,specvalue4)` constructs an object `D` with the default `Specification` property and the values you enter for `specvalue1,specvalue2,...`

`D = fdesign.highpass(...,Fs)` provides the sampling frequency for the filter specification object. `Fs` is in Hz and must be specified as a scalar trailing the other numerical values provided. If you specify a sampling frequency, all other frequency specifications are in Hz.

`D = fdesign.highpass(...,MAGUNITS)` specifies the units for any magnitude specification you provide in the input arguments. `MAGUNITS` can be one of

- 'linear' — specify the magnitude in linear units
- 'dB' — specify the magnitude in dB (decibels)
- 'squared' — specify the magnitude in power units

When you omit the `MAGUNITS` argument, `fdesign` assumes that all magnitudes are in decibels. Note that `fdesign` stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

## Examples

### Design Butterworth Filter

Design a butterworth filter with lowpass and highpass frequency responses. The filter design procedure is:

- 1 Specify the filter design specifications using a `fdesign` function.
- 2 Pick a design method provided by the `designmethods` function.
- 3 To determine the available design options to choose from, use the `designoptions` function.
- 4 Design the filter using the `design` function.

### Lowpass Filter

Construct a default lowpass filter design specification object using `fdesign.lowpass`.

```
designSpecs = fdesign.lowpass

designSpecs =
    lowpass with properties:
        Response: 'Lowpass'
        Specification: 'Fp,Fst,Ap,Ast'
        Description: {4x1 cell}
        NormalizedFrequency: 1
            Fpass: 0.4500
            Fstop: 0.5500
            Apass: 1
            Astop: 60
```

Determine the available design methods using the `designmethods` function. To design a butterworth filter, pick `butter`.

```
designmethods(designSpecs,SystemObject=true)
```

```
Design Methods that support System objects for class fdesign.lowpass (Fp,Fst,Ap,Ast):
```

```
butter
cheby1
cheby2
ellip
equiripple
ifir
```

```
kaiserwin
multistage
```

While designing the filter, you can specify additional design options. View a list of the options using the `designoptions` function. This function also shows the default design options the filter uses.

```
designoptions(designSpecs, 'butter', SystemObject=true)
```

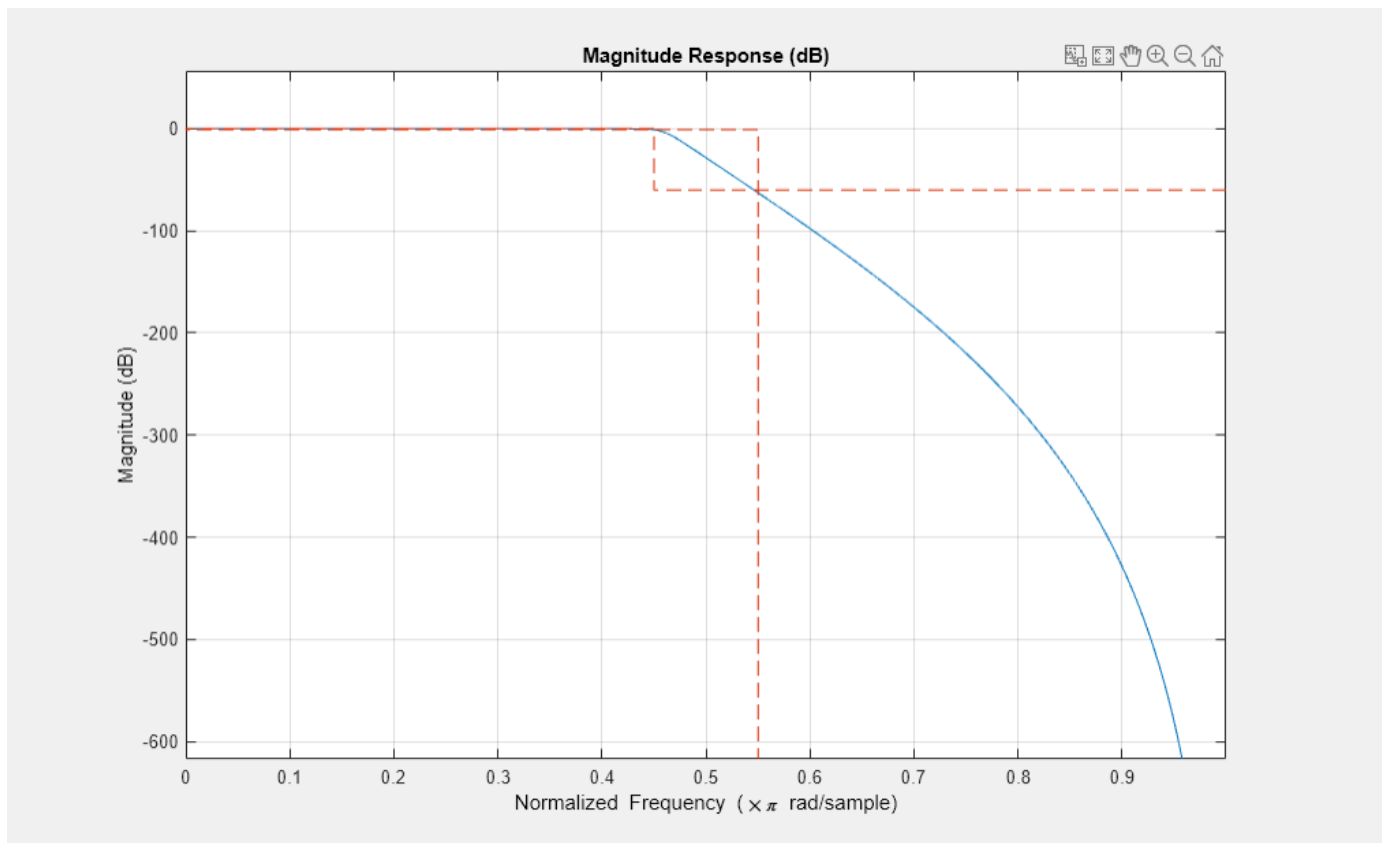
```
ans = struct with fields:
    FilterStructure: {'df1sos' 'df2sos' 'df1tsos' 'df2tsos' 'cascadeallpass' 'cascade...
    SOSScaleNorm: 'ustring'
    SOSScale0pts: 'fdopts.sosscaling'
    MatchExactly: {'passband' 'stopband'}
    DefaultFilterStructure: 'df2sos'
    DefaultMatchExactly: 'stopband'
    DefaultSOSScaleNorm: ''
    DefaultSOSScale0pts: [1x1 fdopts.sosscaling]
```

Use the `design` function to design the filter. Pass `'butter'` and the specifications given by variable `designSpecs`, as input arguments. Specify the `'matchexactly'` design option to `'passband'`.

```
lpFilter = design(designSpecs, 'butter', matchexactly='passband', SystemObject=true);
```

Visualize the frequency response of the designed filter.

```
fvtool(lpFilter)
```



## Highpass Filter

Construct a highpass filter design specification object using `fdesign.highpass`. Specify the order to be 7 and the 3 dB frequency to be  $0.6\pi$  radians/sample.

```
designSpecs = fdesign.highpass('N,F3dB',7,.6);
```

Determine the available design methods. To design a butterworth filter, pick `butter`.

```
designmethods(designSpecs,SystemObject=true)
```

Design Methods that support System objects for class `fdesign.highpass (N,F3dB)`:

```
butter
maxflat
```

While designing the filter, you can specify additional design options. View a list of the options using the `designoptions` function. This function also shows the default design options the filter uses.

```
designoptions(designSpecs,'butter',SystemObject=true)
```

```
ans = struct with fields:
    FilterStructure: {'df1sos' 'df2sos' 'df1tsos' 'df2tsos' 'cascadeallpass' 'cascad
    SOSScaleNorm: 'ustring'
    SOSScaleOpts: 'fdopts.sosscaling'
    DefaultFilterStructure: 'df2sos'
    DefaultSOSScaleNorm: ''
    DefaultSOSScaleOpts: [1x1 fdopts.sosscaling]
```

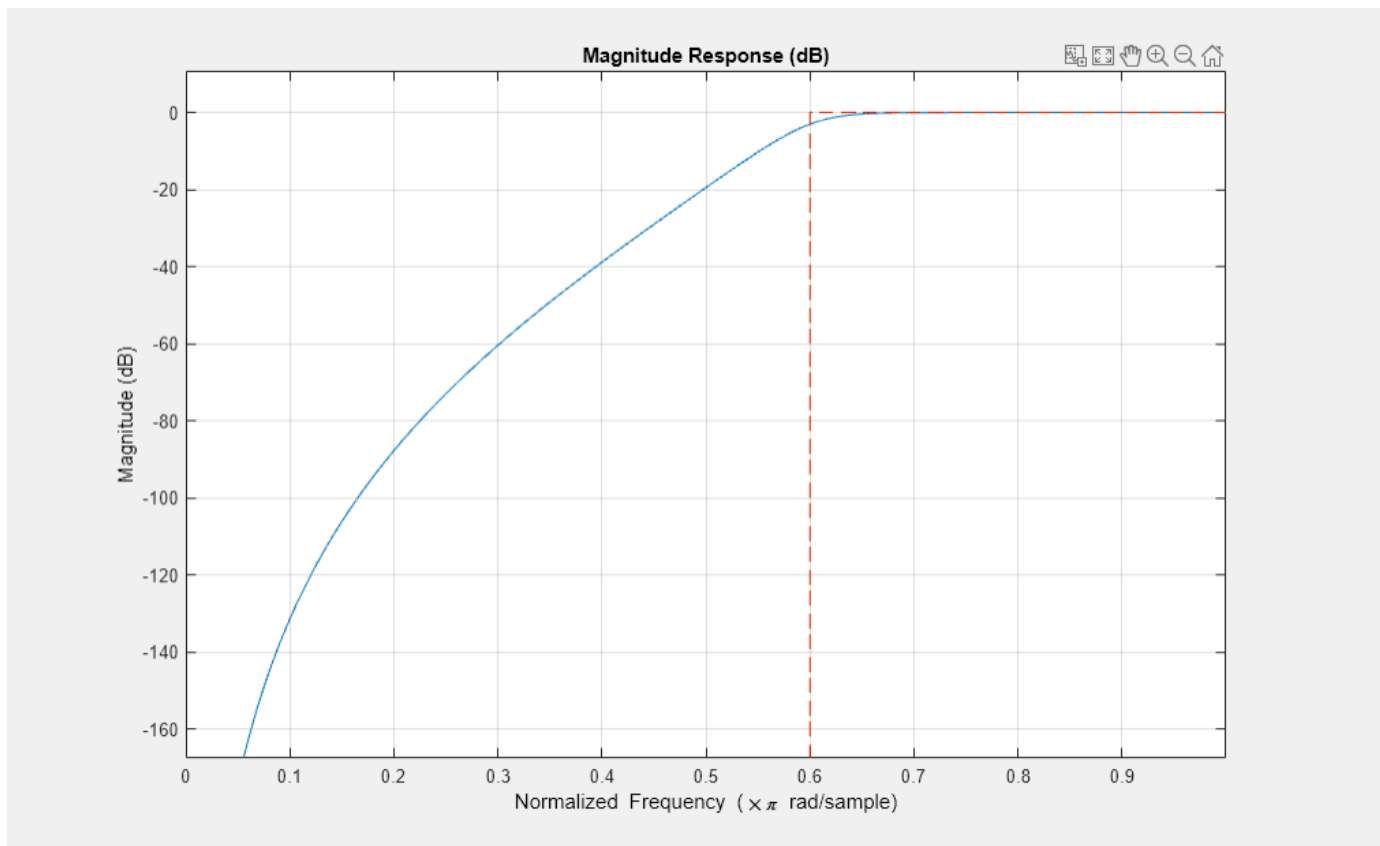
To design the butterworth filter, use the design function and specify `'butter'` as an input. Set `'FilterStructure'` to `'cascadeallpass'`.

```
hpFilter = design(designSpecs,'butter',FilterStructure='cascadeallpass',SystemObject=true);
```

Visualize the highpass frequency response.

```
fvtool(hpFilter)
```





## Highpass Filtering of Sinusoids

Highpass filter a discrete-time signal consisting of two sine waves.

Create a highpass filter specification object. Specify the passband frequency to be  $0.25\pi$  rad/sample and the stopband frequency to be  $0.15\pi$  rad/sample. Specify 1 dB of allowable passband ripple and a stopband attenuation of 60 dB.

```
d = fdesign.highpass('Fst,Fp,Ast,Ap',0.15,0.25,60,1);
```

Query the valid design methods for your filter specification object.

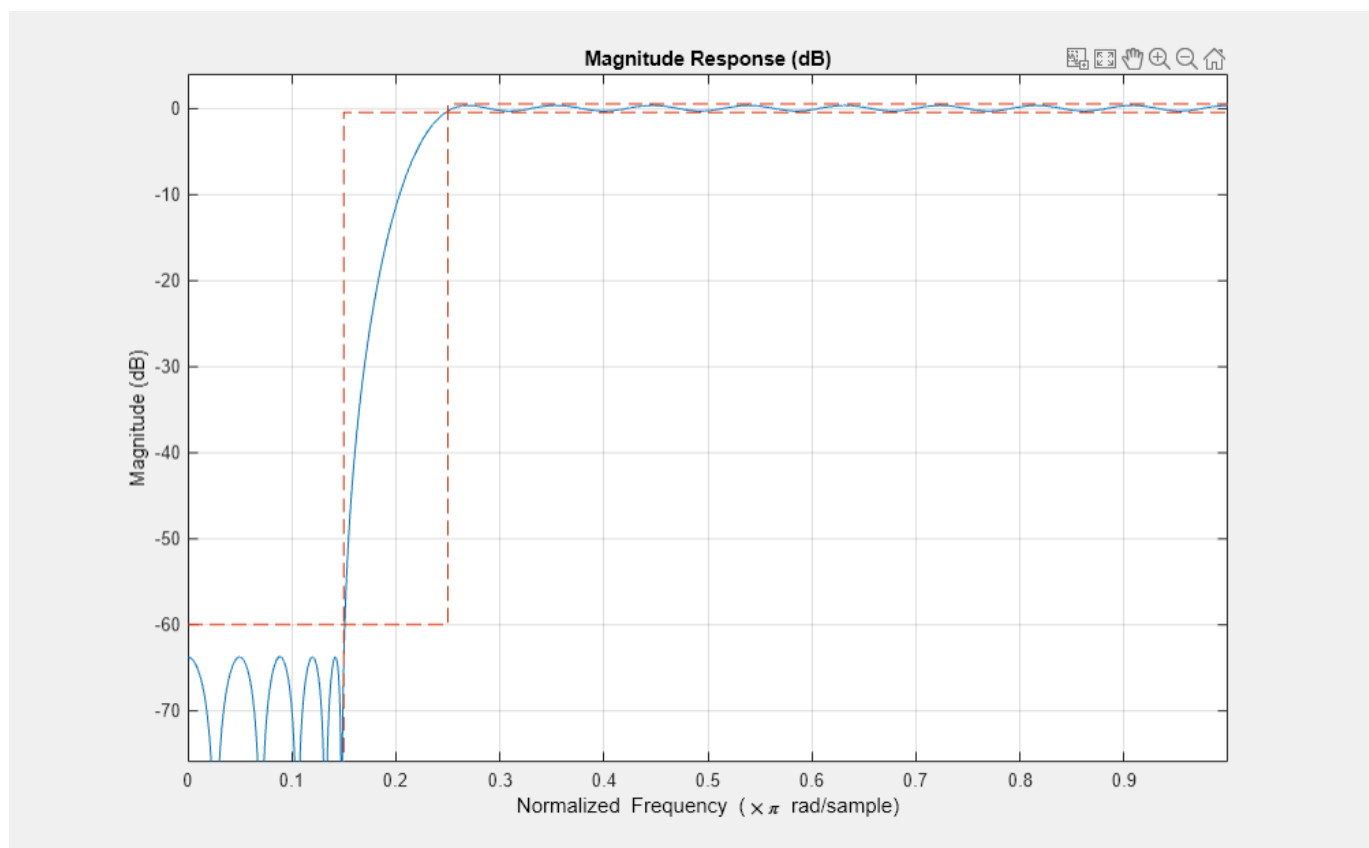
```
designmethods(d)
```

```
Design Methods for class fdesign.highpass (Fst,Fp,Ast,Ap):
```

```
butter
cheby1
cheby2
ellip
equiripple
ifir
kaiserwin
```

Create an FIR equiripple filter and view the filter magnitude response with FVTool.

```
Hd = design(d, 'equiripple');
fvtool(Hd)
```



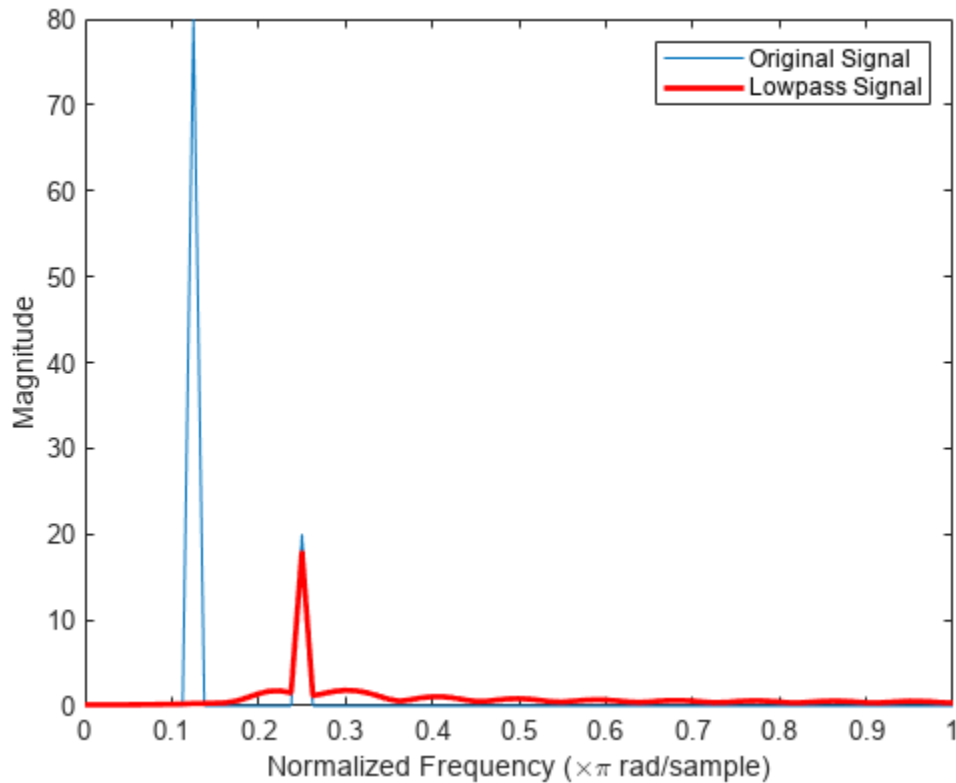
Create a signal consisting of the sum of two discrete-time sinusoids with frequencies of  $\pi/8$  and  $\pi/4$  rad/sample and amplitudes of 1 and 0.25 respectively. Filter the discrete-time signal with the FIR equiripple filter object.

```
n = 0:159;
x = cos(pi/8*n)+0.25*sin(pi/4*n);
y = filter(Hd,x);
```

Plot the original and filtered signals in the frequency domain.

```
freq = 0:(2*pi)/160:pi;
xdft = fft(x);
ydft = fft(y);

plot(freq/pi,abs(xdft(1:length(x)/2+1)))
hold on
plot(freq/pi,abs(ydft(1:length(y)/2+1)),'r','linewidth',2)
hold off
legend('Original Signal','Lowpass Signal','Location','NorthEast')
ylabel('Magnitude')
xlabel('Normalized Frequency (\times\pi rad/sample)')
```



### Window Method Highpass Design

Create a filter of order 10 with a 6-dB frequency of 9.6 kHz and a sample rate of 48 kHz. Look at the available design methods.

```
d=fdesign.highpass('N,Fc',10,9600,48000);
designmethods(d)
```

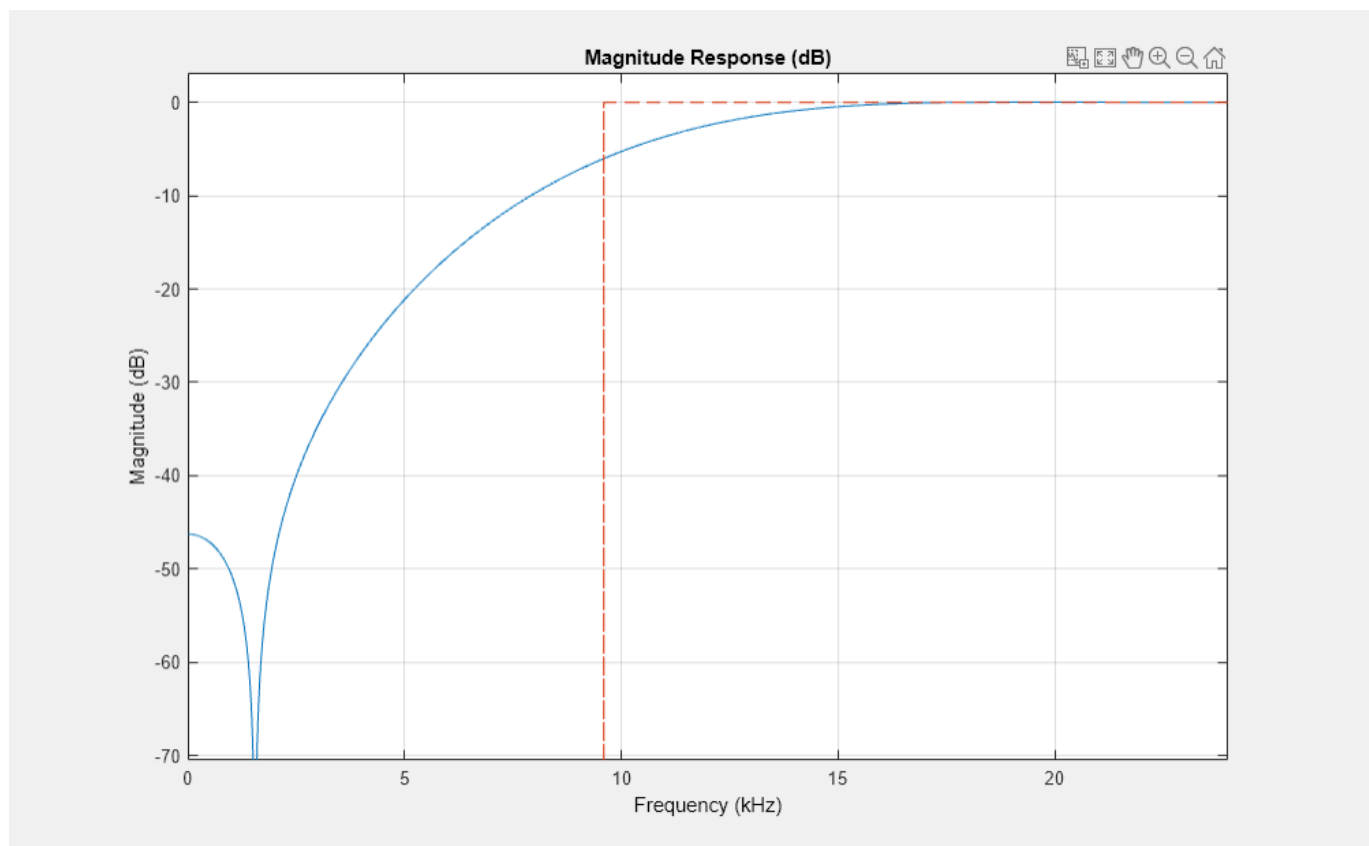
```
Design Methods for class fdesign.highpass (N,Fc):
```

```
window
```

The only available method is the FIR window method. Design the filter and display its magnitude response.

```
Hd = design(d);
```

```
fvtool(Hd)
```



### Stopband Constraints

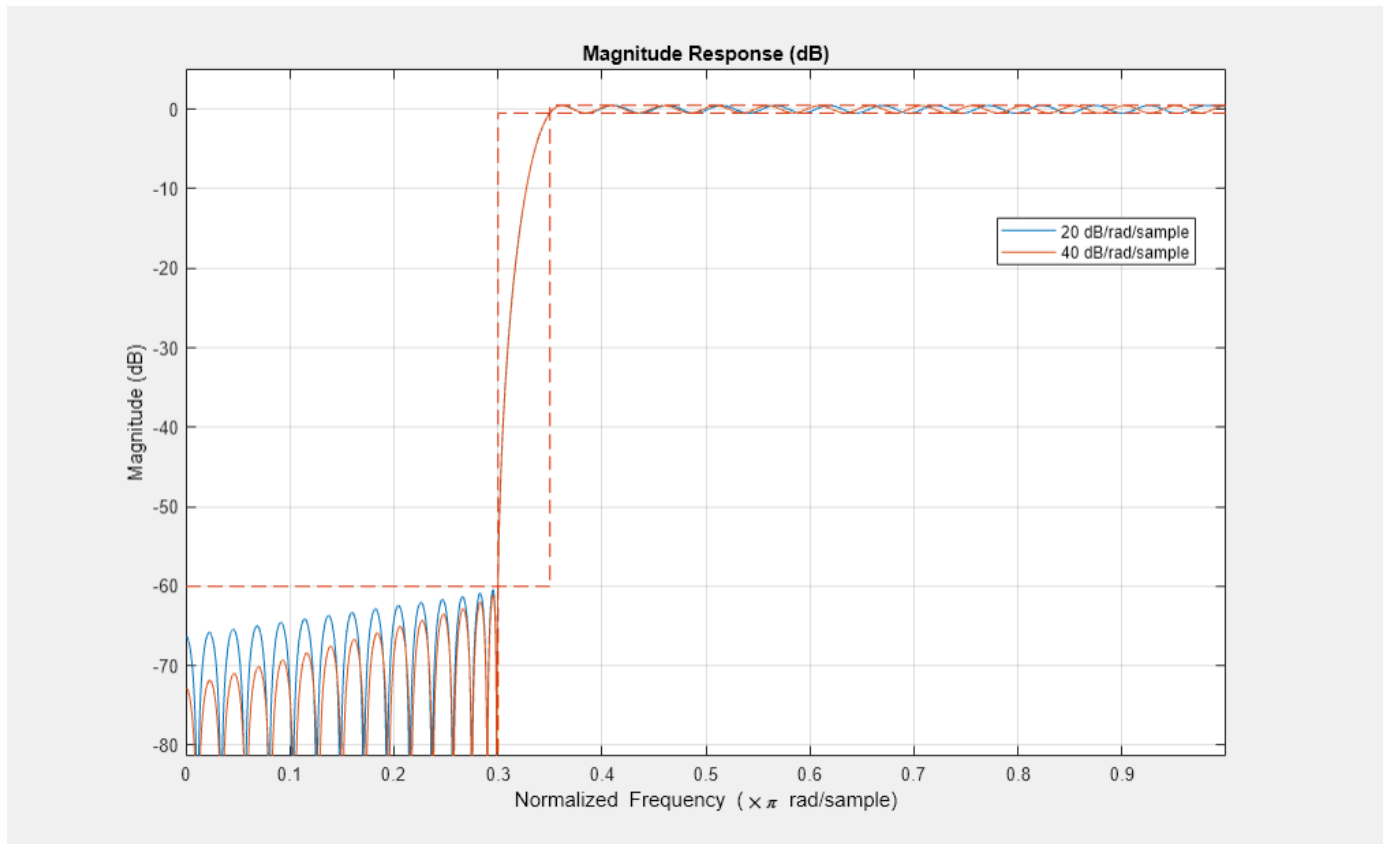
You can specify the shape of the stopband and the rate at which the stopband decays.

Create two FIR equiripple filters with different linear stopband slopes. Specify the passband frequency to be  $0.3\pi$  rad/sample and the stopband frequency to be  $0.35\pi$  rad/sample. Specify 1 dB of allowable passband ripple and a stopband attenuation of 60 dB. Design one filter with a 20 dB/(rad/sample) stopband slope and another filter with a slope of 40 dB/(rad/sample).

```
D = fdesign.highpass('Fst,Fp,Ast,Ap',0.3,0.35,60,1);
Hd1 = design(D,'equiripple','StopBandShape','linear','StopBandDecay',20);
Hd2 = design(D,'equiripple','StopBandShape','linear','StopBandDecay',40);
```

Visualize the magnitude responses of the filters.

```
hfvt = fvtool([Hd1 Hd2]);
legend(hfvt,'20 dB/rad/sample','40 dB/rad/sample')
```



## Version History

Introduced in R2009a

### See Also

[design](#) | [designmethods](#) | [fdesign](#)

## fdesign.hilbert

Hilbert filter specification object

### Syntax

```
d = fdesign.hilbert
d = fdesign.hilbert(specvalue1,specvalue2)
d = fdesign.hilbert(spec)
d = fdesign.hilbert(spec,specvalue1,specvalue2)
d = fdesign.hilbert(...,Fs)
d = fdesign.hilbert(...,MAGUNITS)
```

### Description

`d = fdesign.hilbert` constructs a default Hilbert filter designer `d` with `N`, the filter order, set to 30 and `TW`, the transition width set to  $0.1\pi$  radians/sample.

`d = fdesign.hilbert(specvalue1,specvalue2)` constructs a Hilbert filter designer `d` assuming the default specification `'N,TW'`. You input `specvalue1` and `specvalue2` for `N` and `TW`.

`d = fdesign.hilbert(spec)` initializes the filter designer `Specification` property to `spec`. You provide one of the following as input to replace `spec`. The specification options are not case sensitive.

---

**Note** Specifications marked with an asterisk require the DSP System Toolbox software.

---

- `'N,TW'` default specification option.
- `'TW,Ap'` \*

The filter specifications are defined as follows:

- `Ap` — amount of ripple allowed in the pass band in decibels (the default units). Also called `Apass`.
- `N` — filter order.
- `TW` — width of the transition region between the passband and the stopband.

By default, `fdesign.hilbert` assumes that all frequency specifications are provided in normalized frequency units. Also, decibels is the default for all magnitude specifications.

Different specifications may have different design methods available. Use `designmethods(d)` to get a list of the design methods available for a given specification.

`d = fdesign.hilbert(spec,specvalue1,specvalue2)` initializes the filter designer specifications in `spec` with `specvalue1`, `specvalue2`, and so on. To get a description of the specifications `specvalue1` and `specvalue2`, enter

```
get(d, 'description')
```

at the Command prompt.

`d = fdesign.hilbert(...,Fs)` adds the argument `Fs`, specified in Hz to define the sampling frequency. In this case, all frequencies in the specifications are in Hz as well.

`d = fdesign.hilbert(...,MAGUNITS)` specifies the units for any magnitude specification you provide in the input arguments. `MAGUNITS` can be one of

- 'linear' — specify the magnitude in linear units
- 'dB' — specify the magnitude in dB (decibels)
- 'squared' — specify the magnitude in power units

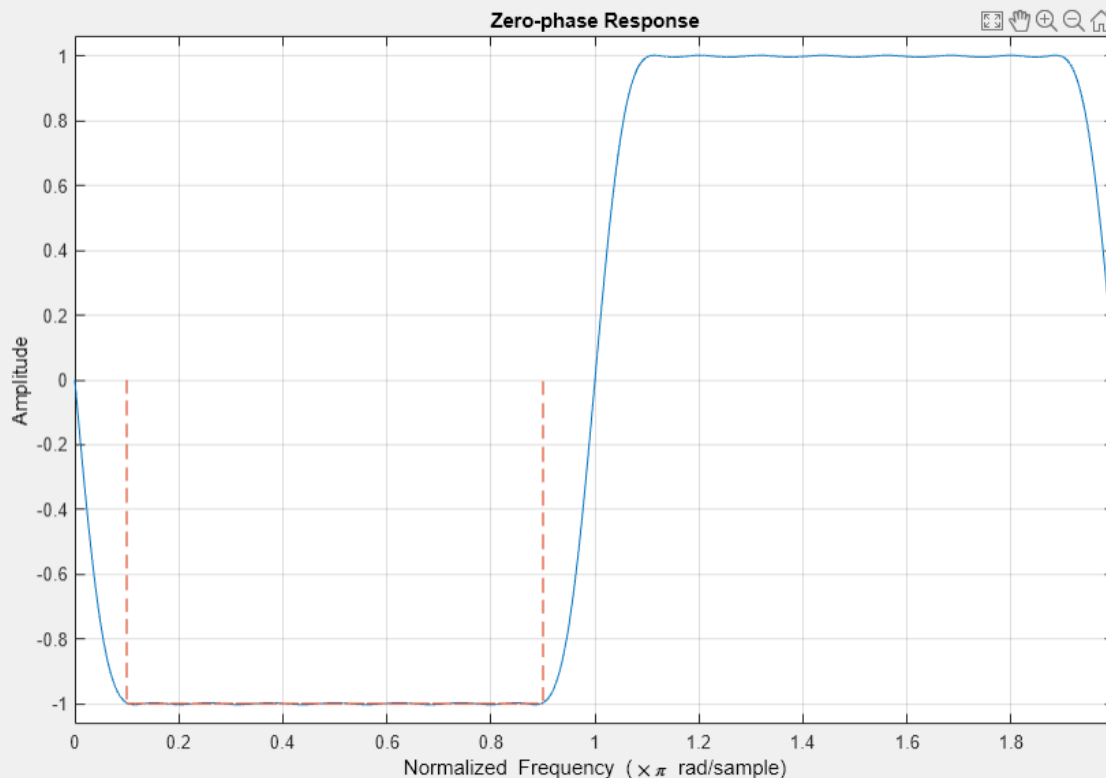
When you omit the `MAGUNITS` argument, `fdesign` assumes that all magnitudes are in decibels. Note that `fdesign` stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

## Examples

### Hilbert Transformers

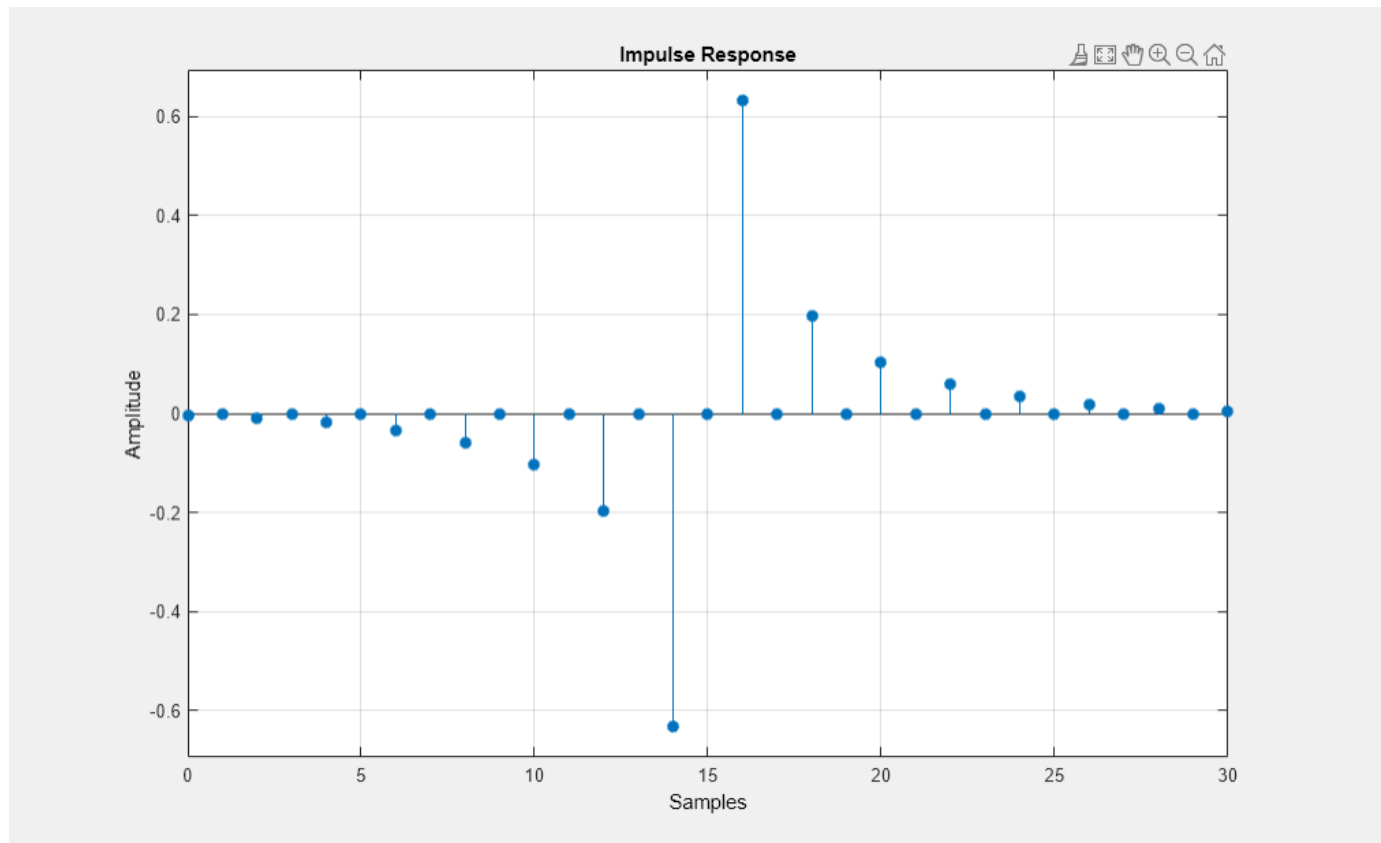
Design a Hilbert transformer of order 30 with a transition width of  $0.2\pi$  rad/sample. Use least-squares minimization to obtain an equiripple linear-phase FIR filter. Plot the zero-phase response in the interval  $[-\pi,\pi)$ .

```
d = fdesign.hilbert('N,TW',30,0.2);
Hd = design(d,'equiripple','SystemObject',true);
zerophase(Hd,'whole')
```



The impulse response of this even-order type-3 filter is antisymmetric.

```
impz(Hd)
```



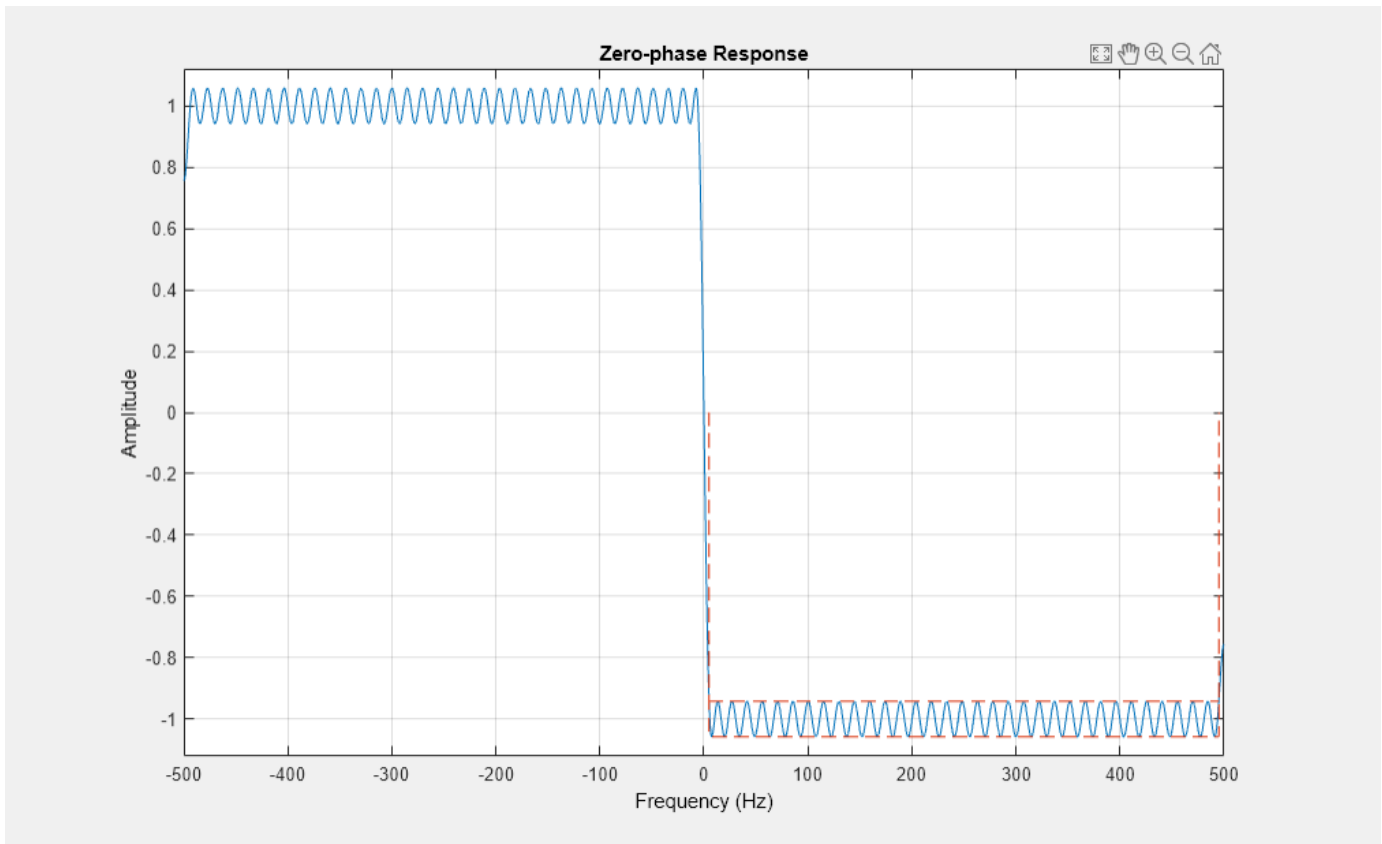
```
ftype = firtype(Hd)
```

```
ftype = 3
```

Design a minimum-order Hilbert transformer that has a sample rate of 1 kHz. Specify the width of the transition region as 10 Hz and the passband ripple as 1 dB. Display the zero-phase response of the filter.

```
fs = 1e3;
d = fdesign.hilbert('TW,Ap',10,1,fs);
hd = design(d,'equiripple','SystemObject',true);
zerophase(hd,-fs/2:0.1:fs/2,fs)
```





## Version History

Introduced in R2009a

### See Also

[design](#) | [fdesign](#) | [setspecs](#)

## fdesign.interpolator

Interpolator filter specification

---

### Note

**Note** The 'Raised Cosine' and 'Square Root Raised Cosine' response methods in the `fdesign.interpolator` object will be removed in a future release. Use `rcosdesign` and `comm.RaisedCosineTransmitFilter` instead.

---

**Note** You can no longer design an `mfilt.firinterp` object using the `fdesign.interpolator` and the `design` functions. Use the `design` function with the `SystemObject=true` flag to design an FIR interpolator System object™.

---

**Note** Support for multistage filter design using the `fdesign.interpolator` object will be removed in a future release. Use the `designMultistageInterpolator` function instead.

For more information, see “Compatibility Considerations”.

---

### Syntax

```
D = fdesign.interpolator(L)
D = fdesign.interpolator(L,RESPONSE)
D = fdesign.interpolator(L,CICRESPONSE,D)
D = fdesign.interpolator(L,RESPONSE,spec)
D = fdesign.interpolator(...,spec,specvalue1,specvalue2,...)
D = fdesign.interpolator(...,Fs)
d = fdesign.interpolator(...,MAGUNITS)
```

### Description

`D = fdesign.interpolator(L)` constructs an interpolator filter specification object `D` with the `InterpolationFactor` property equal to the positive integer `L` and the `Response` property set to 'Nyquist'. The default values for the transition width and stopband attenuation in the Nyquist design are  $0.1\pi$  radians/sample and 80 dB. If `L` is unspecified, `L` defaults to 2.

`D = fdesign.interpolator(L,RESPONSE)` constructs a interpolator specification object with the interpolation factor `L` and the 'Response' property set to one of the supported types.

`D = fdesign.interpolator(L,CICRESPONSE,D)` constructs a CIC or CIC compensator interpolator specification object with the interpolation factor, `L`, and 'Response' property equal to 'CIC' or 'CICCOMP'. `D` is the differential delay. The differential delay, `D`, must precede any specification option.

`D = fdesign.interpolator(L,RESPONSE,spec)` constructs object `D` and sets its `Specification` property to `spec`. Entries in the `spec` represent various filter response features,

such as the filter order, that govern the filter design. Valid entries for `spec` depend on the design type of the specifications object.

When you add the `spec` input argument, you must also add the `RESPONSE` input argument.

Because you are designing multirate filters, the specification options available are not the same as the specifications for designing single-rate filters with design methods such as `fdesign.lowpass`. The options are not case sensitive.

The interpolation factor `L` is not in the specification options. The different filter responses support different specifications. The following table lists the supported response types and specification options.

| Design Method                   | Valid Specification Options  |
|---------------------------------|--|
| 'Arbitrary Magnitude'           | See <code>fdesign.arbmag</code> for a description of the specification entries. <ul style="list-style-type: none"> <li>'N,F,A' (default option)</li> <li>'N,B,F,A'</li> </ul>  |
| 'Arbitrary Magnitude and Phase' | See <code>fdesign.arbmagnphase</code> for a description of the specification entries. <ul style="list-style-type: none"> <li>'N,F,H' (default option)</li> <li>'N,B,F,H'</li> </ul>  |
| 'Bandpass'                      | See <code>fdesign.bandpass</code> for a description of the specification entries. <ul style="list-style-type: none"> <li>'Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2' (default option)</li> <li>'N,Fc1,Fc2'</li> <li>'N,Fst1,Fp1,Fp2,Fst2'</li> </ul>  |
| 'Bandstop'                      | See <code>fdesign.bandstop</code> for a description of the specification entries. <ul style="list-style-type: none"> <li>'N,Fc1,Fc2'</li> <li>'N,Fp1,Fst1,Fst2,Fp2'</li> <li>'Fp1,Fst1,Fst2,Fp2,Ap1,Ast,Ap2' (default option)</li> </ul>   |
| 'CIC'                           | 'Fp,Ast' — Only valid specification. <code>Fp</code> is the passband frequency and <code>Ast</code> is the stopband attenuation in decibels.<br><br>To specify a CIC interpolator, include the differential delay after 'CIC' and before the filter specification: 'Fp,Ast'. For example:<br><code>d = fdesign.interpolator(2,'cic',4,'Fp,Ast',0.4,40);</code> |

| Design Method     | Valid Specification Options  |
|-------------------|--|
| 'CIC Compensator' | <p>See <code>fdesign.ciccomp</code> for a description of the specification entries.</p> <ul style="list-style-type: none"> <li>• 'Fp,Fst,Ap,Ast' (default option)</li> <li>• 'N,Fc,Ap,Ast'</li> <li>• 'N,Fp,Ap,Ast'</li> <li>• 'N,Fp,Fst'</li> <li>• 'N,Fst,Ap,Ast'</li> </ul> <p>To specify a CIC compensator interpolator, include the differential delay after 'CICCOMP' and before the filter specification. For example:<br/> <code>d = fdesign.interpolator(2,'ciccomp',4);</code></p> |
| 'Differentiator'  | 'N' — filter order   |
| 'Gaussian'        | <p>'Nsym,BT — Nsym is the filter order in symbols and BT is the bandwidth-symbol time product.</p> <p>The specification must be preceded by an integer-valued <code>SamplesPerSymbol</code>.</p>   |
| 'Halfband'        | <p>See <code>fdesign.halfband</code> for a description of the specification entries.</p> <ul style="list-style-type: none"> <li>• 'TW,Ast' (default option)</li> <li>• 'N,TW'</li> <li>• 'N'</li> <li>• 'N,Ast'</li> </ul> <p>If you use the quasi-linear IIR design method, <code>iirlinphase</code>, with a halfband specification, the interpolation factor must be 2.</p>  |
| 'Highpass'        | <p>See <code>fdesign.highpass</code> for a description of the specification entries.</p> <ul style="list-style-type: none"> <li>• 'Fst,Fp,Ast,Ap' (default option)</li> <li>• 'N,F3db'</li> <li>• 'N,Fc'</li> <li>• 'N,Fc,Ast,Ap'</li> <li>• 'N,Fp,Ast,Ap'</li> <li>• 'N,Fst,Ast,Ap'</li> <li>• 'N,Fst,Fp'</li> <li>• 'N,Fst,Ast,Ap'</li> <li>• 'N,Fst,Fp,Ast'</li> </ul>  |
| 'Hilbert'         | <p>See <code>fdesign.hilbert</code> for a description of the specification entries.</p> <ul style="list-style-type: none"> <li>• 'N,TW' (default option)</li> <li>• 'TW,Ap'</li> </ul>   |

| Design Method           | Valid Specification Options  |
|-------------------------|--|
| 'Inverse-sinc Lowpass'  | See <code>fdesign.isinclp</code> for a description of the specification entries. <ul style="list-style-type: none"> <li>'Fp,Fst,Ap,Ast' (default option)</li> <li>'N,Fc,Ap,Ast'</li> <li>'N,Fp,Fst'</li> <li>'N,Fst,Ap,Ast'</li> </ul>   |
| 'Inverse-sinc Highpass' | See <code>fdesign.isinchp</code> for a description of the specification entries. <ul style="list-style-type: none"> <li>'Fst,Fp,Ast,Ap' (default option)</li> <li>'N,Fc,Ast,Ap'</li> <li>'N,Fst,Fp'</li> <li>'N,Fst,Ast,Ap'</li> </ul>   |
| 'Lowpass'               | See <code>fdesign.lowpass</code> for a description of the specification entries. <ul style="list-style-type: none"> <li>'Fp,Fst,Ap,Ast' (default option)</li> <li>'N,F3dB'</li> <li>'N,Fc'</li> <li>'N,Fc,Ap,Ast'</li> <li>'N,Fp,Ap,Ast'</li> <li>'N,Fp,Fst'</li> <li>'N,Fp,Fst,Ap'</li> <li>'N,Fp,Fst,Ast'</li> <li>'N,Fst,Ap,Ast'</li> </ul>   |
| 'Nyquist'               | See <code>fdesign.nyquist</code> for a description of the specification entries. For all Nyquist specifications, you must specify the <i>L</i> th band. This typically corresponds to the interpolation factor so that the nonzero samples of the upsampler output are preserved. <ul style="list-style-type: none"> <li>'TW,Ast' (default option)</li> <li>'N'</li> <li>'N,Ast'</li> <li>'N,Ast'</li> </ul> |

`D = fdesign.interpolator(...,spec,specvalue1,specvalue2,...)` constructs an object `D` and sets its specifications at construction time.

`D = fdesign.interpolator(...,Fs)` adds the argument `Fs`, specified in Hz, to define the sampling frequency to use. In this case, all frequencies in the specifications are in Hz as well.

`d = fdesign.interpolator(...,MAGUNITS)` specifies the units for any magnitude specification you provide in the input arguments. `MAGUNITS` can be one of

- 'linear' — specify the magnitude in linear units.
- 'dB' — specify the magnitude in dB (decibels).

- 'squared' — specify the magnitude in power units.

When you omit the MAGUNITS argument, `fdesign` assumes that all magnitudes are in decibels. Note that `fdesign` stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

## Examples

### Create an Interpolator Using `fdesign` Object

These examples show how to construct interpolating filter specification objects.

First, create a default specifications object without using input arguments except for the interpolation factor `l`.

```
l = 2;
d = fdesign.interpolator(l); %#ok
```

Now create an object by passing a specification option 'fst1,fp1,fp2,fst2,ast1,ap,ast2' and a design - the resulting object uses default values for all of the filter specifications. You must provide the design input argument when you include a specification.

```
d = fdesign.interpolator(8, 'bandpass', 'fst1,fp1,fp2,fst2,ast1,ap,ast2'); %#ok
```

Create another interpolating filter object, passing the specification values to the object rather than accepting the default values for, in this case, `fp`, `fst`, `ap`, `ast`.

```
d = fdesign.interpolator(3, 'lowpass', .45,0.55,.1,60); %#ok
```

Now pass the filter specifications that correspond to the specifications - `n`, `fc`, `ap`, `ast`.

```
d = fdesign.interpolator(3, 'ciccomp', 1,2, 'n,fc,ap,ast', ...
    20,0.45,.05,50);
```

With the specifications object in your workspace, design an interpolator using the equiripple design method.

```
hm = design(d, 'equiripple', SystemObject=true); %#ok
```

Pass a new specification type for the filter, specifying the filter order.

```
d = fdesign.interpolator(5, 'CIC', 1, 'fp,ast', 0.05,55);
```

With the specifications object in your workspace, design an interpolator using the multisection design method.

```
hm = design(d, 'multisection', SystemObject=true); %#ok
```

In this example, you specify a sampling frequency as the right most input argument. Here, it is set to 1000 Hz.

```
d = fdesign.interpolator(8, 'bandpass', 'fst1,fp1,fp2,fst2,ast1,ap,ast2', ...
    0.25,0.35,.55,.65,50,.05,1e3); %#ok
```

In this, the last example, use the linear option for the filter specification object and specify the stopband ripple attenuation in linear form.

```
d = fdesign.interpolator(4,'lowpass','n,fst,ap,ast',15,0.55,.05,0.001,...
    'linear'); %#ok
```

Now design a CIC interpolator for a signal sampled at 19200 Hz. Specify the differential delay of 2 and set the attenuation of information beyond 50 Hz to be at least 80 dB.

*% The filter object sampling frequency is (l x fs) where fs is the sampling frequency of the input signal.*

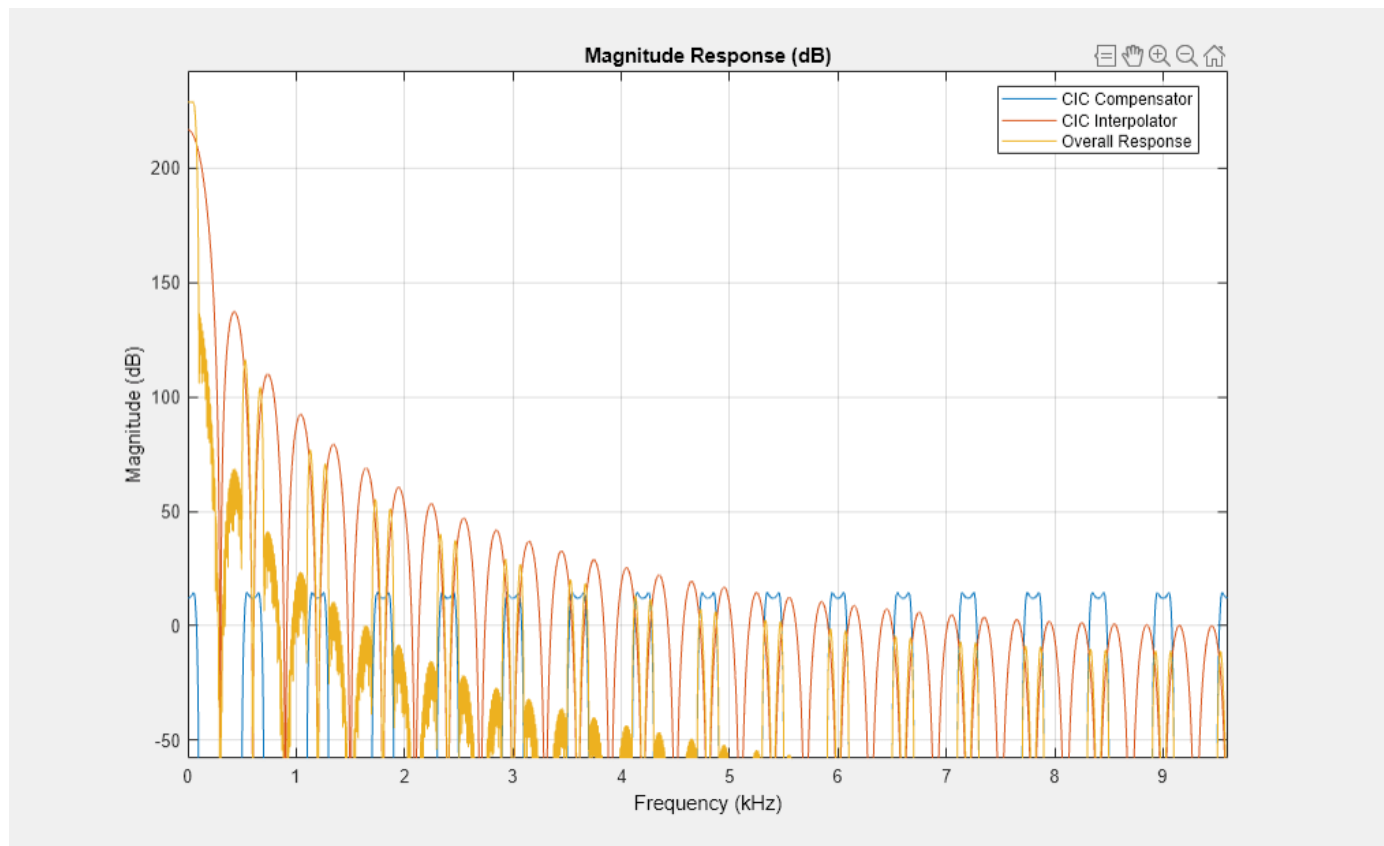
```
dd = 2;      % Differential delay.
fp = 50;     % Passband of interest.
ast = 80;    % Minimum attenuation of alias components in passband.
fs = 600;    % Sampling frequency for input signal.
l = 32;      % Interpolation factor.
d = fdesign.interpolator(l,'cic',dd,'fp,ast',fp,ast,l*fs);
hm = design(d,SystemObject=true); %Use the default design method.
```

This next example results in a minimum-order CIC compensator that interpolates by 4 and compensates for the droop in the passband for the CIC filter hm from the previous example.

```
nsecs = hm.NumSections;
d = fdesign.interpolator(4,'ciccomp',dd,nsecs,...
    50,100,0.1,80,fs);
hmc = design(d,'equiripple',SystemObject=true);
```

hmc is designed to compensate for hm. To see the effect of the compensating CIC filter, use fvtool to analyze both filters individually and include the compound filter response by cascading hm and hmc.

```
hfvt = fvtool(hmc,hm,cascade(hmc,hm),'fs',[fs,l*fs,l*fs],ShowReference='off');
legend(hfvt,'CIC Compensator','CIC Interpolator',...
    'Overall Response');
```



fvtool returns with this plot.

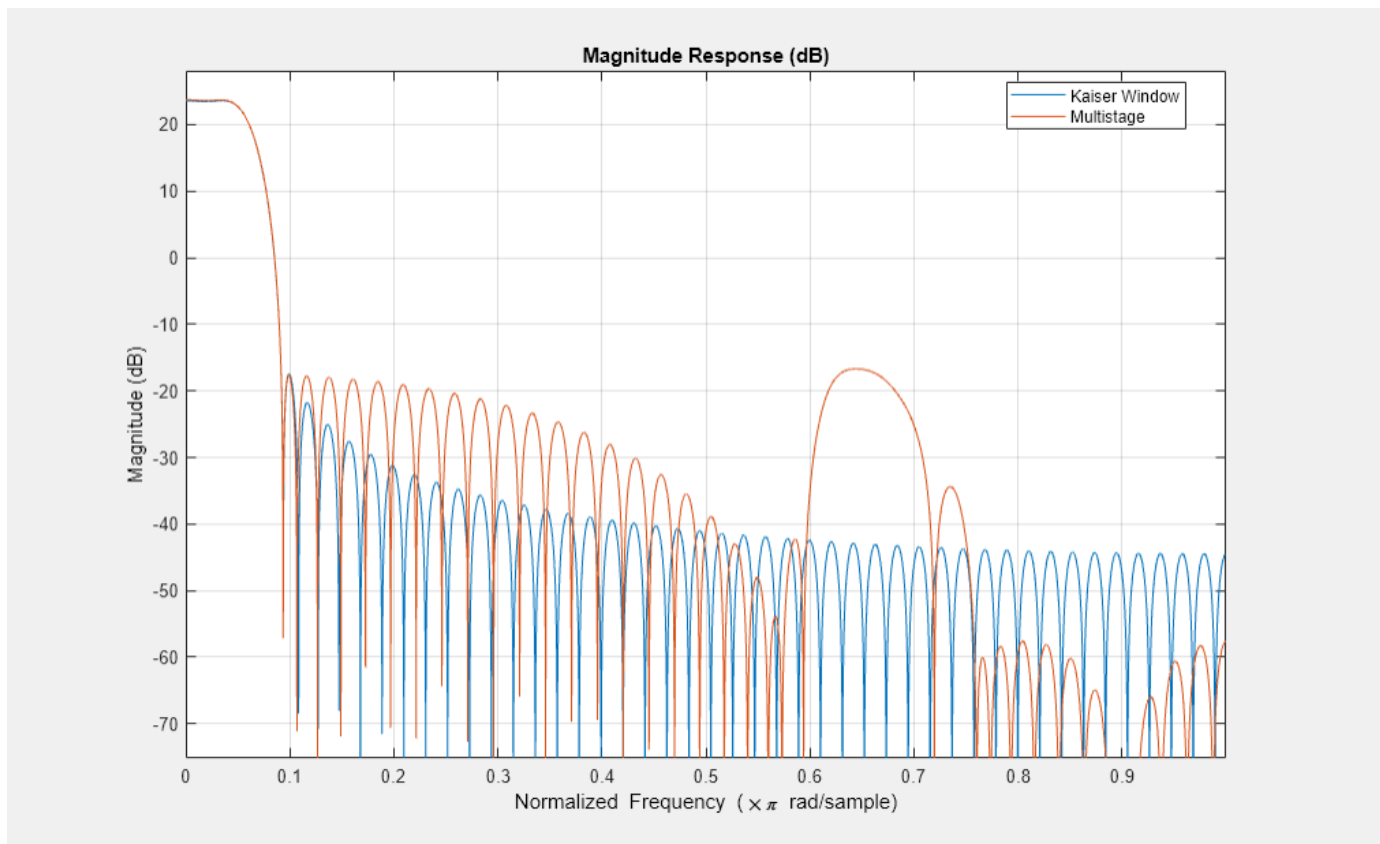
For the third example, use `fdesign.interpolator` to design a minimum-order Nyquist interpolator that uses a Kaiser window. For comparison, design a multistage interpolator as well and compare the responses.

```

l = 15; % Set the interpolation factor and the Nyquist band.
tw = 0.05; % Specify the normalized transition width.
ast = 40; % Set the minimum stopband attenuation in dB.
Fs = 2; % For normalized frequency units
d = fdesign.interpolator(l,'nyquist',l,tw,ast);
hm = design(d,'kaiserwin',SystemObject=true);
hm2 = designMultistageInterpolator(l,Fs,tw,ast); % Design the multistage interpolator.
hfvt = fvtool(hm,hm2);
legend(hfvt,'Kaiser Window','Multistage')

```





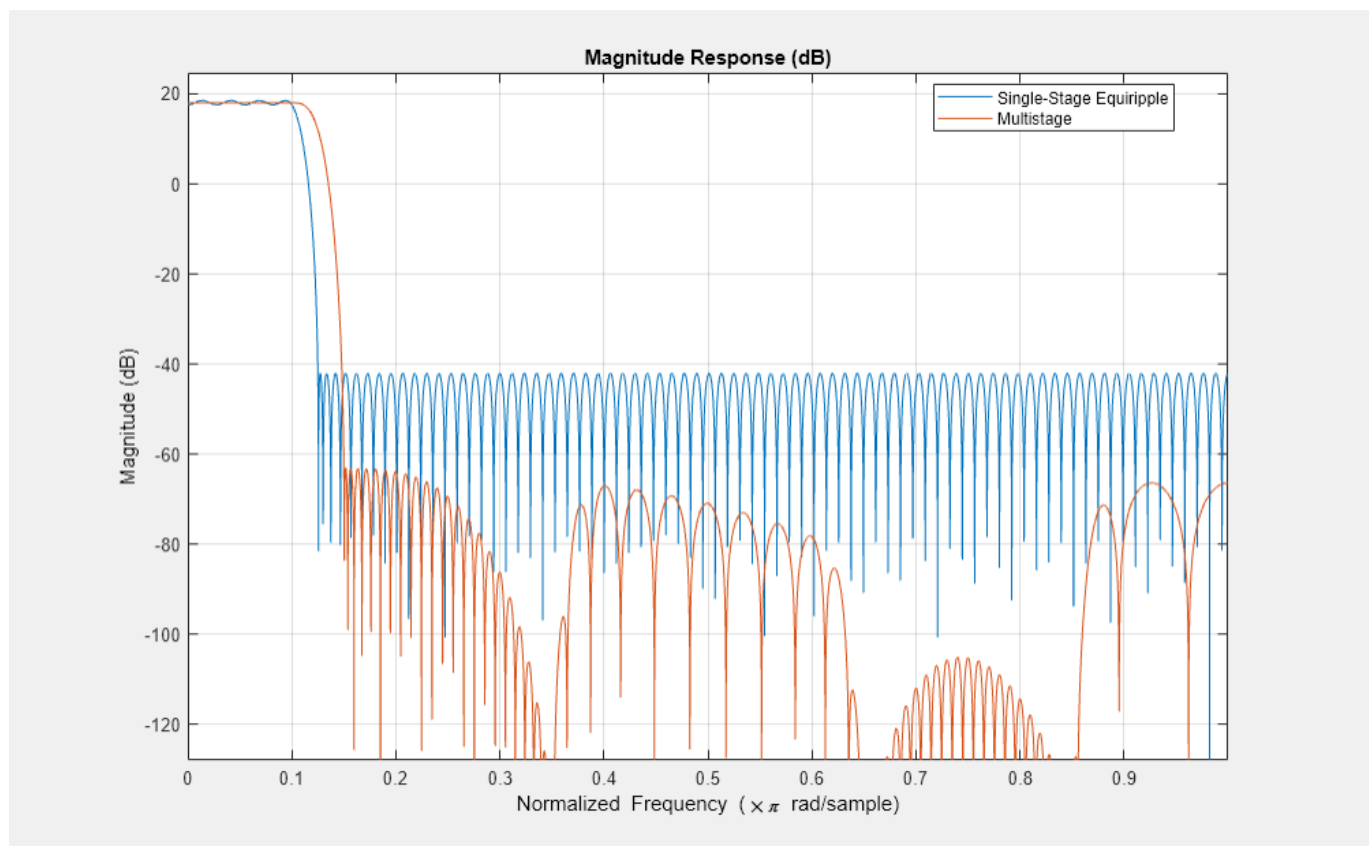
fvtool shows both responses.

Design a lowpass interpolator for an interpolation factor of 8. Compare the single-stage equiripple design to a multistage design with the same interpolation factor.

```
l = 8; % Interpolation factor.
d = fdesign.interpolator(l,'lowpass');
hm1 = design(d,'equiripple',SystemObject=true);
% Use halfband filters whenever possible.
hm2 = designMultistageInterpolator(l)

hm2 =
    dsp.FilterCascade with properties:
        Stage1: [1x1 dsp.FIRInterpolator]
        Stage2: [1x1 dsp.FIRInterpolator]
        Stage3: [1x1 dsp.FIRInterpolator]
        CloneStages: false

hfvtlp = fvtool(hm1,hm2);
legend(hfvtlp,'Single-Stage Equiripple','Multistage')
```



## Version History

### Introduced in R2011a

#### **R2022a: 'Raised Cosine' and 'Square Root Raised Cosine' response methods will be removed**

*Warns starting in R2022a*

The 'Raised Cosine' and 'Square Root Raised Cosine' response methods in the `fdesign.interpolator` object will be removed in a future release. Use `rcosdesign` and `comm.RaisedCosineTransmitFilter` instead.

#### **R2022b: Support for designing `mfilt.firinterp` has been removed**

*Errors starting in R2022b*

You can no longer design an `mfilt.firinterp` object using the `fdesign.interpolator` and the design functions.

To design an FIR interpolator using the `fdesign.interpolator` filter specification object, use the design function with the `SystemObject=true` flag. Setting this flag to true ensures that the output object is a filter System object instead of an `mfilt.firinterp` object.

```
fdesignObj = fdesign.interpolator;
FIRInterpObj = design(fdesignObj, SystemObject=true)
```

```
FIRInterpObj =
    dsp.FIRInterpolator with properties:
        InterpolationFactor: 2
        NumeratorSource: 'Property'
        Numerator: [-3.0325e-05 0 8.3413e-05 0 ... ]

    Show all properties
```

## R2022b: Support for multistage design in fdesign.interpolator will be removed

Warns starting in R2022b

Support for multistage filter design using the `fdesign.interpolator` object will be removed in a future release. Use the `designMultistageInterpolator` function instead.

### Update Code

This table shows how to update your code.

| Discouraged Usage   | Recommended Replacement   |
|---|---|
| Using the <code>design</code> function and passing the 'multistage' flag as an input.<br><br><pre>L = 12; % Interpolation factor TW = 2205; % Transition width Astop = 80; % Stopband attenuation Fs = 44100; % Output sample rate filtSpec = fdesign.interpolator(L, 'nyquist', L, TW, Astop, Fs); filtObj = design(filtSpec, 'multistage', SystemObject=true)</pre> | When you set the cost method to 'design', the function designs an optimal number of stages so that the overall cost of implementation is low.<br><br><pre>filtObj = designMultistageInterpolator(L, Fs, TW, Astop, CostMethod='design')</pre> |
| Using the <code>multistage</code> function directly.<br><br><pre>filtObj = multistage(filtSpec, SystemObject=true)</pre>  |   |

### See Also

`fdesign` | `fdesign.arbmagnphase` | `fdesign.decimator` | `fdesign.rsrc` | `setspecs`

## fdesign.isinchnp

Inverse sinc highpass filter specification

### Syntax

```
D = fdesign.isinchnp
D = fdesign.isinchnp(SPEC)
D = fdesign.isinchnp(SPEC,specvalue1,specvalue2,...)
D = fdesign.isinchnp(specvalue1,specvalue2,specvalue3,specvalue4)
D = fdesign.isinchnp(...,Fs)
D = fdesign.isinchnp(...,MAGUNITS)
```

### Description

`D = fdesign.isinchnp` constructs an inverse sinc highpass filter specification object `D`, applying default values for the default specification `'Fst,Fp,Ast,Ap'`.

`D = fdesign.isinchnp(SPEC)` constructs object `D` and sets the `Specification` property to `SPEC`. Entries in the `SPEC` represent various filter response features, such as the filter order, that govern the filter design. Valid entries for `SPEC` are shown below. The entries are not case sensitive.

- `'Fst,Fp,Ast,Ap'` (default spec)
- `'N,Fc,Ast,Ap'`
- `'N,Fst,Fp'`
- `'N,Fp,Ast,Ap'`
- `'N,Fst,Ast,Ap'`

The filter specifications are defined as follows:

- `Ast` — attenuation in the stopband in decibels (the default units). Also called `Astop`.
- `Ap` — amount of ripple allowed in the passband in decibels (the default units). Also called `Apass`.
- `Fp` — frequency at the start of the passband. Specified in normalized frequency units. Also called `Fpass`.
- `Fst` — frequency at the end of the stopband. Specified in normalized frequency units. Also called `Fstop`.
- `N` — filter order.

The filter design methods that apply to an inverse sinc highpass filter specification object change depending on the value of the `Specification` property. Use `designmethods` to determine which design method applies to a specific `Specification`.

Use `designopts` to see the available design options for a specific design method. Enter `help(D,METHOD)` at the MATLAB command line to obtain detailed information on the design options for a given design method, `METHOD`.

`D = fdesign.isinchnp(SPEC,specvalue1,specvalue2,...)` constructs an object `D` and sets the specifications at construction time.

`D = fdesign.isinchnp(specvalue1,specvalue2,specvalue3,specvalue4)` constructs an object `D` assuming the default `Specification` property `'Fst,Fp,Ast,Ap'`, using the values you provide in `specvalue1,specvalue2, specvalue3, and specvalue4`.

`D = fdesign.isinchnp(...,Fs)` adds the argument `Fs`, specified in Hz to define the sampling frequency to use. In this case, all frequencies in the specifications are in Hz as well.

`D = fdesign.isinchnp(...,MAGUNITS)` specifies the units for any magnitude specification you provide in the input arguments. `MAGUNITS` can be one of

- `'linear'` — specify the magnitude in linear units
- `'dB'` — specify the magnitude in dB (decibels)
- `'squared'` — specify the magnitude in power units

When you omit the `MAGUNITS` argument, `fdesign` assumes that all magnitudes are in decibels. Note that `fdesign` stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

The design method of `fdesign.isinchnp` implements a filter with a passband magnitude response equal to:

$$H(\omega) = \text{sinc}(C(1 - \omega))^{-P}$$

You can control the values of the sinc frequency factor,  $C$ , and the sinc power,  $P$ , using the `'SincFrequencyFactor'` and `'SincPower'` options in the design method. `'SincFrequencyFactor'` and `'SincPower'` default to 0.5 and 1 respectively.

## Examples

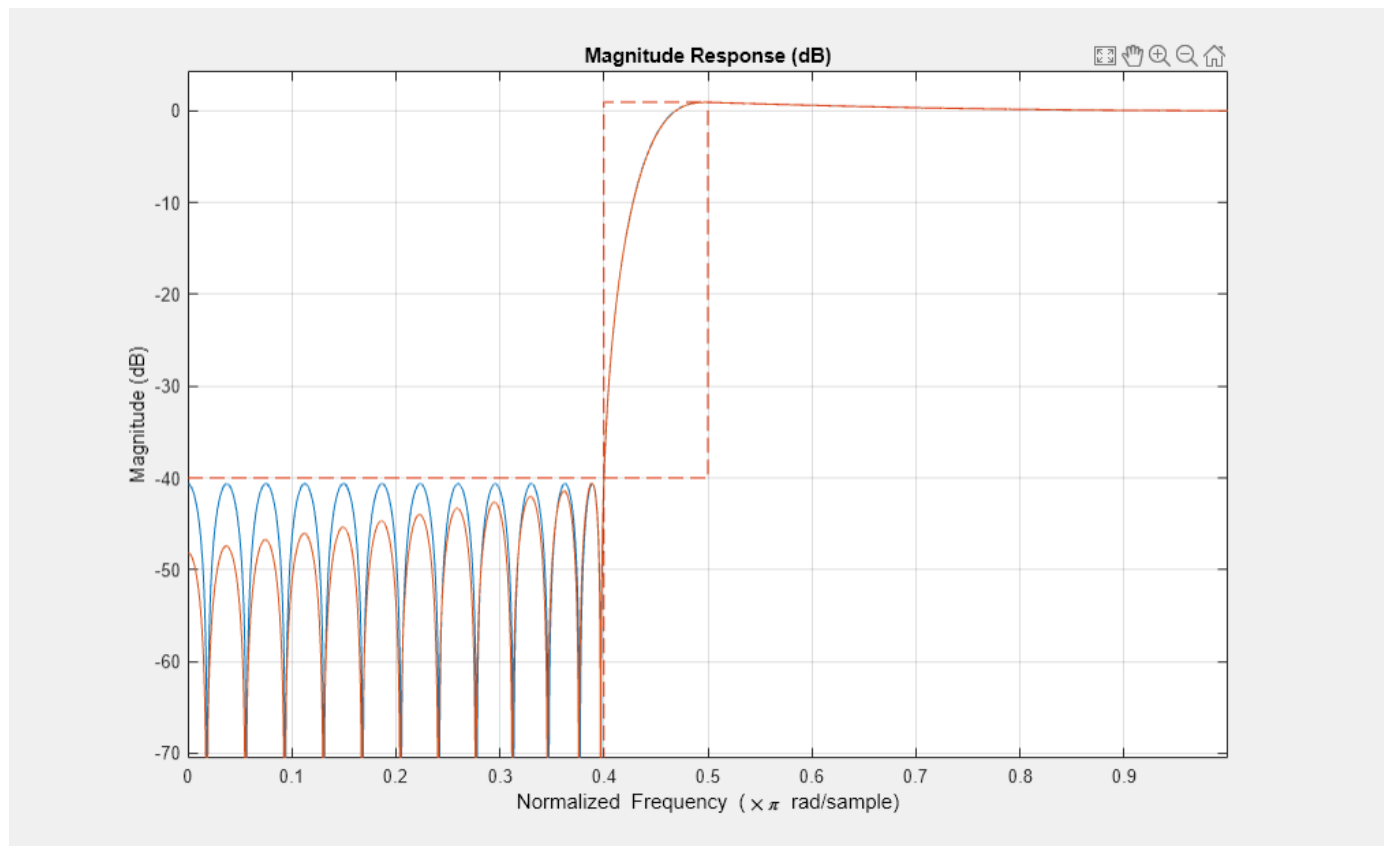
### Construct an Inverse Sinc Highpass Filter Using `fdesign` Object

Design a minimum order inverse sinc highpass filter and shape the stopband to have a slope of 20 dB/radian/sample.

```
d = fdesign.isinchnp('Fst,Fp,Ast,Ap',.4,.5,40,0.01);
Hd = design(d,'SystemObject',true);
```

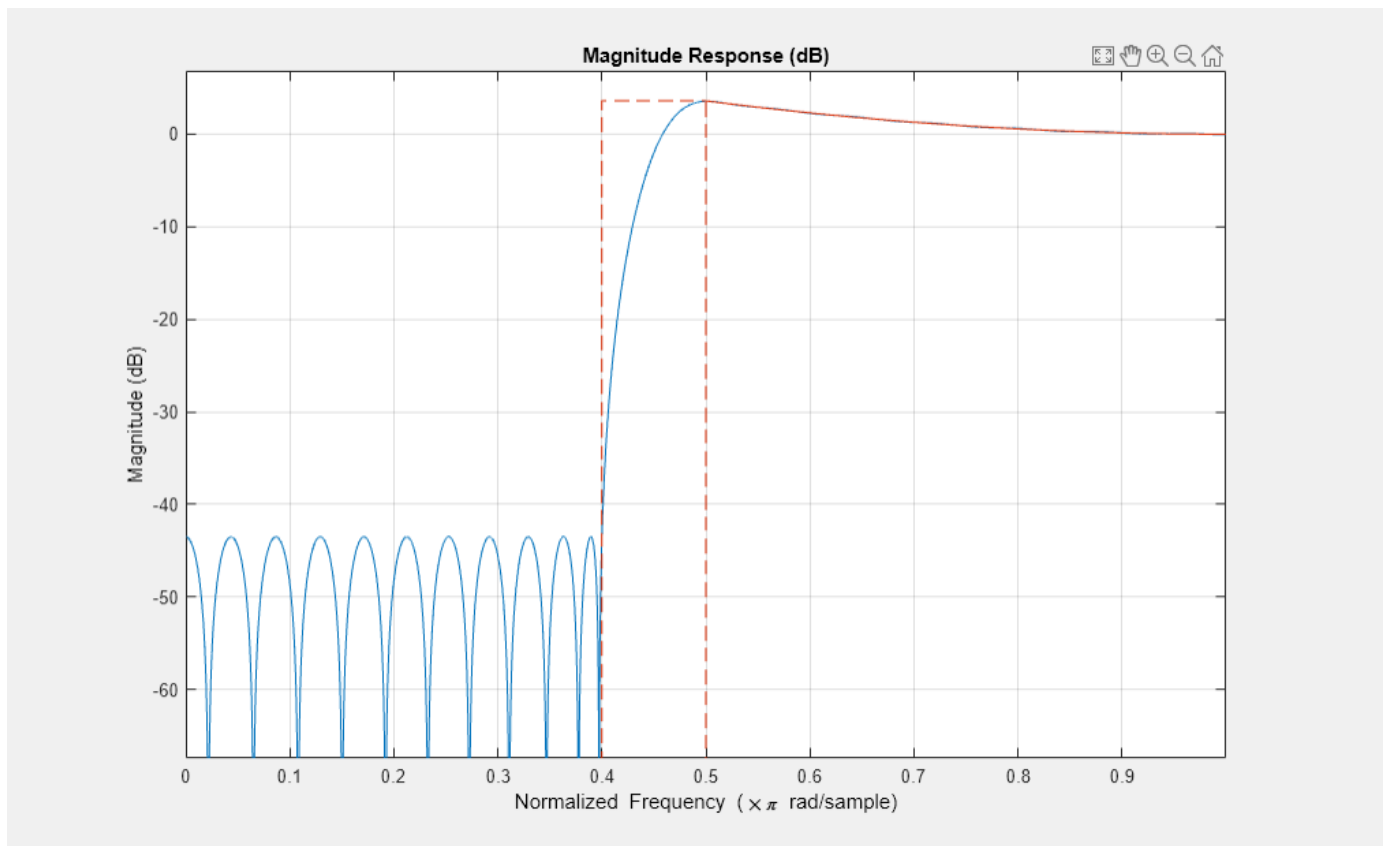
Shape the stopband to have a linear slope of 20 dB/rad/sample

```
Hd1 = design(d,'StopbandShape','linear','StopbandDecay',20,'SystemObject',...
    true);
fvtool(Hd,Hd1);
```



Design a 50th order inverse sinc highpass filter. Set the sinc frequency factor to 0.25, and the sinc power to 16 to achieve a magnitude response in the passband of the form  $H(\omega) = \text{sinc}(0.25*(1-\omega))^{-16}$ .

```
d = fdesign.isinchp('N,Fst,Fp',50,.4,.5);
Hd = design(d,'SincFrequencyFactor',0.25,'SincPower',16,...
'SystemObject',true);
fvtool(Hd);
```



## Version History

Introduced in R2011b

### See Also

`design` | `designmethods` | `fdesign` | `fdesign.ciccomp` | `fdesign.highpass` | `fdesign.isinchnp` | `fdesign.nyquist`

## fdesign.isinclp

Inverse sinc lowpass filter specification

### Syntax

```
d = fdesign.isinclp
d = fdesign.isinclp(spec)
d = fdesign.isinclp(spec,specvalue1,specvalue2,...)
d = fdesign.isinclp(specvalue1,specvalue2,specvalue3,specvalue4)
d = fdesign.isinclp(...,Fs)
d = fdesign.isinclp(...,MAGUNITS)
```

### Description

`d = fdesign.isinclp` constructs an inverse sinc lowpass filter specification object `d`, applying default values for the default specification, 'Fp,Fst,Ap,Ast'.

`d = fdesign.isinclp(spec)` constructs object `d` and sets its 'Specification' to `spec`. Entries in the `spec` represent various filter response features, such as the filter order, that govern the filter design. Valid entries for `spec` are shown below. The options are not case sensitive.

- 'Fp,Fst,Ap,Ast' (default option)
- 'N,Fc,Ap,Ast'
- 'N,Fp,Ap,Ast'
- 'N,Fp,Fst'
- 'N,Fst,Ap,Ast'

The filter specifications are defined as follows:

- `Ast` — attenuation in the stopband in decibels (the default units). Also called `Astop`.
- `Ap` — amount of ripple allowed in the passband in decibels (the default units). Also called `Apass`.
- `Fp` — frequency at the start of the passband. Specified in normalized frequency units. Also called `Fpass`.
- `Fst` — frequency at the end of the stopband. Specified in normalized frequency units. Also called `Fstop`.
- `N` — filter order.

The filter design methods that apply to an inverse sinc lowpass filter specification object change depending on the Specification. Use `designmethods` to determine which design method applies to an object and its specification.

`d = fdesign.isinclp(spec,specvalue1,specvalue2,...)` constructs an object `d` and sets its specifications at construction time.

`d = fdesign.isinclp(specvalue1,specvalue2,specvalue3,specvalue4)` constructs an object `d` assuming the default Specification property 'Fp,Fst,Ap,Ast', using the values you provide in `specvalue1`, `specvalue2`, `specvalue3`, and `specvalue4`.



`d = fdesign.isinclp(...,Fs)` adds the argument `Fs`, specified in Hz to define the sampling frequency to use. In this case, all frequencies in the specifications are in Hz as well.

`d = fdesign.isinclp(...,MAGUNITS)` specifies the units for any magnitude specification you provide in the input arguments. `MAGUNITS` can be one of

- 'linear' — specify the magnitude in linear units
- 'dB' — specify the magnitude in dB (decibels)
- 'squared' — specify the magnitude in power units

When you omit the `MAGUNITS` argument, `fdesign` assumes that all magnitudes are in decibels. Note that `fdesign` stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

The design method of `fdesign.isinclp` implements a filter with a passband magnitude response equal to:

$$H(\omega) = \text{sinc}(C\omega)^{-P}$$

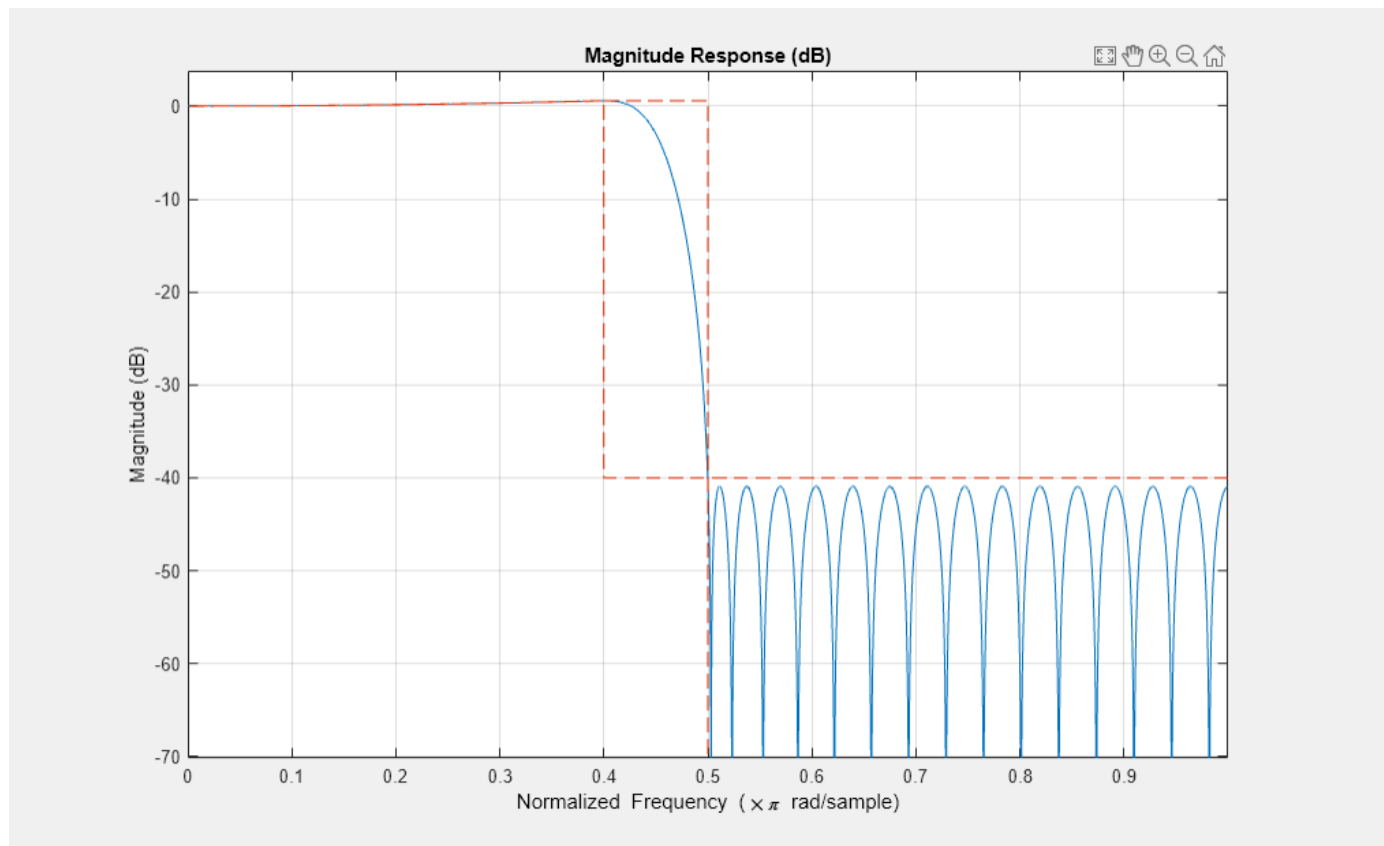
You can control the values of the sinc frequency factor,  $C$ , and the sinc power,  $P$ , using the 'SincFrequencyFactor' and 'SincPower' options in the design method. 'SincFrequencyFactor' and 'SincPower' default to 0.5 and 1 respectively.

## Examples

### Construct an Inverse Sinc Lowpass Filter Using fdesign Object

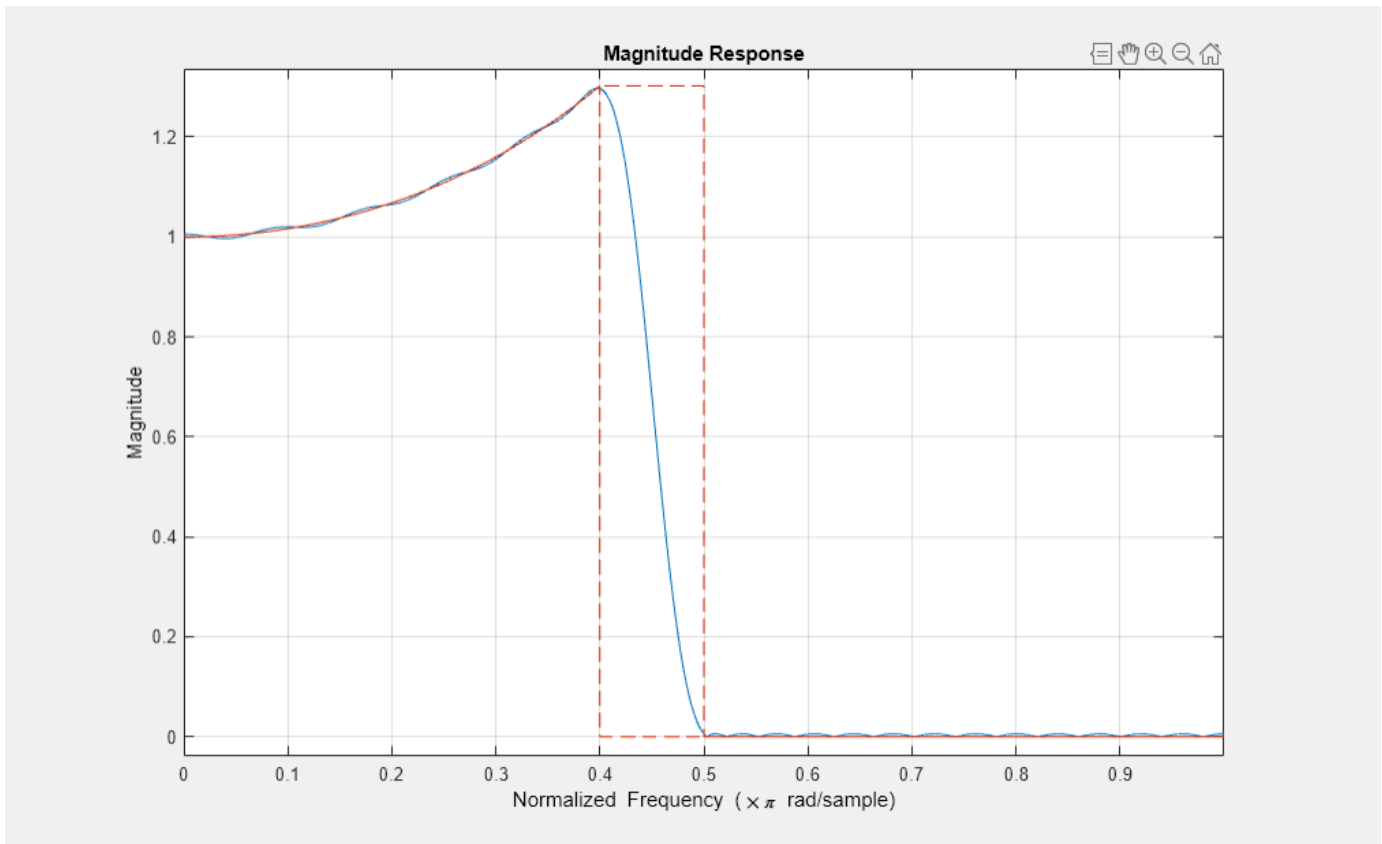
Pass the specifications for the default specification - 'Fp,Fst,Ap,Ast', as input arguments to the specifications object.

```
d = fdesign.isinclp(.4,.5,.01,40);
hd = design(d,'equiripple','SystemObject',true);
fvtool(hd);
```



Design a 50th order inverse sinc lowpass filter. Set the sinc frequency factor to 0.25 and the sinc power to 16 to achieve a magnitude response in the passband of the form  $H(w) = \text{sinc}(0.25*w)^{-16}$ .

```
d = fdesign.isinclp('N,Fp,Fst',50,.4,.5);
Hd = design(d,'SincFrequencyFactor',0.25,'SincPower',16,'SystemObject',...
    true);
fvtool(Hd,'MagnitudeDisplay','Magnitude');
```



## Version History

Introduced in R2011a

### See Also

fdesign | fdesign.bandpass | fdesign.bandstop | fdesign.halfband | fdesign.highpass  
| fdesign.lowpass | fdesign.nyquist

## fdesign.lowpass

Lowpass filter specification

---

### Note

**Note** Multistage filter design using the `fdesign.lowpass` and `design` functions now requires you to set the `SystemObject` flag to `true` unless the resulting design is a single-stage filter and not a cascade. This setting generates a `dsp.FilterCascade` object instead of the `mfilt.cascade` object.

For more information, see “Compatibility Considerations”.

---

### Syntax

```
D = fdesign.lowpass
D = fdesign.lowpass(SPEC)
D = fdesign.lowpass(SPEC,specvalue1,specvalue2,...)
D = fdesign.lowpass(specvalue1,specvalue2,specvalue3,specvalue4)
D = fdesign.lowpass(...,Fs)
D = fdesign.lowpass(...,MAGUNITS)
```

### Description

`D = fdesign.lowpass` constructs a lowpass filter specification object `D`, applying default values for the default specification option `'Fp,Fst,Ap,Ast'`.

`D = fdesign.lowpass(SPEC)` constructs object `D` and sets the `Specification` property to the entry in `SPEC`. Entries in `SPEC` represent various filter response features, such as the filter order, that govern the filter design. Valid entries for `SPEC` are shown below. The options are not case sensitive.

---

**Note** Specifications options marked with an asterisk require the DSP System Toolbox software.

---

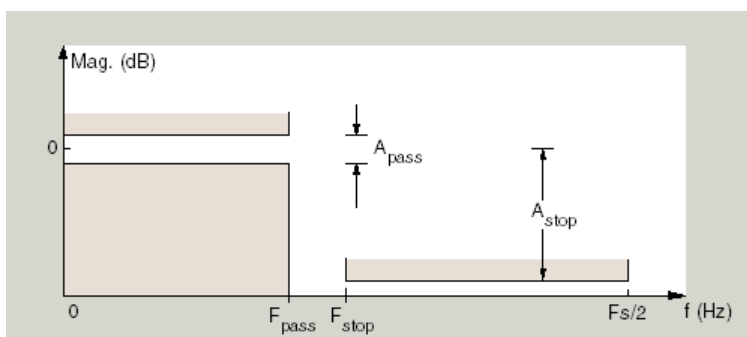
- `'Fp,Fst,Ap,Ast'` (default option)
- `'N,F3db'`
- `'N,F3db,Ap' *`
- `'N,F3db,Ap,Ast' *`
- `'N,F3db,Ast' *`
- `'N,F3db,Fst' *`
- `'N,Fc'`
- `'N,Fc,Ap,Ast'`
- `'N,Fp,Ap'`

- 'N,Fp,Ap,Ast'
- 'N,Fp,Fst,Ap' \*
- 'N,Fp,F3db' \*
- 'N,Fp,Fst'
- 'N,Fp,Fst,Ast' \*
- 'N,Fst,Ap,Ast' \*
- 'N,Fst,Ast'
- 'Nb,Na,Fp,Fst' \*

The filter specifications are defined as follows:

- $A_p$  — amount of ripple allowed in the pass band in decibels (the default units). Also called  $A_{pass}$ .
- $A_s$  — attenuation in the stop band in decibels (the default units). Also called  $A_{stop}$ .
- $F_{3db}$  — cutoff frequency for the point 3 dB point below the passband value. Specified in normalized frequency units.
- $F_c$  — cutoff frequency for the point 6 dB point below the passband value. Specified in normalized frequency units.
- $F_p$  — frequency at the start of the pass band. Specified in normalized frequency units. Also called  $F_{pass}$ .
- $F_{st}$  — frequency at the end of the stop band. Specified in normalized frequency units. Also called  $F_{stop}$ .
- $N$  — filter order.
- $N_a$  and  $N_b$  are the order of the denominator and numerator.

Graphically, the filter specifications look similar to those shown in the following figure.



Regions between specification values like  $F_p$  and  $F_{st}$  are transition regions where the filter response is not explicitly defined.

$D = \text{fdesign.lowpass}(\text{SPEC}, \text{specvalue1}, \text{specvalue2}, \dots)$  constructs an object  $D$  and sets the specification values at construction time using  $\text{specvalue1}$ ,  $\text{specvalue2}$ , and so on for all of the specification variables in  $\text{SPEC}$ .

$D = \text{fdesign.lowpass}(\text{specvalue1}, \text{specvalue2}, \text{specvalue3}, \text{specvalue4})$  constructs an object  $D$  with values for the default Specification property 'Fp,Fst,Ap,Ast' using the specifications you provide as input arguments  $\text{specvalue1}$ ,  $\text{specvalue2}$ ,  $\text{specvalue3}$ ,  $\text{specvalue4}$ .

`D = fdesign.lowpass(...,Fs)` adds the argument `Fs`, specified in Hz to define the sampling frequency to use. In this case, all frequencies in the specifications are in Hz as well.

`D = fdesign.lowpass(...,MAGUNITS)` specifies the units for any magnitude specification you provide in the input arguments. `MAGUNITS` can be one of

- 'linear' — specify the magnitude in linear units
- 'dB' — specify the magnitude in dB (decibels)
- 'squared' — specify the magnitude in power units

When you omit the `MAGUNITS` argument, `fdesign` assumes that all magnitudes are in decibels. Note that `fdesign` stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

## Examples

### Design Butterworth Filter

Design a butterworth filter with lowpass and highpass frequency responses. The filter design procedure is:

- 1 Specify the filter design specifications using a `fdesign` function.
- 2 Pick a design method provided by the `designmethods` function.
- 3 To determine the available design options to choose from, use the `designoptions` function.
- 4 Design the filter using the `design` function.

### Lowpass Filter

Construct a default lowpass filter design specification object using `fdesign.lowpass`.

```
designSpecs = fdesign.lowpass

designSpecs =
    lowpass with properties:

        Response: 'Lowpass'
        Specification: 'Fp,Fst,Ap,Ast'
        Description: {4x1 cell}
        NormalizedFrequency: 1
            Fpass: 0.4500
            Fstop: 0.5500
            Apass: 1
            Astop: 60
```

Determine the available design methods using the `designmethods` function. To design a butterworth filter, pick `butter`.

```
designmethods(designSpecs,SystemObject=true)
```

```
Design Methods that support System objects for class fdesign.lowpass (Fp,Fst,Ap,Ast):
```

```

butter
cheby1
cheby2
ellip
equiripple
ifir
kaiserwin
multistage

```

While designing the filter, you can specify additional design options. View a list of the options using the `designoptions` function. This function also shows the default design options the filter uses.

```
designoptions(designSpecs, 'butter', SystemObject=true)
```

```

ans = struct with fields:
    FilterStructure: {'df1sos' 'df2sos' 'df1tsos' 'df2tsos' 'cascadeallpass' 'cascad
    SOSScaleNorm: 'ustring'
    SOSScaleOpts: 'fdopts.sosscaling'
    MatchExactly: {'passband' 'stopband'}
    DefaultFilterStructure: 'df2sos'
    DefaultMatchExactly: 'stopband'
    DefaultSOSScaleNorm: ''
    DefaultSOSScaleOpts: [1x1 fdopts.sosscaling]

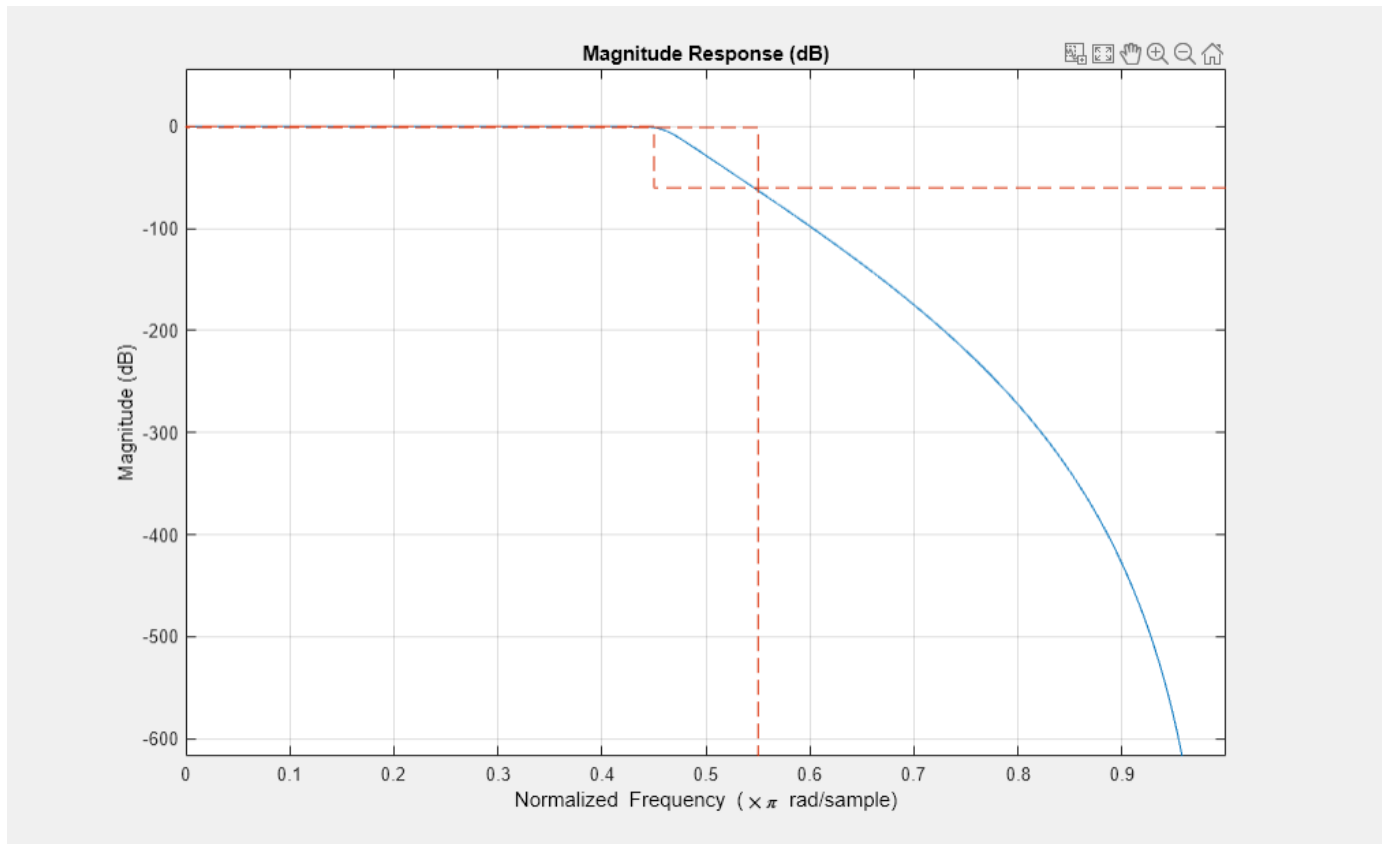
```

Use the `design` function to design the filter. Pass `'butter'` and the specifications given by variable `designSpecs`, as input arguments. Specify the `'matchexactly'` design option to `'passband'`.

```
lpFilter = design(designSpecs, 'butter', matchexactly='passband', SystemObject=true);
```

Visualize the frequency response of the designed filter.

```
fvtool(lpFilter)
```



## Highpass Filter

Construct a highpass filter design specification object using `fdesign.highpass`. Specify the order to be 7 and the 3 dB frequency to be  $0.6\pi$  radians/sample.

```
designSpecs = fdesign.highpass('N,F3dB',7,.6);
```

Determine the available design methods. To design a butterworth filter, pick `butter`.

```
designMethods(designSpecs,SystemObject=true)
```

Design Methods that support System objects for class `fdesign.highpass(N,F3dB)`:

```
butter
maxflat
```

While designing the filter, you can specify additional design options. View a list of the options using the `designoptions` function. This function also shows the default design options the filter uses.

```
designoptions(designSpecs,'butter',SystemObject=true)
```

```
ans = struct with fields:
```

```
    FilterStructure: {'df1sos' 'df2sos' 'df1tsos' 'df2tsos' 'cascadeallpass' 'cascadeallpass'}
    SOSScaleNorm: 'ustring'
    SOSScaleOpts: 'fdopts.sosscaling'
    DefaultFilterStructure: 'df2sos'
    DefaultSOSScaleNorm: ''
```



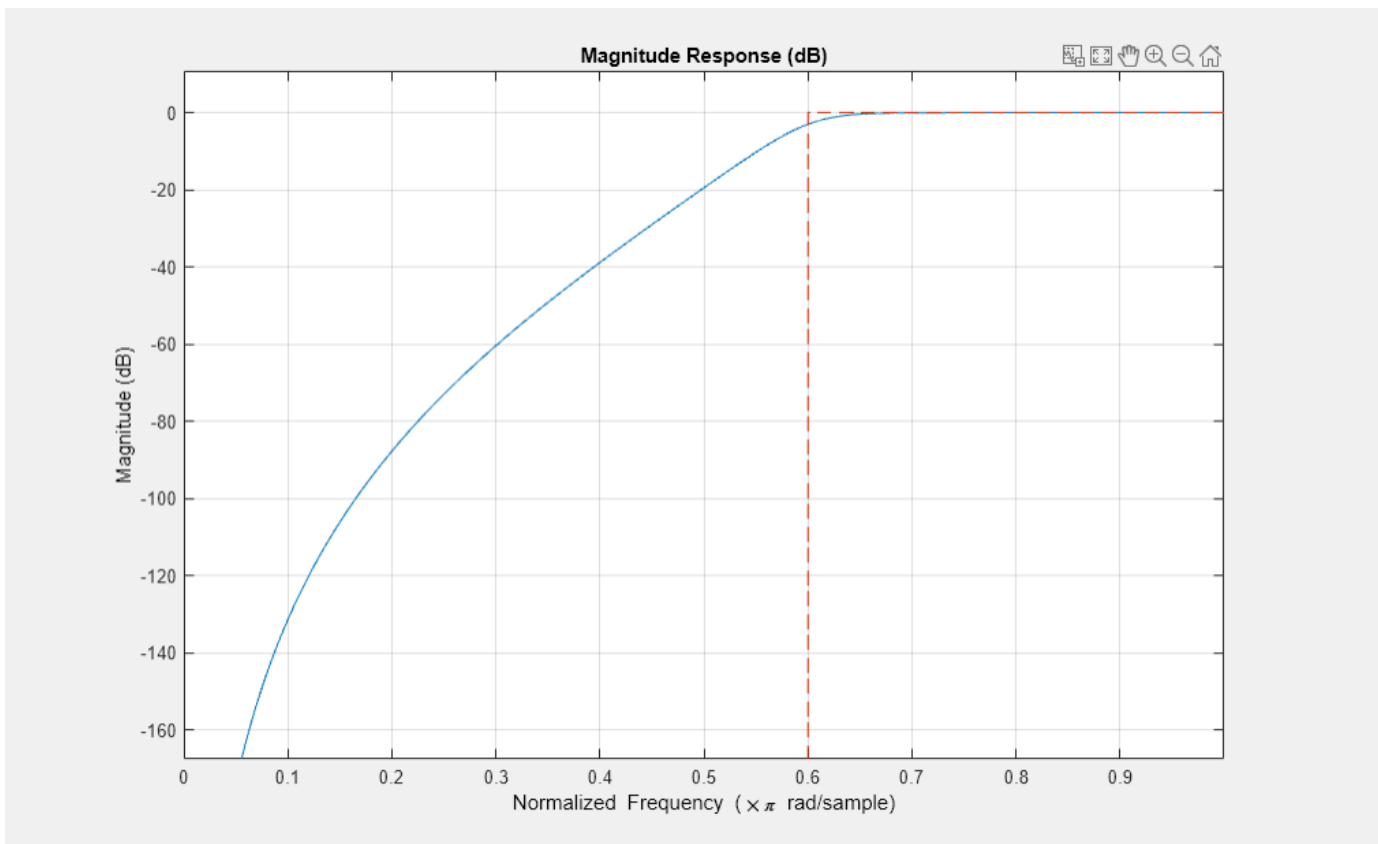
```
DefaultSOSScale0pts: [1x1 fdopts.sosscaling]
```

To design the butterworth filter, use the design function and specify 'butter' as an input. Set 'FilterStructure' to 'cascadeallpass'.

```
hpFilter = design(designSpecs, 'butter', FilterStructure='cascadeallpass', SystemObject=true);
```

Visualize the highpass frequency response.

```
fvtool(hpFilter)
```



### Lowpass Filtering of Sinusoids

Lowpass filter a discrete-time signal consisting of two sine waves.

Create a lowpass filter specification object. Specify the passband frequency to be  $0.15\pi$  rad/sample and the stopband frequency to be  $0.25\pi$  rad/sample. Specify 1 dB of allowable passband ripple and a stopband attenuation of 60 dB.

```
d = fdesign.lowpass('Fp,Fst,Ap,Ast',0.15,0.25,1,60);
```

Query the valid design methods for your filter specification object, d.

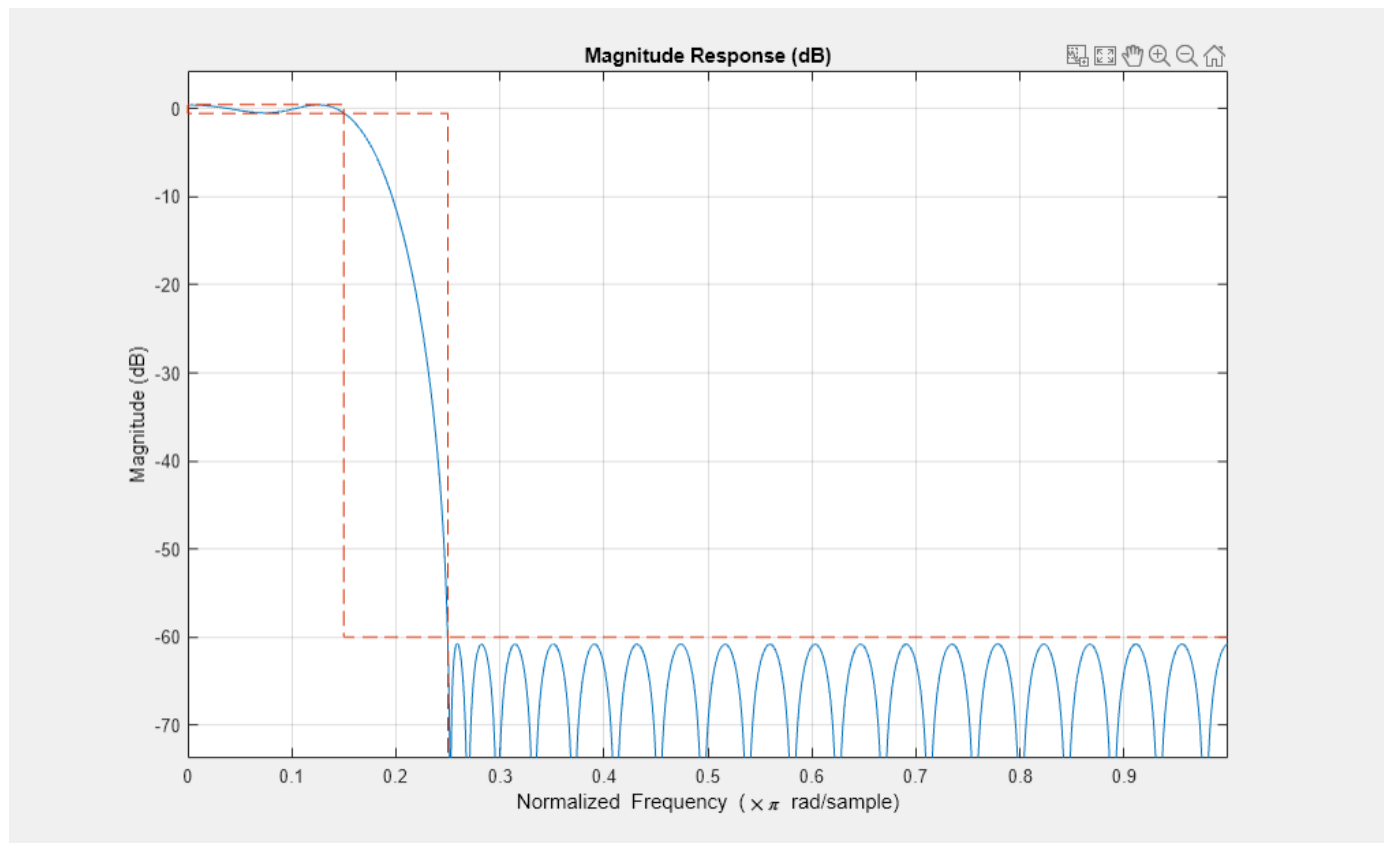
```
designmethods(d)
```

Design Methods for class `fdesign.lowpass (Fp,Fst,Ap,Ast)`:

```
butter
cheby1
cheby2
ellip
equiripple
ifir
kaiserwin
multistage
```

Create an FIR equiripple filter and view the filter magnitude response with `fvtool`.

```
Hd = design(d,'equiripple');
fvtool(Hd)
```



Create a signal consisting of the sum of two discrete-time sinusoids with frequencies of  $\pi/8$  and  $\pi/4$  rad/sample and amplitudes of 1 and 0.25, respectively. Filter the discrete-time signal with the FIR equiripple filter object, `Hd`.

```
n = 0:159;
x = (0.25*cos((pi/8)*n)+sin((pi/4)*n));
y = filter(Hd,x);
```

Compute the Fourier transform of the original signal and the filtered signal. Verify that the high-frequency component has been filtered out.

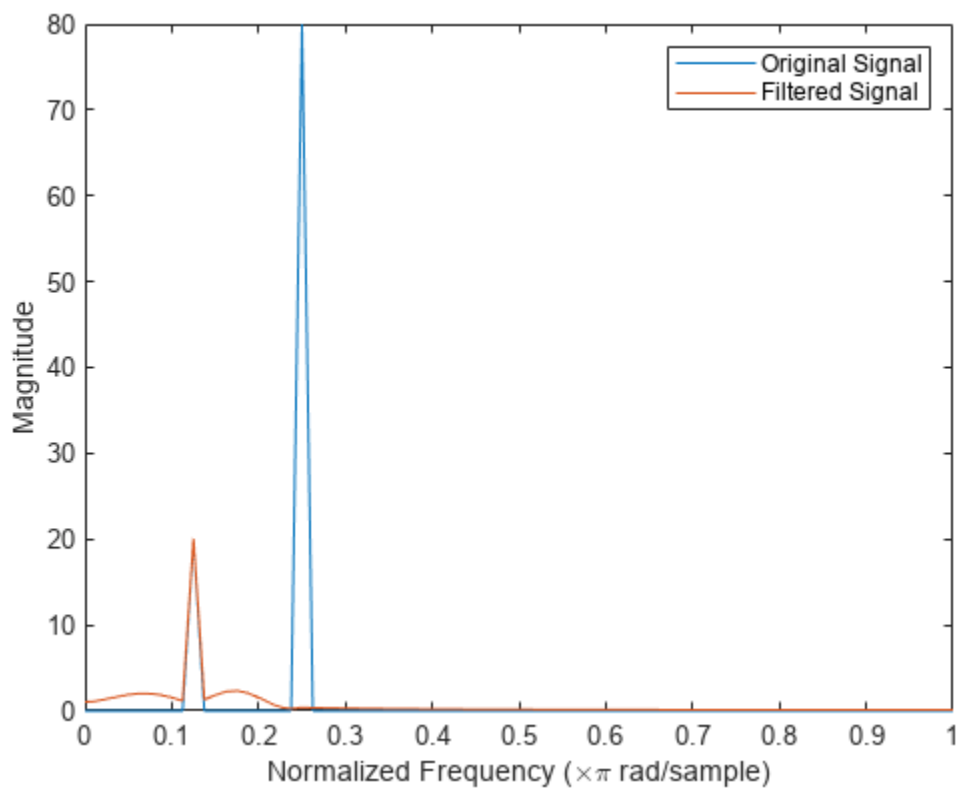
```

freq = 0:(2*pi)/160:pi;
xdft = fft(x);
ydft = fft(y);

figure
plot(freq/pi,abs(xdft(1:length(x)/2+1)))
hold on
plot(freq/pi,abs(ydft(1:length(y)/2+1)))
hold off

legend('Original Signal','Filtered Signal')
ylabel('Magnitude')
xlabel('Normalized Frequency (\times\pi rad/sample)')

```



### Window Method Lowpass Design

Create a filter of order 10 with a 6-dB frequency of 9.6 kHz and a sampling frequency of 48 kHz. Look at the available design methods.

```

d = fdesign.lowpass('N,Fc',10,9600,48000);
designmethods(d)

```

Design Methods for class fdesign.lowpass (N,Fc):

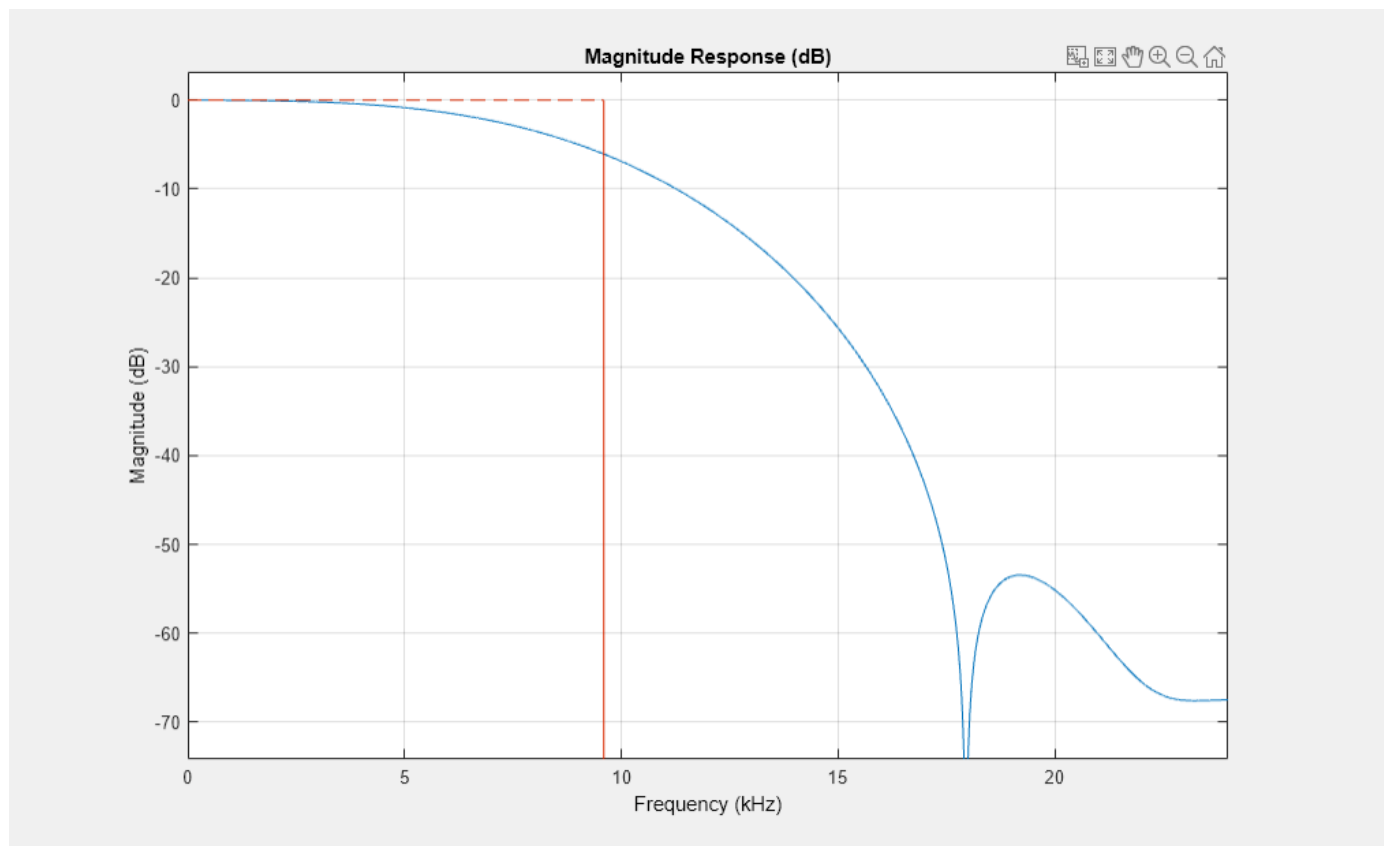
```
window
```

The only valid design method is the FIR window method. Design the filter.

```
Hd = design(d);
```

Display the filter magnitude response. The -6 dB point is at 9.6 kHz, as expected.

```
fvtool(Hd)
```



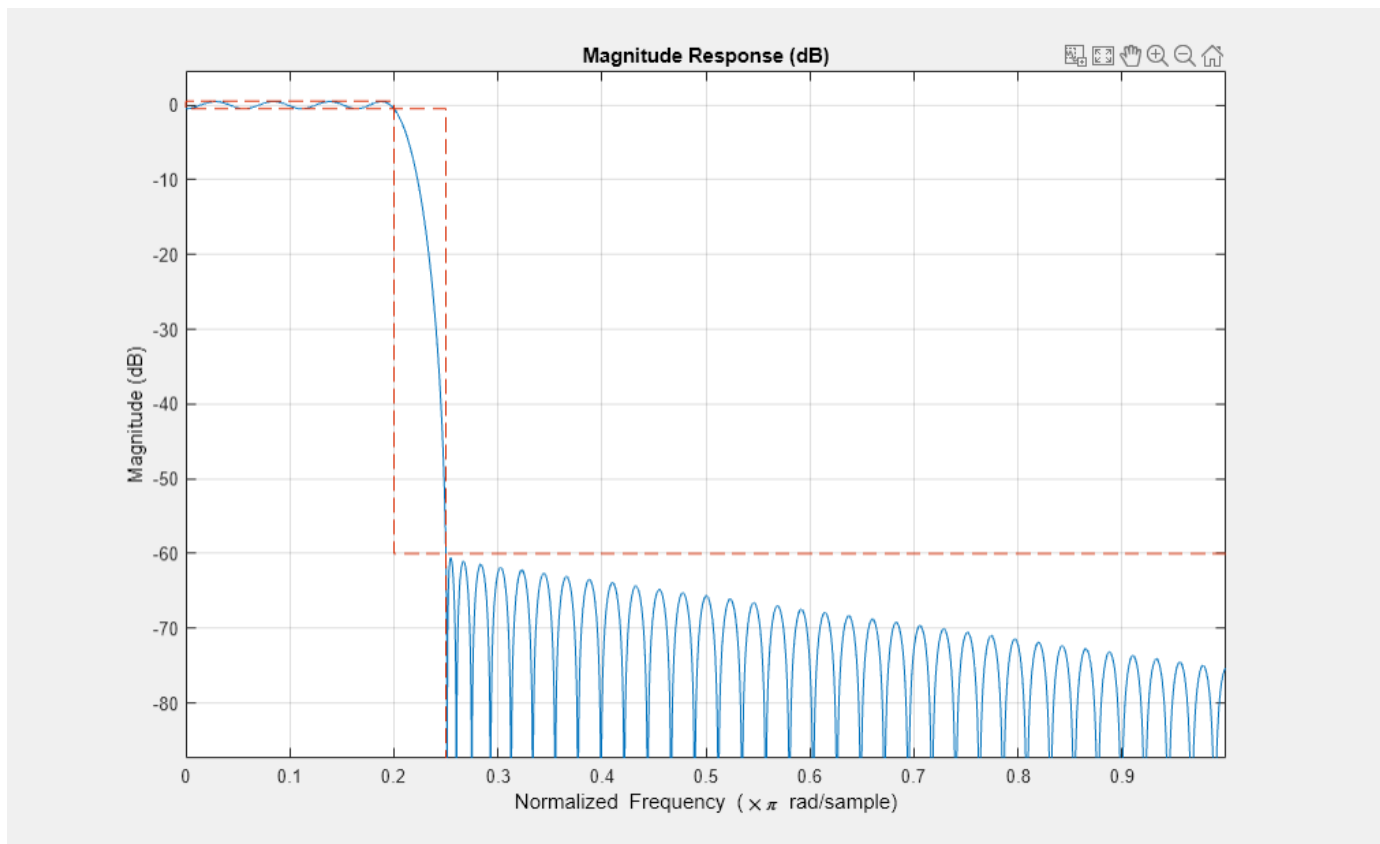
### Stopband Shape and Decay

Create an FIR equiripple filter with a passband frequency of  $0.2\pi$  rad/sample, a stopband frequency of  $0.25\pi$  rad/sample, a passband ripple of 1 dB, and a stopband attenuation of 60 dB. Design the filter with a 20 dB/rad/sample linear stopband.

```
D = fdesign.lowpass('Fp,Fst,Ap,Ast',0.2,0.25,1,60);
Hd = design(D,'equiripple','StopbandShape','linear','StopbandDecay',20);
```

Visualize the frequency response of the filter.

```
fvtool(Hd)
```



## Version History

Introduced in R2009a

**R2022b: Support for designing `mfilt.cascade` object using the `fdesign.lowpass` object has been removed**

*Errors starting in R2022b*

Multistage filter design using the `fdesign.lowpass` object and the design function now requires you to set the `SystemObject` flag to `true` unless the resulting design is a single-stage filter and not a cascade. This setting generates a `dsp.FilterCascade` object instead of the `mfilt.cascade` object.

```
fdesignObj = fdesign.lowpass('Fp,Fst,Ap,Ast',...
0.1,0.12,0.1,1e-4,'linear');
FIRInterpObj = design(fdesignObj,'multistage',...
SystemObject=true)
```

FIRInterpObj =

dsp.FilterCascade with properties:

```
Stage1: [1x1 dsp.FIRDecimator]
Stage2: [1x1 dsp.FIRDecimator]
Stage3: [1x1 dsp.FIRDecimator]
```

```
Stage4: [1x1 dsp.FIRInterpolator]  
Stage5: [1x1 dsp.FIRInterpolator]  
Stage6: [1x1 dsp.FIRInterpolator]  
CloneStages: true
```

### **See Also**

[design](#) | [designmethods](#) | [fdesign](#)

# fdesign.notch

Notch filter specification

## Syntax

```
notchSpecs = fdesign.notch
notchSpecs = fdesign.notch(n, f0, q)
notchSpecs = fdesign.notch(spec, value1, ..., valueN)
notchSpecs = fdesign.notch( ____, Fs)
notchSpecs = fdesign.notch( ____, magunits)
```

## Description

The `fdesign.notch` function returns a notch filter design specification object that contains the specifications for a filter, such as passband ripple, stopband attenuation, and filter order. Then, use the `design` function to design the filter from the filter design specifications object.

For more control options, see “Filter Design Procedure” on page 5-399. For a complete workflow, see “Design a Filter in Fdesign — Process Overview”.

`notchSpecs = fdesign.notch` constructs a notch filter specification object with the filter order set to 10, center frequency set to 0.5, and quality factor set to 2.5.

`notchSpecs = fdesign.notch(n, f0, q)` constructs a notch filter specification object with the filter order, center frequency, and quality factor specified by `n`, `f0`, and `q`, respectively.

`notchSpecs = fdesign.notch(spec, value1, ..., valueN)` constructs a notch filter specification object with a particular filter order, center frequency, and other specification options. Indicate the options you want to specify in the expression `spec`. After the expression, specify a value for each option.

`notchSpecs = fdesign.notch( ____, Fs)` provides the sample rate of the signal to be filtered.

`notchSpecs = fdesign.notch( ____, magunits)` provides the units for any magnitude specification given.

`magunits` can be one of the following: 'linear', 'dB', or 'squared'. If this argument is omitted, 'dB' is assumed. The magnitude specifications are always converted and stored in decibels regardless of how they were specified. If `Fs` is provided, `magunits` must follow `Fs` in the input argument list.

## Examples

### Design Notch Filter

Design a direct-form I notching filter that has a filter order of 6, center frequency of 0.5, quality factor of 10, and a passband ripple of 1 dB.

Create a notch filter design specification object using the `fdesign.notch` function and specify these design parameters.

```
notchSpecs = fdesign.notch('N,F0,Q,Ap',6,0.5,10,1);
```

Design the notch filter using the `design` function. The resulting filter is a `dsp.SOSFilter` System object™. For details on how to apply this filter on streaming data, refer to `dsp.SOSFilter`.

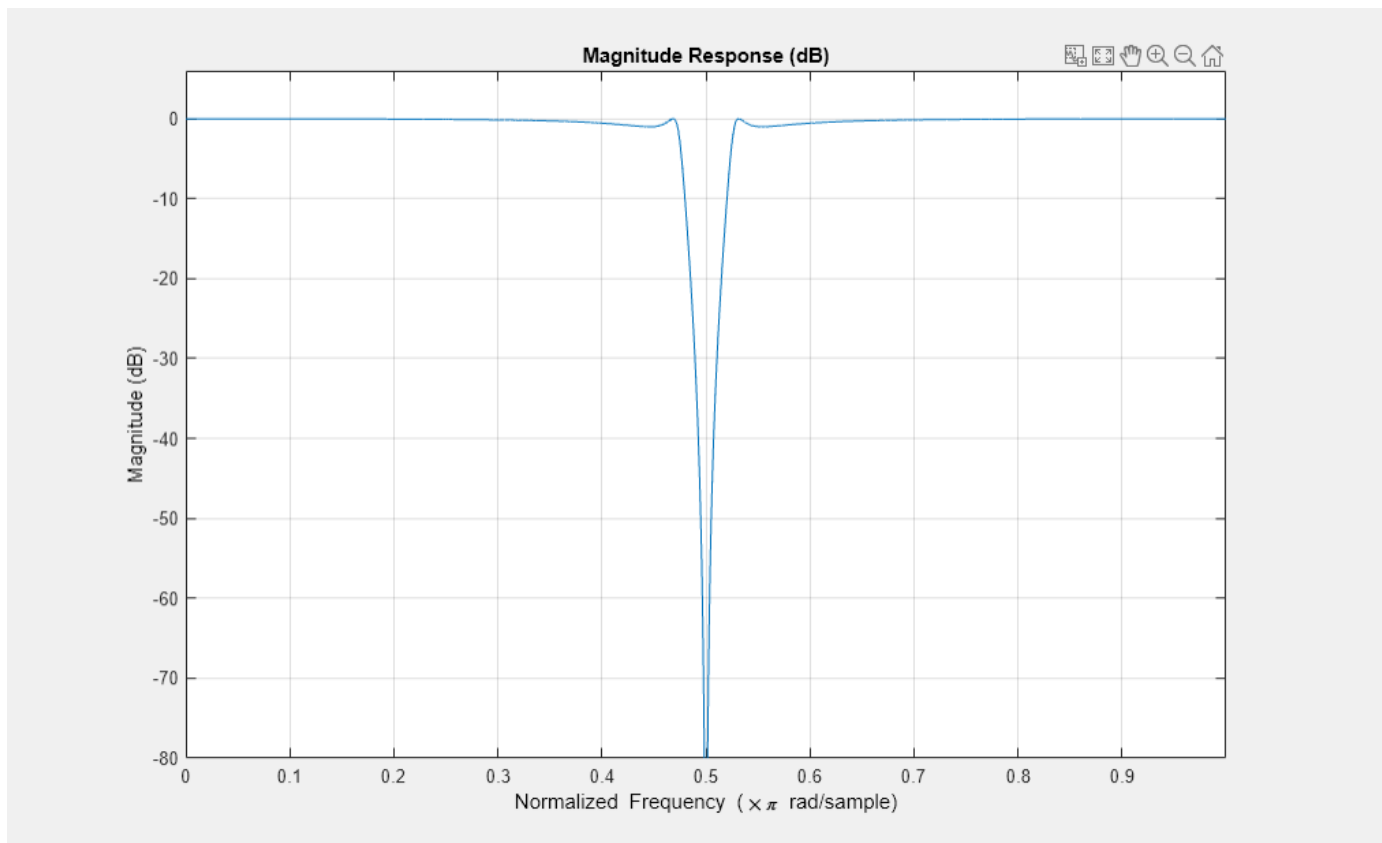
```
notchFilt = design(notchSpecs,SystemObject=true)
```

```
notchFilt =  
  dsp.SOSFilter with properties:  
    Structure: 'Direct form II'  
  CoefficientSource: 'Property'  
    Numerator: [3x3 double]  
   Denominator: [3x3 double]  
  HasScaleValues: true  
   ScaleValues: [0.7722 0.7722 1.3597 1]
```

Show all properties

Visualize the frequency response of the designed filter using `fvtool`.

```
fvtool(notchFilt)
```





## Input Arguments

### spec — Specification

'N,F0,Q' (default) | 'N,F0,Q,Ap' | 'N,F0,Q,Ast' | ...

Specification expression, specified as one of these character vectors:

- 'N,F0,Q' (default)
- 'N,F0,Q,Ap'
- 'N,F0,Q,Ast'
- 'N,F0,Q,Ap,Ast'
- 'N,F0,BW'
- 'N,F0,BW,Ap'
- 'N,F0,BW,Ast'
- 'N,F0,BW,Ap,Ast'

This table describes each option that can appear in the expression.

| Specification option | Description                 |
|----------------------|-----------------------------|
| N                    | Filter order (must be even) |
| F0                   | Center frequency            |
| Q                    | Quality factor              |
| BW                   | 3 dB bandwidth              |
| Ap                   | Passband ripple (dB)        |
| Ast                  | Stopband attenuation (dB)   |

The design methods available for designing the filter depend on the specification expression. You can obtain these methods using the `designmethods` function. The table lists each specification expression supported by `fdesign.notch` and the corresponding design methods available.

| Specification expression | Supported design method | Filter description               |
|--------------------------|-------------------------|----------------------------------|
| 'N,F0,Q'                 | butter                  | Butterworth digital filter       |
| 'N,F0,Q,Ap'              | cheby1                  | Chebyshev Type I digital filter  |
| 'N,F0,Q,Ast'             | cheby2                  | Chebyshev Type II digital filter |
| 'N,F0,Q,Ap,Ast'          | ellip                   | Elliptical digital filter        |
| 'N,F0,BW'                | butter                  | Butterworth digital filter       |
| 'N,F0,BW,Ap'             | cheby1                  | Chebyshev Type I digital filter  |
| 'N,F0,BW,Ast'            | cheby2                  | Chebyshev Type II digital filter |
| 'N,F0,BW,Ap,Ast'         | ellip                   | Elliptical digital filter        |

To design the filter, call the `design` function with one of these design methods as an input. You can choose the type of filter response by passing 'FIR' or 'IIR' to the `design` function. For more details, see `design`.

For more details on the procedure, see “Filter Design Procedure” on page 5-399. For an example, see “Design Notch Filter” on page 5-543.

**value1, ..., valueN — Specification values**

comma-separated list of values

Specification values, specified as a comma-separated list of values. Specify a value for each option in `spec` in the same order that the options appear in the expression.

Example: `d = fdesign.notch('N,F0,BW,Ast',n,f0,bw,ast)`

The arguments below describe more details for each option in the expression.

**n — Filter order**

even positive integer

Filter order, specified as an even positive integer.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**f0 — Center frequency**

scalar

Center frequency of the filter, specified as a scalar. When the input sampling frequency  $F_s$  is specified, the center frequency is in Hz. When the input sample rate is not specified, the center frequency is in normalized units between 0 and 1.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**q — Quality factor**

positive scalar

Quality factor of the filter, specified as a real positive scalar.

Quality factor of the filter is defined as the ratio of the center frequency to the 3 dB bandwidth.

$$q = f_0/bw$$

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**bw — 3 dB Bandwidth**

scalar

3 dB bandwidth of the filter, specified as a real scalar.

Specify the 3 dB bandwidth value in normalized frequency units between 0 and 1. If you specify the sample rate  $F_s$ , then specify the bandwidth value in Hz instead.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**ap — Passband ripple**

positive scalar

Passband ripple, specified as a positive scalar in dB. If `magunits` is `'linear'` or `'squared'`, the passband ripple is converted and stored in dB by the function regardless of how it has been specified.

Data Types: `double`

**ast — Stopband attenuation**

positive scalar

Stopband attenuation of the filter, specified as a positive scalar in dB. If `magunits` is 'linear' or 'squared', the stopband attenuation is converted and stored in dB by the function regardless of how it has been specified.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **Fs — Sample rate**

scalar

Sample rate of the signal to be filtered, specified as a scalar in Hz. Specify the sample rate as a scalar trailing the other numerical values provided. When `Fs` is provided, `Fs` is assumed to be in Hz, as are all other frequency values provided. Note that you do not have to change the specification string.

Consider a design specification where `N` is set to 4, `F0` is set to 1200 Hz, and `Q` is set to 6.5. Specify the sample rate of the input signal as 8000 Hz. Here is how the design looks:

```
d = fdesign.notch('N,F0,Q',4,1200,6.5,8e3); filt =
design(d,'Systemobject',true);
```

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **magunits — Magnitude units**

'dB' (default) | 'linear' | 'squared'

Magnitude specification units, specified as 'dB', 'linear', or 'squared'. If this argument is omitted, 'dB' is assumed. Note that the magnitude specifications are always converted and stored in dB regardless of how they were specified. If `Fs` is one of the input arguments, `magunits` must be specified after `Fs` in the input argument list.

## **Output Arguments**

### **notchSpecs — Notch filter design specification object**

`notch` object

Notch filter design specification object, returned as a `notch` object. The fields of the object depend on the `spec` input character vector.

Consider an example where the `spec` argument is set to 'N,F0,Q,Ap,Ast', and the corresponding values are set to 6, 0.5, 10, 1, 80, respectively. The `notch` filter design specification object is populated with the following fields:

```
>> notchSpecs = fdesign.notch('N,F0,Q,Ap,Ast',6,0.5,10,1,80)
```

```
notchSpecs =
```

```
notch with properties:
```

```
    Response: 'Notching Filter'  
Specification: 'N,F0,Q,Ap,Ast'  
Description: {5×1 cell}  
NormalizedFrequency: 1  
FilterOrder: 6  
           F0: 0.5000  
           Q: 10  
           Apass: 1  
           Astop: 80
```

## Version History

Introduced in R2011a

## See Also

### Functions

[design](#) | [fdesign](#) | [fdesign.peak](#) | [designmethods](#) | [designoptions](#)

### Objects

[dsp.SOSFilter](#)

### Topics

[“Design a Filter in Fdesign — Process Overview”](#)

# fdesign.nyquist

Nyquist filter specification

## Syntax

```
d = fdesign.nyquist
d = fdesign.nyquist(l, spec)
d = fdesign.nyquist(l, spec, specvalue1, specvalue2, ...)
d = fdesign.nyquist(l, specvalue1, specvalue2)
d = fdesign.nyquist(..., fs)
d = fdesign.nyquist(..., magunits)
```

## Description

`d = fdesign.nyquist` constructs a Nyquist or L-band filter specification object `d`, applying default values for the properties `tw` and `ast`. By default, the filter object designs a minimum-order half-band ( $L=2$ ) Nyquist filter.

Using `fdesign.nyquist` along with `design` method generates a System object, if the 'SystemObject' flag in the `design` method is set to `true`.

`d = fdesign.nyquist(l, spec)` constructs object `d` and sets its `Specification` property to `spec`. Use `l` to specify the desired value for  $L$ .  $L = 2$  designs a half-band FIR filter,  $L = 3$  a third-band FIR filter, and so on. When you use a Nyquist filter as an interpolator,  $l$  or  $L$  is the interpolation factor. The first input argument must be `l` when you are not using the default syntax `d = fdesign.nyquist`.

Entries in the `spec` represent various filter response features, such as the filter order, that govern the filter design. Valid entries for `spec` are shown below. The entries are not case sensitive.

- `tw,ast` (default option)
- `n,tw`
- `n`
- `n,ast`

where,

- `ast` — attenuation in the stop band in decibels (the default units).
- `n` — filter order.
- `tw` — width of the transition region between the pass and stop bands. Specified in normalized frequency units.

The filter design methods that apply to a Nyquist filter specification object change depending on the `Specification` option. Use `designmethods` to determine which design method applies to an object and its specification option. Different filter design methods also have options that you can specify. Use `designopts` with the design method to see the available options. For example:

```
f=fdesign.nyquist(4, 'N,TW');
designmethods(f)
```

`d = fdesign.nyquist(l, spec, specvalue1, specvalue2, ...)` constructs an object `d` and sets its specification to `spec`, and the specification values to `specvalue1`, `specvalue2`, and so on at construction time.

`d = fdesign.nyquist(l, specvalue1, specvalue2)` constructs an object `d` with the values you provide in `l`, `specvalue1`, `specvalue2` as the values for `l`, `tw` and `ast`.

`d = fdesign.nyquist(..., fs)` adds the argument `fs`, specified in Hz to define the sampling frequency to use. In this case, all frequencies in the specifications are in Hz as well.

`d = fdesign.nyquist(..., magunits)` specifies the units for any magnitude specification you provide in the input arguments. `magunits` can be one of

- `linear` — specify the magnitude in linear units
- `dB` — specify the magnitude in dB (decibels)
- `squared` — specify the magnitude in power units

When you omit the `magunits` argument, `fdesign` assumes that all magnitudes are in decibels. Note that `fdesign` stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

### Limitations of the Nyquist `fdesign` Object

Using Nyquist filter specification objects with the `equiripple` design method imposes a few limitations on the resulting filter, caused by the `equiripple` design algorithm.

- When you request a minimum-order design from `equiripple` with your Nyquist object, the design algorithm might not converge and can fail with a filter convergence error.
- When you specify the order of your desired filter, and use the `equiripple` design method, the design might not converge.
- Generally, the following specifications, alone or in combination with one another, can cause filter convergence problems with Nyquist objects and the `equiripple` design method.
  - very high order
  - small transition width
  - very large stopband attenuation

Note that halfband filters (filters where `band = 2`) do not exhibit convergence problems.

When convergence issues arise, either in the cases mentioned or in others, you might be able to design your filter with the `kaiserwin` method.

In addition, if you use Nyquist objects to design decimators or interpolators (where the interpolation or decimation factor is not a prime number), using multistage filter designs might be your best approach.

## Examples

### Construct a Nyquist Filter Using `fdesign` Object

These examples show how to construct a Nyquist filter specification object.

First, create a default specifications object without using input arguments.

```
d = fdesign.nyquist; %#ok
```

Now create an object by passing a specification type 'n,ast' - the resulting object uses default values for n and ast.

```
d = fdesign.nyquist(2, 'n,ast'); %#ok
```

Create another Nyquist filter object, passing the specification values to the object rather than accepting the default values for n and ast.

```
d = fdesign.nyquist(3, 'n,ast', 42, 80) %#ok
```

```
d =
  nyquist with properties:
      Response: 'Nyquist'
  Specification: 'N,Ast'
  Description: {2x1 cell}
NormalizedFrequency: 1
  FilterOrder: 42
      Astop: 80
      Band: 3
```

Finally, pass the filter specifications that correspond to the default Specification - tw,ast. When you pass only the values, fdesign.nyquist assumes the default Specification option.

```
d = fdesign.nyquist(4, .01, 80)
```

```
d =
  nyquist with properties:
      Response: 'Nyquist'
  Specification: 'TW,Ast'
  Description: {2x1 cell}
NormalizedFrequency: 1
  TransitionWidth: 0.0100
      Astop: 80
      Band: 4
```

Now design a Nyquist filter using the kaiserwin design method.

```
hd = design(d, 'kaiserwin', 'SystemObject', true);
```

Create two equiripple Nyquist 4th-band filters with and without a nonnegative zero phase response:

```
f = fdesign.nyquist(4, 'N,TW', 12, 0.2);
```

Equiripple Nyquist 4th-band filter with nonnegative zero phase response

```
Hd1 = design(f, 'equiripple', 'zerophase', true, 'SystemObject', true);
```

Equiripple Nyquist 4th-band filter with 'ZeroPhase' set to false 'zerophase', false is the default

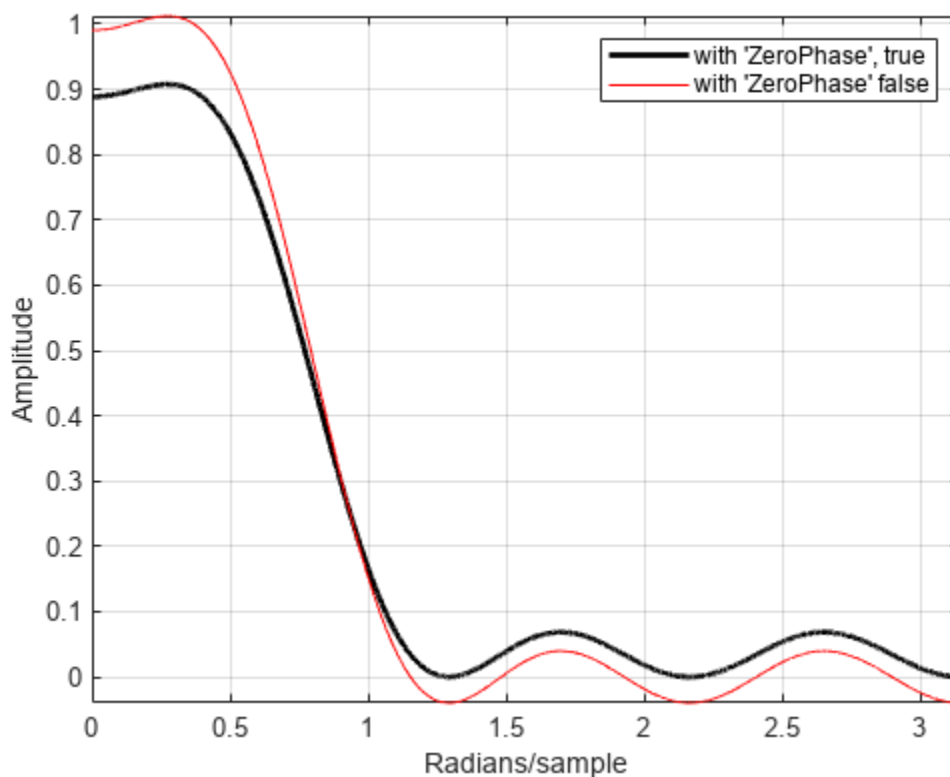
```
Hd2 = design(f, 'equiripple', 'zerophase', false, 'SystemObject', true);
```

Obtain real-valued amplitudes (not magnitudes)

```
[Hr_zerophase,~] = zerophase(Hd1);
[Hr,W] = zerophase(Hd2);
```

Plot and compare response

```
plot(W,Hr_zerophase,'k','linewidth',2);
xlabel('Radians/sample'); ylabel('Amplitude');
hold on;
plot(W,Hr,'r');
axis tight; grid on;
legend('with ''ZeroPhase'', true','with ''ZeroPhase'' false');
```



Note that the amplitude of the zero phase response (black line) is nonnegative for all frequencies.

The 'ZeroPhase' option is valid only for equiripple Nyquist designs with the 'N,TW' specification. You cannot specify 'MinPhase' and 'ZeroPhase' to be simultaneously 'true'.

## Version History

Introduced in R2011a



### **See Also**

fdesign | fdesign.interpolator | fdesign.halfband | fdesign.interpolator |  
fdesign.rsrc | zerophase

## fdesign.octave

(To be removed) Octave filter specification

### Compatibility

---

**Note** The `fdesign.octave` filter specification object will be removed from DSP System Toolbox in a future release. Existing instances of the object continue to run. For new code, use the `octaveFilter` System object instead. For more information on how to update your existing code, see “Compatibility Considerations” on page 5-556.

---

### Syntax

```
d = fdesign.octave(l)
d = fdesign.octave(l, MASK)
d = fdesign.octave(l, MASK, spec)
d = fdesign.octave(..., Fs)
```

### Description

`d = fdesign.octave(l)` constructs an octave filter specification object `d`, with `l` bands per octave. The default value for `l` is one.

---

**Note** The filters created by `fdesign.octave` comply with the ANSI<sup>®</sup> S1.11-2004 and IEC 61260:1995 standards.

---

`d = fdesign.octave(l, MASK)` constructs an octave filter specification object `d` with `l` bands per octave and `MASK` specification for the FVTool. The available values for mask are:

- 'class 0'
- 'class 1'
- 'class 2'

`d = fdesign.octave(l, MASK, spec)` constructs an octave filter specification object `d` with `l` bands per octave, `MASK` specification for the FVTool, and the `spec` specification character vector. The specifications available are:

- 'N, F0'

(not case sensitive), where:

- `N` is the filter order
- `F0` is the center frequency. The center frequency is specified in normalized frequency units assuming a sampling frequency of 48 kHz, unless a sampling frequency in Hz is included in the specification: `d = fdesign.octave(..., Fs)`. If you specify an invalid center frequency, a warning is issued and the center frequency is rounded to the nearest valid value. You can

determine the valid center frequencies for your design by using `validfrequencies` with your octave filter specification object. For example:

```
d = fdesign.octave(1, 'Class 1', 'N,F0', 6, 1000, 44.1e3);
validcenterfreq = validfrequencies(d);
```

Valid center frequencies:

- Must be greater than 20 Hz and less than 20 kHz if you specify a sampling frequency. The range 20 Hz to 20 kHz is the standard range of human hearing.
- Are calculated according to the following algorithm if the number of bands per octave,  $L$ , is even:

```
G = 10^(3/10);
x = -1000:1350;
validcenterfreq = 1000*(G.^((2*x-59)/(2*L)));
validcenterfreq = validcenterfreq(validcenterfreq>20 & validcenterfreq<2e4);
```

- Are calculated according to the following algorithm if the number of bands per octave,  $L$ , is odd:

```
G = 10^(3/10);
x = -1000:1350;
validcenterfreq = 1000*(G.^((x-30)/L));
validcenterfreq = validcenterfreq(validcenterfreq>20 & validcenterfreq<2e4);
```

Only center frequencies greater than 20 and less than 20,000 are retained. The center frequencies and the corresponding upper band frequencies must be less than the Nyquist frequency, which is half the sampling rate (`samplingfreq`). The vector of upper band frequencies (`upperbandfreq`) corresponding to the center frequencies (`validcenterfreq`) is computed using the following algorithm:

```
upperbandfreq = validcenterfreq.*(G^(1/(2*L)));
```

The algorithm removes the center frequencies whose corresponding upper band frequencies do not obey the Nyquist rule.

```
validcenterfreq = validcenterfreq(upperbandfreq < samplingfreq/2);
```

If you do not specify a sampling frequency, `fdesign.octave` assumes a `samplingfreq` of 48 kHz. To obtain valid normalized center frequencies, the remaining center frequencies are divided by 24,000.

```
validcenterfreq = validcenterfreq/24000;
```

## Examples

### Design an Octave Band Filter

Design a sixth order, octave-band class 0 filter with a center frequency of 1000 Hz and, a sampling frequency of 44.1 kHz.

```
d = fdesign.octave(1, 'Class 0', 'N,F0', 6, 1000, 44100)
d = octave with properties:
    Response: 'Octave and Fractional Octave'
    BandsPerOctave: 1
    Mask: 'Class 0'
    Specification: 'N,F0'
```

```

        Description: {2x1 cell}
NormalizedFrequency: 0
                Fs: 44100
        FilterOrder: 6
                F0: 1000

```

```

biquad = design(d,'SystemObject',true)

biquad = BiquadFilter with properties:
    Structure: 'Direct form II'
    SOSMatrixSource: 'Property'
    SOSMatrix: [3x6 double]
    ScaleValues: [4x1 double]
    InitialConditions: 0
    OptimizeUnityScaleValues: true

Show all properties

```

Plot the magnitude response of the filter using `fvtool`. The logarithmic scale for frequency is automatically set by `fvtool` for the octave filters.

```
fvtool(biquad)
```

Filter a sinusoidal signal using the above designed filter. The input is a sum of two tones - one at 0.3 kHz and the other at 3 kHz. Initialize a spectrum analyzer to view the filtered output power spectrum.

```

Fs = 2500;
Sineobject1 = dsp.SineWave('SamplesPerFrame',1024,...
    'SampleRate',Fs,'Frequency',100);
Sineobject2 = dsp.SineWave('SamplesPerFrame',1024,...
    'SampleRate',Fs,'Frequency',1000);

sa = spectrumAnalyzer('SampleRate',Fs,'NumInputPorts',2,...
    'PlotAsTwoSidedSpectrum',false,'YLimits',[-250,50],...
    'ChannelNames',{'Input','FilteredOutput'},'ShowLegend',true);

```

Stream the sinusoidal signal and pass it as an input to the biquadratic filter. View the input and the filtered output power spectra using the spectrum analyzer.

```

for Iter = 1:3000
    Sinewave1 = Sineobject1();
    Sinewave2 = Sineobject2();
    Input = Sinewave1 + Sinewave2;
    filteredOutput = biquad(Input);
    sa(Input,filteredOutput);
end

```

## Version History

**Introduced in R2011a**

**R2022a: `fdesign.octave` will be removed**

*Warns starting in R2022a*

The `fdesign.octave` filter specification object will be removed in a future release. Existing instances of the object continue to run. For new code, use the `octaveFilter` System object instead.

### Update Code

This table shows how the object is typically used and explains how to update the existing code to use the `octaveFilter` object. The `octaveFilter` object requires Audio Toolbox.

| Discouraged Usage   | Recommended Replacement  |
|---|--|
| <p>Design a 6th order, 1/3rd octave-band Class 0 filter that is centered at 1000 Hz and has a sample rate of 44100 Hz.</p> <pre>Fs = 44100; d = fdesign.octave(3,'Class 0','N,F0',6,1000); octFilt = design(d,'Systemobject',true);</pre> | <p>Design a 6th order, 1/3rd octave-band Class 0 filter that is centered at 1000 Hz and has a sample rate of 44100 Hz.</p> <pre>Fs = 44100; octFilt = octaveFilter(1000,'1/3 octave',... 'SampleRate',Fs);</pre> |
| <p>Visualize the filter response and verify it fits within the Class 0 mask of the ANSI S1.11-2004 standard.</p> <pre>fvtool(octFilt,'Fs',44100,'FrequencyScale','log');</pre>  | <p>Visualize the filter response and verify it fits within the Class 0 mask of the ANSI S1.11-2004 standard.</p> <pre>visualize(octFilt,'class 0');</pre>  |

### See Also

#### Objects

`octaveFilter`

## fdesign.parmeq

(To be removed) Parametric equalizer filter specification

### Compatibility

---

**Note** The `fdesign.parmeq` filter specification object will be removed from DSP System Toolbox in a future release. Existing instances of the object continue to run. For new code, use the `designParamEQ` function instead. For more information on how to update your existing code, see “Compatibility Considerations” on page 5-561.

---

### Syntax

```
d = fdesign.parmeq(spec, specvalue1, specvalue2, ...)
d = fdesign.parmeq(... fs)
```

### Description

`d = fdesign.parmeq(spec, specvalue1, specvalue2, ...)` constructs a parametric equalizer filter design object, where `spec` is a non-case sensitive specification character vector. The choices for `spec` are as follows:

- 'F0, BW, BWp, Gref, G0, GBW, Gp' (minimum order default)
- 'F0, BW, BWst, Gref, G0, GBW, Gst'
- 'F0, BW, BWp, Gref, G0, GBW, Gp, Gst'
- 'N, F0, BW, Gref, G0, GBW'
- 'N, F0, BW, Gref, G0, GBW, Gp'
- 'N, F0, Fc, Qa, G0'
- 'N, F0, Fc, S, G0'
- 'N, F0, BW, Gref, G0, GBW, Gst'
- 'N, F0, BW, Gref, G0, GBW, Gp, Gst'
- 'N, Flow, Fhigh, Gref, G0, GBW'
- 'N, Flow, Fhigh, Gref, G0, GBW, Gp'
- 'N, Flow, Fhigh, Gref, G0, GBW, Gst'
- 'N, Flow, Fhigh, Gref, G0, GBW, Gp, Gst'

where the parameters are defined as follows:

- BW — Bandwidth
- BWp — Passband Bandwidth
- BWst — Stopband Bandwidth
- Gref — Reference Gain (decibels)
- G0 — Center Frequency Gain (decibels)

- GBW — Gain at which Bandwidth (BW) is measured (decibels)
- Gp — Passband Gain (decibels)
- Gst — Stopband Gain (decibels)
- N — Filter Order
- F0 — Center Frequency
- Fc— Cutoff frequency
- Fhigh - Higher Frequency at Gain GBW
- Flow - Lower Frequency at Gain GBW
- Qa-Quality Factor
- S-Slope Parameter for Shelving Filters

Regardless of the specification chosen, there are some conditions that apply to the specification parameters. These are as follows:

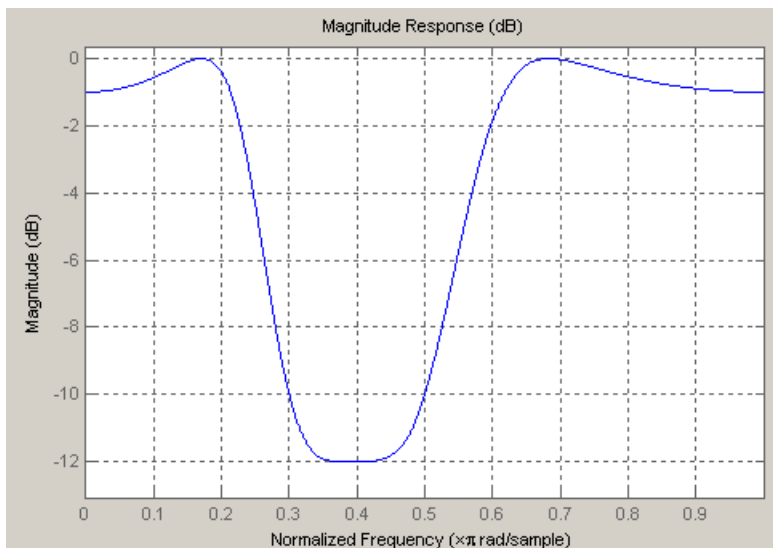
- Specifications for parametric equalizers must be given in decibels
- To boost the input signal, set  $G_0 > G_{ref}$ ; to cut, set  $G_{ref} > G_0$
- For boost:  $G_0 > G_p > GBW > G_{st} > G_{ref}$ ; For cut:  $G_0 < G_p < GBW < G_{st} < G_{ref}$
- Bandwidth must satisfy:  $BW_{st} > BW > BW_p$

`d = fdesign.parmeq(... fs)` adds the input sampling frequency. Fs must be specified as a scalar trailing the other numerical values provided, and is assumed to be in Hz.

## Examples

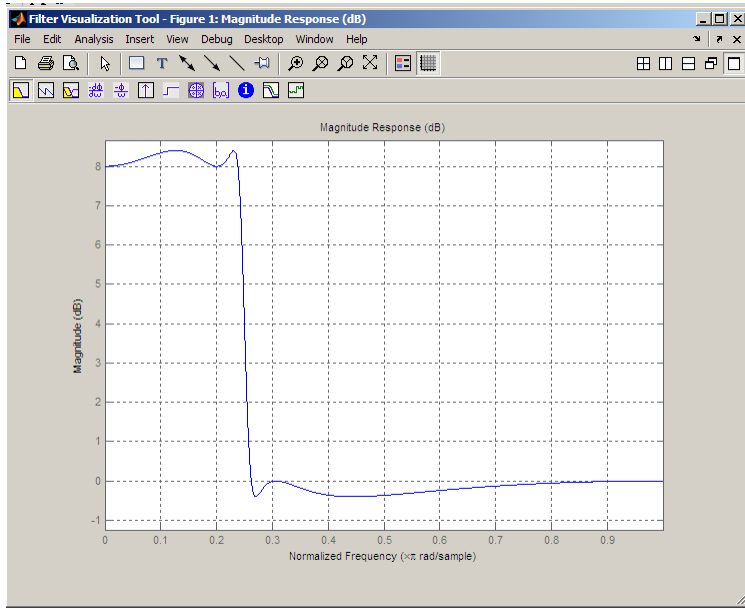
Design a Chebyshev Type II parametric equalizer filter that cuts by 12 dB:

```
d = fdesign.parmeq('N,Flow,Fhigh,Gref,G0,GBW,Gst',...
    4, .3, .5, 0, -12, -10, -1);
Hd = design(d, 'cheby2');
fvtool(Hd)
```



Design a 4th order audio lowpass ( $F_0 = 0$ ) shelving filter with cutoff frequency of  $F_c = 0.25$ , quality factor  $Q_a = 10$ , and boost gain of  $G_0 = 8$  dB:

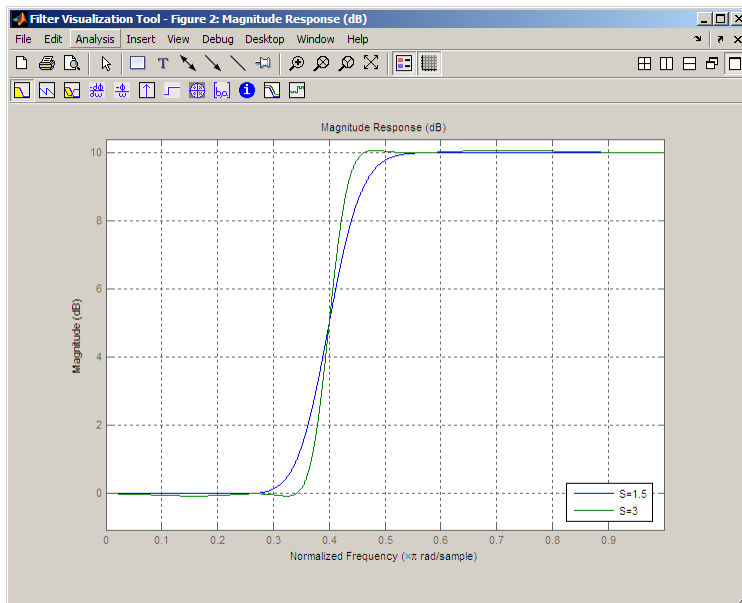
```
d = fdesign.parmeq('N,F0,Fc,Qa,G0',4,0,0.25,10,8);
Hd = design(d);
fvtool(Hd)
```



Design 4th-order highpass shelving filters with  $S=1.5$  and  $S=3$ :

```
N=4;
F0 = 1;
Fc = .4; % Cutoff Frequency
G0 = 10;
S = 1.5;
S2=3;
f = fdesign.parmeq('N,F0,Fc,S,G0',N,F0,Fc,S,G0);
h1 = design(f);
f.S=3;
h2=design(f);
hfvt=fvtool([h1 h2]);
set(hfvt,'Filters',[h1 h2]);
legend(hfvt,'S=1.5','S=3');
```





## Version History

Introduced in R2011a

### R2022a: fdesign.parmeq will be removed

Warns starting in R2022a

The `fdesign.parmeq` filter specification object will be removed in a future release. Existing instances of the object continue to run. For new code, use the `designParamEQ` function instead.

### Update Code

This table shows how the object is typically used and explains how to update the existing code to use the `designParamEQ` function. The `designParamEQ` function requires Audio Toolbox.

| Discouraged Usage  | Recommended Replacement   |
|--|---|
| <p><b>Design based on Filter Bandwidth</b></p> <pre> Fs = 48e3; N = 2; Q = 10; G = 9; % 9 dB  % Normalized center frequency Wo1 = 2000/(Fs/2); Wo2 = 12000/(Fs/2);  % Normalized bandwidth BW1 = Wo1/Q; BW2 = Wo2/Q;  PEQ = fdesign.paremeq('N,F0,BW,Gref,G0,GBW',N,F0,BW,Gref,G0,GBW); BQ1 = design(PEQ,'SystemObject',true);  PEQ.BW = BW2; PEQ.F0 = Wo2; BQ2 = design(PEQ,'SystemObject',true);  % Visualize the filters hfvt = fvtool(BQ1,BQ2,'Fs',Fs,'Color','white'); legend(hfvt,'BW1 = 200 Hz; Q = 10','BW2 = 1200 Hz; Q = 10');</pre>                           | <p><b>Design based on Filter Bandwidth</b></p> <pre> Fs = 48e3; N = 2; Q = 10; G = 9; % 9 dB  % Normalized center frequency Wo1 = 2000/(Fs/2); Wo2 = 12000/(Fs/2);  % Normalized bandwidth BW1 = Wo1/Q; BW2 = Wo2/Q;  [B1,A1] = designParamEQ(N,G,Wo1,BW1); [B2,A2] = designParamEQ(N,G,Wo2,BW2); BQ1 = dsp.BiquadFilter('SOSMatrix',[B1.',[1,A1.]]); BQ2 = dsp.BiquadFilter('SOSMatrix',[B2.',[1,A2.]]);  % Visualize the filters hfvt = fvtool(BQ1,BQ2,'Fs',Fs,'Color','white'); legend(hfvt,'BW1 = 200 Hz; Q = 10','BW2 = 1200 Hz; Q = 10');</pre> |
| <p><b>Design based on Quality factor</b></p> <pre> Fs = 48e3; N = 2; G = 15; % 15 dB  % Quality factor Q1 = 0.48; Q2 = 1/sqrt(2);  % Normalized center frequency % F0 = 1 designs a highpass filter % F0 can either be 0 or 1 in this configuration F0 = 1;  % Cutoff Frequency Fc = 6e3/(Fs/2);  PEQ = fdesign.paremeq('N,F0,Fc,Qa,G0',N,F0,Fc,Qa,G0); BQ1 = design(PEQ,'SystemObject',true);  PEQ.Qa = Q2; BQ2 = design(PEQ,'SystemObject',true);  % Visualize the filters hfvt = fvtool(BQ1,BQ2,'Fs',Fs,'Color','white'); legend(hfvt,'Q = 0.48','Q = 0.7071');</pre> | <p><b>Design based on Quality factor</b></p> <pre> Fs = 48e3; N = 2; G = 15; % 15 dB  % Quality factor Q1 = 0.48; Q2 = 1/sqrt(2);  % Normalized center frequency Wo = 6000/(Fs/2);  % Normalized bandwidth BW1 = Wo/Q1; BW2 = Wo/Q2;  [B1,A1] = designParamEQ(N,G,Wo,BW1); [B2,A2] = designParamEQ(N,G,Wo,BW2); BQ1 = dsp.BiquadFilter('SOSMatrix',[B1.',[1,A1.]]); BQ2 = dsp.BiquadFilter('SOSMatrix',[B2.',[1,A2.]]);  % Visualize the filters hfvt = fvtool(BQ1,BQ2,'Fs',Fs,'Color','white'); legend(hfvt,'Q = 0.48','Q = 0.7071');</pre>          |

| Discouraged Usage  | Recommended Replacement  |
|--|--|
| <p><b>Low shelf and high shelf filters</b></p> <pre> Fs = 48e3; N = 4; G = 10; % 10 dB  % Normalized center frequency Wo1 = 0; % Lowpass filter % Corresponds to Fs/2 (Hz) or pi (rad/sample) Wo2 = 1; % Highpass filter  % Bandwidth occurs at 7.4 dB in this case BW = 1000/(Fs/2);  PEQ = fdesign.parmeq('N,F0,BW,Gref,G0,GBW',N,F0,BW,Gref,G0,GBW); BQ1 = design(PEQ,'SystemObject',true);  PEQ.F0 = Wo2; BQ2 = design(PEQ,'SystemObject',true);  % Visualize the filters hfvt = fvtool(BQ1,BQ2,'Fs',Fs,'Color','white'); legend(hfvt,'Low Shelf Filter','High Shelf Filter');</pre> | <p><b>Low shelf and high shelf filters</b></p> <pre> Fs = 48e3; N = 4; G = 10; % 10 dB  % Normalized center frequency Wo1 = 0; % Lowpass filter % Corresponds to Fs/2 (Hz) or pi (rad/sample) Wo2 = 1; % Highpass filter  % Bandwidth occurs at 7.4 dB in this case BW = 1000/(Fs/2);  [B1,A1] = designParamEQ(N,G,Wo1,BW); [B2,A2] = designParamEQ(N,G,Wo2,BW); BQ1 = dsp.BiquadFilter('SOSMatrix',[B1.],[ones(2,1),A1]); BQ2 = dsp.BiquadFilter('SOSMatrix',[B2.],[ones(2,1),A2]);  % Visualize the filters hfvt = fvtool(BQ1,BQ2,'Fs',Fs,'Color','white'); legend(hfvt,'Low Shelf Filter','High Shelf Filter');</pre> |

## See Also

designParamEQ

## fdesign.peak

Peak filter specification

### Syntax

```
d = fdesign.peak(specstring, value1, value2, ...)  
d = fdesign.peak(n,f0,q)  
d = fdesign.peak(...,Fs)  
d = fdesign.peak(...,MAGUNITS)
```

### Description

`d = fdesign.peak(specstring, value1, value2, ...)` constructs a peaking filter specification object `d`, with specification set to `specstring` and values provided for all members of the `specstring`. The possible specification options, which are not case sensitive, are listed as follows:

- 'N,F0,Q' (default)
- 'N,F0,Q,Ap'
- 'N,F0,Q,Ast'
- 'N,F0,Q,Ap,Ast'
- 'N,F0,BW'
- 'N,F0,BW,Ap'
- 'N,F0,BW,Ast'
- 'N,F0,BW,Ap,Ast'

where the variables are defined as follows:

- N - Filter Order (must be even)
- F0 - Center Frequency
- Q - Quality Factor
- BW - 3-dB Bandwidth
- Ap - Passband Ripple (decibels)
- Ast - Stopband Attenuation (decibels)

Different specification options, resulting in different specification objects, may have different design methods available. Use the function `designmethods` to get a list of design methods available for a given specification. For example:

```
>> d = fdesign.peak('N,F0,Q,Ap',6,0.5,10,1);  
>> designmethods(d)
```

Design Methods for class `fdesign.peak (N,F0,Q,Ap)`:

`cheby1`

`d = fdesign.peak(n,f0,q)` constructs a peaking filter specification object using the default specstring ('N,F0,Q') and setting the corresponding values to `n`, `f0`, and `q`.

By default, all frequency specifications are assumed to be in normalized frequency units. All magnitude specifications are assumed to be in decibels.

`d = fdesign.peak(...,Fs)` constructs a peak filter specification object while providing the sampling frequency of the signal to be filtered. `Fs` must be specified as a scalar trailing the other values provided. If you specify an `Fs`, it is assumed to be in Hz, as all the other frequency values provided.

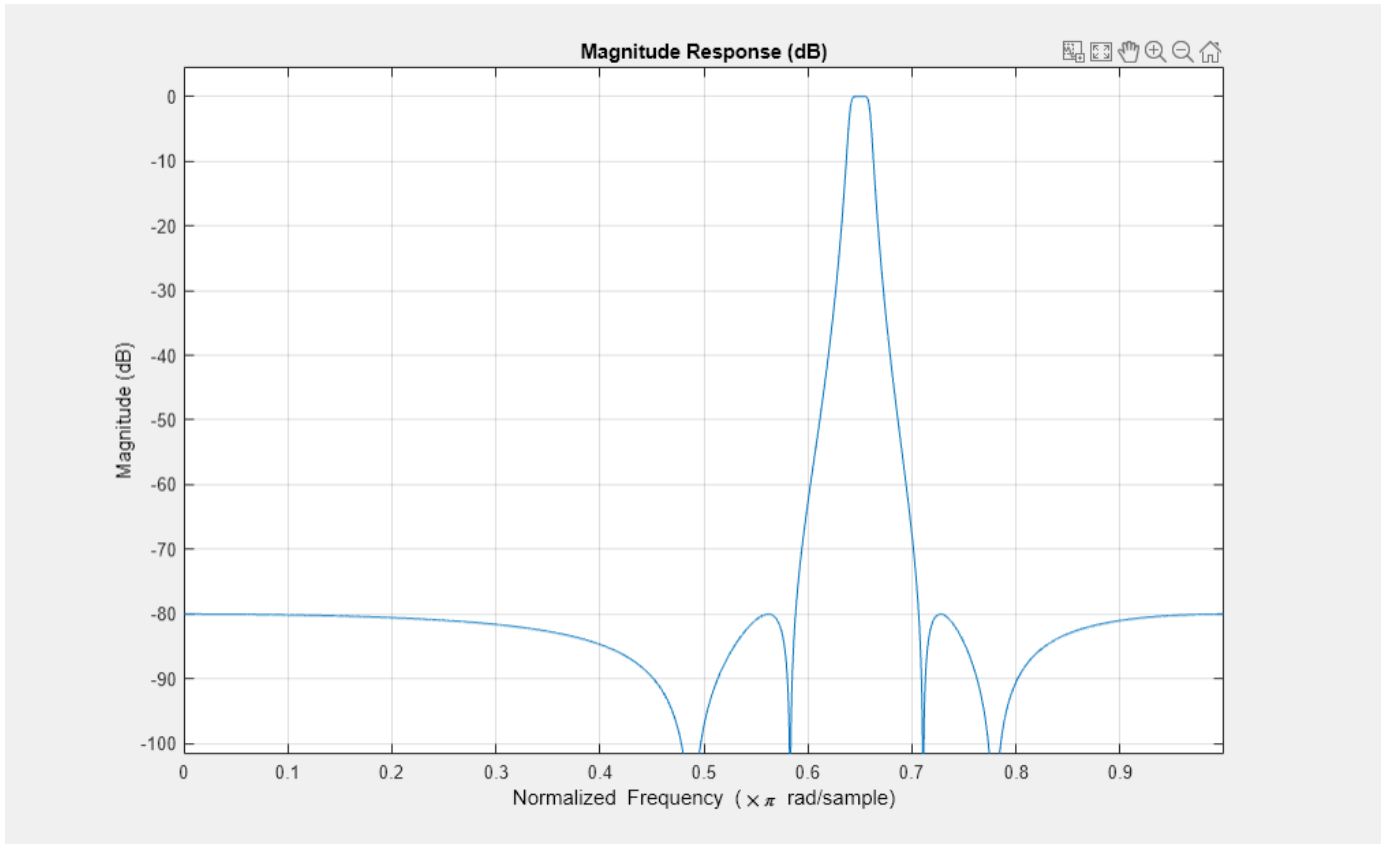
`d = fdesign.peak(...,MAGUNITS)` constructs a notch filter specification while providing the units for any magnitude specification given. `MAGUNITS` can be one of the following: 'linear', 'dB', or 'squared'. If this argument is omitted, 'dB' is assumed. The magnitude specifications are always converted and stored in decibels regardless of how they were specified. If `Fs` is provided, `MAGUNITS` must follow `Fs` in the input argument list.

## Examples

### Design a Chebyshev Type II Peaking Filter

Design a Chebyshev Type II peaking filter with a stopband attenuation of 80 dB.

```
d = fdesign.peak('N,F0,BW,Ast',8,.65,.02,80);  
Hd = design(d,'cheby2','SystemObject',true);  
fvtool(Hd)
```



## Version History

Introduced in R2011a

### See Also

`fdesign` | `fdesign.notch`

## fdesign.polysrc

Construct polynomial sample-rate converter (POLYSRC) filter designer

### Syntax

```
d = fdesign.polysrc(l,m)
d = fdesign.polysrc(l,m,'Fractional Delay','Np',Np)
d = fdesign.polysrc(...,Fs)
```

### Description

`d = fdesign.polysrc(l,m)` constructs a polynomial sample-rate converter filter designer `D` with an interpolation factor `L` and a decimation factor `M`. `L` defaults to 3. `M` defaults to 2. `L` and `M` can be arbitrary positive numbers.

`d = fdesign.polysrc(l,m,'Fractional Delay','Np',Np)` initializes the filter designer specification with `Np` and sets the polynomial order to the value `Np`. If omitted `Np` defaults to 3.

`d = fdesign.polysrc(...,Fs)` specifies the sampling frequency (in Hz).

### Examples

#### Design a Sample-Rate Converter

This example shows how to design sample-rate converter that uses a third order Lagrange interpolation filter to convert from 44.1kHz to 48kHz.

```
FsIn = 44.1e3;
FsOut = 48e3;
[L,M] = rat(FsOut/FsIn);
f = fdesign.polysrc(L,M,'Fractional Delay',Np=3);
FRCObj = design(f,'lagrange',SystemObject=true);
```

9408 samples of data is 0.2133 seconds long.

```
n = (0:9407)';
```

Input signal is a sinusoid at 1 kHz.

```
F0 = 1e3;
x = sin(2*pi*F0/FsIn*n);
```

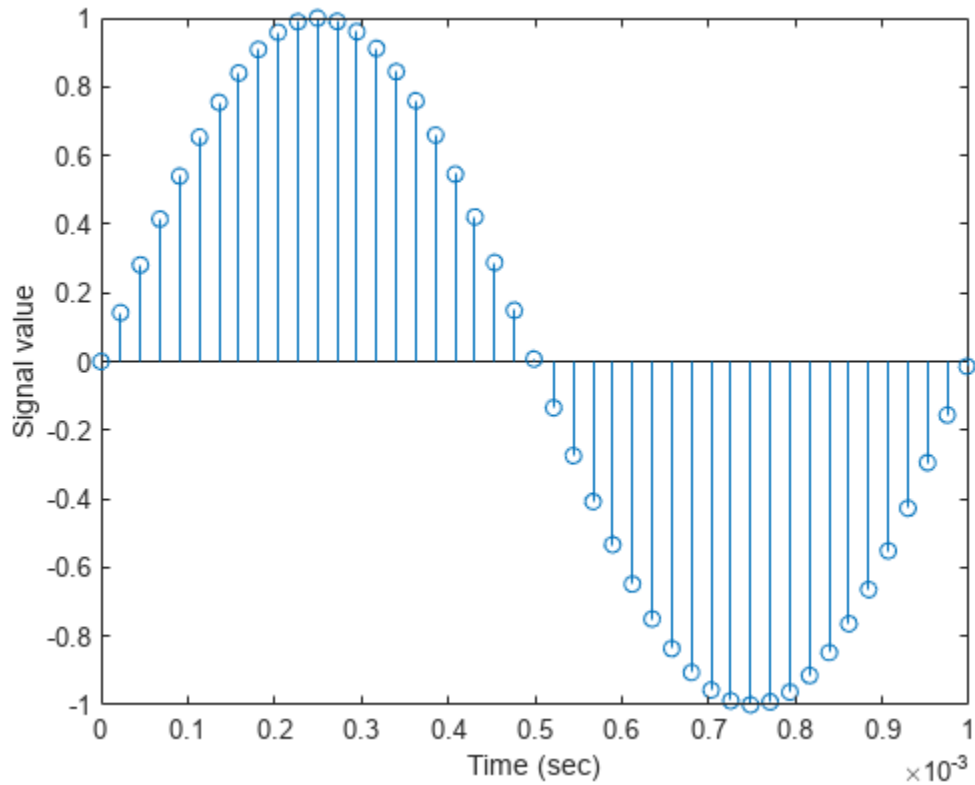
The output has 10241 samples (still 0.2133 seconds long).

```
y = FRCObj(x);
```

Plot one cycle of the input signal, sampled at 44.1kHz.

```
stem(n/FsIn, x);
hold on;
xlim([0, 1/F0])
```

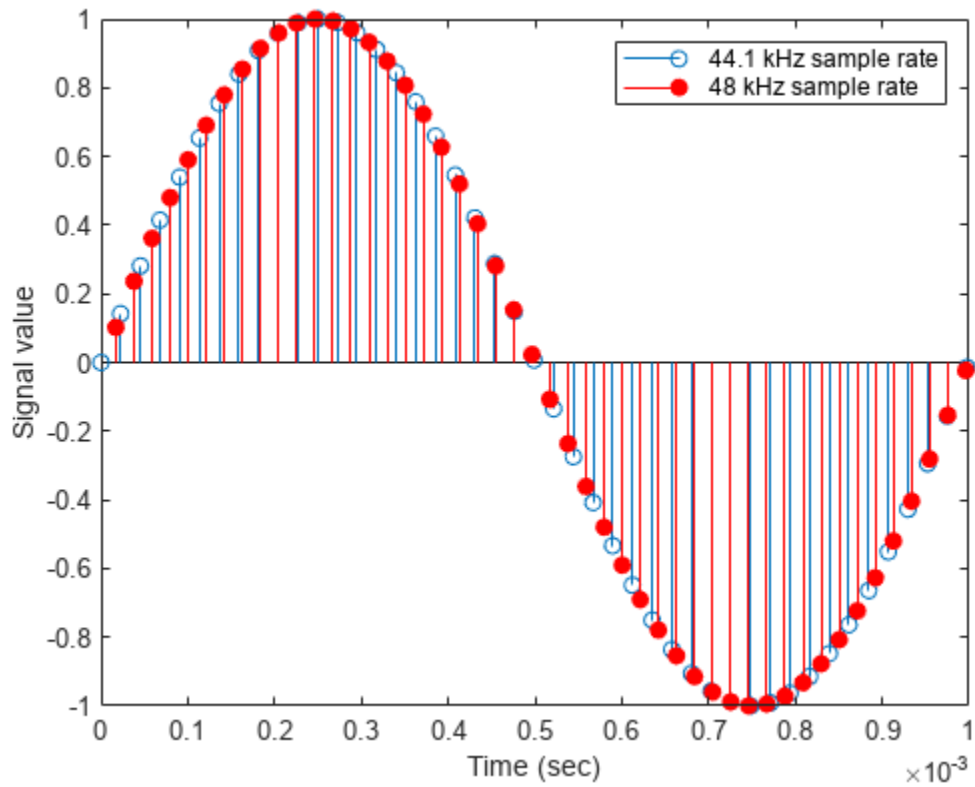
```
xlabel('Time (sec)');
ylabel('Signal value')
```



Plot fractionally interpolated signal (48kHz) in red.

```
D = outputDelay(FRCObj, Fc=0, FsIn=FsIn);
k = (0:length(y)-1);
stem(k/FsOut-D, y, 'r', 'filled');
hold off;
legend('44.1 kHz sample rate', '48 kHz sample rate')
```





## Version History

Introduced in R2011a

## See Also

fdesign

## fdesign.pulseshaping

(To be removed) Pulse-shaping filter specification object

---

**Note** The `fdesign.pulseshaping` filter specification object will be removed in a future release. Use `rcosdesign` or `gaussdesign` instead. For more information on how to update your existing code, see “Compatibility Considerations”.

---

### Syntax

```
D = fdesign.pulseshaping
D = fdesign.pulseshaping(sps)
D = fdesign.pulseshaping(sps,shape)
d = fdesign.pulseshaping(sps,shape,spec,value1,value2,...)
d = fdesign.pulseshaping(...,fs)
d = fdesign.pulseshaping(...,magunits)
```

### Description

`D = fdesign.pulseshaping` constructs a specification object `D`, which can be used to design a minimum-order raised cosine filter object with a default stop band attenuation of 60 dB and a rolloff factor of 0.25.

`D = fdesign.pulseshaping(sps)` constructs a minimum-order raised cosine filter specification object `d` with a positive integer-valued oversampling factor, `SamplesPerSymbol`.

`D = fdesign.pulseshaping(sps,shape)` constructs `d` where `shape` specifies the `PulseShape` property. Valid entries for `shape` are:

- 'Raised Cosine'
- 'Square Root Raised Cosine'
- 'Gaussian'

`d = fdesign.pulseshaping(sps,shape,spec,value1,value2,...)` constructs `d` where `spec` defines the `Specification` properties. The entries for `spec` specify various properties of the filter, including the order and frequency response. Valid entries for `spec` depend upon the `shape` property. For 'Raised Cosine' and 'Square Root Raised Cosine' filters, the valid entries for `spec` are:

- 'Ast,Beta' (minimum order; default)
- 'Nsym,Beta'
- 'N,Beta'

The filter specifications are defined as follows:

- `Ast` —stopband attenuation (in dB). The default stopband attenuation for a raised cosine filter is 60 dB. The default stopband attenuation for a square root raised cosine filter is 30 dB. If `Ast` is specified, the minimum-order filter is returned.

- **Beta** —rolloff factor expressed as a real-valued scalar ranging from 0 to 1. Smaller rolloff factors result in steeper transitions between the passband and stopband of the filter.
- **Nsym** —filter order in symbols. The length of the impulse response is given by  $Nsym * SamplesPerSymbol + 1$ . The product  $Nsym * SamplesPerSymbol$  must be even.
- **N** —filter order (must be even). The length of the impulse response is  $N + 1$ .

If the `shape` property is specified as `'Gaussian'`, the valid entries for `spec` are:

- `'Nsym,BT'` (default)

The filter specifications are defined as follows:

- **Nsym**—filter order in symbols. `Nsym` defaults to 6. The length of the filter impulse response is  $Nsym * SamplesPerSymbol + 1$ . The product  $Nsym * SamplesPerSymbol$  must be even.
- **BT** —the 3 dB bandwidth-symbol time product. `BT` is a positive real-valued scalar, which defaults to 0.3. Larger values of `BT` produce a narrower pulse width in time with poorer concentration of energy in the frequency domain.

`d = fdesign.pulseshaping(...,fs)` specifies the sampling frequency of the signal to be filtered. `fs` must be specified as a scalar trailing the other numerical values provided. For this case, `fs` is assumed to be in Hz and is used for analysis and visualization.

`d = fdesign.pulseshaping(...,magunits)` specifies the units for any magnitude specification you provide in the input arguments. Valid entries for `magunits` are:

- `linear` — specify the magnitude in linear units
- `dB` — specify the magnitude in dB (decibels)
- `squared` — specify the magnitude in power units

When you omit the `magunits` argument, `fdesign` assumes that all magnitudes are in decibels. Note that `fdesign` stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

After creating the specification object `d`, you can use the `design` function to create a filter object such as `h` in the following example:

```
d = fdesign.pulseshaping(8,'Raised Cosine','Nsym,Beta',6,0.25);
h = design(d);
```

Normally, the `Specification` property of the specification object also determines which design methods you can use when you create the filter object. Currently, regardless of the `Specification` property, the `design` function uses the `window` design method with all `fdesign.pulseshaping` specification objects. The `window` method creates an FIR filter with a windowed impulse response.

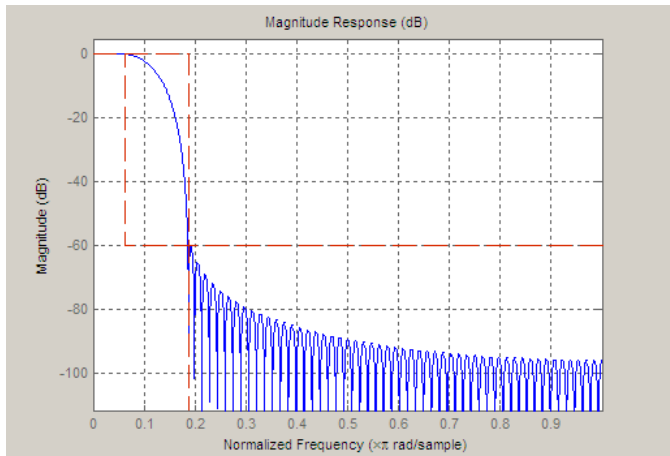
## Examples

Pulse-shaping can be used to change the waveform of transmitted pulses so the signal bandwidth matches that of the communication channel. This helps to reduce distortion and intersymbol interference (ISI).

This example shows how to design a minimum-order raised cosine filter that provides a stop band attenuation of 60 dB, rolloff factor of 0.50, and 8 samples per symbol.

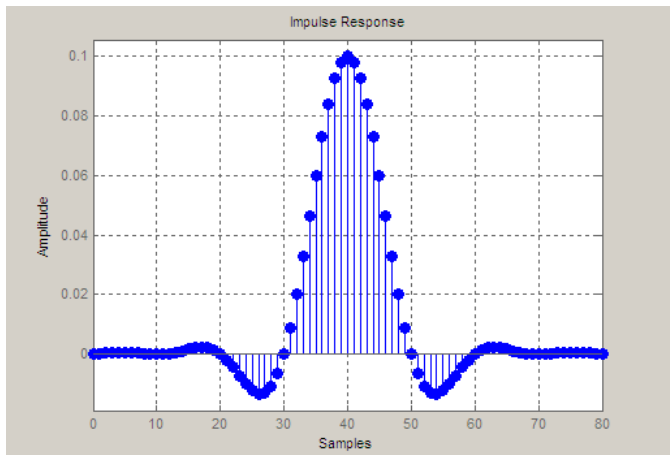
```
h = fdesign.pulseshaping(8,'Raised Cosine','Ast,Beta',60,0.50);
Hd = design(h);
fvtool(Hd)
```

This code generates the following figure.



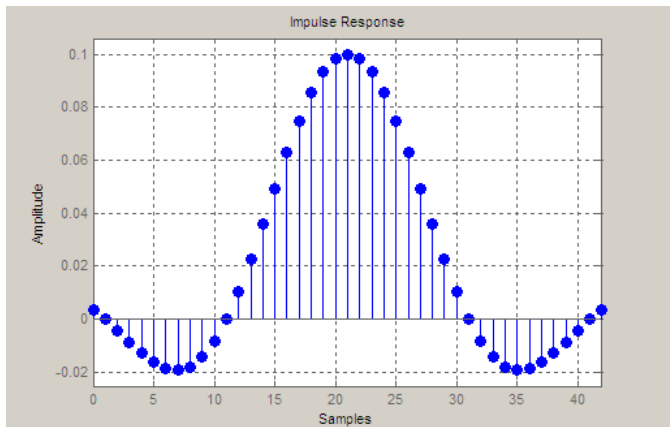
This example shows how to design a raised cosine filter that spans 8 symbol durations (i.e., of order 8 symbols), has a rolloff factor of 0.50, and oversampling factor of 10.

```
h = fdesign.pulseshaping(10,'Raised Cosine','Nsym,Beta',8,0.50);
Hd = design(h);
fvtool(Hd, 'impulse')
```

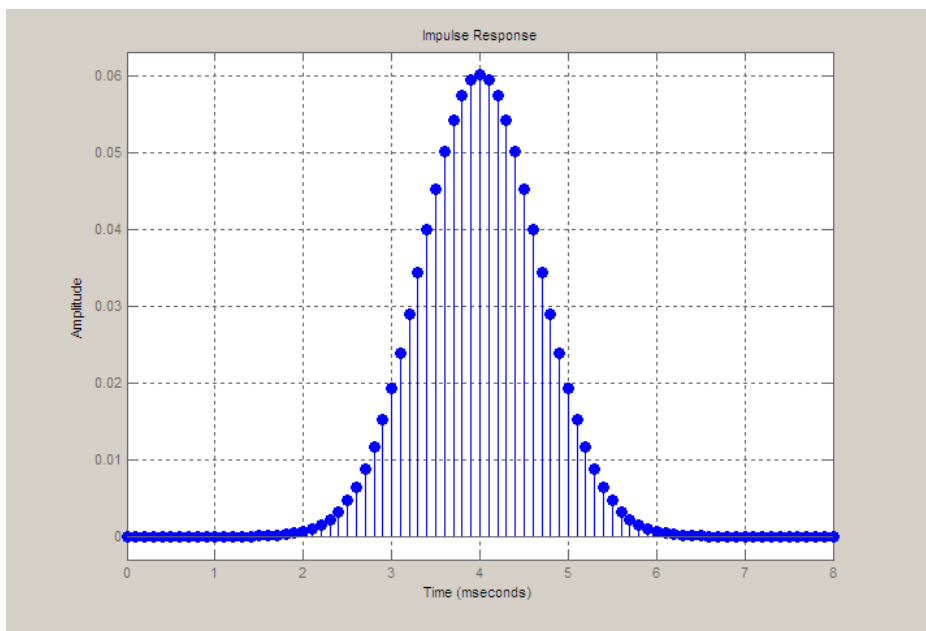


This example shows how to design a square root raised cosine filter of order 42, rolloff factor of 0.25, and 10 samples per symbol.

```
h = fdesign.pulseshaping(10,'Square Root Raised Cosine','N,Beta',42);
Hd = design(h);
fvtool(Hd, 'impulse')
```



The following example demonstrates how to create a Gaussian pulse-shaping filter with an oversampling factor (sps) of 10, a bandwidth-time symbol product of 0.2, and 8 symbol periods. The sampling frequency is specified as 10 kHz.



## Version History

Introduced in R2011a

**R2022a: fdesign.pulseshaping will be removed**

*Warns starting in R2022a*

The `fdesign.pulseshaping` filter specification object will be removed in a future release. Existing instances of the object continue to run. For new code, use the `rcosdesign` or `gaussdesign` functions instead.

**Update Code**

This table shows how the object is typically used and explains how to update the existing code to use the `rcosdesign` and `gaussdesign` functions.

| Discouraged Usage   | Recommended Replacement  |
|---|--|
| <p><b>Square Root Raised Cosine Filter</b></p> <p>Design a square root raised cosine FIR filter than spans 4 symbol durations and has a roll-off factor of 0.25 and an oversampling factor of 6.</p> <pre>d = fdesign.pulseshaping(4, 'Square Root Raised Cosine',     'Nsym,Beta',6,0.25); filt = design(d, 'Systemobject', true); fvtool(filt, 'impulse')</pre> | <p><b>Square Root Raised Cosine Filter</b></p> <p>Design a square root raised cosine FIR filter than spans 4 symbol durations and has a roll-off factor of 0.25 and an oversampling factor of 6.</p> <pre>filtCoeffs = rcosdesign(0.25,6,4, 'sqrt'); fvtool(filtCoeffs, 'Analysis', 'impulse')</pre> |
| <p><b>Gaussian windowed FIR filter</b></p> <p>Design a Gaussian-window FIR filter of order 3 symbols, with the bandwidth-symbol time product of 0.4, and with 10 samples per symbol.</p> <pre>d = fdesign.pulseshaping(10, 'Gaussian', ...     'Nsym,BT',3,0.4); filt = design(d, 'Systemobject', true); fvtool(filt)</pre>                                       | <p><b>Gaussian windowed FIR filter</b></p> <p>Design a Gaussian-window FIR filter of order 3 symbols, with the bandwidth-symbol time product of 0.4, and with 10 samples per symbol.</p> <pre>filtCoeffs = gaussdesign(0.4,3,10) fvtool(filtCoeffs)</pre>  |

## See Also

### Functions

`rcosdesign` | `gaussdesign`

# fdesign.rsrc

Rational-factor sample-rate converter specification

---

## Note

**Note** Support for designing an `mfilt.firsrc` object using the `fdesign.rsrc` and the `design` functions will be removed in a future release. Use the `design` function with the `SystemObject=true` flag to design an FIR rate converter System object™. For more information, see “Compatibility Considerations”.

---

## Syntax

```
D = fdesign.rsrc(L,M)
D = fdesign.rsrc(L,M,RESPONSE)
D = fdesign.rsrc(L,M,CICRESPONSE,D)
D = fdesign.rsrc(L,M,RESPONSE,SPEC)
D = fdesign.rsrc(L,M,SPEC,specvalue1,specvalue2,...)
D = fdesign.rsrc(...,Fs)
D = fdesign.rsrc(...,MAGUNITS)
```

## Description

`D = fdesign.rsrc(L,M)` constructs a rational-factor sample-rate filter specification object `D` with the `InterpolationFactor` property equal to the positive integer `L`, the `DecimationFactor` property equal to the positive integer `M` and the `Response` property set to `'Nyquist'`. The default values for the transition width and stopband attenuation in the Nyquist design are  $0.1\pi$  radians/sample and 80 dB. If `L` is unspecified, `L` defaults to 3. If `M` is unspecified, `M` defaults to 2.

`D = fdesign.rsrc(L,M,RESPONSE)` constructs an rational-factor sample-rate converter with the interpolation factor `L`, decimation factor `M`, and the response you specify in `RESPONSE`.

`D = fdesign.rsrc(L,M,CICRESPONSE,D)` constructs a CIC or CIC compensator rational-factor sample-rate convertor filter specification object with the `'RESPONSE'` property equal to `'CIC'` or `'CICCOMP'`. `D` is the differential delay. The differential delay, `D`, must precede the filter specification.

Because you are designing multirate filters, the specification options available are not the same as the specification options for designing single-rate filters. The interpolation and decimation factors are not included in the specification. Different filter responses support different specifications. The following table lists the supported response types and specification options. These options are not case sensitive.

| Design Method                   | Valid Specification Options  |
|---------------------------------|--|
| 'Arbitrary Magnitude'           | See <code>fdesign.arbmag</code> for a description of the specification entries. <ul style="list-style-type: none"> <li>'N,F,A' (default option)</li> <li>'N,B,F,A'</li> </ul>  |
| 'Arbitrary Magnitude and Phase' | See <code>fdesign.arbmagnphase</code> for a description of the specification entries. <ul style="list-style-type: none"> <li>'N,F,H' (default option)</li> <li>'N,B,F,H'</li> </ul>  |
| 'Bandpass'                      | See <code>fdesign.bandpass</code> for a description of the specification entries. <ul style="list-style-type: none"> <li>'Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2' (default option)</li> <li>'N,Fc1,Fc2'</li> <li>'N,Fst1,Fp1,Fp2,Fst2'</li> </ul>  |
| 'Bandstop'                      | See <code>fdesign.bandstop</code> for a description of the specification entries. <ul style="list-style-type: none"> <li>'N,Fc1,Fc2'</li> <li>'N,Fp1,Fst1,Fst2,Fp2'</li> <li>'Fp1,Fst1,Fst2,Fp2,Ap1,Ast,Ap2' (default option)</li> </ul>   |
| 'CIC'                           | 'Fp,Fst,Ap,Ast' — Only valid specification. Fp is the passband frequency, Fst is the stopband frequency, Ap is the passband ripple, and Ast is the stopband attenuation in decibels.<br><br>To specify a CIC rational-factor sample-rate convertor, include the differential delay after 'CIC' and before the filter specification: 'Fp,Ast'. For example:<br><code>d = fdesign.rsrc(2,2,'cic',4);</code>  |
| 'CIC Compensator'               | See <code>fdesign.ciccomp</code> for a description of the specification entries. <ul style="list-style-type: none"> <li>'Fp,Fst,Ap,Ast' (default option)</li> <li>'N,Fc,Ap,Ast'</li> <li>'N,Fp,Ap,Ast'</li> <li>'N,Fp,Fst'</li> <li>'N,Fst,Ap,Ast'</li> </ul><br>To specify a CIC compensator rational-factor sample-rate convertor, include the differential delay after 'CICCOMP' and before the filter specification. For example:<br><code>d = fdesign.rsrc(2,2,'ciccomp',4);</code> |
| 'Differentiator'                | 'N' — filter order   |
| 'Gaussian'                      | 'Nsym,BT — Nsym is the filter order in symbols and BT is the bandwidth-symbol time product.<br><br>The filter specification must be preceded by an integer-valued <code>SamplesPerSymbol</code> .  |



| Design Method           | Valid Specification Options   |
|-------------------------|---|
| 'Halfband'              | <p>See <code>fdesign.halfband</code> for a description of the specification entries.</p> <ul style="list-style-type: none"> <li>• 'TW,Ast' (default option)</li> <li>• 'N,TW'</li> <li>• 'N'</li> <li>• 'N,Ast'</li> </ul> <p>If you use the quasi-linear IIR design method, <code>iirlinphase</code>, with a halfband specification, the interpolation factor must be 2.</p> |
| 'Highpass'              | <p>See <code>fdesign.highpass</code> for a description of the specification entries.</p> <ul style="list-style-type: none"> <li>• 'Fst,Fp,Ast,Ap' (default option)</li> <li>• 'N,F3db'</li> <li>• 'N,Fc'</li> <li>• 'N,Fc,Ast,Ap'</li> <li>• 'N,Fp,Ast,Ap'</li> <li>• 'N,Fst,Ast,Ap'</li> <li>• 'N,Fst,Fp'</li> <li>• 'N,Fst,Ast,Ap'</li> <li>• 'N,Fst,Fp,Ast'</li> </ul>     |
| 'Hilbert'               | <p>See <code>fdesign.hilbert</code> for a description of the specification entries.</p> <ul style="list-style-type: none"> <li>• 'N,TW' (default option)</li> <li>• 'TW,Ap'</li> </ul>  |
| 'Inverse-sinc Lowpass'  | <p>See <code>fdesign.isinclp</code> for a description of the specification entries.</p> <ul style="list-style-type: none"> <li>• 'Fp,Fst,Ap,Ast' (default option)</li> <li>• 'N,Fc,Ap,Ast'</li> <li>• 'N,Fp,Fst'</li> <li>• 'N,Fst,Ap,Ast'</li> </ul>   |
| 'Inverse-sinc Highpass' | <p>See <code>fdesign.isinchp</code> for a description of the specification entries.</p> <ul style="list-style-type: none"> <li>• 'Fst,Fp,Ast,Ap' (default option)</li> <li>• 'N,Fc,Ast,Ap'</li> <li>• 'N,Fst,Fp'</li> <li>• 'N,Fst,Ast,Ap'</li> </ul>   |

| Design Method | Valid Specification Options  |
|---------------|--|
| 'Lowpass'     | See <code>fdesign.lowpass</code> for a description of the specification entries. <ul style="list-style-type: none"> <li>'Fp,Fst,Ap,Ast' (default option)</li> <li>'N,F3dB'</li> <li>'N,Fc'</li> <li>'N,Fc,Ap,Ast'</li> <li>'N,Fp,Ap,Ast'</li> <li>'N,Fp,Fst'</li> <li>'N,Fp,Fst,Ap'</li> <li>'N,Fp,Fst,Ast'</li> <li>'N,Fst,Ap,Ast'</li> </ul>   |
| 'Nyquist'     | See <code>fdesign.nyquist</code> for a description of the specification entries. For all Nyquist specifications, you must specify the <i>L</i> th band. This typically corresponds to the interpolation factor so that the nonzero samples of the upsampler output are preserved. <ul style="list-style-type: none"> <li>'TW,Ast' (default option)</li> <li>'N'</li> <li>'N,Ast'</li> <li>'N,Ast'</li> </ul> |

`D = fdesign.rsrc(L,M,RESPONSE,SPEC)` constructs object `D` and sets its `Specification` property to `SPEC`. Entries in the `SPEC` represent various filter response features, such as the filter order, that govern the filter design. Valid entries for `SPEC` depend on the design type of the specifications object.

When you add the `SPEC` input argument, you must also add the `RESPONSE` input argument.

`D = fdesign.rsrc(L,M,SPEC,specvalue1,specvalue2,...)` constructs an object `D` and sets its specifications at construction time.

`D = fdesign.rsrc(...,Fs)` provides the sampling frequency of the signal to be filtered. `Fs` must be specified as a scalar trailing the other numerical values provided. `Fs` is assumed to be in Hz as are all other frequency values provided.

`D = fdesign.rsrc(...,MAGUNITS)` specifies the units for any magnitude specification you provide in the input arguments. `MAGUNITS` can be one of

- 'linear' — specify the magnitude in linear units.
- 'dB' — specify the magnitude in dB (decibels).
- 'squared' — specify the magnitude in power units.

When you omit the `MAGUNITS` argument, `fdesign` assumes that all magnitudes are in decibels. Note that `fdesign` stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

## Examples

### Construct a Sample-Rate Converter Using fdesign Object

Design a rational-factor sample-rate converter. Set the rational sample-rate change to 5/3. Use the default Nyquist design with a transition width of  $0.05\pi$  radians/sample and stopband attenuation of 40 dB. The Lth band factor in the Nyquist design is equal to the interpolation factor.

```
d = fdesign.rsrc(5,3,'nyquist',5,.05,40);
%design with Kaiser window
hm = design(d,'kaiserwin','SystemObject',true);
```

Design a rational-factor sample-rate converter. Set the rational sample-rate change to 5/3. Use a Nyquist design with the filter specification set to 'N,TW'. Set the order equal to 12 and the transition width to  $0.1\pi$  radians/sample. The Lth band factor in the Nyquist design is equal to the interpolation factor.

```
d = fdesign.rsrc(5,3,'nyquist',5,'N,TW',12,0.1); %#ok
```

Design a rational-factor sample-rate converter. Assume the data are sampled at 10 kHz. Set the rational sample-rate change to 3/2. Use a Nyquist design with the filter specification set to 'N,TW'. Set the order equal to 12 and the transition width to 100 Hz. The Lth band factor in the Nyquist design is equal to the interpolation factor.

```
d = fdesign.rsrc(3,2,'nyquist',3,'N,TW',12,100,1e4);
hd = design(d,'equiripple','SystemObject',true);
```

## Version History

### Introduced in R2011a

### R2022b: Support for designing mfilt.firsrc will be removed

*Warns starting in R2022b*

Support for designing an `mfilt.firsrc` object using the `fdesign.rsrc` and the `design` functions will be removed in a future release.

To design an FIR rate converter using the `fdesign.rsrc` filter specification object, use the `design` function with the `SystemObject=true` flag. Setting this flag to `true` ensures that the output object is a filter System object instead of an `mfilt.firsrc` object.

```
fdesignObj = fdesign.rsrc;
FIRRCobj = design(fdesignObj,SystemObject=true)
```

```
FIRRCobj =
```

```
    dsp.FIRRateConverter with properties:
```

```
    InterpolationFactor: 3
    DecimationFactor: 2
    NumeratorSource: 'Property'
    Numerator: [-3.2490e-05 0 9.1277e-05 1.3523e-04 ... ]
```

```
Show all properties
```

**See Also**

`design` | `designmethods` | `fdesign.interpolator` | `setspecs` | `fdesign.arbmag` |  
`fdesign.arbmagnphase`

## fftcoeffs

Frequency-domain coefficients

### Syntax

```
c = fftcoeffs(hd)
```

### Description

`c = fftcoeffs(hd)` return the frequency-domain coefficients used when filtering with the `dfilt.fftfilter` object. `c` contains the coefficients

### Examples

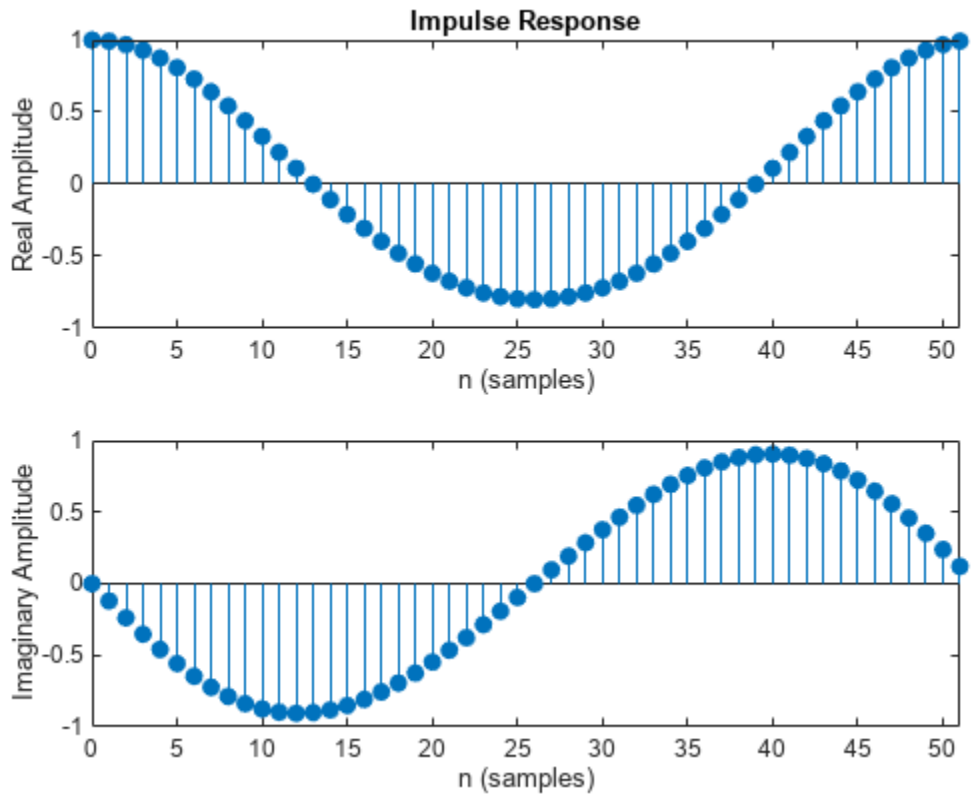
#### Frequency Domain Coefficients of FIR Filter

This example demonstrates returning the FFT coefficients from the discrete-time filter `hd`.

```
b = [0.05 0.9 0.05];  
len = 50;  
hd = dfilt.fftfilter(b, len);  
c = fftcoeffs(hd);
```

Plot the impulse response of the coefficients.

```
impz(c);
```



## Version History

Introduced in R2011a

# filterBuilder

Interactive filter design

## Syntax

```
filterBuilder(h)
filterBuilder('response')
```

## Description

`filterBuilder` starts an interactive tool for building filters. It relies on the `fdesign` object-oriented filter design paradigm, and is intended to reduce development time during the filter design process. `filterBuilder` uses a specification-centered approach to find the best algorithm for the desired response.

---

**Note** You must have the Signal Processing Toolbox installed to use `fdesign` and `filterBuilder`. Some of the features described below may be unavailable if your installation does not additionally include the DSP System Toolbox. You can verify the presence of both toolboxes by typing `ver` at the command prompt.

---

For more information on how to use `filterBuilder`, see “Filter Builder Design Process”.

To use `filterBuilder`, enter `filterBuilder` at the MATLAB command line using one of three approaches:

- Simply enter `filterBuilder`. MATLAB opens a dialog for you to select a filter response type. After you select a filter response type, `filterBuilder` launches the appropriate filter design dialog box.
- Enter `filterBuilder(h)`, where `h` is an existing filter object. For example, if `h` is a bandpass filter, `filterBuilder(h)` opens the bandpass filter design dialog box. The `h` object must have been created using `filterBuilder` or using `fdesign`.

---

**Note** You must have the DSP System Toolbox software to create and import filter System objects.

- Enter `filterBuilder('response')` to replace `response` with a response method from the following table. MATLAB opens a filter design dialog that corresponds to the specified response.

---

**Note** You must have the DSP System Toolbox software to implement a number of the filter designs listed in the following table. If you only have the Signal Processing Toolbox software, you can design a limited set of the following filter-response types.

---

| Response Method           | Description of Resulting Filter Design | Filter Object                    |
|---------------------------|--|----------------------------------|
| arbgrpdelay on page 5-594 | Arbitrary group delay filter design    | <code>fdesign.arbgrpdelay</code> |

| Response Method  | Description of Resulting Filter Design          | Filter Object  |
|--|---|--|
| arbmag on page 5-594   | Arbitrary magnitude filter design               | fdesign.arbmag   |
| arbmagnphase on page 5-594   | Arbitrary response filter (magnitude and phase) | fdesign.arbmagnphase   |
| audioweighting on page 5-597   | Audio weighting filter                          | fdesign.audioweighting   |
| bandpass on page 5-598 or bp   | Bandpass filter                                 | fdesign.bandpass   |
| bandstop on page 5-602 or bs   | Bandstop filter                                 | fdesign.bandstop   |
| cic on page 5-607  | CIC filter                                      | fdesign.decimator(M, 'cic', ...) or<br>fdesign.interpolator(L, 'cic', ...)<br>See fdesign.decimator and fdesign.interpolator |
| ciccomp on page 5-608  | CIC compensator                                 | fdesign.ciccomp  |
| comb on page 5-611   | Comb filter                                     | fdesign.comb   |
| diff on page 5-613   | Differentiator filter                           | fdesign.differentiator   |
| fracdelay on page 5-616  | Fractional delay filter                         | fdesign.fracdelay  |
| halfband on page 5-616 or hb   | Halfband filter                                 | fdesign.halfband   |
| highpass on page 5-619 or hp   | Highpass filter                                 | fdesign.highpass   |
| hilb on page 5-623   | Hilbert filter                                  | fdesign.hilbert  |
| isinc on page 5-626, isinclp on page 5-626, or isinchp on page 5-626 | Inverse sinc lowpass or highpass filter         | fdesign.isinclp and fdesign.isinchp  |
| lowpass on page 5-630 or lp  | Lowpass filter (default)                        | fdesign.lowpass  |
| notch on page 5-634  | Notch filter                                    | fdesign.notch  |
| nyquist on page 5-634  | Nyquist filter                                  | fdesign.nyquist  |
| octave on page 5-638   | Octave filter                                   | fdesign.octave   |
| parameq on page 5-639  | Parametric equalizer filter                     | fdesign.parameq  |
| peak on page 5-642   | Peak filter                                     | fdesign.peak   |

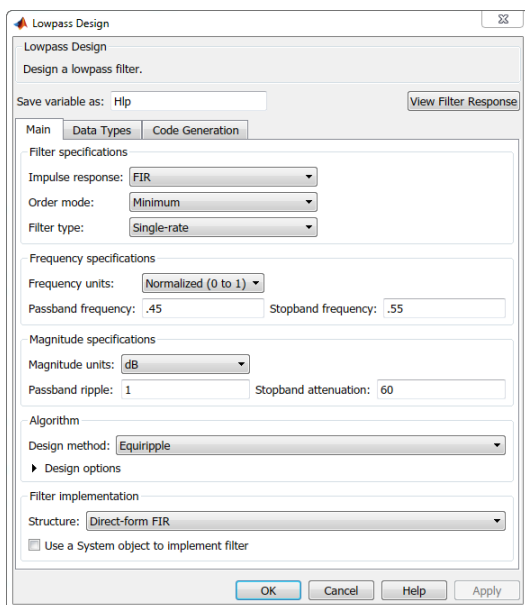
**Note** Because they do not change the filter structure, the magnitude specifications and design method are tunable when using `filterBuilder`.

## Filter Builder Design Panes

### Main Design Pane

The main pane of Filter Builder varies depending on the filter response type, but the basic structure is the same. The following figure shows the basic layout of the dialog box.





As you choose the response for the filter, the available options and design parameters displayed in the dialog box change. This display allows you to focus only on parameters that make sense in the context of your filter design.

Every filter design dialog box includes the options displayed at the top of the dialog box, shown in the following figure.



- **Save variable as** — When you click **Apply** to apply your changes or **OK** to close this dialog box, filterBuilder saves the current filter to your MATLAB workspace as a filter object with the name you enter.
- **View Filter Response** — Displays the magnitude response for the current filter specifications and design method by opening the Filter Visualization Tool (**FVTool**). If you specify the input sample rate, then **FVTool** plots the magnitude response as a function of frequency in Hz. If you do not specify the input sample rate, then **FVTool** plots the magnitude response as a function of normalized frequency in rad/sample.

---

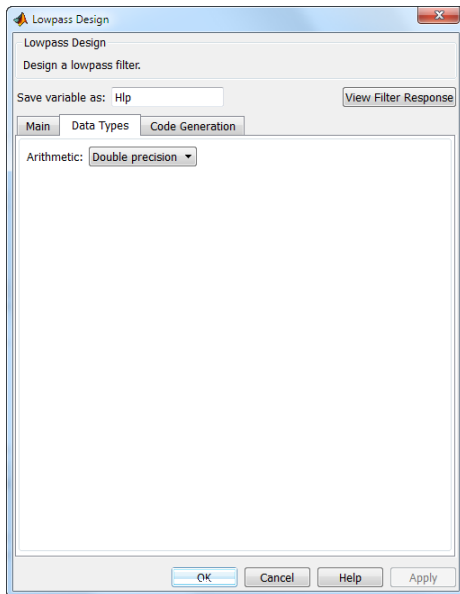
**Note** The filterBuilder dialog box includes an **Apply** option. Each time you click **Apply**, filterBuilder writes the modified filter to your MATLAB workspace. This modified filter has the variable name you assign in **Save variable as**. To apply changes without overwriting the variable in your workspace, change the variable name in **Save variable as** before you click **Apply**.

---

There are three tabs in the Filter Builder dialog box, containing three panes: **Main**, **Data Types**, and **Code Generation**. The first pane changes according to the filter being designed. The last two panes are the same for all filters. These panes are discussed in the following sections.

### Data Types Pane

The second tab in the Filter Builder dialog box is shown in the following figure.



The **Arithmetic** drop down box allows the choice of **Double precision**, **Single precision**, or **Fixed point**. Some of these options may be unavailable depending on the filter parameters. The following table describes these options.

| Arithmetic List Entry | Effect on the Filter  |
|-----------------------|---|
| Double precision      | All filtering operations and coefficients use double-precision, floating-point representations and math. When you use <code>filterBuilder</code> to create a filter, <code>double precision</code> is the default value for the <code>Arithmetic</code> property.   |
| Single precision      | All filtering operations and coefficients use single-precision floating-point representations and math.   |
| Fixed point           | This entry applies selected default values, typically used on many digital processors, for the properties in the fixed-point filter. These properties include coefficient word lengths, fraction lengths, and various operating modes. This setting allows signed fixed data types only. Fixed-point filter design with <code>filterBuilder</code> is available only when you install Fixed-Point Designer software along with DSP System Toolbox software. |

The following figure shows the **Data Types** pane after you select **Fixed point** for **Arithmetic** and set **Filter internals** to **Specify precision**. This figure shows the **Data Types** pane for the case where the **Use a System object to implement filter** check box is not selected in the **Main** pane.

Lowpass Design

Design a lowpass filter.

Filter output variable name:

Main Data Types Code Generation

Arithmetic:

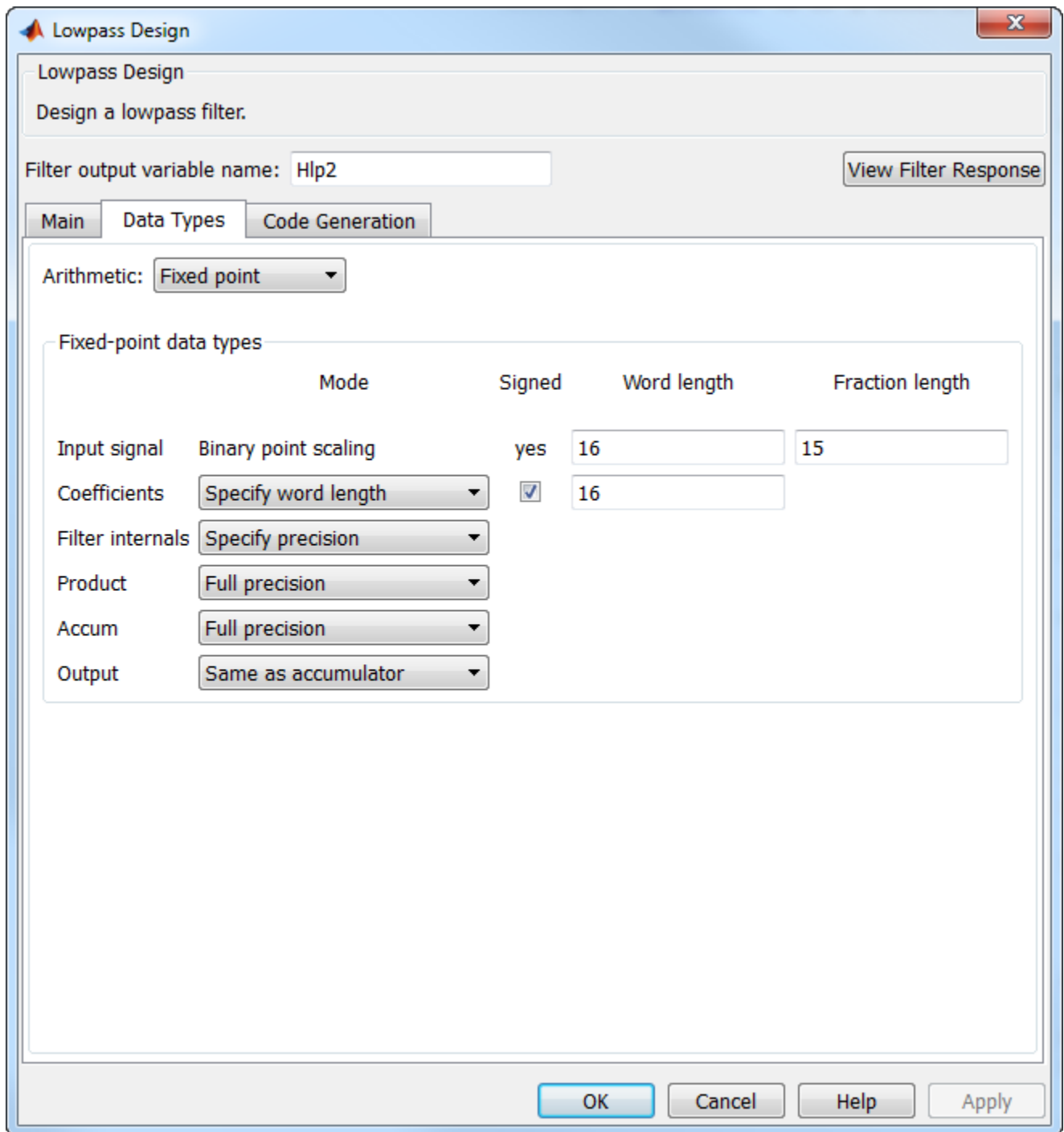
Fixed-point data types

|                  | Mode   | Signed                              | Word length                     | Fraction length                 |
|------------------|--|-------------------------------------|---------------------------------|---------------------------------|
| Input signal     | Binary point scaling                             | yes                                 | <input type="text" value="16"/> | <input type="text" value="15"/> |
| Coefficients     | <input type="text" value="Specify word length"/> | <input checked="" type="checkbox"/> | <input type="text" value="16"/> |                                 |
| Filter internals | <input type="text" value="Specify precision"/>   |                                     |                                 |                                 |
| Product          | Binary point scaling                             | yes                                 | <input type="text" value="32"/> | <input type="text" value="29"/> |
| Accum            | Binary point scaling                             | yes                                 | <input type="text" value="40"/> | <input type="text" value="29"/> |
| Output           | Binary point scaling                             | yes                                 | <input type="text" value="16"/> | <input type="text" value="15"/> |

Fixed-point operational parameters

Rounding mode:  Overflow mode:

When you select **Use a System object to implement filter** check box in the **Main** pane, the **Data Types** pane appears as below:



Not all parameters described in the following section apply to all filters. For example, FIR filters do not have the **Section input** and **Section output** parameters.

### Input signal

Specify the format the filter applies to data to be filtered. For all cases, `filterBuilder` implements filters that use binary point scaling and signed input. You set the word length and fraction length as needed.

## Coefficients

Choose how you specify the word length and the fraction length of the filter numerator and denominator coefficients:

- **Specify word length** enables you to enter the word length of the coefficients in bits. In this mode, `filterBuilder` automatically sets the fraction length of the coefficients to the binary-point only scaling that provides the best possible precision for the value and word length of the coefficients.
- **Binary point scaling** enables you to enter the word length and the fraction length of the coefficients in bits. If applicable, enter separate fraction lengths for the numerator and denominator coefficients.
- The filter coefficients do not obey the **Rounding mode** and **Overflow mode** parameters that are available when you select **Specify precision** from the Filter internals list. Coefficients are always saturated and rounded to Nearest.

## Section Input

Choose how you specify the word length and the fraction length of the fixed-point data type going into each section of an SOS filter. This parameter is visible only when the selected filter structure is IIR and SOS.

- **Binary point scaling** enables you to enter the word and fraction lengths of the section input in bits.
- **Specify word length** enables you to enter the word lengths in bits.

## Section Output

Choose how you specify the word length and the fraction length of the fixed-point data type coming out of each section of an SOS filter. This parameter is visible only when the selected filter structure is IIR and SOS.

- **Binary point scaling** enables you to enter the word and fraction lengths of the section output in bits.
- **Specify word length** enables you to enter the output word lengths in bits.

## State

Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. Use this parameter to specify how to designate the state word and fraction lengths. This parameter is not visible for direct form and direct form I filter structures because `filterBuilder` deduces the state directly from the input format. States always use signed representation:

- **Binary point scaling** enables you to enter the word length and the fraction length of the accumulator in bits.
- **Specify precision** enables you to enter the word length and fraction length in bits (if available).

## Product

Determines how the filter handles the output of product operations. Choose from the following options:

- **Full precision** — Maintain full precision in the result.

- **Keep LSB** — Keep the least significant bit in the result when you need to shorten the data words.
- **Specify Precision** — Enables you to set the precision (the fraction length) used by the output from the multiplies.

**Filter internals**

Specify how the fixed-point filter performs arithmetic operations within the filter. The affected filter portions are filter products, sums, states, and output. Select one of these options:

- **Full precision** — Specifies that the filter maintains full precision in all calculations for products, output, and in the accumulator.
- **Specify precision** — Set the word and fraction lengths applied to the results of product operations, the filter output, and the accumulator. Selecting this option enables the word and fraction length controls.

**Signed**

Selecting this option directs the filter to use signed representations for the filter coefficients.

**Word length**

Sets the word length for the associated filter parameter in bits.

**Fraction length**

Sets the fraction length for the associate filter parameter in bits.

**Accum**

Use this parameter to specify how you would like to designate the accumulator word and fraction lengths.

Determines how the accumulator outputs stored values. Choose from the following options:

- **Full precision** — Maintain full precision in the accumulator.
- **Keep MSB** — Keep the most significant bit in the accumulator.
- **Keep LSB** — Keep the least significant bit in the accumulator when you need to shorten the data words.
- **Specify Precision** — Enables you to set the precision (the fraction length) used by the accumulator.

**Output**

Sets the mode the filter uses to scale the output data after filtering. You have the following choices:

- **Avoid Overflow** — Set the output data fraction length to avoid causing the data to overflow. **Avoid overflow** is considered the conservative setting because it is independent of the input data values and range.
- **Best Precision** — Set the output data fraction length to maximize the precision in the output data.
- **Specify Precision** — Set the fraction length used by the filtered data.

**Fixed-point operational parameters**

Parameters in this group control how the filter rounds fixed-point values and how it treats values that overflow.

## Rounding mode

Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths).

- `ceil` — Round toward positive infinity.
- `convergent` — Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.
- `zero/fix` — Round toward zero.
- `floor` — Round toward negative infinity.
- `nearest` — Round toward nearest. Ties round toward positive infinity.
- `round` — Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.

The choice you make affects everything except coefficient values and input data which always round. In most cases, products do not overflow—they maintain full precision.

## Overflow mode

Sets the mode the filter uses to respond to overflow conditions in fixed-point arithmetic. Choose from the following options:

- `Saturate` — Limit the output to the largest positive or negative representable value.
- `Wrap` — Set overflowing values to the nearest representable value using modular arithmetic.

The choice you make affects everything except coefficient values and input data which always round. In most cases, products do not overflow—they maintain full precision.

## Cast before sum

Specifies whether to cast numeric data to the appropriate accumulator format before performing sum operations. Selecting **Cast before sum** ensures that the results of the affected sum operations match most closely the results found on most digital signal processors. Performing the cast operation before the summation adds one or two additional quantization operations that can add error sources to your filter results.

If you clear **Cast before sum**, the filter prevents the addends from being cast to the sum format before the addition operation. Choose this setting to get the most accurate results from summations without considering the hardware your filter might use. The input format referenced by **Cast before sum** depends on the filter structure you are using.

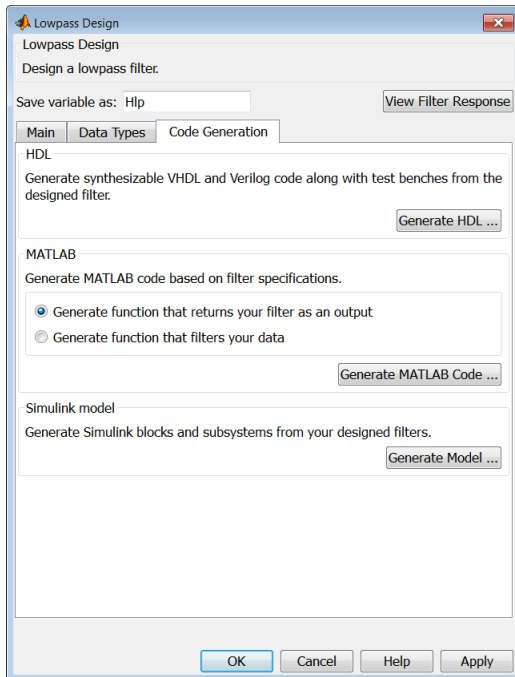
The effect of clearing or selecting **Cast before sum** is as follows:

- `Cleared` — Configures filter summation operations to retain the addends in the format carried from the previous operation.
- `Selected` — Configures filter summation operations to convert the input format of the addends to match the summation output format before performing the summation operation. Usually, selecting **Cast before sum** generates results from the summation that more closely match those found from digital signal processors.

## Code Generation Pane

The code generation pane contains options for various implementations of the completed filter design. Depending on your installation, you can generate MATLAB, VHDL, and Verilog code from the

designed filter. You can also choose to create or update a Simulink model from the designed filter. The following section explains these options.



## HDL

For more information on this option, see “Opening the Filter Design HDL Coder UI from the Filter Builder” (Filter Design HDL Coder).

## MATLAB

### Generate MATLAB code based on filter specifications

- **Generate function that returns your filter as an output**

Selecting this option generates a function that designs a filter object using `fdesign`.

- **Generate function that filters your data**

Selecting this option generates a function that takes data as input, and outputs data filtered with the designed filter. The data type of the filter output is set according to the data type settings in the **Data Types** pane.

Clicking on the **Generate MATLAB code** button, brings up a Save File dialog. Specify the file name and location, and save. The filter is now contained in an editable file.

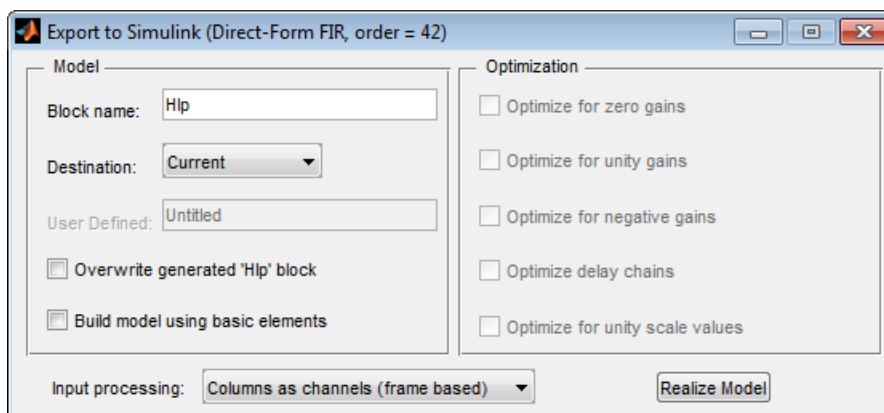
## Simulink Model

### Generate Simulink blocks and subsystems from your designed filters

When you click **Generate Model**, the filter builder generates Simulink blocks and subsystems from your designed filters.

Clicking on the **Generate Model** button opens the Export to Simulink dialog box.





- **Block Name** — The name for the new subsystem block, set to **Filter** by default.
- **Destination** — **Current** saves the generated model to the current Simulink model. **New** creates a new model to contain the generated block. **User Defined** creates a new model or subsystem at the location specified in **User Defined**.
- **Overwrite generated 'Filter' block** — Overwrites an existing block with the name specified in **Block Name**. Clear this check box to create a new block with the same name.
- **Build model using basic elements** — Builds the model using only basic blocks.
- **Optimize for zero gains** — Removes all zero-gain blocks from the model.
- **Optimize for unity gains** — Replaces all unity gains with direct connections.
- **Optimize for negative gains** — Removes all negative unity gain blocks, and changes sign at the nearest summation block.
- **Optimize delay chains** — Replaces delay chains made up of  $n$  unit delays with a single delay by  $n$ .
- **Optimize for unity scale values** — Removes all scale value multiplications by 1 from the filter structure.
- **Input processing** — Specify how the generated filter block or subsystem block processes the input. Depending on the type of filter you are designing, one or both of the following options may be available:
  - **Columns as channels (frame based)** — The block treats each column of the input as a separate channel.
  - **Elements as channels (sample based)** — The block treats each element of the input as a separate channel.

For more information about sample-based and frame-based processing, see “Sample- and Frame-Based Concepts”.

- **Realize Model** — Builds the model with the set parameters.

When the **Use a System object to implement filter** check box is selected in the **Main** pane, the **Generate Model** button in the **Simulink model** panel is disabled under the following conditions:

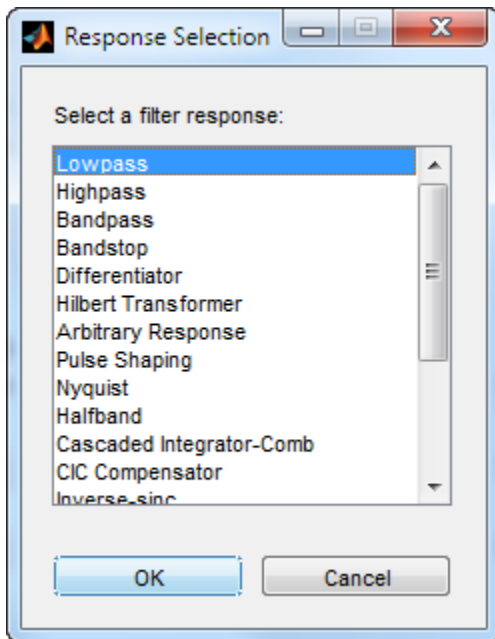
- Select **Filter response** as **Comb** and **Arithmetic** on the **Data Types** pane as **Fixed point**.
- Select **Filter response** as **Arbitrary Response**, **Impulse response** as **IIR**, set **Specify response as** to either **Magnitudes and phases** or **Frequency response**, and **Arithmetic** on the **Data Types** pane as **Fixed point**.

These settings design a `dsp.IIRFilter` System object with fixed point arithmetic. Generating a Simulink model for fixed point `dsp.IIRFilter` object is not supported.

## Filter Responses

Select your filter response from the `filterBuilder` **Response Selection** main menu.

If you have the DSP System Toolbox software, the following **Response Selection** menu appears.



Select your desired filter response from the menu and design your filter.

The following sections describe the options available for each response type.

### Arbitrary Response Filter Design – Main Pane

#### Filter specifications

Parameters in this group enable you to specify your filter format, such as the impulse response and the filter order.

#### Impulse response

This dialog only applies if you have the DSP System Toolbox software. Select either FIR or IIR from the drop down list, where FIR is the default impulse response. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly. Arbitrary group delay designs are only available if **Impulse response** is IIR.

Without the DSP System Toolbox, the only available arbitrary response filter design is FIR.

#### Order mode

This dialog only applies if you have the DSP System Toolbox software. Choose `Minimum` or `Specify`. Choosing `Specify` enables the **Order** dialog.

#### Order

This dialog only applies when **Order mode** is `Specify`. For an FIR design, specify the filter order. For an IIR design, you can specify an equal order for the numerator and denominator, or

you can specify different numerator and denominator orders. The default is equal orders. To specify a different denominator order, check the **Denominator order** box. Because the Signal Processing Toolbox only supports FIR arbitrary-magnitude filters, you do not have the option to specify a denominator order.

### Denominator order

Select the check box and enter the denominator order. This option is enabled only if IIR is selected for **Impulse response**.

### Filter type

This dialog only applies if you have the DSP System Toolbox software and is only available for FIR filters. Select **Single-rate**, **Decimator**, **Interpolator**, or **Sample-rate converter**. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, filterBuilder specifies single-rate filters.

- Selecting **Decimator** or **Interpolator** activates the **Decimation Factor** or the **Interpolation Factor** options respectively.
- Selecting **Sample-rate converter** activates both factors.

When you design either a decimator or interpolator, the resulting filter is a bandpass filter that either decimates or interpolates your input signal.

### Decimation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to **Decimator** or **Sample-rate converter**. The default factor value is 2 for **Decimator** and 3 for **Sample-rate converter**.

### Interpolation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to **Interpolator** or **Sample-rate converter**. The default factor value is 2.

## Response Specification

### Number of Bands

Select the number of bands in the filter. Multiband design is available for both FIR and IIR filters.

### Specify response as:

Specify the response as **Amplitudes**, **Magnitudes and phase**, **Frequency response**, or **Group delay**. **Amplitudes** is the only option if you do not have the DSP System Toolbox software. **Group delay** is only available for IIR designs.

### Frequency units

Specify frequency units as either **Normalized**, **Hz**, **kHz**, **MHz**, or **GHz**.

### Input sample rate

Enter the input sampling frequency in the units specified in the **Frequency units** drop-down box. This option is enabled when **Frequency units** is set to an option in hertz.

## Band Properties

These properties are modified automatically depending on the response chosen in the **Specify response as** drop-down box. Two or three columns are presented for input. The first column is always **Frequencies**. The other columns are either **Amplitudes**, **Magnitudes**, **Phases**, or **Frequency Response**. Enter the corresponding vectors of values for each column.

- **Frequencies** and **Amplitudes** — These columns are presented for input if you select **Amplitudes** in the **Specify response as** drop-down box.
- **Frequencies**, **Magnitudes**, and **Phases** — These columns are presented for input if the response chosen in the **Specify response as** drop-down box is **Magnitudes and phases**.
- **Frequencies** and **Frequency response** — These columns are presented for input if the response chosen in the **Specify response as** drop-down box is **Frequency response**.

### Algorithm

The options for each design are specific for each design method. In the arbitrary response design, the available options also depend on the **Response specifications**. This section does not present all of the available options for all designs and design methods.

### Design Method

Select the design method for the filter. Different methods are enabled depending on the defining parameters entered in the previous sections.

### Design Options

- **Window** — Valid when the **Design method** is **Frequency Sampling**. Replace the square brackets with the name of a window function or function handle. For example, `'hamming'` or `@hamming`. If the window function takes parameters other than the length, use a cell array. For example, `{'kaiser', 3.5}` or `{@chebwin, 60}`.
- **Density factor** — Valid when the **Design method** is **equiripple**. Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 16 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

The default changes to 20 for an IIR arbitrary group delay design.

- **Phase constraint** — Valid when the **Design method** is **equiripple**, you have the DSP System Toolbox installed, and **Specify response as** is set to **Amplitudes**. Choose one of **Linear**, **Minimum**, or **Maximum**.
- **Weights** — Uses the weights in **Weights** to weight the error for a single-band design. If you have multiple frequency bands, the **Weights** design option changes to **B1 Weights**, **B2 Weights** to designate the separate bands. Use **Bi Weights** to specify weights for the *i*-th band. The **Bi Weights** design option is only available when you specify the *i*-th band as an unconstrained.
- **Bi forced frequency point** — This option is only available in a multi-band constrained equiripple design when **Specify response as** is **Amplitudes**. **Bi forced frequency point** is the frequency point in the *i*-th band at which the response is forced to be zero. The index *i* corresponds to the frequency bands in **Band properties**. For example, if you specify two bands in **Band properties**, you have **B1 forced frequency point** and **B2 forced frequency point**.
- **Norm** — Valid only for IIR arbitrary group delay designs. **Norm** is the norm used in the optimization. The default value is 128, which essentially equals the L-infinity norm. The norm must be even.

- **Max pole radius** — Valid only for IIR arbitrary group delay designs. Constrains the maximum pole radius. The default is 0.999999. Reducing the **Max pole radius** can produce a transfer function more resistant to quantization.
- **Init norm** — Valid only for IIR arbitrary group delay designs. The initial norm used in the optimization. The default initial norm is 2.
- **Init numerator** — Specifies an initial estimate of the filter numerator coefficients.
- **Init denominator** — Specifies an initial estimate of the filter denominator coefficients. This may be useful in difficult optimization problems. In allpass filters, you only have to specify either the denominator or numerator coefficients. If you specify the denominator coefficients, you can obtain the numerator coefficients.

## Filter implementation

### Structure

Select the structure for the filter. The available filter structures depend on the parameters you select for your filter.

### Use a System object to implement filter

This check box appears when you set **Filter type** to **Single-rate**. Selecting this check box gives you the choice of using a System object to implement the filter. By default, this check box is cleared.

This check box no longer appears when you set **Filter type** to **Interpolator**, **Decimator**, or **Sample-rate converter**. The filter builder always implements the filter as a System object.

## Audio Weighting Filter Design — Main Pane

### Filter specifications

- **Weighting type** — The weighting type defines the frequency response of the filter. The valid weighting types are: A, C, C-message, ITU-T 0.41, and ITU-R 468-4 weighting. See `fdesign.audioweighting` for definitions of the weighting types.
- **Class** — Filter class is only applicable for A weighting and C weighting filters. The filter class describes the frequency-dependent tolerances specified in the relevant standards. There are two possible class values: 1 and 2. Class 1 weighting filters have stricter tolerances than class 2 filters. The filter class value does not affect the design. The class value is only used to provide a specification mask in **FVTool** for the analysis of the filter design.
- **Impulse response** — Impulse response type as one of IIR or FIR. For A, C, C-message, and ITU-R 468-4 filter, IIR is the only option. For a ITU-T 0.41 weighting filter, FIR is the only option.
- **Frequency units** — Choose Hz, kHz, MHz, or GHz. Normalized frequency designs are not supported for audio weighting filters.
- **Input sample rate** — The sampling frequency in **Frequency units**. For example, if **Frequency units** is set to kHz, setting **Input sample rate** to 40 is equivalent to a 40 kHz sampling frequency.

### Algorithm

- **Design method** — Valid design methods depend on the weighting type. For type A and C weighting filters, the only valid design type is **ANSI S1.42**. This is an IIR design method that follows ANSI standard S1.42-2001. For a C message filter, the only valid design method is **Bell 41009**, which is an IIR design method following the Bell System Technical Reference PUB 41009. For a ITU-R 468-4 weighting filter, you can design an IIR or FIR filter. If you choose an IIR design,

the design method is IIR least  $p$ -norm. If you choose an FIR design, the design method choices are: Equiripple or Frequency Sampling. For an ITU-T 0.41 weighting filter, the available FIR design methods are Equiripple or Frequency Sampling

- **Scale SOS filter coefficients to reduce chance of overflow** — Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

### Filter implementation

- **Structure** — For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. For audio weighting IIR filter designs, you can choose direct form I or II biquad (SOS). You can also choose to implement these structures in transposed form.

For FIR designs, you can choose direct form, direct-form transposed, direct-form symmetric, direct-form asymmetric structures, or an overlap and add structure.

- **Use a System object to implement filter** — Selecting this check box gives you the choice of using a System object to implement the filter. By default, this check box is cleared. When the current design method or structure is not supported by a System object filter, then this check box is disabled.

### Bandpass Filter Design — Main Pane

#### Filter specifications

Parameters in this group enable you to specify your filter format, such as the impulse response and the filter order.

#### Impulse response

Select FIR or IIR from the drop-down list, where FIR is the default impulse response. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly.

---

**Note** The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

---

#### Order mode

Select Minimum (the default) or Specify from the drop-down box. Selecting Specify enables the **Order** option so you can enter the filter order.

If you have the DSP System Toolbox software installed, you can specify IIR filters with different numerator and denominator orders. The default is equal orders. To specify a different denominator order, check the **Denominator order** box.

**Filter type** — This dialog only applies if you have the DSP System Toolbox software.

Select Single-rate, Decimator, Interpolator, or Sample-rate converter. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, `filterBuilder` specifies single-rate filters.

- Selecting Decimator or Interpolator activates the **Decimation Factor** or the **Interpolation Factor** options respectively.

- Selecting **Sample-rate converter** activates both factors.

### Order

Enter the filter order. This option is enabled only if you select **Specify** for **Order mode**.

### Decimation Factor

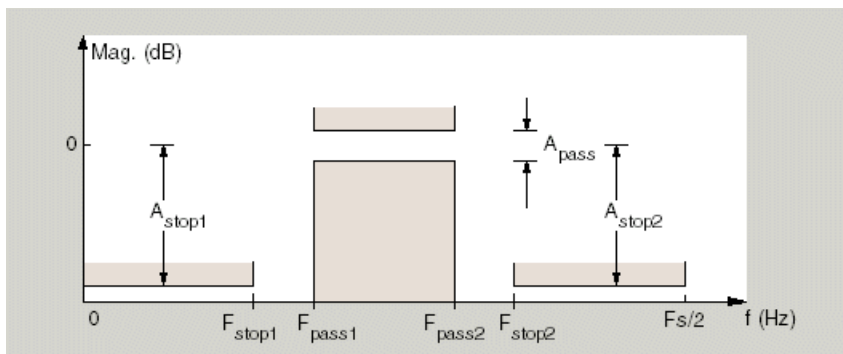
Enter the decimation factor. This option is enabled only if the **Filter type** is set to **Decimator** or **Sample-rate converter**. The default factor value is 2.

### Interpolation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to **Interpolator** or **Sample-rate converter**. The default factor value is 2.

### Frequency specifications

The parameters in this group allow you to specify your filter response curve. Graphically, the filter specifications look similar to those shown in the following figure.



In the figure, regions between specification values such as Stopband frequency 1 ( $F_{stop1}$ ) and Passband frequency 1 ( $F_{pass1}$ ) represent transition regions where the filter response is not explicitly defined.

### Frequency constraints

Select the filter features to use to define the frequency response characteristics. This dialog applies only when **Order mode** is **Specify**.

- **Passband and stopband frequencies** — Define the filter by specifying the frequencies for the edges for the stop- and passbands.
- **Passband frequency** — Define the filter by specifying frequencies for the edges of the passband.
- **Stopband frequency** — Define the filter by specifying frequencies for the edges of the stopbands.
- **Half power (3dB) frequency** — Define the filter response by specifying the locations of the 3 dB points (IIR filters). The 3-dB point is the frequency for the point 3 dB below the passband value.
- **Half power (3dB) frequencies and passband width** — Define the filter by specifying frequencies for the 3-dB points in the filter response and the width of the passband. (IIR filters)

- **Half power (3dB) frequencies and stopband width** — Define the filter by specifying frequencies for the 3-dB points in the filter response and the width of the stopband. (IIR filters)
- **Cutoff (6dB) frequency** — Define the filter response by specifying the locations of the 6-dB points. The 6-dB point is the frequency for the point 6 dB below the passband value. (FIR filters)

### Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select **Normalized (0 to 1)** to enter frequencies in normalized form. This behavior is the default. To enter frequencies in hertz, select one of the frequency units from the drop-down list—Hz, kHz, MHz, or GHz. Selecting one of the unit options enables the **Input sample rate** parameter.

### Input sample rate

F<sub>s</sub>, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

### Stopband frequency 1

Enter the frequency at the edge of the end of the first stopband. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

### Passband frequency 1

Enter the frequency at the edge of the start of the passband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

### Passband frequency 2

Enter the frequency at the edge of the end of the passband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

### Stopband frequency 2

Enter the frequency at the edge of the start of the second stopband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

### Magnitude specifications

The parameters in this group let you specify the filter response in the passbands and stopbands.

### Magnitude constraints

Specify as **Unconstrained** or **Constrained bands**. You must have the DSP System Toolbox software to select **Constrained bands**. Selecting **Constrained bands** enables dialogs for both stopbands and the passband: **Stopband attenuation 1**, **Stopband attenuation 2**, and **Passband ripple**. You cannot specify constraints for all three bands simultaneously.

Setting **Magnitude constraints** to **Constrained bands** enables the **Wstop** and **Wpass** options under **Design options**.

### Magnitude units

Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.

- **Linear** — Specify the magnitude in linear units.



- dB — Specify the magnitude in dB (decibels). This is the default setting.
- Squared — Specify the magnitude in squared units.

**Stopband attenuation 1**

Enter the filter attenuation in the first stopband in the units you choose for **Magnitude units**, either linear or decibels.

**Passband ripple**

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels.

**Stopband attenuation 2**

Enter the filter attenuation in the second stopband in the units you choose for **Magnitude units**, either linear or decibels.

**Algorithm**

The parameters in this group allow you to specify the design method and structure that filterBuilder uses to implement your filter.

**Design Method**

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is usually Butterworth, and the default FIR method is equiripple.

**Scale SOS filter coefficients to reduce chance of overflow**

Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

**Design Options**

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

**Density factor**

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 16 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

**Phase constraint**

Valid when the **Design method** is equiripple and you have the DSP System Toolbox installed. Choose one of Linear, Minimum, or Maximum.

**Minimum order**

This option only applies when you have the DSP System Toolbox software and **Order mode** is Minimum.

Select Any (default), Even, or Odd. Selecting Even or Odd forces the minimum-order design to be an even or odd order.

**Wstop1**

Weight for the first stopband.

**Wpass**

Passband weight.

**Wstop2**

Weight for the second stopband.

**Max pole radius**

Valid only for IIR designs. Constrains the maximum pole radius. The default is 1. Reducing the max pole radius can produce a transfer function more resistant to quantization.

**Init norm**

Valid only for IIR designs. The initial norm used in the optimization. The default initial norm is 2.

**Init numerator**

Specifies an initial estimate of the filter numerator coefficients. This may be useful in difficult optimization problems.

**Init denominator**

Specifies an initial estimate of the filter denominator coefficients. This may be useful in difficult optimization problems.

**Filter implementation****Structure**

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure, and IIR filters use direct-form II filters with SOS.

**Use a System object to implement filter**

This check box appears when you set **Filter type** to `Single-rate`. Selecting this check box gives you the choice of using a System object to implement the filter. By default, this check box is cleared.

This check box no longer appears when you set **Filter type** to `Interpolator`, `Decimator`, or `Sample-rate converter`. The filter builder always implements the filter as a System object.

**Bandstop Filter Design — Main Pane****Filter specifications**

Parameters in this group enable you to specify your filter format, such as the impulse response and the filter order.

**Impulse response**

Select FIR or IIR from the drop-down list, where FIR is the default impulse response. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly.

---

**Note** The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

---

### Order mode

Select **Minimum** (the default) or **Specify** from the drop-down list. Selecting **Specify** enables the **Order** option so you can enter the filter order.

If you have the DSP System Toolbox software installed, you can specify IIR filters with different numerator and denominator orders. The default is equal orders. To specify a different denominator order, check the **Denominator order** box.

### Filter type

Select **Single-rate**, **Decimator**, **Interpolator**, or **Sample-rate converter**. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, filterBuilder specifies single-rate filters.

- Selecting **Decimator** or **Interpolator** activates the **Decimation Factor** or the **Interpolation Factor** options respectively.
- Selecting **Sample-rate converter** activates both factors.

When you design either a decimator or an interpolator, the resulting filter is a bandpass filter that either decimates or interpolates your input signal.

### Order

Enter the filter order. This option is enabled only if **Specify** was selected for **Order mode**.

### Decimation Factor

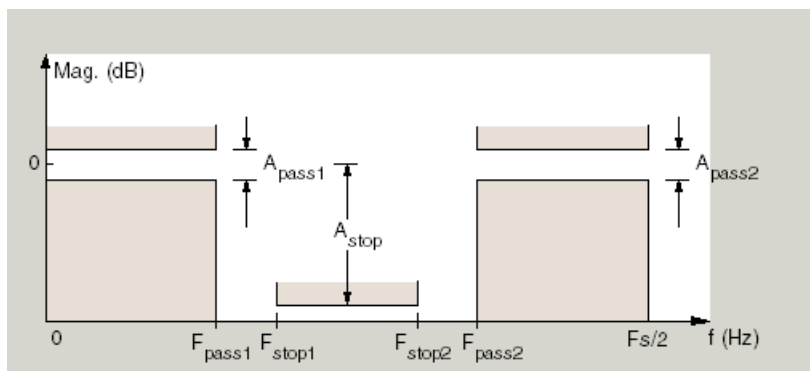
Enter the decimation factor. This option is enabled only if the **Filter type** is set to **Decimator** or **Sample-rate converter**. The default factor value is 2.

### Interpolation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to **Interpolator** or **Sample-rate converter**. The default factor value is 2.

### Frequency specifications

The parameters in this group allow you to specify your filter response curve. Graphically, the filter specifications look similar to those shown in the following figure.



### Frequency constraints

Select the filter features to use to define the frequency response characteristics. This dialog applies only when **Order mode** is **Specify**.

- **Passband and stopband frequencies** — Define the filter by specifying the frequencies for the edges for the stop- and passbands.
- **Passband frequency** — Define the filter by specifying frequencies for the edges of the passband.
- **Stopband frequency** — Define the filter by specifying frequencies for the edges of the stopbands.
- **Half power (3dB) frequency** — Define the filter response by specifying the locations of the 3 dB points (IIR filters). The 3 dB point is the frequency for the point 3 dB point below the passband value.
- **Half power (3dB) frequencies and passband width** — Define the filter by specifying frequencies for the 3 dB points in the filter response and the width of the passband (IIR filters).
- **Half power (3dB) frequencies and stopband width** — Define the filter by specifying frequencies for the 3 dB points in the filter response and the width of the stopband (IIR filters).
- **Cutoff (6dB) frequency** — Define the filter response by specifying the locations of the 6-dB points (FIR filters). The 6-dB point is the frequency for the point 6 dB point below the passband value.

### Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select **Normalized** (0 to 1) to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—Hz, kHz, MHz, or GHz. Selecting one of the unit options enables the **Input sample rate** parameter.

### Input sample rate

$F_s$ , specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

### Output sample rate

When you design an interpolator,  $F_s$  represents the sampling frequency at the filter output rather than the filter input. This option is available only when you set **Filter type** is **interpolator**.

### Passband frequency 1

Enter the frequency at the edge of the end of the first passband. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

### Stopband frequency 1

Enter the frequency at the edge of the start of the stopband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

### Stopband frequency 2

Enter the frequency at the edge of the end of the stopband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

### Passband frequency 2

Enter the frequency at the edge of the start of the second passband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

### Magnitude specifications

The parameters in this group let you specify the filter response in the passbands and stopbands.

#### Magnitude constraints

Specify as **Unconstrained** or **Constrained** bands. You must have the DSP System Toolbox software to select **Constrained** bands. Selecting **Constrained** bands enables dialogs for both passbands and the stopband: **Passband ripple 1**, **Passband ripple 2**, and **Stopband attenuation**. You cannot specify constraints for all three bands simultaneously.

Setting **Magnitude constraints** to **Constrained** bands enables the **Wstop** and **Wpass** options under **Design options**.

#### Magnitude units

Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.

- **Linear** — Specify the magnitude in linear units.
- **dB** — Specify the magnitude in decibels (default).
- **Squared** — Specify the magnitude in squared units.

#### Passband ripple 1

Enter the filter ripple allowed in the first passband in the units you choose for **Magnitude units**, either linear or decibels.

#### Stopband attenuation

Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels

#### Passband ripple 2

Enter the filter ripple allowed in the second passband in the units you choose for **Magnitude units**, either linear or decibels

### Algorithm

The parameters in this group allow you to specify the design method and structure that **filterBuilder** uses to implement your filter.

#### Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is usually Butterworth, and the default FIR method is equiripple.

#### Scale SOS filter coefficients to reduce chance of overflow

Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

### Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

**Density factor**

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 16 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

**Phase constraint**

Valid when the **Design method** is equiripple and you have the DSP System Toolbox installed. Choose one of Linear, Minimum, or Maximum.

**Minimum order**

This option only applies when you have the DSP System Toolbox software and **Order mode** is Minimum.

Select Any (default), Even, or Odd. Selecting Even or Odd forces the minimum-order design to be an even or odd order.

**Wpass1**

Weight for the first passband.

**Wstop**

Stopband weight.

**Wpass2**

Weight for the second passband.

**Match exactly**

Specifies that the resulting filter design matches either the passband or stopband or both bands when you select passband or stopband .

**Max pole radius**

Valid only for IIR designs. Constrains the maximum pole radius. The default is 1. Reducing the max pole radius can produce a transfer function more resistant to quantization.

**Init norm**

Valid only for IIR designs. The initial norm used in the optimization. The default initial norm is 2.

**Init numerator**

Specifies an initial estimate of the filter numerator coefficients. This may be useful in difficult optimization problems.

**Init denominator**

Specifies an initial estimate of the filter denominator coefficients. This may be useful in difficult optimization problems.

## Filter implementation

### Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure, and IIR filters use direct-form II filters with SOS.

### Use a System object to implement filter

This check box appears when you set **Filter type** to `Single-rate`. Selecting this check box gives you the choice of using a System object to implement the filter. By default, this check box is cleared.

This check box no longer appears when you set **Filter type** to `Interpolator`, `Decimator`, or `Sample-rate converter`. The filter builder always implements the filter as a System object.

## CIC Filter Design — Main Pane

### Filter specifications

Parameters in this group enable you to specify your CIC filter format, such as the filter type and the differential delay.

#### Filter type

Select whether your filter will be a `decimator` or an `interpolator`. Your choice determines the type of filter and the design methods and structures that are available to implement your filter. Selecting `decimator` or `interpolator` activates the **Factor** option. When you design an interpolator, you enable the **Output sample rate** parameter.

When you design either a decimator or interpolator, the resulting filter is a CIC filter that decimates or interpolates your input signal.

#### Differential Delay

Specify the differential delay of your CIC filter as an integer value greater than or equal to 1. The default value is 1. The differential delay changes the shape, number, and location of nulls in the filter response. Increasing the differential delay increases the sharpness of the nulls and the response between the nulls. In practice, differential delay values of 1 or 2 are the most common.

#### Factor

Specify the decimation or interpolation factor for your filter as an integer value greater than or equal to 1. The default value is 2.

### Frequency specifications

#### Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0 to 1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `KHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input sample rate** parameter.

#### Input sample rate

$F_s$ , specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

**Output sample rate**

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter output. When you provide an output sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available only when you design interpolators.

**Passband frequency**

Enter the frequency at the end of the passband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

**Magnitude specifications****Magnitude units**

Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.

- Linear — Specify the magnitude in linear units.
- dB — Specify the magnitude in decibels (default).
- Squared — Specify the magnitude in squared units.

**Stopband attenuation**

Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels.

**CIC Compensator Design — Main Pane****Filter specifications**

Parameters in this group enable you to specify your filter format, such as the filter order mode and the filter type.

**Order mode**

Select Minimum (the default) or Specify from the drop-down list. Selecting Specify enables the **Order** option (see the following sections) so you can enter the filter order.

**Filter type**

Select Single-rate, Decimator, Interpolator, or Sample-rate converter. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, `filterBuilder` specifies single-rate filters.

- Selecting Decimator or Interpolator activates the **Decimation Factor** or the **Interpolation Factor** options respectively.
- Selecting Sample-rate converter activates both factors.

When you design either a decimator or an interpolator, the resulting filter is a bandpass filter that either decimates or interpolates your input signal.

**Order**

Enter the filter order. This option is enabled only if Specify was selected for **Order mode**.

**Decimation Factor**

Enter the decimation factor. This option is enabled only if the **Filter type** is set to Decimator or Sample-rate converter. The default factor value is 2.



### Interpolation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to **Interpolator** or **Sample-rate converter**. The default factor value is 2.

### Number of CIC sections

Specify the number of sections in the CIC filter for which you are designing this compensator. Select the number of sections from the drop-down list or enter the number.

### Differential Delay

Specify the differential delay of your target CIC filter. The default value is 1. Most CIC filters use 1 or 2.

### Frequency specifications

The parameters in this group allow you to specify your filter response curve.

#### Frequency specifications

##### Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select **Normalized (0 to 1)** to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—Hz, kHz, MHz, or GHz. Selecting one of the unit options enables the **Input sample rate** parameter.

##### Input sample rate

$F_s$ , specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

##### Output sample rate

$F_s$ , specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter output. When you provide an output sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available only when you design interpolators.

##### Passband frequency

Enter the frequency at the end of the passband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

##### Stopband frequency

Enter the frequency at the start of the stopband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

### Magnitude specifications

The parameters in this group let you specify the filter response in the passbands and stopbands.

#### Magnitude units

Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.

- **Linear** — Specify the magnitude in linear units.

- dB — Specify the magnitude in decibels (default).
- Squared — Specify the magnitude in squared units.

**Passband ripple**

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels

**Algorithm**

The parameters in this group allow you to specify the design method and structure that `filterBuilder` uses to implement your filter.

**Design Method**

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is usually Butterworth, and the default FIR method is `equiripple`.

**Design Options**

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

**Density factor**

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal `equiripple` filter but increases the time required to design the filter. The default value of 16 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

**Minimum phase**

To design a filter that is minimum phase, select **Minimum phase**. Clearing the **Minimum phase** option removes the phase constraint—the resulting design is not minimum phase.

**Minimum order**

When you select this parameter, the design method determines and design the minimum order filter to meet your specifications. Some filters do not provide this parameter. Select `Any`, `Even`, or `Odd` from the drop-down list to direct the design to be any minimum order, or minimum even order, or minimum odd order.

---

**Note** Generally, **Minimum order** designs are not available for IIR filters.

---

**Match exactly**

Specifies that the resulting filter design matches either the passband or stopband or both bands when you select `passband` or `stopband` or both from the drop-down list.

## Stopband Shape

Stopband shape lets you specify how the stopband changes with increasing frequency. Choose one of the following options:

- **Flat** — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.
- **Linear** — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.
- **1/f** — Specifies that the stopband attenuation changes exponentially as the frequency increases, where  $f$  is the frequency. Set the power (exponent) for the decay in **Stopband decay**.

## Stopband Decay

When you set Stopband shape, Stopband decay specifies the amount of decay applied to the stopband. The following conditions apply to Stopband decay based on the value of Stopband Shape:

- When you set **Stopband shape** to **Flat**, **Stopband decay** has no effect on the stopband.
- When you set **Stopband shape** to **Linear**, enter the slope of the stopband in units of dB/rad/s. `filterBuilder` applies that slope to the stopband.
- When you set **Stopband shape** to **1/f**, enter a value for the exponent  $n$  in the relation  $(1/f)^n$  to define the stopband decay. `filterBuilder` applies the  $(1/f)^n$  relation to the stopband to result in an exponentially decreasing stopband attenuation.

## Filter implementation

### Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure, and IIR filters use direct-form II filters with SOS.

### Use a System object to implement filter

This check box appears when you set **Filter type** to **Single-rate**. Selecting this check box gives you the choice of using a System object to implement the filter. By default, this check box is cleared.

This check box no longer appears when you set **Filter type** to **Interpolator**, **Decimator**, or **Sample-rate converter**. The filter builder always implements the filter as a System object.

## Comb Filter Design —Main Pane

### Filter specifications

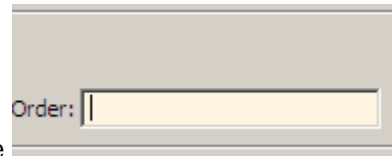
Parameters in this group enable you to specify the type of comb filter and the number of peaks or notches.

### Comb Type

Select **Notch** or **Peak** from the drop-down list. **Notch** creates a comb filter that attenuates a set of harmonically related frequencies. **Peak** creates a comb filter that amplifies a set of harmonically related frequencies.

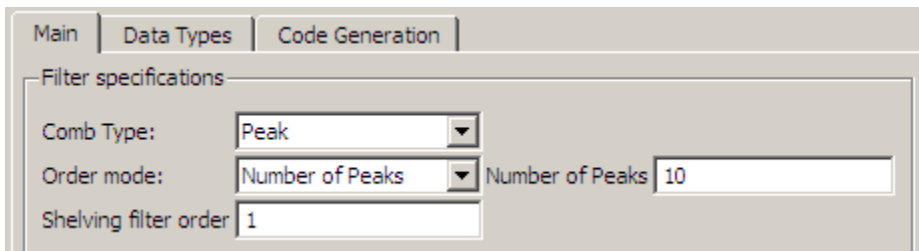
### Order mode

Select **Order** or **Number of Peaks/Notches** from the drop-down menu.



Select **Order** to enter the desired filter order in the dialog box. The comb filter has notches or peaks at increments of  $2/\text{Order}$  in normalized frequency units.

Select **Number of Peaks** or **Number of Notches** to specify the number of peaks or notches and the **Shelving filter order**



### Shelving filter order

The **Shelving filter order** is a positive integer that determines the sharpness of the peaks or notches. Larger values result in sharper peaks or notches.

### Frequency specifications

Parameters in this group enable you to specify the frequency constraints and frequency units.

#### Frequency specifications

Select **Quality factor** or **Bandwidth**.

**Quality factor** is the ratio of the center frequency of the peak or notch to the bandwidth calculated at the -3 dB point.

**Bandwidth** specifies the bandwidth of the peak or notch. By default the bandwidth is measured at the -3 dB point. For example, setting the bandwidth equal to 0.1 results in 3 dB frequencies at normalized frequencies 0.05 above and below the center frequency of the peak or notch.

#### Frequency Units

Specify the frequency units. The default is normalized frequency. Choosing an option in Hz enables the **Input sample rate** dialog box.

#### Magnitude specifications

Specify the units for the magnitude specification and the gain at which the bandwidth is measured. This menu is disabled if you specify a filter order. Select one of the following magnitude units from the drop down list:

- dB — Specify the magnitude in decibels (default).
- Squared — Specify the magnitude in squared units.

**Bandwidth gain** — Specify the gain at which the bandwidth is measured. The default is -3 dB.

## Algorithm

The parameters in this group allow you to specify the design method and structure that filterBuilder uses to implement your filter.

### Design Method

The IIR Butterworth design is the only option for peaking or notching comb filters.

### Filter implementation

#### Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter.

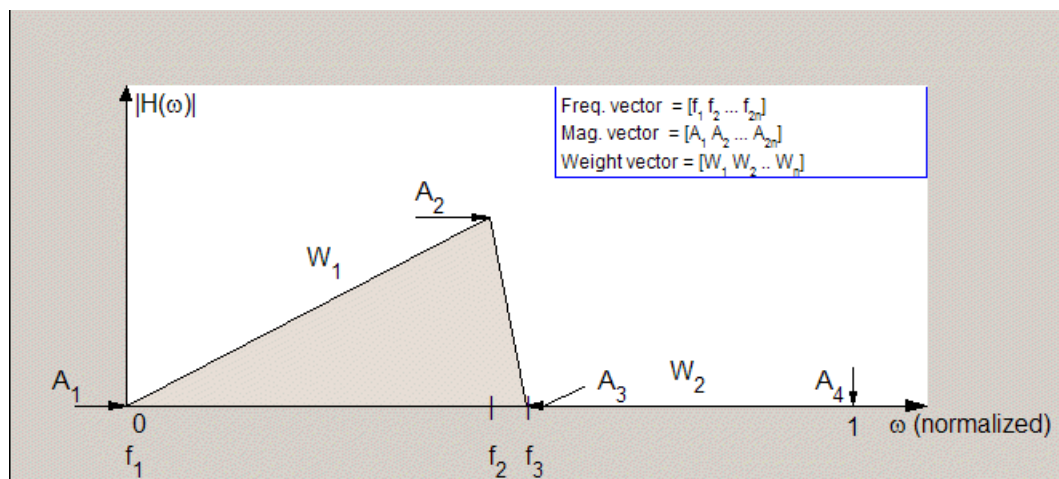
#### Use a System object to implement filter

Selecting this check box gives you the choice of using a System object to implement the filter. By default, the check box is cleared.

## Differentiator Filter Design — Main Pane

### Filter specifications

Parameters in this group enable you to specify your filter format, such as the impulse response and the filter order. Graphically, the filter specifications look similar to those shown in the following figure.



In the figure, regions between specification values such as **Passband frequency** ( $f_1$ ) and **Stopband frequency** ( $f_3$ ) represent transition regions where the filter response is not explicitly defined.

### Order mode

Select Minimum (the default) or Specify from the drop-down list. Selecting Specify enables the **Order** option (see the following sections) so you can enter the filter order.

### Filter type

Select Single-rate, Decimator, Interpolator, or Sample-rate converter. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, filterBuilder specifies single-rate filters.

- Selecting **Decimator** or **Interpolator** activates the **Decimation Factor** or the **Interpolation Factor** options respectively.
- Selecting **Sample-rate converter** activates both factors.

When you design either a decimator or an interpolator, the resulting filter is a bandpass filter that either decimates or interpolates your input signal.

### **Order**

Enter the filter order. This option is enabled only if **Specify** was selected for **Order mode**.

### **Decimation Factor**

Enter the decimation factor. This option is enabled only if the **Filter type** is set to **Decimator** or **Sample-rate converter**. The default factor value is 2.

### **Interpolation Factor**

Enter the decimation factor. This option is enabled only if the **Filter type** is set to **Interpolator** or **Sample-rate converter**. The default factor value is 2.

### **Frequency specifications**

The parameters in this group allow you to specify your filter response curve.

#### **Frequency constraints**

This option is only available when you specify the order of the filter design. Supported options are **Unconstrained** and **Passband edge and stopband edge**.

#### **Frequency units**

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select **Normalized (0 to 1)** to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—**Hz**, **KHz**, **MHz**, or **GHz**. Selecting one of the unit options enables the **Input sample rate** parameter.

#### **Input sample rate**

$F_s$ , specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

#### **Passband frequency**

Enter the frequency at the end of the passband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

#### **Stopband frequency**

Enter the frequency at the start of the stopband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

### **Magnitude specifications**

The parameters in this group let you specify the filter response in the passbands and stopbands.

#### **Magnitude constraints**

This option is only available when you specify the order of your filter design. The options for **Magnitude constraints** depend on the value of the **Frequency constraints**. If the value of **Frequency constraints** is **Unconstrained**, **Magnitude constraints** must be **Unconstrained**.

If the value of **Frequency constraints** is Passband edge and stopband edge, **Magnitude constraints** can be Unconstrained, Passband ripple, or Stopband attenuation.

### Magnitude units

Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.

- Linear — Specify the magnitude in linear units.
- dB — Specify the magnitude in decibels (default).
- Squared — Specify the magnitude in squared units.

### Passband ripple

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels.

### Stopband attenuation 2

Enter the filter attenuation in the second stopband in the units you choose for **Magnitude units**, either linear or decibels.

### Algorithm

The parameters in this group allow you to specify the design method and structure that filterBuilder uses to implement your filter.

### Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is usually Butterworth, and the default FIR method is equiripple.

### Scale SOS filter coefficients to reduce chance of overflow

Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

### Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications.

### Density factor

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 16 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

### Wpass

Passband weight. This option is only available for a specified-order design when **Frequency constraints** is equal to Passband edge and stopband edge and the **Design method** is Equiripple.

**Wstop**

Stopband weight. This option is only available for a specified-order design when **Frequency constraints** is equal to Passband edge and stopband edge and the **Design method** is Equiripple.

**Filter implementation****Structure**

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure, and IIR filters use direct-form II filters with SOS.

**Use a System object to implement filter**

This check box appears when you set **Filter type** to **Single-rate**. Selecting this check box gives you the choice of using a System object to implement the filter. By default, the check box is cleared.

This check box no longer appears when you set **Filter type** to **Interpolator**, **Decimator**, or **Sample-rate converter**. The filter builder always implements the filter as a System object.

**Fractional Delay Design — Main Pane****Frequency specifications**

Parameters in this group enable you to specify your filter format, such as the fractional delay and the filter order.

**Order**

If you choose **Specify** for **Order mode**, enter your filter order in this field, or select the order from the drop-down list. `filterBuilder` designs a filter with the order you specify.

**Fractional delay**

Specify a value between 0 and 1 samples for the filter fractional delay. The default value is 0.5 samples.

**Frequency units**

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select **Normalized (0 to 1)** to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—Hz, kHz, MHz, or GHz. Selecting one of the unit options enables the **Input sample rate** parameter.

**Input sample rate**

$F_s$ , specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

**Halfband Filter Design — Main Pane****Filter specifications**

Parameters in this group enable you to specify your filter type and order.



### Impulse response

Select FIR or IIR from the drop-down list, where FIR is the default impulse response. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly.

---

**Note** The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

---

### Order mode

Select Minimum (the default) or Specify from the drop-down list. Selecting Specify enables the **Order** option (see the following sections) so you can enter the filter order.

### Filter type

Select Single-rate, Decimator, or Interpolator. By default, filterBuilder specifies single-rate filters.

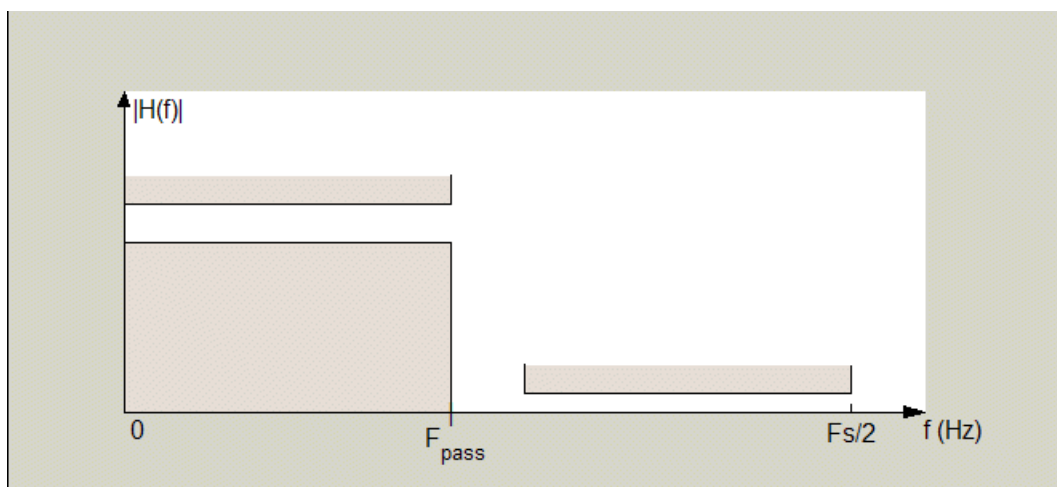
When you design either a decimator or an interpolator, the resulting filter is a bandpass filter that decimates or interpolates your input by a factor of two.

### Order

Enter the filter order. This option is enabled only if Specify was selected for **Order mode**.

### Frequency specifications

The parameters in this group allow you to specify your filter response curve. Graphically, the filter specifications for a halfband lowpass filter look similar to those shown in the following figure.



In the figure, the transition region lies between the end of the passband and the start of the stopband. The width is defined explicitly by the value of **Transition width**.

### Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select Normalized (0 to 1) to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—Hz, kHz, MHz, or GHz. Selecting one of the unit options enables the **Input sample rate** parameter.

**Input sample rate**

$F_s$ , specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

**Transition width**

Specify the width of the transition between the end of the passband and the edge of the stopband. Specify the value in normalized frequency units or the absolute units you select in **Frequency units**.

**Magnitude specifications**

The parameters in this group let you specify the filter response in the passbands and stopbands.

**Magnitude units**

Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.

- Linear — Specify the magnitude in linear units.
- dB — Specify the magnitude in decibels (default).

**Stopband attenuation**

Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels.

**Algorithm**

The parameters in this group allow you to specify the design method and structure that `filterBuilder` uses to implement your filter.

**Design Method**

Lists the design methods available for the frequency and magnitude specifications you entered. For FIR halfband filters, the available design options are `Equiripple` and `Kaiser window`. For IIR halfband filters, the available design options are `Butterworth`, `Elliptic`, and `IIR quasi-linear phase`.

**Design Options**

The following design options are available for FIR halfband filters when the user specifies an equiripple design:

**Minimum phase**

To design a filter that is minimum phase, select **Minimum phase**. Clearing the **Minimum phase** option removes the phase constraint—the resulting design is not minimum phase.

**Stopband Shape**

Stopband shape lets you specify how the stopband changes with increasing frequency. Choose one of the following options:

- Flat — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.
- Linear — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.

- $1/f$  — Specifies that the stopband attenuation changes exponentially as the frequency increases, where  $f$  is the frequency. Set the power (exponent) for the decay in **Stopband decay**.

### Stopband Decay

When you set Stopband shape, Stopband decay specifies the amount of decay applied to the stopband. The following conditions apply to Stopband decay based on the value of Stopband Shape:

- When you set **Stopband shape** to Flat, **Stopband decay** has no effect on the stopband.
- When you set **Stopband shape** to Linear, enter the slope of the stopband in units of dB/rad/s. filterBuilder applies that slope to the stopband.
- When you set **Stopband shape** to  $1/f$ , enter a value for the exponent  $n$  in the relation  $(1/f)^n$  to define the stopband decay. filterBuilder applies the  $(1/f)^n$  relation to the stopband to result in an exponentially decreasing stopband attenuation.

### Filter implementation

#### Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter.

#### Use a System object to implement filter

This check box appears when you set **Filter type** to Single-rate. Selecting this check box gives you the choice of using a System object to implement the filter. By default, the check box is cleared.

This check box no longer appears when you set **Filter type** to either Interpolator or Decimator. The filter builder always implements the filter as a System object.

### Highpass Filter Design — Main Pane

#### Filter specifications

Parameters in this group enable you to specify your filter format, such as the impulse response and the filter order.

#### Impulse response

Select FIR or IIR from the drop-down list, where FIR is the default impulse response. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly.

---

**Note** The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

---

#### Order mode

Select Minimum (the default) or Specify from the drop-down list. Selecting Specify enables the **Order** option so you can enter the filter order.

If your **Impulse response** is IIR, you can specify an equal order for the numerator and denominator, or different numerator and denominator orders. The default is equal orders. To specify a different denominator order, check the **Denominator order** box.

### Filter type

This option is only available if you have the DSP System Toolbox software. Select **Single-rate**, **Decimator**, **Interpolator**, or **Sample-rate converter**. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, `filterBuilder` specifies single-rate filters.

- Selecting **Decimator** or **Interpolator** activates the **Decimation Factor** or the **Interpolation Factor** options respectively.
- Selecting **Sample-rate converter** activates both factors.

When you design either a decimator or an interpolator, the resulting filter is a highpass filter that either decimates or interpolates your input signal.

### Order

Enter the filter order. This option is enabled only if `Specify` was selected for **Order mode**.

### Decimation Factor

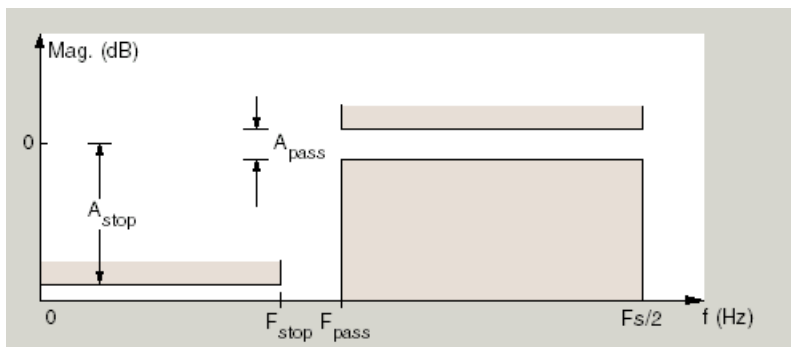
Enter the decimation factor. This option is enabled only if the **Filter type** is set to **Decimator** or **Sample-rate converter**. The default factor value is 2.

### Interpolation Factor

Enter the interpolation factor. This option is enabled only if the **Filter type** is set to **Interpolator** or **Sample-rate converter**. The default factor value is 2.

### Frequency specifications

The parameters in this group allow you to specify your filter response curve. Graphically, the filter specifications look similar to those shown in the following figure.



In the figure, the region between specification values Stopband frequency ( $F_{stop}$ ) and Passband frequency ( $F_{pass}$ ) represents the transition region where the filter response is not explicitly defined.

### Frequency constraints

Select the filter features to use to define the frequency response characteristics. The list contains the following options, when available for the filter specifications.

- **Stopband edge and passband edge** — Define the filter by specifying the frequencies for the edges for the stopband and passband.
- **Passband frequency** — Define the filter by specifying the frequency for the edge of the passband.
- **Stopband frequency** — Define the filter by specifying the frequency for the edges of the stopband.

- **Stopband and half power (3dB) frequencies** — Define the filter by specifying the stopband edge frequency and the 3-dB down point (IIR designs).
- **Half power (3dB) and passband frequencies** — Define the filter by specifying the 3-dB down point and passband edge frequency (IIR designs).
- **Half power (3dB) frequency** — Define the filter by specifying the frequency for the 3-dB point (IIR designs or maxflat FIR).
- **Cutoff (6dB) frequency** — Define the filter by specifying the frequency for the 6-dB point in the filter response (FIR designs).

### Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select **Normalized (0 to 1)** to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—Hz, kHz, MHz, or GHz. Selecting one of the unit options enables the **Input sample rate** parameter.

### Input sample rate

F<sub>s</sub>, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

### Passband frequency

Enter the frequency at the of the passband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

### Stopband frequency

Enter the frequency at the start of the stopband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

### Magnitude specifications

The parameters in this group let you specify the filter response in the passbands and stopbands.

### Magnitude units

Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.

- **Linear** — Specify the magnitude in linear units.
- **dB** — Specify the magnitude in decibels (default).
- **Squared** — Specify the magnitude in squared units.

### Stopband attenuation

Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels.

### Passband ripple

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels.

## Algorithm

The parameters in this group allow you to specify the design method and structure that `filterBuilder` uses to implement your filter.

### Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is usually Butterworth, and the default FIR method is equiripple.

### Scale SOS filter coefficients to reduce chance of overflow

Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

### Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

### Density factor

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 16 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

### Phase constraint

This option only applies when you have the DSP System Toolbox software and when the **Design method** is equiripple. Select one of **Linear**, **Minimum**, or **Maximum**.

**Minimum order** — This option only applies when you have the DSP System Toolbox software and the **Order mode** is **Minimum**.

Select **Any** (default), **Even**, or **Odd**. Selecting **Even** or **Odd** forces the minimum-order design to be an even or odd order.

### Match Exactly

Specifies that the resulting filter design matches either the passband or stopband when you select **Passband** or **Stopband**.

### Stopband Shape

Stopband shape lets you specify how the stopband changes with increasing frequency. Choose one of the following options:

- **Flat** — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.
- **Linear** — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.

- $1/f$  — Specifies that the stopband attenuation changes exponentially as the frequency increases, where  $f$  is the frequency. Set the power (exponent) for the decay in **Stopband decay**.

### Stopband Decay

When you set Stopband shape, Stopband decay specifies the amount of decay applied to the stopband. The following conditions apply to Stopband decay based on the value of Stopband Shape:

- When you set **Stopband shape** to Flat, **Stopband decay** has no effect on the stopband.
- When you set **Stopband shape** to Linear, enter the slope of the stopband in units of dB/rad/s. filterBuilder applies that slope to the stopband.
- When you set **Stopband shape** to  $1/f$ , enter a value for the exponent  $n$  in the relation  $(1/f)^n$  to define the stopband decay. filterBuilder applies the  $(1/f)^n$  relation to the stopband to result in an exponentially decreasing stopband attenuation.

### Wpass

Passband weight. This option only applies when **Impulse response** is FIR and **Order mode** is Specify.

### Wstop

Stopband weight. This option only applies when **Impulse response** is FIR and **Order mode** is Specify.

## Filter implementation

### Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure, and IIR filters use direct-form II filters with SOS.

### Use a System object to implement filter

This check box appears when you set **Filter type** to Single-rate. Selecting this check box gives you the choice of using a System object to implement the filter. By default, the check box is cleared.

This check box no longer appears when you set **Filter type** to Interpolator, Decimator, or Sample-rate converter. The filter builder always implements the filter as a System object.

## Hilbert Filter Design — Main Pane

### Filter specifications

Parameters in this group enable you to specify your filter format, such as the impulse response and the filter order.

### Impulse response

Select FIR or IIR from the drop-down list, where FIR is the default impulse response. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly.

---

**Note** The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

---

### Order mode

This option is only available if you have the DSP System Toolbox software. Select either **Minimum** (the default) or **Specify** from the drop-down list. Selecting **Specify** enables the **Order** option (see the following sections) so you can enter the filter order.

### Filter type

This option is only available if you have the DSP System Toolbox software. Select **Single-rate**, **Decimator**, **Interpolator**, or **Sample-rate converter**. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, `filterBuilder` specifies single-rate filters.

- Selecting **Decimator** or **Interpolator** activates the **Decimation Factor** or the **Interpolation Factor** options respectively.
- Selecting **Sample-rate converter** activates both factors.

When you design either a decimator or an interpolator, the resulting filter is a bandpass filter that either decimates or interpolates your input signal.

### Order

Enter the filter order. This option is enabled only if **Specify** was selected for **Order mode**.

### Decimation Factor

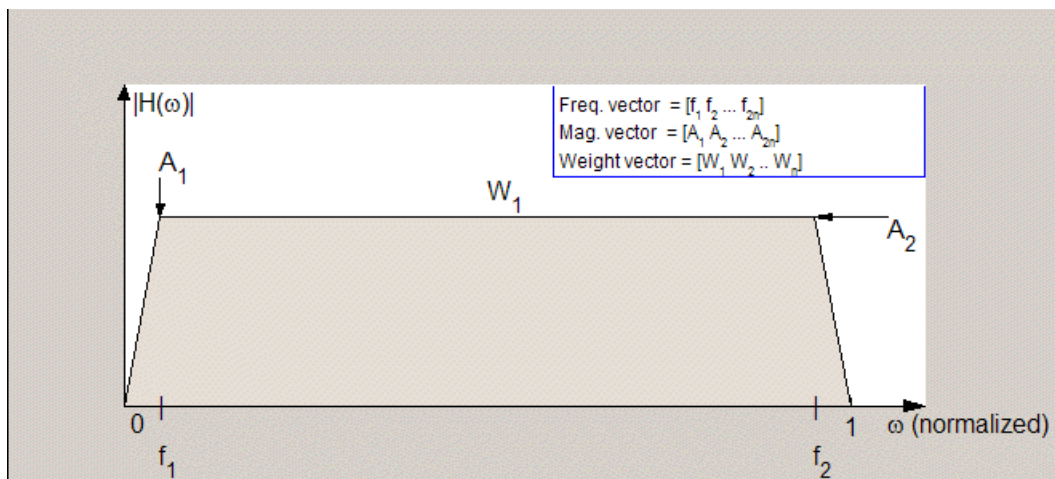
Enter the decimation factor. This option is enabled only if the **Filter type** is set to **Decimator** or **Sample-rate converter**. The default factor value is 2.

### Interpolation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to **Interpolator** or **Sample-rate converter**. The default factor value is 2.

### Frequency specifications

The parameters in this group allow you to specify your filter response curve. Graphically, the filter specifications look similar to those shown in the following figure.



In the figure, the regions between 0 and  $f_1$  and between  $f_2$  and 1 represent the transition regions where the filter response is explicitly defined by the transition width.



### Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select **Normalized (0 to 1)** to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—Hz, kHz, MHz, or GHz. Selecting one of the unit options enables the **Input sample rate** parameter.

### Input sample rate

F<sub>s</sub>, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

### Transition width

Specify the width of the transitions at the ends of the passband. Specify the value in normalized frequency units or the absolute units you select in **Frequency units**.

### Magnitude specifications

The parameters in this group let you specify the filter response in the passbands and stopbands.

### Magnitude units

Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.

- **Linear** — Specify the magnitude in linear units.
- **dB** — Specify the magnitude in decibels (default)
- **Squared** — Specify the magnitude in squared units.

### Passband ripple

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels.

### Algorithm

The parameters in this group allow you to specify the design method and structure that filterBuilder uses to implement your filter.

### Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is usually Butterworth, and the default FIR method is equiripple.

### Scale SOS filter coefficients to reduce chance of overflow

Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

### Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as

you select different design methods and filter specifications. The following options represent some of the most common ones available.

### **Density factor**

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 16 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

### **FIR Type**

This option is only available in a minimum-order design. Specify whether to design a type 3 or a type 4 FIR filter. The filter type is defined as follows:

- Type 3 — FIR filter with even order antisymmetric coefficients
- Type 4 — FIR filter with odd order antisymmetric coefficients

Select 3 or 4 from the drop-down list.

### **Filter implementation**

#### **Structure**

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure, and IIR filters use direct-form II filters with SOS.

#### **Use a System object to implement filter**

This check box appears when you set **Filter type** to **Single-rate**. Selecting this check box gives you the choice of using a System object to implement the filter. By default, the check box is cleared.

This check box no longer appears when you set **Filter type** to **Interpolator**, **Decimator**, or **Sample-rate converter**. The filter builder always implements the filter as a System object.

### **Inverse Sinc Filter Design — Main Pane**

#### **Filter specifications**

Parameters in this group enable you to specify your filter format, such as the impulse response and the filter order.

#### **Order mode**

Select **Minimum** (the default) or **Specify** from the drop-down list. Selecting **Specify** enables the **Order** option (see the following sections) so you can enter the filter order.

#### **Response type**

Select **Lowpass** or **Highpass** to design an inverse sinc lowpass or highpass filter.

#### **Filter type**

Select **Single-rate**, **Decimator**, **Interpolator**, or **Sample-rate converter**. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, `filterBuilder` specifies single-rate filters.

- Selecting **Decimator** or **Interpolator** activates the **Decimation Factor** or the **Interpolation Factor** options respectively.
- Selecting **Sample-rate converter** activates both factors.

When you design either a decimator or an interpolator, the resulting filter is a bandpass filter that either decimates or interpolates your input signal.

### Order

Enter the filter order. This option is enabled only if **Specify** was selected for **Order mode**.

### Decimation Factor

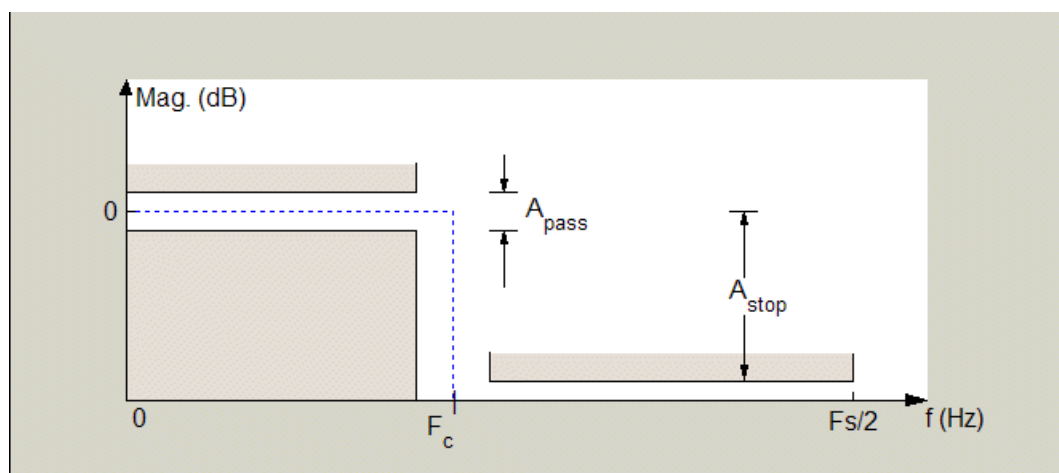
Enter the decimation factor. This option is enabled only if the **Filter type** is set to **Decimator** or **Sample-rate converter**. The default factor value is 2.

### Interpolation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to **Interpolator** or **Sample-rate converter**. The default factor value is 2.

### Frequency specifications

The parameters in this group allow you to specify your filter response curve. Graphically, the filter specifications look similar to those shown in the following figure.



Regions between specification values such as Passband frequency ( $F_{pass}$ ) and Stopband frequency ( $F_{stop}$ ) represent transition regions where the filter response is not explicitly defined.

### Frequency constraints

This option is only available when you specify the filter order. The following options are available:

- **Passband and stopband frequencies** — Define the filter by specifying the frequencies for the edges for the stop- and passbands.
- **Passband frequency** — Define the filter by specifying frequencies for the edges of the passband.
- **Stopband frequency** — Define the filter by specifying frequencies for the edges of the stopbands.
- **Cutoff (6dB) frequency** — The 6-dB point is the frequency for the point 6 dB below the passband value.

**Frequency units**

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select **Normalized (0 to 1)** to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—Hz, kHz, MHz, or GHz. Selecting one of the unit options enables the **Input sample rate** parameter.

**Input sample rate**

F<sub>s</sub>, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

**Passband frequency**

Enter the frequency at the end of the passband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

**Stopband frequency**

Enter the frequency at the start of the stopband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

**Magnitude specifications**

The parameters in this group let you specify the filter response in the passbands and stopbands.

**Magnitude units**

Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.

- **Linear** — Specify the magnitude in linear units.
- **dB** — Specify the magnitude in decibels (default)
- **Squared** — Specify the magnitude in squared units.

**Passband ripple**

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels.

**Stopband attenuation**

Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels.

**Algorithm**

The parameters in this group allow you to specify the design method and structure that `filterBuilder` uses to implement your filter.

**Design Method**

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is usually Butterworth, and the default FIR method is equiripple.

**Design Options**

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

### Density factor

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 16 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

### Phase constraint

Available options are Linear, Minimum, and Maximum.

### Stopband Shape

Stopband shape lets you specify how the stopband changes with increasing frequency. Choose one of the following options;

- Flat — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.
- Linear — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.
- 1/f — Specifies that the stopband attenuation changes exponentially as the frequency increases, where  $f$  is the frequency. Set the power (exponent) for the decay in **Stopband decay**.

### Stopband Decay

When you set Stopband shape, Stopband decay specifies the amount of decay applied to the stopband. the following conditions apply to Stopband decay based on the value of Stopband Shape:

- When you set **Stopband shape** to Flat, **Stopband decay** has no affect on the stopband.
- When you set **Stopband shape** to Linear, enter the slope of the stopband in units of dB/rad/s. filterBuilder applies that slope to the stopband.
- When you set **Stopband shape** to 1/f, enter a value for the exponent  $n$  in the relation  $(1/f)^n$  to define the stopband decay. filterBuilder applies the  $(1/f)^n$  relation to the stopband to result in an exponentially decreasing stopband attenuation.

### Sinc frequency factor

A frequency dilation factor. The sinc frequency factor,  $C$ , parameterizes the passband magnitude response for a lowpass design through  $H(\omega) = \text{sinc}(C\omega)^{-P}$  and for a highpass design through  $H(\omega) = \text{sinc}(C(1-\omega))^{-P}$ .

### Sinc power

Negative power of passband magnitude response. The sinc power,  $P$ , parameterizes the passband magnitude response for a lowpass design through  $H(\omega) = \text{sinc}(C\omega)^{-P}$  and for a highpass design through  $H(\omega) = \text{sinc}(C(1-\omega))^{-P}$ .

## Filter implementation

### Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure.

### Use a System object to implement filter

This check box appears when you set **Filter type** to `Single-rate`. Selecting this check box gives you the choice of using a System object to implement the filter. By default, the check box is cleared.

This check box no longer appears when you set **Filter type** to `Interpolator`, `Decimator`, or `Sample-rate converter`. The filter builder always implements the filter as a System object.

## Lowpass Filter Design — Main Pane

### Filter specifications

Parameters in this group enable you to specify your filter format, such as the impulse response and the filter order.

### Impulse response

Select FIR or IIR from the drop-down list, where FIR is the default impulse response. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly.

---

**Note** The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

---

### Order mode

Select `Minimum` (the default) or `Specify` from the drop-down list. Selecting `Specify` enables the **Order** option (see the following sections) so you can enter the filter order.

If your **Impulse response** is IIR, you can specify an equal order for the numerator and denominator, or different numerator and denominator orders. The default is equal orders. To specify a different denominator order, check the **Denominator order** box.

### Filter type

This option is only available if you have the DSP System Toolbox. Select `Single-rate`, `Decimator`, `Interpolator`, or `Sample-rate converter`. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, `filterBuilder` specifies single-rate filters.

- Selecting `Decimator` or `Interpolator` activates the **Decimation Factor** or the **Interpolation Factor** options respectively.
- Selecting `Sample-rate converter` activates both factors.

When you design either a decimator or an interpolator, the resulting filter is a bandpass filter that either decimates or interpolates your input signal.

### Order

Enter the filter order. This option is enabled only if `Specify` was selected for **Order mode**.

### Decimation Factor

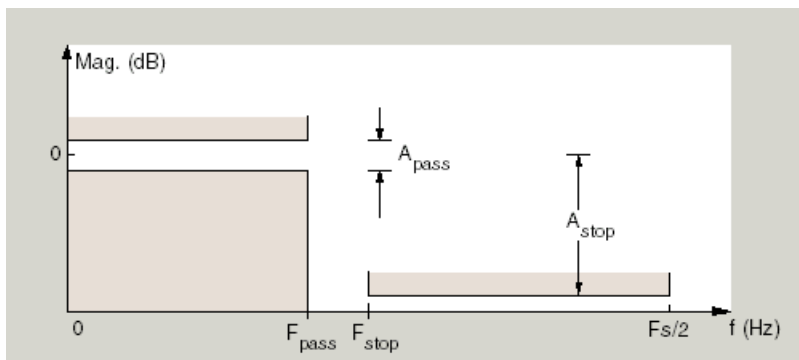
Enter the decimation factor. This option is enabled only if the **Filter type** is set to Decimator or Sample-rate converter. The default factor value is 2.

### Interpolation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to Interpolator or Sample-rate converter. The default factor value is 2.

### Frequency specifications

The parameters in this group allow you to specify your filter response curve. Graphically, the filter specifications look similar to the one shown in the following figure.



In the figure, regions between specification values such as Passband frequency ( $F_{\text{pass}}$ ) and Stopband frequency ( $F_{\text{stop}}$ ) represent transition regions where the filter response is not explicitly defined.

### Frequency constraints

Select the filter features to use to define the frequency response characteristics. The list contains the following options, when available for the filter specifications.

- **Passband and stopband frequencies** — Define the filter by specifying the frequencies for the edge of the stopband and passband.
- **Passband frequency** — Define the filter by specifying the frequency for the edge of the passband.
- **Stopband frequency** — Define the filter by specifying the frequency for the edges of the stopband.
- **Passband edge and 3dB point** — Define the filter by specifying the passband edge frequency and the 3-dB down point (IIR designs).
- **Half power (3dB) and stopband frequencies** — Define the filter by specifying the 3-dB down point and stopband edge frequency (IIR designs).
- **Half power (3dB) frequency** — Define the filter by specifying the frequency for the 3-dB point (IIR designs or maxflat FIR).
- **Cutoff (6dB) frequency** — Define the filter by specifying the frequency for the 6-dB point in the filter response (FIR designs).

### Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select **Normalized (0 to 1)** to enter frequencies in normalized form. This behavior

is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—Hz, kHz, MHz, or GHz. Selecting one of the unit options enables the **Input sample rate** parameter.

### **Input sample rate**

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

### **Passband frequency**

Enter the frequency at the of the passband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

### **Stopband frequency**

Enter the frequency at the start of the stopband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

### **Magnitude specifications**

The parameters in this group let you specify the filter response in the passbands and stopbands.

#### **Magnitude units**

Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.

- Linear — Specify the magnitude in linear units.
- dB — Specify the magnitude in decibels (default)
- Squared — Specify the magnitude in squared units.

#### **Passband ripple**

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels.

#### **Stopband attenuation**

Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels.

### **Algorithm**

The parameters in this group allow you to specify the design method and structure that `filterBuilder` uses to implement your filter.

#### **Design Method**

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is usually Butterworth, and the default FIR method is equiripple.

#### **Scale SOS filter coefficients to reduce chance of overflow**

Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

### **Design Options**



The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

### Density factor

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 16 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

### Phase constraint

This option only applies when you have the DSP System Toolbox software and when the **Design method** is equiripple. Select one of **Linear**, **Minimum**, or **Maximum**.

**Minimum order** — This option only applies when you have the DSP System Toolbox software and the **Order mode** is **Minimum**.

Select **Any** (default), **Even**, or **Odd**. Selecting **Even** or **Odd** forces the minimum-order design to be an even or odd order.

### Match Exactly

Specifies that the resulting filter design matches either the passband or stopband when you select **Passband** or **Stopband**.

### Stopband Shape

Stopband shape lets you specify how the stopband changes with increasing frequency. Choose one of the following options:

- **Flat** — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.
- **Linear** — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.
- **1/f** — Specifies that the stopband attenuation changes exponentially as the frequency increases, where  $f$  is the frequency. Set the power (exponent) for the decay in **Stopband decay**.

### Stopband Decay

When you set Stopband shape, Stopband decay specifies the amount of decay applied to the stopband. The following conditions apply to Stopband decay based on the value of Stopband Shape:

- When you set **Stopband shape** to **Flat**, **Stopband decay** has no effect on the stopband.
- When you set **Stopband shape** to **Linear**, enter the slope of the stopband in units of dB/rad/s. `filterBuilder` applies that slope to the stopband.
- When you set **Stopband shape** to **1/f**, enter a value for the exponent  $n$  in the relation  $(1/f)^n$  to define the stopband decay. `filterBuilder` applies the  $(1/f)^n$  relation to the stopband to result in an exponentially decreasing stopband attenuation.

**Wpass**

Passband weight. This option only applies when **Impulse response** is FIR and **Order mode** is Specify.

**Wstop**

Stopband weight. This option only applies when **Impulse response** is FIR and **Order mode** is Specify.

**Filter implementation****Structure**

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure, and IIR filters use direct-form II filters with SOS.

**Use a System object to implement filter**

This check box appears when you set **Filter type** to Single-rate. Selecting this check box gives you the choice of using a System object to implement the filter. By default, the check box is cleared.

This check box no longer appears when you set **Filter type** to Interpolator, Decimator, or Sample-rate converter. The filter builder always implements the filter as a System object.

**Notch**

See “Peak/Notch Filter Design — Main Pane” on page 5-642.

**Nyquist Filter Design — Main Pane****Filter specifications**

Parameters in this group enable you to specify your filter format, such as the impulse response and the filter order.

**Band**

Specifies the location of the center of the transition region between the passband and the stopband. The center of the transition region, *bw*, is calculated using the value for Band:

$$bw = Fs/(2 \times \text{Band}).$$

**Impulse response**

Select FIR or IIR from the drop-down list, where FIR is the default impulse response. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly.

---

**Note** The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

---

**Order mode**

Select Minimum (the default) or Specify from the drop-down list. Selecting Specify enables the **Order** option (see the following sections) so you can enter the filter order.

### Filter type

Select **Single-rate**, **Decimator**, **Interpolator**, or **Sample-rate converter**. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, filterBuilder specifies single-rate filters.

- Selecting **Decimator** or **Interpolator** activates the **Decimation Factor** or the **Interpolation Factor** options respectively.
- Selecting **Sample-rate converter** activates both factors.

When you design either a decimator or an interpolator, the resulting filter is a bandpass filter that either decimates or interpolates your input signal.

### Order

Enter the filter order. This option is enabled only if **Specify** was selected for **Order mode**.

### Decimation Factor

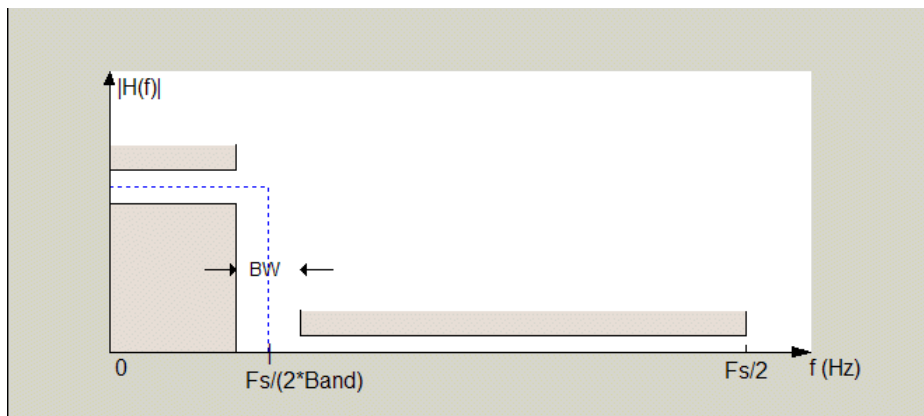
Enter the decimation factor. This option is enabled only if the **Filter type** is set to **Decimator** or **Sample-rate converter**. The default factor value is 2.

### Interpolation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to **Interpolator** or **Sample-rate converter**. The default factor value is 2.

### Frequency specifications

The parameters in this group allow you to specify your filter response curve. Graphically, the filter specifications look similar to those shown in the following figure.



In the figure,  $BW$  is the width of the transition region and **Band** determines the location of the center of the region.

### Frequency constraints

Select the filter features to use to define the frequency response characteristics. The list contains the following options, when available for the filter specifications.

- **Transition width** — Define the filter using transition width and stopband attenuation or transition width and order.
- **Unconstrained** — Define the filter by specifying the filter order and having no constraints on the transition width and stopband attenuation. You can add constraints on the magnitude by specifying the stopband attenuation.

**Frequency units**

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select **Normalized** (0 to 1) to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—Hz, kHz, MHz, or GHz. Selecting one of the unit options enables the **Input sample rate** parameter.

**Input sample rate**

F<sub>s</sub>, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

**Transition width**

Specify the width of the transition between the end of the passband and the edge of the stopband. Specify the value in normalized frequency units or the absolute units you select in **Frequency units**.

**Magnitude specifications**

The parameters in this group let you specify the filter response in the passbands and stopbands.

**Magnitude units**

Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.

- **Linear** — Specify the magnitude in linear units.
- **dB** — Specify the magnitude in decibels (default)
- **Squared** — Specify the magnitude in squared units.

**Stopband attenuation**

Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels.

**Algorithm**

The parameters in this group allow you to specify the design method and structure that `filterBuilder` uses to implement your filter.

**Design Method**

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is usually Butterworth, and the default FIR method is equiripple.

**Scale SOS filter coefficients to reduce chance of overflow**

Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

**Design Options**

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as

you select different design methods and filter specifications. The following options represent some of the most common ones available.

### Density factor

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 16 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

### Minimum phase

To design a filter that is minimum phase, select **Minimum phase**. Clearing the **Minimum phase** option removes the phase constraint—the resulting design is not minimum phase.

### Minimum order

When you select this parameter, the design method determines and designs the minimum order filter to meet your specifications. Some filters do not provide this parameter. Select Any, Even, or Odd from the drop-down list to direct the design to be any minimum order, or minimum even order, or minimum odd order.

---

**Note** Generally, **Minimum order** designs are not available for IIR filters.

---

### Match Exactly

Specifies that the resulting filter design matches either the passband or stopband or both bands when you select passband or stopband or both from the drop-down list.

### Stopband Shape

Stopband shape lets you specify how the stopband changes with increasing frequency. Choose one of the following options:

- **Flat** — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.
- **Linear** — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.
- **1/f** — Specifies that the stopband attenuation changes exponentially as the frequency increases, where  $f$  is the frequency. Set the power (exponent) for the decay in **Stopband decay**.

### Stopband Decay

When you set Stopband shape, Stopband decay specifies the amount of decay applied to the stopband. the following conditions apply to Stopband decay based on the value of Stopband Shape:

- When you set **Stopband shape** to **Flat**, **Stopband decay** has no affect on the stopband.
- When you set **Stopband shape** to **Linear**, enter the slope of the stopband in units of dB/rad/s. `filterBuilder` applies that slope to the stopband.
- When you set **Stopband shape** to **1/f**, enter a value for the exponent  $n$  in the relation  $(1/f)^n$  to define the stopband decay. `filterBuilder` applies the  $(1/f)^n$  relation to the stopband to result in an exponentially decreasing stopband attenuation.

## Filter implementation

### Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure, and IIR filters use direct-form II filters with SOS.

### Use a System object to implement filter

This check box appears when you set **Filter type** to **Single-rate**. Selecting this check box gives you the choice of using a System object to implement the filter. By default, the check box is cleared.

This check box no longer appears when you set **Filter type** to **Interpolator**, **Decimator**, or **Sample-rate converter**. The filter builder always implements the filter as a System object.

## Octave Filter Design — Main Pane

### Filter specifications

#### Order

Specify filter order. Possible values are: 4, 6, 8, 10.

#### Bands per octave

Specify the number of bands per octave. Possible values are: 1, 3, 6, 12, 24.

#### Frequency units

Specify frequency units as Hz or kHz.

#### Input sample rate

Specify the input sampling frequency in the frequency units specified previously.

#### Center Frequency

Select from the drop-down list of available center frequency values.

### Algorithm

#### Design Method

Butterworth is the design method used for this type of filter.

#### Scale SOS filter coefficients to reduce chance of overflow

Select the check box to scale the filter coefficients.

## Filter implementation

### Structure

Specify filter structure. Choose from:

- Direct-form I SOS
- Direct-form II SOS
- Direct-form I transposed SOS
- Direct-form II transposed SOS

### Use a System object to implement filter

Selecting this check box gives you the choice of using a System object to implement the filter. By default, the check box is cleared. When the current design method or structure is not supported by a system object filter, then this check box is disabled.

## Parametric Equalizer Filter Design — Main Pane

### Filter specifications

#### Order mode

Select **Minimum** to design a minimum order filter that meets the design specifications, or **Specify** to enter a specific filter order. The order mode also affects the possible frequency constraints, which in turn limit the gain specifications. For example, if you specify a **Minimum** order filter, the available frequency constraints are:

- Center frequency, bandwidth, passband width
- Center frequency, bandwidth, stopband width

If you select **Specify**, the available frequency constraints are:

- Center frequency, bandwidth
- Center frequency, quality factor
- Shelf type, cutoff frequency, quality factor
- Shelf type, cutoff frequency, shelf slope parameter
- Low frequency, high frequency

#### Order

This parameter is enabled only if the **Order mode** is set to **Specify**. Enter the filter order in this text box.

### Frequency specifications

Depending on the filter order, the possible frequency constraints change. Once you choose the frequency constraints, the input boxes in this area change to reflect the selection.

#### Frequency constraints

Select the specification to represent the frequency constraints. The following options are available:

- Center frequency, bandwidth, passband width (available for minimum order only)
- Center frequency, bandwidth, stopband width (available for minimum order only)
- Center frequency, bandwidth (available for a specified order only)
- Center frequency, quality factor (available for a specified order only)
- Shelf type, cutoff frequency, quality factor (available for a specified order only)
- Shelf type, cutoff frequency, shelf slope parameter (available for a specified order only)
- Low frequency, high frequency (available for a specified order only)

**Frequency units**

Select the frequency units from the available drop down list (Normalized, Hz, kHz, MHz, GHz). If Normalized is selected, then the **Input sample rate** box is disabled for input.

**Input sample rate**

Enter the input sampling frequency. This input box is disabled for input if Normalized is selected in the **Frequency units** input box.

**Center frequency**

Enter the center frequency in the units specified by the value in **Frequency units**.

**Bandwidth**

The bandwidth determines the frequency points at which the filter magnitude is attenuated by the value specified as the **Bandwidth gain** in the **Gain specifications** section. By default, the **Bandwidth gain** defaults to  $\text{db}(\text{sqrt}(.5))$ , or -3 dB relative to the center frequency. The **Bandwidth** property only applies when the **Frequency constraints** are: Center frequency, bandwidth, passband width, Center frequency, bandwidth, stopband width, or Center frequency, bandwidth.

**Passband width**

The passband width determines the frequency points at which the filter magnitude is attenuated by the value specified as the **Passband gain** in the **Gain specifications** section. This option is enabled only if the filter is of minimum order, and the frequency constraint selected is Center frequency, bandwidth, passband width.

**Stopband width**

The stopband width determines the frequency points at which the filter magnitude is attenuated by the value specified as the **Stopband gain** in the **Gain specifications** section. This option is enabled only if the filter is of minimum order, and the frequency constraint selected is Center frequency, bandwidth, stopband width.

**Low frequency**

Enter the low frequency cutoff. This option is enabled only if the filter order is user specified and the frequency constraint selected is Low frequency, high frequency. The filter magnitude is attenuated by the amount specified in **Bandwidth gain**.

**High frequency**

Enter the high frequency cutoff. This option is enabled only if the filter order is user specified and the frequency constraint selected is Low frequency, high frequency. The filter magnitude is attenuated by the amount specified in **Bandwidth gain**.

**Gain specifications**

Depending on the filter order and frequency constraints, the possible gain constraints change. Also, once you choose the gain constraints the input boxes in this area change to reflect the selection.

**Gain constraints**

Select the specification array to represent gain constraints, and remember that not all of these options are available for all configurations. The following is a list of all available options:

- Reference, center frequency, bandwidth, passband
- Reference, center frequency, bandwidth, stopband
- Reference, center frequency, bandwidth, passband, stopband



- Reference, center frequency, bandwidth

### Gain units

Specify the gain units either dB or squared. These units are used for all gain specifications in the dialog box.

### Reference gain

The reference gain determines the level to which the filter magnitude attenuates in **Gain units**. The reference gain is a *floor* gain for the filter magnitude response. For example, you may use the reference gain together with the **Center frequency gain** to leave certain frequencies unattenuated (reference gain of 0 dB) while boosting other frequencies.

### Bandwidth gain

Specifies the gain in **Gain units** at which the bandwidth is defined. This property applies only when the **Frequency constraints** specification contains a bandwidth parameter, or is Low frequency, high frequency.

### Center frequency gain

Specify the center frequency in **Gain units**

### Passband gain

The passband gain determines the level in **Gain units** at which the passband is defined. The passband is determined either by the **Passband width** value, or the **Low frequency** and **High frequency** values in the **Frequency specifications** section.

### Stopband gain

The stopband gain is the level in **Gain units** at which the stopband is defined. This property applies only when the **Order mode** is minimum and the **Frequency constraints** are Center frequency, bandwidth, stopband width.

### Boost/cut gain

The boost/cut gain applies only when the designing a shelving filter. Shelving filters include the Shelf type parameter in the **Frequency constraints** specification. The gain in the passband of the shelving filter is increased by **Boost/cut gain** dB from a *floor* gain of 0 dB.

## Algorithm

### Design method

Select the design method from the drop-down list. Different IIR design methods are available depending on the filter constraints you specify.

### Scale SOS filter coefficients to reduce chance of overflow

Select the check box to scale the filter coefficients.

## Filter implementation

### Structure

Select filter structure. The possible choices are:

- Direct-form I SOS
- Direct-form II SOS
- Direct-form I transposed SOS
- Direct-form II transposed SOS

**Use a System object to implement filter**

Selecting this check box gives you the choice of using a System object to implement the filter. By default, the check box is cleared. When the current design method or structure is not supported by a System object filter, then this check box is disabled.

**Peak/Notch Filter Design – Main Pane****Filter specifications**

In this area you can specify whether you want to design a peaking filter or a notching filter, as well as the order of the filter.

**Response**

Select Peak or Notch from the drop-down box.

**Order**

Enter the filter order. The order must be even.

**Frequency specifications**

This group of parameters allows you to specify frequency constraints and units.

**Frequency Constraints**

Select the frequency constraints for filter specification. There are two choices as follows:

- Center frequency and quality factor
- Center frequency and bandwidth

**Frequency units**

The frequency units are normalized by default. If you specify units other than normalized, `filterBuilder` assumes that you wish to specify an input sampling frequency, and enables this input box. The choice of frequency units are: Normalized (0 to 1), Hz, kHz, MHz, GHz.

**Input sample rate**

This input box is enabled if **Frequency units** other than Normalized (0 to 1) are specified. Enter the input sampling frequency.

**Center frequency**

Enter the center frequency in the units you specified in **Frequency units**.

**Quality Factor**

This input box is enabled only when Center frequency and quality factor is chosen for the **Frequency Constraints**. Enter the quality factor.

**Bandwidth**

This input box is enabled only when Center frequency and bandwidth is chosen for the **Frequency Constraints**. Enter the bandwidth.

**Magnitude specifications**

This group of parameters allows you to specify the magnitude constraints, as well as their values and units.

### **Magnitude Constraints**

Depending on the choice of constraints, the other input boxes are enabled or disabled. Select from four magnitude constraints available:

- Unconstrained
- Passband ripple
- Stopband attenuation
- Passband ripple and stopband attenuation

### **Magnitude units**

Select the magnitude units: either dB or squared.

### **Passband ripple**

This input box is enabled if the magnitude constraints selected are Passband ripple or Passband ripple and stopband attenuation. Enter the passband ripple.

### **Stopband attenuation**

This input box is enabled if the magnitude constraints selected are Stopband attenuation or Passband ripple and stopband attenuation. Enter the stopband attenuation.

### **Algorithm**

The parameters in this group allow you to specify the design method and structure that filterBuilder uses to implement your filter.

### **Design Method**

Lists all design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter the methods available to design filters changes as well.

### **Scale SOS filter coefficients to reduce chance of overflow**

Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

### **Filter implementation**

#### **Structure**

Lists all available filter structures for the filter specifications and design method you select. The typical options are:

- Direct-form I SOS
- Direct-form II SOS
- Direct-form I transposed SOS
- Direct-form II transposed SOS

#### **Use a System object to implement filter**

Selecting this check box gives you the choice of using a System object to implement the filter. By default, the check box is cleared. When the current design method or structure is not supported by a System object filter, then this check box is disabled.

## Pulse-shaping Filter Design —Main Pane

### Filter specifications

Parameters in this group enable you to specify the shape and length of the filter.

#### Pulse shape

Select the shape of the impulse response from the following options:

- Raised Cosine
- Square Root Raised Cosine
- Gaussian

#### Order mode

This specification is only available for raised cosine and square root raised cosine filters. For these filters, select one of the following options:

- **Minimum**— This option will result in the minimum-length filter satisfying the user-specified **Frequency specifications**.
- **Specify order**—This option allows the user to construct a raised cosine or square root cosine filter of a specified order by entering an even number in the **Order** input box. The length of the impulse response will be  $\text{Order}+1$ .
- **Specify symbols**—This option enables the user to specify the length of the impulse response in an alternative manner. If **Specify symbols** is chosen, the **Order** input box changes to the **Number of symbols** input box.

#### Samples per symbol

Specify the oversampling factor. Increasing the oversampling factor guards against aliasing and improves the FIR filter approximation to the ideal frequency response. If **Order** is specified in **Number of symbols**, the filter length will be  $\text{Number of symbols} * \text{Samples per symbol} + 1$ . The product  $\text{Number of symbols} * \text{Samples per symbol}$  must be an even number.

If a Gaussian filter is specified, the filter length must be specified in **Number of symbols** and **Samples per symbol**. The product  $\text{Number of symbols} * \text{Samples per symbol}$  must be an even number. The filter length will be  $\text{Number of symbols} * \text{Samples per symbol} + 1$ .

#### Filter Type

This option is only available if you have the DSP System Toolbox software. Choose **Single rate**, **Decimator**, **Interpolator**, or **Sample-rate converter**. If you select **Decimator** or **Interpolator**, the decimation and interpolation factors default to the value of the **Samples per symbol**. If you select **Sample-rate converter**, the interpolation factor defaults to **Samples per symbol** and the decimation factor defaults to 3.

### Frequency specifications

Parameters in this group enable you to specify the frequency response of the filter. For raised cosine and square root raised cosine filters, the frequency specifications include:

#### Rolloff factor

The rolloff factor takes values in the range [0,1]. The smaller the rolloff factor, the steeper the transition in the stopband.

### Frequency units

The frequency units are normalized by default. If you specify units other than normalized, `filterBuilder` assumes that you wish to specify an input sampling frequency, and enables this input box. The choice of frequency units are: Normalized (0 to 1), Hz, kHz, MHz, GHz

For a Gaussian pulse shape, the available frequency specifications are:

### Bandwidth-time product

This option allows the user to specify the width of the Gaussian filter. Note that this is independent of the length of the filter. The bandwidth-time product (BT) must be a positive real number. Smaller values of the bandwidth-time product result in larger pulse widths in time and steeper stopband transitions in the frequency response.

### Frequency units

The frequency units are normalized by default. If you specify units other than normalized, `filterBuilder` assumes that you wish to specify an input sampling frequency, and enables this input box. The choice of frequency units are: Normalized (0 to 1), Hz, kHz, MHz, GHz

### Magnitude specifications

If the **Order mode** is specified as Minimum, the **Magnitude units** may be selected from:

- dB— Specify the magnitude in decibels (default).
- Linear— Specify the magnitude in linear units.

### Algorithm

The only **Design method** available for FIR pulse-shaping filters is the Window method.

### Filter implementation

#### Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure.

#### Use a System object to implement filter

This check box appears when you set **Filter type** to `Single-rate`. Selecting this check box gives you the choice of using a System object to implement the filter. By default, the check box is cleared.

This check box no longer appears when you set **Filter type** to `Interpolator`, `Decimator`, or `Sample-rate converter`. The filter builder always implements the filter as a System object.

## Version History

Introduced in R2009a

## filtstates.cic

Store CIC filter states

### Description

`filtstates.cic` objects hold the states information for CIC filters. Once you create a CIC filter, the states for the filter are stored in `filtstates.cic` objects, and you can access them and change them as you would any property of the filter. This arrangement parallels that of the `filtstates` object that IIR direct-form I filters use (refer to `filtstates` for more information).

Each `States` property in the CIC filter comprises two properties — `Numerator` and `Comb` — that hold `filtstates.cic` objects. Within the `filtstates.cic` objects are the numerator-related and comb-related filter states. The states are column vectors, where each column represents the states for one section of the filter. For example, a CIC filter with four decimator sections and four interpolator sections has `filtstates.cic` objects that contain four columns of states each.

### Examples

#### Integrator and Comb States of a CIC Filter

Construct an object with integrator and comb states as vectors of zeros.

```
h = filtstates.cic(zeros(4,1),zeros(4,1));
```

`h` has zero states now. You can use `int` to see the states as 32-bit integers.

```
intStates = int(h.Integrator)
```

```
intStates = 4x1 int32 column vector
```

```
0  
0  
0  
0
```

```
combStates = int(h.Comb)
```

```
combStates = 4x1 int32 column vector
```

```
0  
0  
0  
0
```

### Version History

Introduced in R2011a

**See Also**

`dsp.CICDecimator` | `dsp.CICInterpolator` | `filtstates`

## firband

Constrained-band equiripple FIR filter

### Syntax

```
b = firband(n,f,a,w,c)
b = firband(n,f,a,s)
b = firband( __ , '1')
b = firband( __ , 'minphase')
b = firband( __ , 'check')
b = firband( __ , {lgrid})
[b,err] = firband( __ )
[b,err,res] = firband( __ )
```

### Description

`b = firband(n,f,a,w,c)` designs filters having constrained error magnitudes (ripples) using the minimax filter design algorithm.

`b = firband(n,f,a,s)` is used to design filters with special properties at certain frequency points specified by `s`.

`b = firband( __ , '1')` designs a type 1 filter (even-order symmetric). You can also specify type 2 (odd-order symmetric), type 3 (even-order antisymmetric), or type 4 (odd-order antisymmetric) filters. Note there are restrictions on `a` at `f = 0` or `f = 1` for types 2, 3, and 4.

`b = firband( __ , 'minphase')` designs a minimum-phase FIR filter. You can also specify `'maxphase'`.

`b = firband( __ , 'check')` produces a warning when there are potential transition-region anomalies in the filter response.

`b = firband( __ , {lgrid})`, where `{lgrid}` is a scalar cell array containing an integer, controls the density of the frequency grid.

`[b,err] = firband( __ )` returns the unweighted approximation error magnitudes. `err` has one element for each independent approximation error.

`[b,err,res] = firband( __ )` returns a structure `res` of optional results computed by `firband`.

### Examples

#### Design a Constrained Lowpass Filter

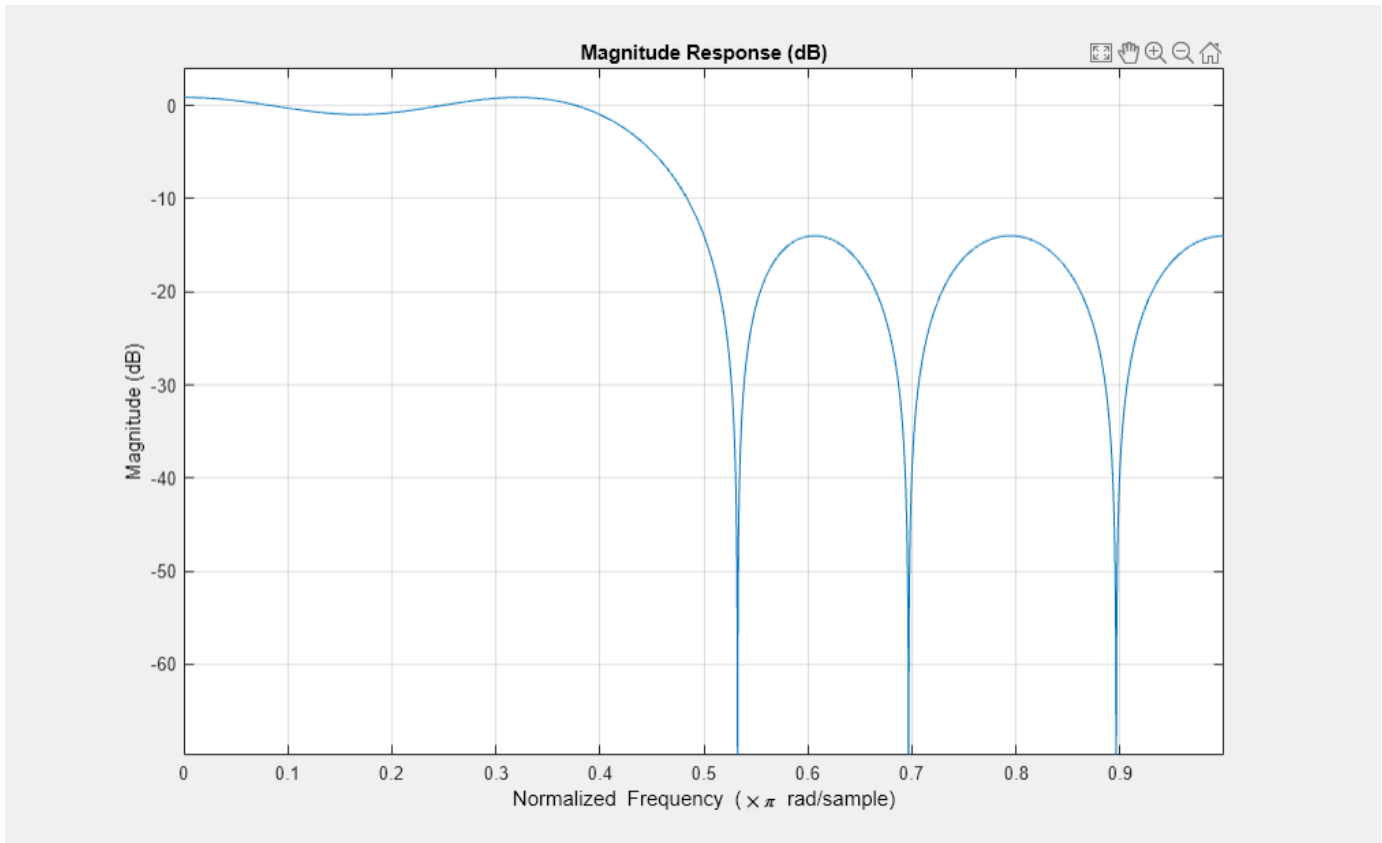
Design a 12th-order lowpass filter with a constraint on the filter response.

```
b = firband(12,[0 0.4 0.5 1], [1 1 0 0], ...
[1 0.2], {'w' 'c'});
```



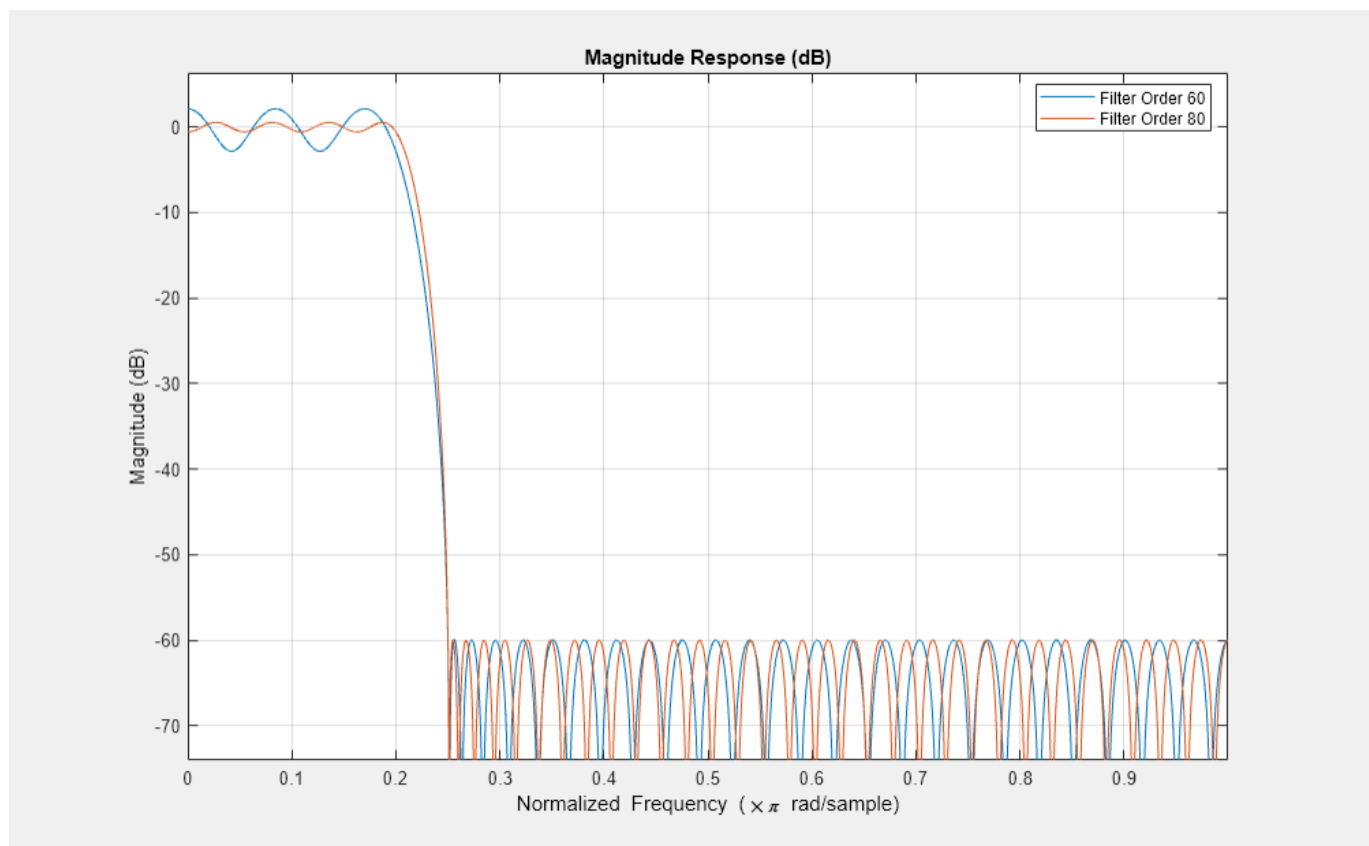
Using fvtool to display the result b shows you the response of the filter you designed.

```
fvtool(b)
```



Design two filters of different order with the stopband constrained to 60 dB. Use excess order (80) in the second filter to improve the passband ripple.

```
b1 = firband(60,[0 .2 .25 1],[1 1 0 0],...
[1 .001],{'w','c'});
b2 = firband(80,[0 .2 .25 1],[1 1 0 0],...
[1 .001],{'w','c'});
hfvt = fvtool(b1,1,b2,1);
legend(hfvt,'Filter Order 60','Filter Order 80');
```



## Input Arguments

### **n** — Filter order

integer greater than or equal to 3

Filter order, specified as an integer greater than or equal to 3.

### **f** — Normalized frequency points

real-valued vector

Normalized frequency points, specified as a real-valued vector. The argument must be in the range  $[0, 1]$ , where 1 corresponds to the Nyquist frequency. The number of elements in the vector is always a multiple of 2. The frequencies must be in increasing order.

### **a** — Desired amplitude

vector

Desired amplitudes at the points specified in **f**, specified as a vector. **f** and **a** must be of the same length. The length must be an even number.

### **w** — Weights

real-valued vector

Weights used to adjust the fit in each frequency band, specified as a real-valued vector. The length of **w** is half the length of **f** and **a**, so there is exactly one weight per band.

**c — Constrained error magnitudes (ripples)**

cell array of character vectors | array of strings

Constrained error magnitudes (ripples), specified as a cell array of character vectors or an array of strings. *c* must have the same length as *w*.

The entries of *c* must be either:

- 'c' or "c" -- To indicate that the corresponding element in *w* is a constraint (the ripple for that band cannot exceed that value).
- 'w' or "w" -- To indicate that the corresponding entry in *w* is a weight.

There must be at least one unconstrained band — *c* must contain at least one *w* entry. For instance, the “Design a Constrained Lowpass Filter” on page 5-648 example uses a weight of 1 in the passband, and constrains the stopband ripple not to exceed 0.2 (about 14 dB).

---

**Note** If your constrained filter does not touch the constraints, increase the error weighting you apply to the unconstrained bands.

---



---

**Note** When you work with constrained stopbands, enter the stopband constraint according to the standard conversion formula for power — the resulting filter attenuation or constraint equals  $20 \cdot \log(\text{constraint})$  where *constraint* is the value you enter in the function. For example, to set 20 dB of attenuation, use a value for the constraint equal to 0.1. This applies to constrained stopbands only.

---

Data Types: char | string

**s — Special properties at certain frequency points**

cell array of character vectors | array of strings

Special properties at certain frequency points, specified as a cell array of character vectors or an array of strings. *s* must have the same length as *f* and *a*. Entries of *s* must be one of:

- 'n' or "n" — Normal frequency point.
- 's' or "s" — Single-point band. The frequency “band” is given by a single point. Specify the corresponding gain at this frequency point in *a*.
- 'f' or "f" — Forced frequency point. Forces the gain at the specified frequency band to be the value specified.
- 'i' or "i" — Indeterminate frequency point. Use this argument when adjacent bands abut one another (no transition region).

Data Types: char | string

**lgrid — Density of frequency grid**

16 (default) | 1-by-1 cell array with integer value

Control the density of the frequency grid, which has roughly  $(\text{lgrid} \cdot n) / (2 \cdot \text{bw})$  frequency points, where *bw* is the fraction of the total frequency band interval [0,1] covered by *f*. Increasing *lgrid* often results in filters that more exactly match an equiripple filter, but that take longer to compute. The default value of 16 is the minimum value that should be specified for *lgrid*.

## Output Arguments

### **b** — Filter coefficients

row vector

Filter coefficients, returned as a row vector of length  $n + 1$ .

The `firband` uses the minimax filter design algorithm to design the following types of real FIR filters:

- Types 1-4 linear phase
  - Type 1 is even order, symmetric
  - Type 2 is odd order, symmetric
  - Type 3 is even order, antisymmetric
  - Type 4 is odd order, antisymmetric
- Minimum phase
- Maximum phase
- Minimum order (even or odd)
- Extra ripple
- Maximal ripple
- Constrained ripple
- Single-point band (notching and peaking)
- Forced gain
- Arbitrary shape frequency response curve filters

### **err** — Unweighted approximation error magnitude

scalar | vector

Unweighted approximation error magnitudes, returned as a scalar or a vector. `err` contains one element for each independent approximation error returned by the function.

### **res** — Frequency response characteristics

structure

Frequency response characteristics, returned as a structure. The structure `res` has the following fields:

| Structure Field        | Contents   |
|------------------------|--|
| <code>res.fgrid</code> | Vector containing the frequency grid used in the filter design optimization    |
| <code>res.des</code>   | Desired response on <code>fgrid</code>   |
| <code>res.wt</code>    | Weights on <code>fgrid</code>  |
| <code>res.h</code>     | Actual frequency response on the frequency grid                                |
| <code>res.error</code> | Error at each point (desired response - actual response) on the frequency grid |
| <code>res.iextr</code> | Vector of indices into <code>fgrid</code> of external frequencies              |

| Structure Field | Contents  |
|-----------------|---|
| res.fextr       | Vector of extremely frequencies   |
| res.order       | Filter order  |
| res.edgecheck   | Transition-region anomaly check. One element per band edge. Element values have the following meanings: 1 = OK, 0 = probable transition-region anomaly, -1 = edge not checked. Computed when you specify the 'check' input option in the function syntax. |
| res.iterations  | Number of Remez iterations for the optimization   |
| res.evals       | Number of function evaluations for the optimization   |

## Version History

Introduced in R2011a

### R2022b: Constrained error magnitudes and special properties support array of strings

Starting in R2022b, you can specify the constrained error magnitudes (c) and the special properties (s) as an array of strings.

```
b1 = fircband(60,[0 .2 .25 1],[1 1 0 0],...
[1 .001],["w","c"]);
```

### See Also

firceqrip | firgr | firfs | firpm

## fireqint

Equiripple FIR interpolators

### Syntax

```
b = fireqint(n,l,alpha)
b = fireqint(n,l,alpha,w)
b = fireqint('minorder', l,alpha,r)
b = fireqint({'minorder',initord},l,alpha,r)
```

### Description

`b = fireqint(n,l,alpha)` designs an FIR equiripple filter useful for interpolating input signals.

`b = fireqint(n,l,alpha,w)` allows you to specify a vector of weights in `w`.

`b = fireqint('minorder', l,alpha,r)` allows you to design a minimum-order filter that meets the design specifications. When you use the input argument `'minorder'`, you must provide the maximum ripple vector `r`.

`b = fireqint({'minorder',initord},l,alpha,r)` allows you to provide an initial estimate of the filter order in the input argument `initord`. Again, you must provide `r`, the vector of maximum deviation or ripples from the ideal filter magnitude response.

### Examples

#### Design an FIR Equiripple Filter

Design a minimum order interpolation filter with interpolation factor set to 6, and inverse measure of transition bandwidth set to 0.8. A vector of ripples must be supplied with the input argument, `minorder`.

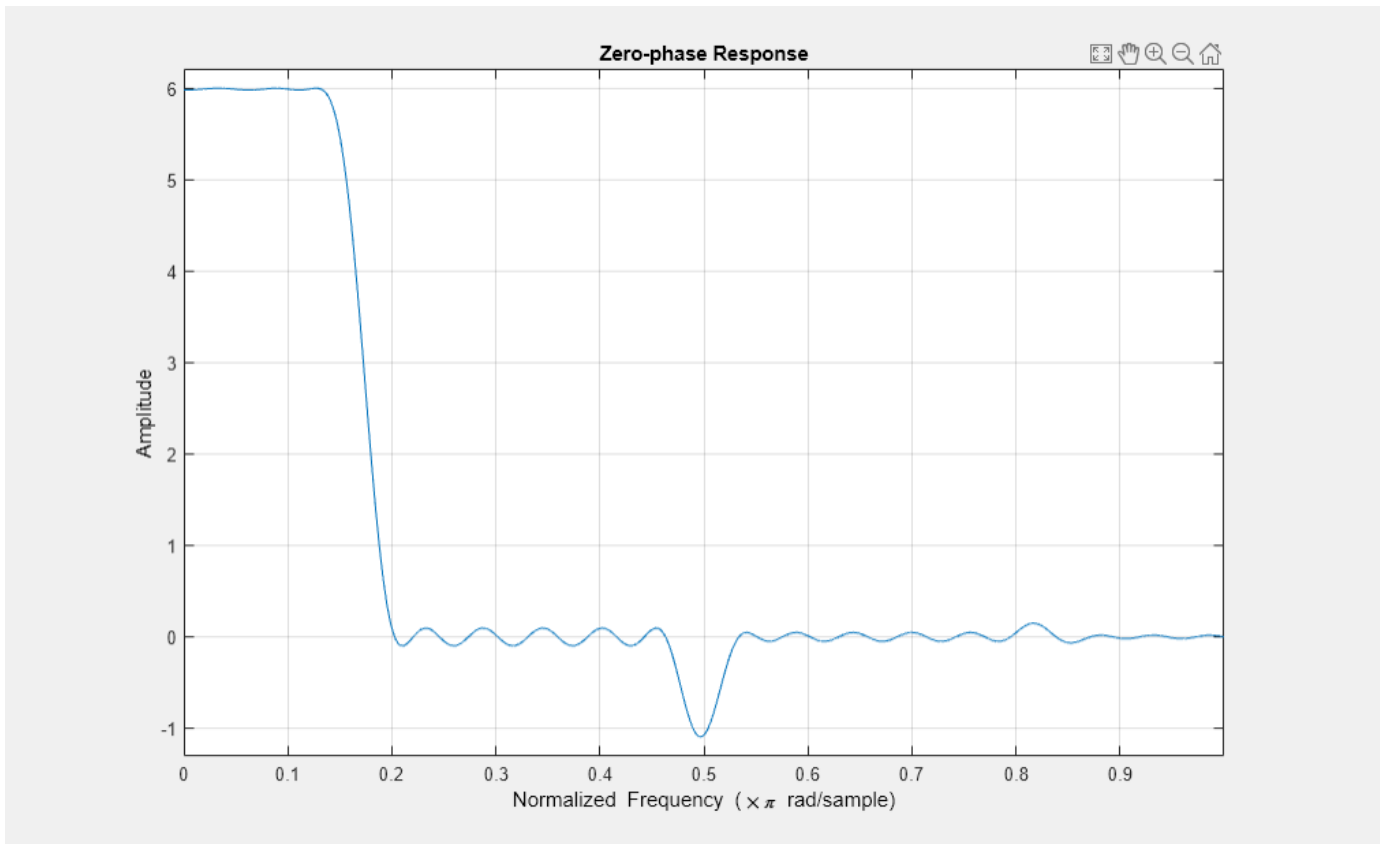
```
b = fireqint('minorder',6,.8,[0.01 .1 .05 .02]);
```

Create a polyphase interpolation filter.

```
hm = dsp.FIRInterpolator(6,'Numerator',b);
```

Plot the zero-phase response of the interpolator.

```
zerophase(hm)
```



## Input Arguments

### **n** — Order of equiripple filter

positive integer

Order of the equiripple filter, specified as a positive integer. Filter order must be three or more.

Data Types: `double`

### **l** — Interpolation factor

positive integer

Interpolation factor, specified as a positive integer.

Data Types: `double`

### **alpha** — Inverse measure of transition bandwidth

positive scalar in the range  $(0, 1)$

Inverse measure of the transition bandwidth, specified as a positive scalar in the range  $(0, 1)$ .

The `alpha` argument is inversely proportional to the transition bandwidth of the filter. It also affects the bandwidth of the don't-care regions in the stopband. Specifying `alpha` allows you to control how much of the Nyquist interval your input signal occupies. This can be beneficial for signals to be interpolated because it allows you to increase the transition bandwidth without affecting the

interpolation, resulting in better stopband attenuation for a given  $l$ . If you set `alpha` argument to 1, then `fireqint` function assumes that your signal occupies the entire Nyquist interval. Setting the `alpha` argument to a value less than one allows for don't-care regions in the stopband. For example, if your input occupies half the Nyquist interval, you could set the `alpha` to 0.5.

The signal to be interpolated is assumed to have zero (or negligible) power in the frequency band between  $(\alpha*\pi)$  and  $\pi$ . The input argument, `alpha` must therefore be a positive scalar between 0 and 1. The `fireqint` function treats such bands as don't-care regions for assessing filter design.

Data Types: `double`

### **w — Weights**

row vector of positive scalars

Weights applied to passband ripple and stopband attenuation, specified as a row vector of positive scalars. The number of weights required in `w` is given by  $1 + \text{floor}(l/2)$ . Using weights enables you to specify different attenuations in different parts of the stopband, as well as providing the ability to adjust the compromise between passband ripple and stopband attenuation.

Data Types: `double`

### **r — Maximum deviation from ideal filter magnitude response**

vector

Maximum deviations or ripples from the ideal filter magnitude response, specified as a vector. The number of elements required in `r` is given by  $1 + \text{floor}(l/2)$ .

Data Types: `double`

### **initord — Initial filter order**

positive integer

Initial filter order when designing a minimum order filter that meets design specifications, specified as a positive integer.

Data Types: `double`

## **Output Arguments**

### **b — Filter coefficients**

real-valued row vector

Filter coefficients of the equiripple filter, returned as a real-valued row vector.

Data Types: `double`

## **Version History**

Introduced in R2011a

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.



Usage notes and limitations:

All inputs must be constant. Expressions or variables are allowed if their values do not change.

**See Also**

`firgr` | `firhalfband` | `firls` | `firnyquist` | `intfilt`

## firceqrip

Constrained equiripple FIR filter

### Syntax

```
B = firceqrip(n,Fo,DEV)
B = firceqrip(...,'slope',r)
B = firceqrip('minorder',[Fp Fst],DEV)
B = firceqrip(...,'passedge')
B = firceqrip(...,'stopedge')
B = firceqrip(...,'high')
B = firceqrip(...,'min')
B = firceqrip(...,'invsinc',C)
B = firceqrip(...,'invdiric',C)
```

### Description

`B = firceqrip(n,Fo,DEV)` designs an order  $n$  filter (filter length equal  $n + 1$ ) lowpass FIR filter with linear phase.

`firceqrip` produces the same equiripple lowpass filters that `firpm` produces using the Parks-McClellan algorithm. The difference is how you specify the filter characteristics for the function.

The input argument `Fo` specifies the frequency at the upper edge of the passband in normalized frequency ( $0 < Fo < 1$ ). The two-element vector `dev` specifies the peak or maximum error allowed in the passband and stopbands. Enter `[d1 d2]` for `dev` where `d1` sets the passband error and `d2` sets the stopband error.

`B = firceqrip(...,'slope',r)` uses the input keyword 'slope' and input argument `r` to design a filter with a nonequiripple stopband. `r` is specified as a positive constant and determines the slope of the stopband attenuation in dB/normalized frequency. Greater values of `r` result in increased stopband attenuation in dB/normalized frequency.

`B = firceqrip('minorder',[Fp Fst],DEV)` designs filter with the minimum number of coefficients required to meet the deviations in  $DEV = [d1 d2]$  while having a transition width no greater than  $Fst - Fp$ , the difference between the stopband and passband edge frequencies. You can specify 'mineven' or 'minodd' instead of 'minorder' to design minimum even order (odd length) or minimum odd order (even length) filters, respectively. The 'minorder' option does not apply when you specify the 'min' (minimum-phase), 'invsinc', or the 'invdiric' options.

`B = firceqrip(...,'passedge')` designs a filter where `Fo` specifies the frequency at which the passband starts to rolloff.

`B = firceqrip(...,'stopedge')` designs a filter where `Fo` specifies the frequency at which the stopband begins.

`B = firceqrip(...,'high')` designs a high pass FIR filter instead of a lowpass filter.

`B = firceqrip(...,'min')` designs a minimum-phase filter.

$B = \text{firceqrip}(\dots, 'invsinc', C)$  designs a lowpass filter whose magnitude response has the shape of an inverse sinc function. This may be used to compensate for sinc-like responses in the frequency domain such as the effect of the zero-order hold in a D/A converter. The amount of compensation in the passband is controlled by  $C$ , which is specified as a scalar or two-element vector. The elements of  $C$  are specified as follows:

- If  $C$  is supplied as a real-valued scalar or the first element of a two-element vector, `firceqrip` constructs a filter with a magnitude response of  $1/\text{sinc}(C*\pi*F)$  where  $F$  is the normalized frequency.
- If  $C$  is supplied as a two-element vector, the inverse-sinc shaped magnitude response is raised to the positive power  $C(2)$ . If we set  $P=C(2)$ , `firceqrip` constructs a filter with a magnitude response  $1/\text{sinc}(C*\pi*F)^P$ .

If this FIR filter is used with a cascaded integrator-comb (CIC) filter, setting  $C(2)$  equal to the number of stages compensates for the multiplicative effect of the successive sinc-like responses of the CIC filters.

---

**Note** Since the value of the inverse sinc function becomes unbounded at  $C=1/F$ , the value of  $C$  should be greater than the reciprocal of the passband edge frequency. This can be expressed as  $F_0 < 1/C$ . For users familiar with CIC decimators,  $C$  is equal to  $1/2$  the product of the differential delay and decimation factor.

---

$B = \text{firceqrip}(\dots, 'invdiric', C)$  designs a lowpass filter with a passband that has the shape of an inverse Dirichlet sinc function. The frequency response of the inverse Dirichlet sinc function is given by  $\left\{ rC \left( \frac{\sin(f/2r)}{\sin(Cf/2)} \right)^p \right\}$  where  $C$ ,  $r$ , and  $p$  are scalars. The input  $C$  can be a scalar or vector containing 2 or 3 elements. If  $C$  is a scalar,  $p$  and  $r$  equal 1. If  $C$  is a two-element vector, the first element is  $C$  and the second element is  $p$ ,  $[C \ p]$ . If  $C$  is a three-element vector, the third element is  $r$ ,  $[C \ p \ r]$ .

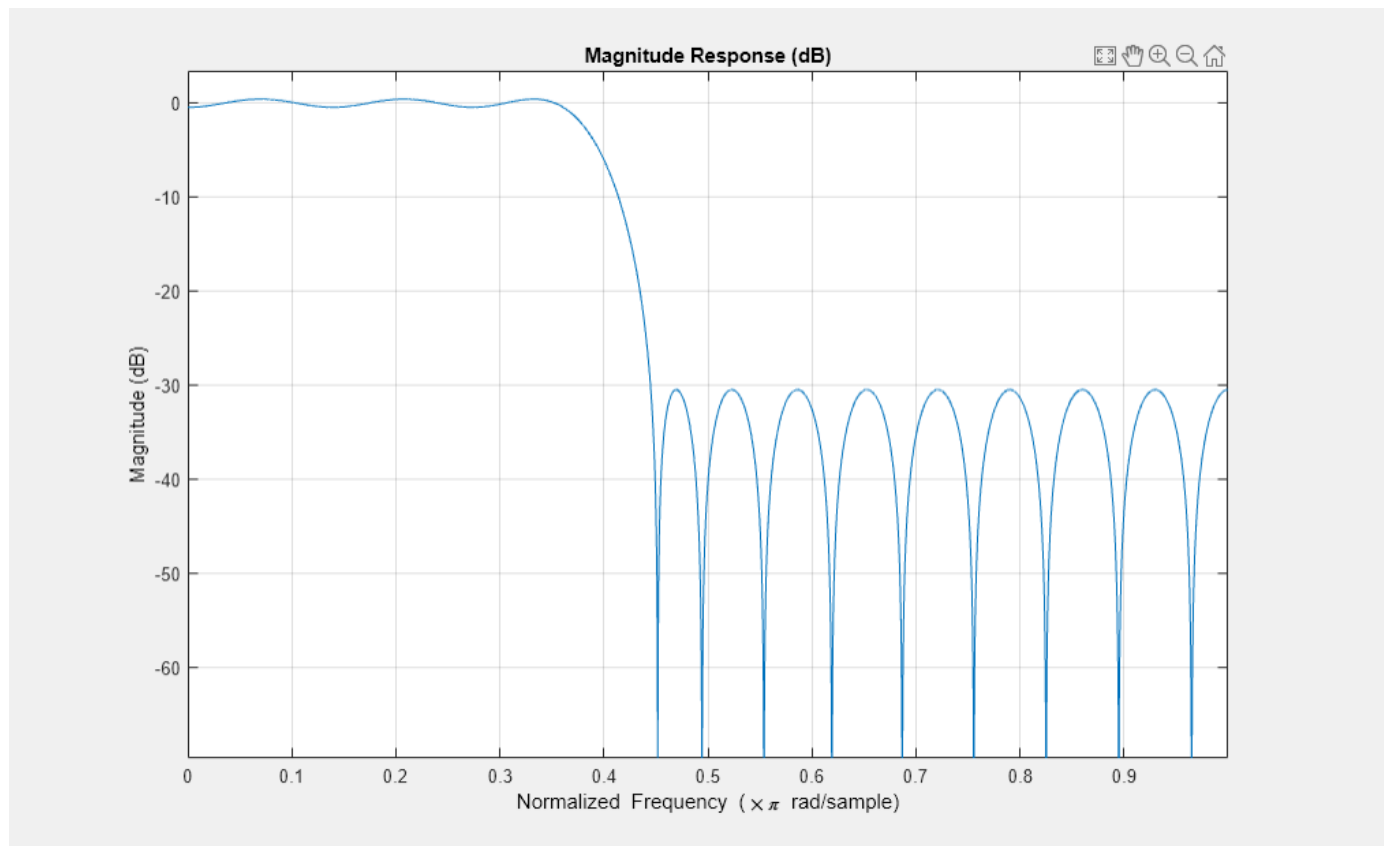
## Examples

To introduce a few of the variations on FIR filters that you design with `firceqrip`, these five examples cover both the default syntax  $b = \text{firceqrip}(n, wo, del)$  and some of the optional input arguments. For each example, the input arguments  $n$ ,  $wo$ , and  $del$  remain the same.

### Filter design using `firceqrip`

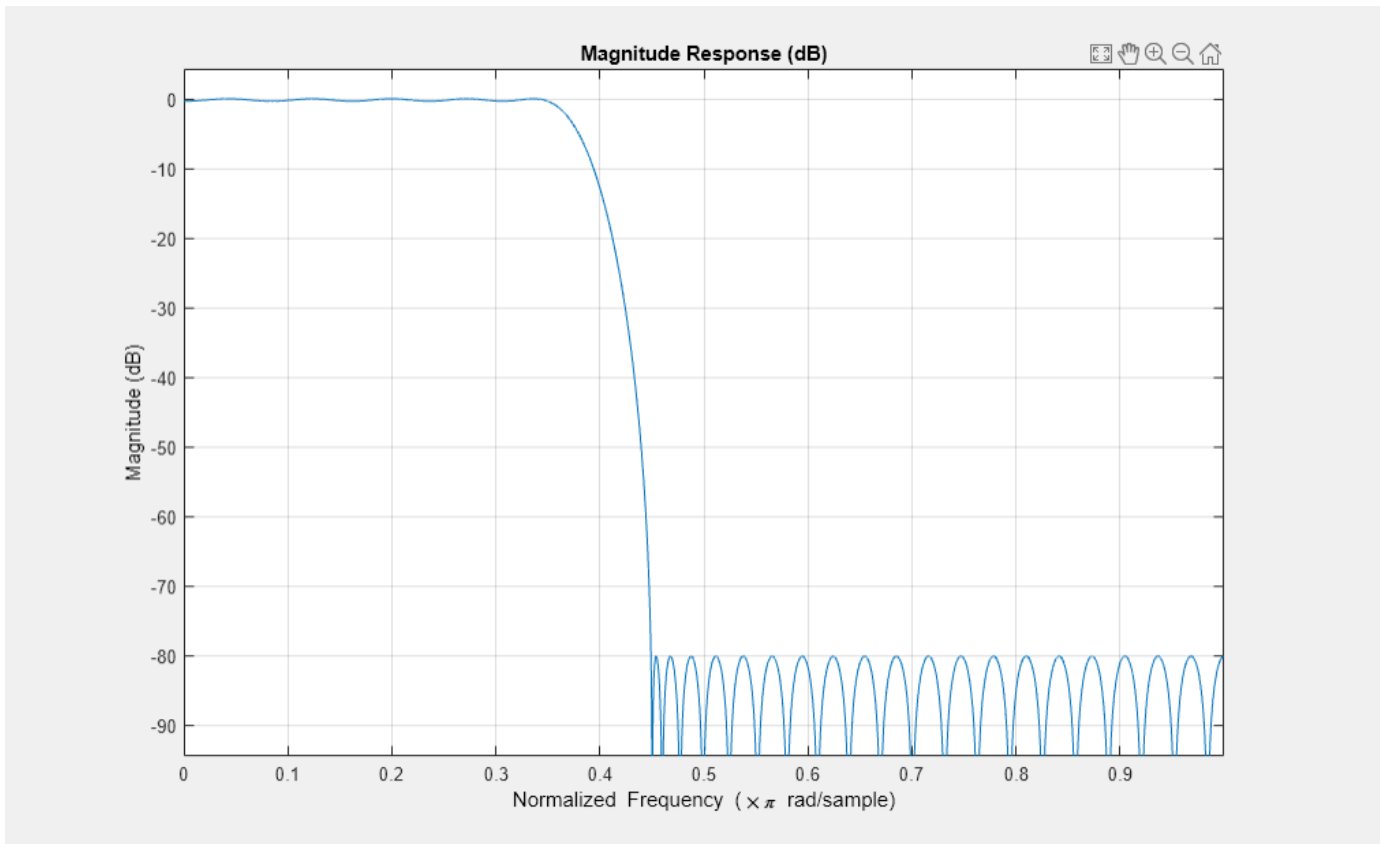
Design a 30th order FIR filter using `firceqrip`.

```
b = firceqrip(30,0.4,[0.05 0.03]); fvtool(b)
```



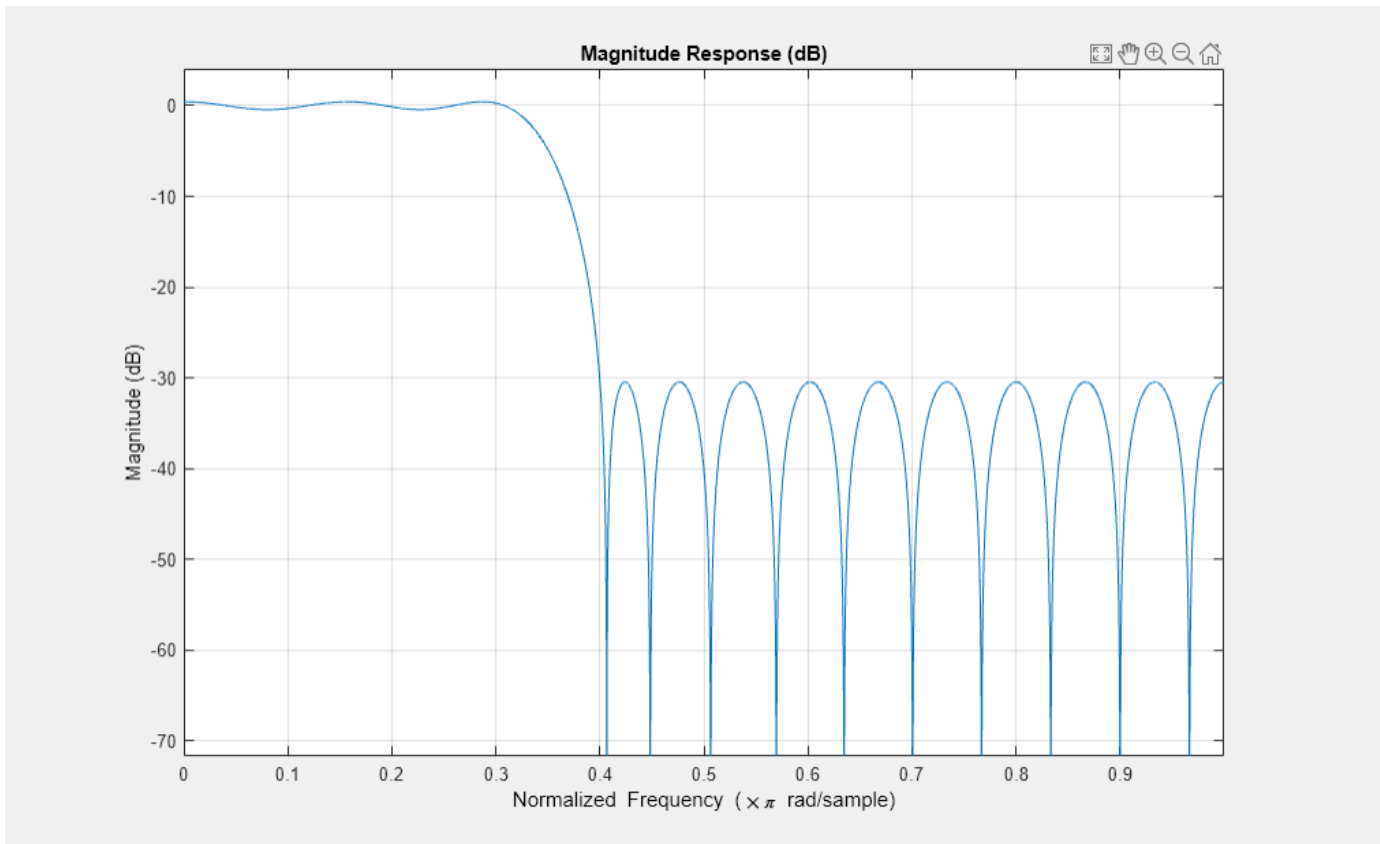
Design a minimum order FIR filter using `firceqrip`. The passband edge and stopband edge frequencies are  $0.35\pi$  and  $0.45\pi$  rad/sample. The allowed deviations are 0.02 and  $1e-4$ .

```
b = firceqrip('minorder',[0.35 0.45],[0.02 1e-4]); fvtool(b)
```



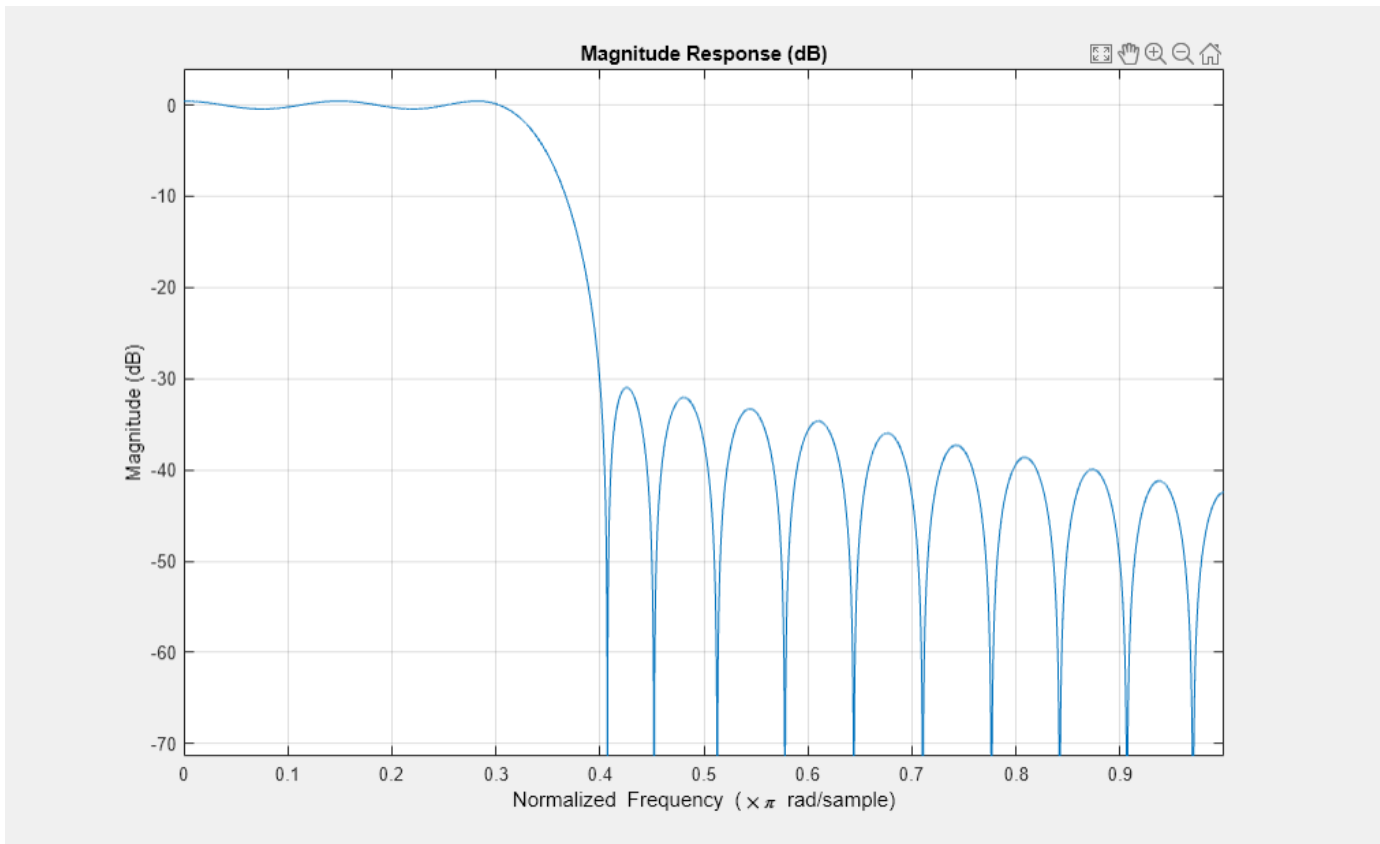
Design a 30th order FIR filter with the stopedge keyword to define the response at the edge of the filter stopband.

```
b = firceqrip(30,0.4,[0.05 0.03], 'stopedge'); fvtool(b)
```



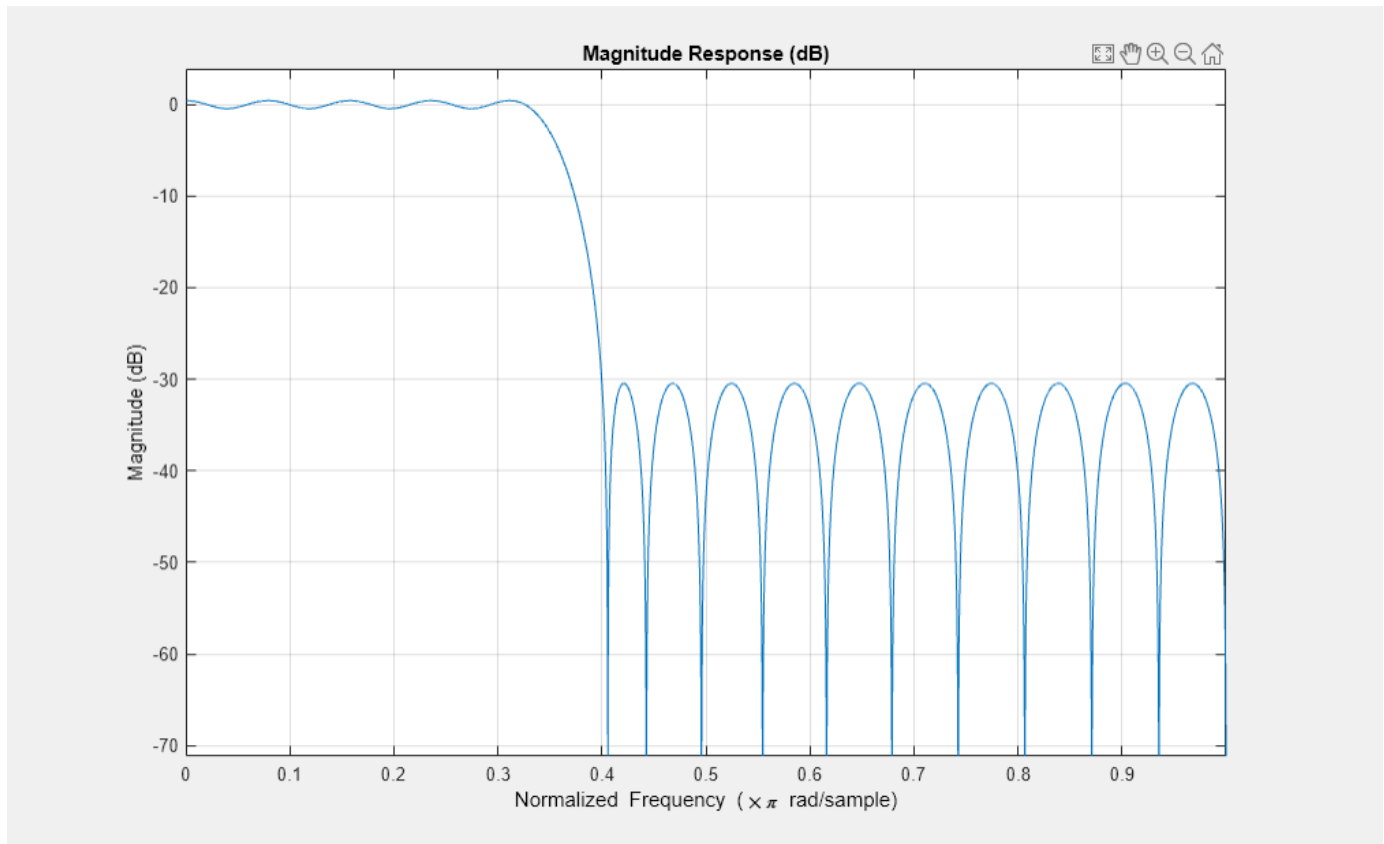
Design a 30th order FIR filter with the `slope` keyword and `r = 20`.

```
b = firceqrip(30,0.4,[0.05 0.03], 'slope',20, 'stopedge'); fvtool(b)
```



Design a 30th order FIR filter defining the stopband and specifying that the resulting filter is minimum phase with the `min` keyword.

```
b = firceqrip(30,0.4,[0.05 0.03], 'stopedge', 'min'); fvtool(b)
```

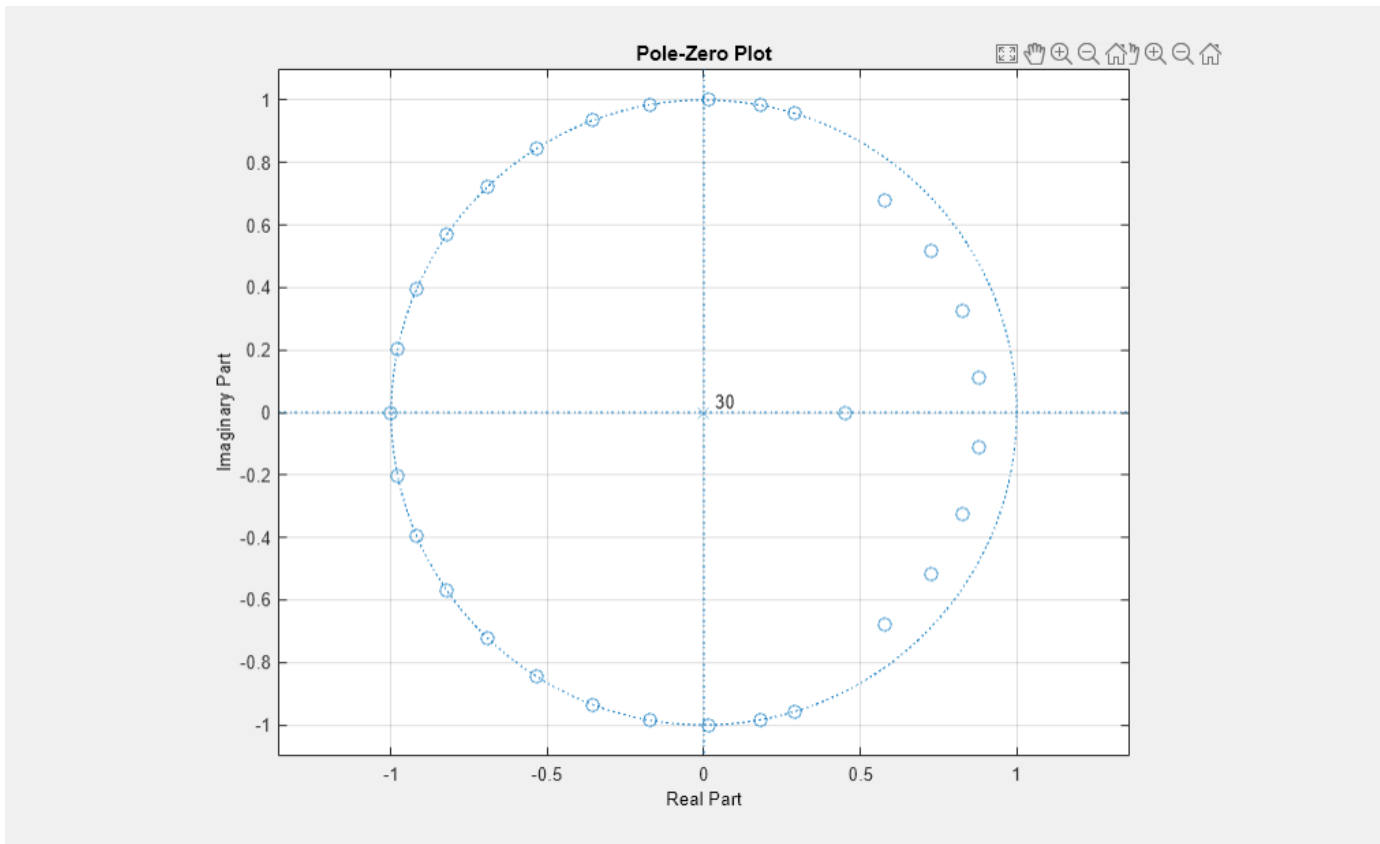


Comparing this filter to the filter in Figure 1. The cutoff frequency  $\omega_0 = 0.4$  now applies to the edge of the stopband rather than the point at which the frequency response magnitude is 0.5.

Viewing the zero-pole plot shown here reveals this is a minimum phase FIR filter - the zeros lie on or inside the unit circle,  $z = 1$

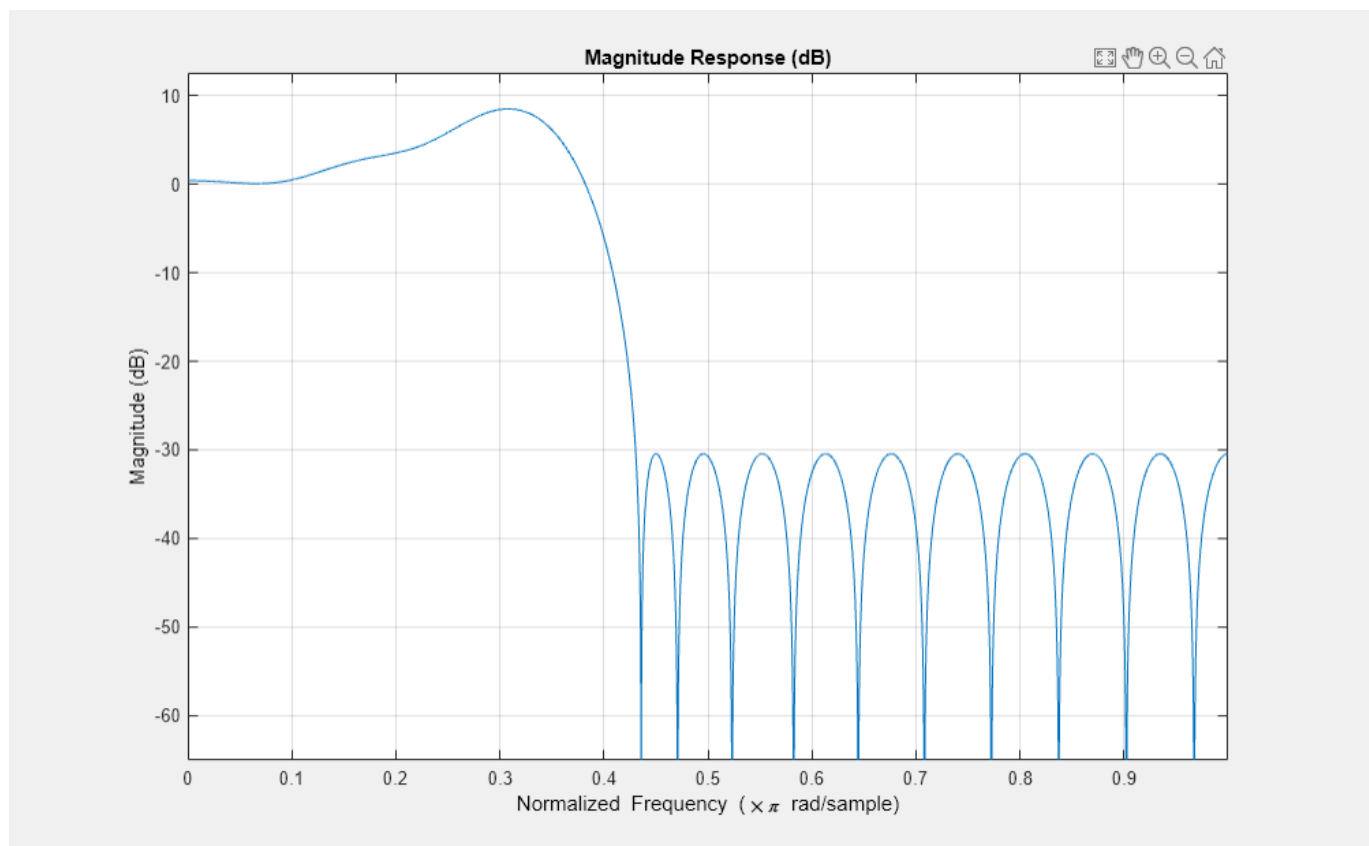
```
fvtool(b, 'polezero')
```





Design a 30th order FIR filter with the `invsinc` keyword to shape the filter passband with an inverse sinc function.

```
b = firceqip(30,0.4,[0.05 0.03], 'invsinc', [2 1.5]); fvtool(b)
```



The inverse sinc function being applied is defined as  $1/\text{sinc}(2*w)^{1.5}$ .

### Inverse-Dirichlet-Sinc-Shaped Passband

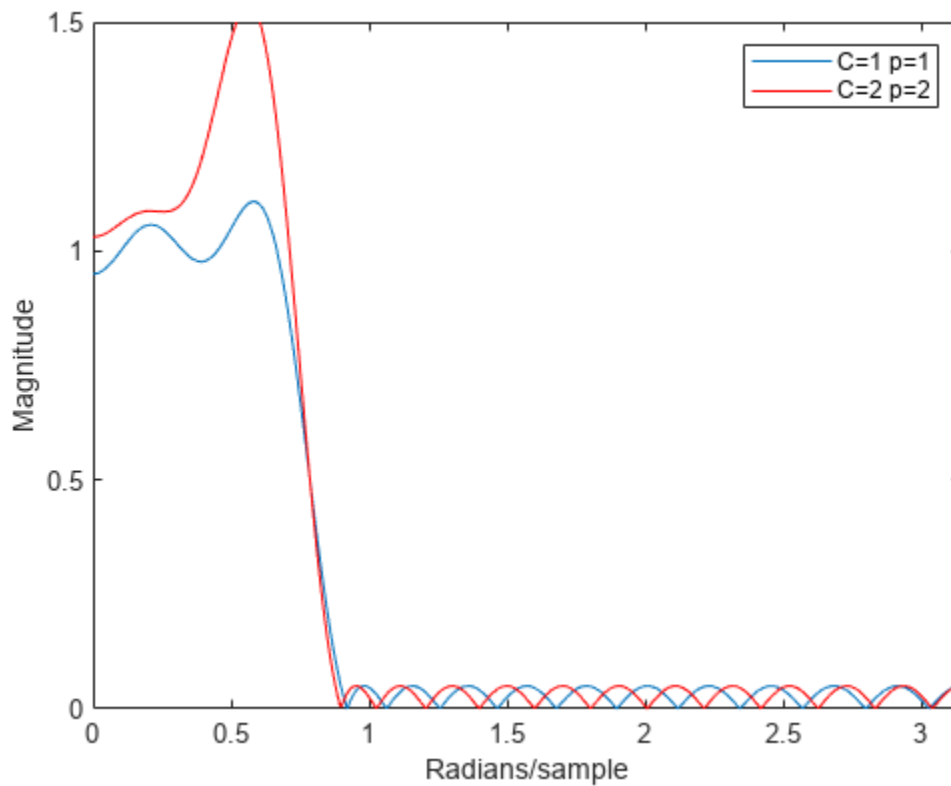
Design two order 30 constrained equiripple FIR filters with inverse-Dirichlet-sinc-shaped passbands. The cutoff frequency in both designs is  $\pi/4$  radians/sample. Set  $C=1$  in one design  $C=2$  in the second design. The maximum passband and stopband ripple is 0.05. Set  $p=1$  in one design and  $p=2$  in the second design.

Design the filters.

```
b1 = firceqrip(30,0.25,[0.05 0.05],'invdiric',[1 1]);
b2 = firceqrip(30,0.25,[0.05 0.05],'invdiric',[2 2]);
```

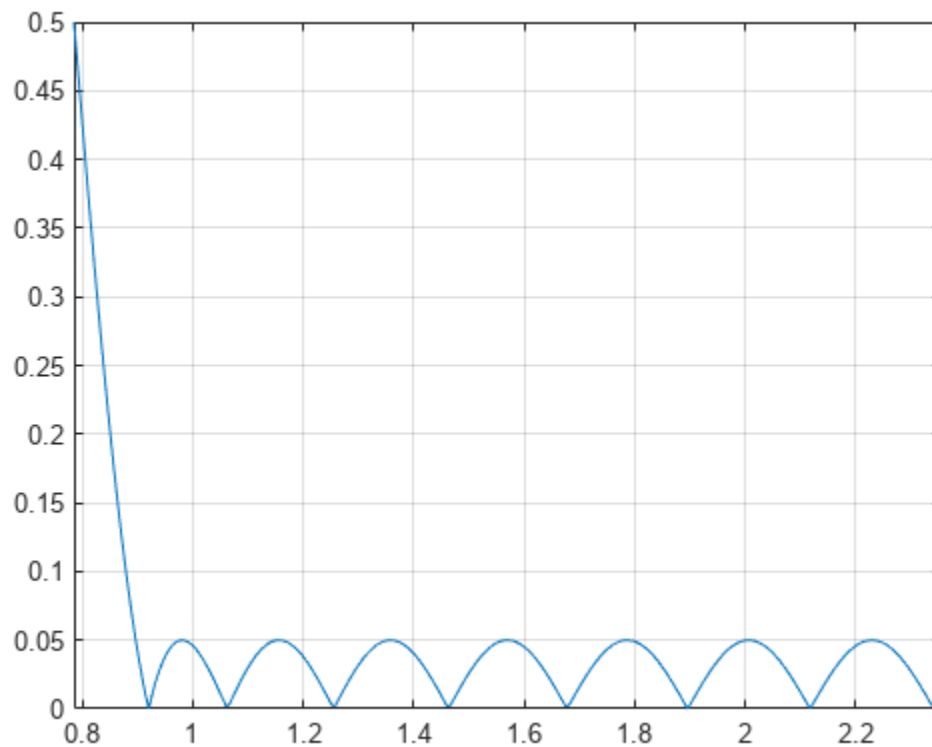
Obtain the filter frequency responses using `freqz`. Plot the magnitude responses.

```
[h1,~] = freqz(b1,1);
[h2,w] = freqz(b2,1);
plot(w,abs(h1)); hold on;
plot(w,abs(h2),'r');
axis([0 pi 0 1.5]);
xlabel('Radians/sample');
ylabel('Magnitude');
legend('C=1 p=1','C=2 p=2');
```



Inspect the stopband ripple in the design with  $C=1$  and  $p=1$ . The constrained design sets the maximum ripple to be 0.05. Zoom in on the stopband from the cutoff frequency of  $\pi/4$  radians/sample to  $3\pi/4$  radians/sample.

```
figure;  
plot(w,abs(h1));  
set(gca,'xlim',[pi/4 3*pi/4]);  
grid on;
```



## Version History

Introduced in R2011a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

All inputs must be constant. Expressions or variables are allowed if their values do not change.

## See Also

`diric` | `fdesign.decimator` | `firhalfband` | `firnyquist` | `firgr` | `ifir` | `iirgrpdelay` | `iirlpnorm` | `iirlpnormc` | `fircls` | `firls` | `firpm` | `sinc`

# fircls

FIR Constrained Least Squares filter

## Syntax

```
clsFilter = design(d,'fircls','SystemObject',true)
clsFilter = design(d,'fircls','FilterStructure',value,'SystemObject',true)
clsFilter = design(d,'fircls','PassbandOffset',value,'SystemObject',true)
clsFilter = design(d,'fircls','zerophase',value,'SystemObject',true)
```

## Description

`clsFilter = design(d,'fircls','SystemObject',true)` designs a FIR Constrained Least Squares (CLS) filter, `clsFilter`, from a filter specifications object, `d`.

`clsFilter = design(d,'fircls','FilterStructure',value,'SystemObject',true)` where `value` is one of the following filter structures:

- 'dffir', a discrete-time, direct-form FIR filter (the default value)
- 'dffirt', a discrete-time, direct-form FIR transposed filter
- 'dfsymfir', a discrete-time, direct-form symmetric FIR filter

`clsFilter = design(d,'fircls','PassbandOffset',value,'SystemObject',true)` where `value` sets the passband band gain in dB. The `PassbandOffset` and `Ap` values affect the upper and lower approximation bound in the passband as follows:

- Lower bound =  $(\text{PassbandOffset} - A_p/2)$
- Upper bound =  $(\text{PassbandOffset} + A_p/2)$

For bandstop filters, the `PassbandOffset` is a vector of length two that specifies the first and second passband gains. The `PassbandOffset` value defaults to 0 for lowpass, highpass and bandpass filters. The `PassbandOffset` value defaults to [0 0] for bandstop filters.

`clsFilter = design(d,'fircls','zerophase',value,'SystemObject',true)` where `value` is either 'true' ('1') or 'false' ('0'). If `zerophase` is true, the lower approximation bound in the stopband is forced to zero (i.e., the filter has a zero phase response). `Zerophase` is false (0) by default.

To determine the available design options, use `designopts` with the specification object and the design method as input arguments as shown.

```
designopts(d,'fircls')
```

For complete help about using `fircls`, refer to the command line help system. For example, to get specific information about using `fircls` with `d`, the specification object, enter the following at the MATLAB prompt.

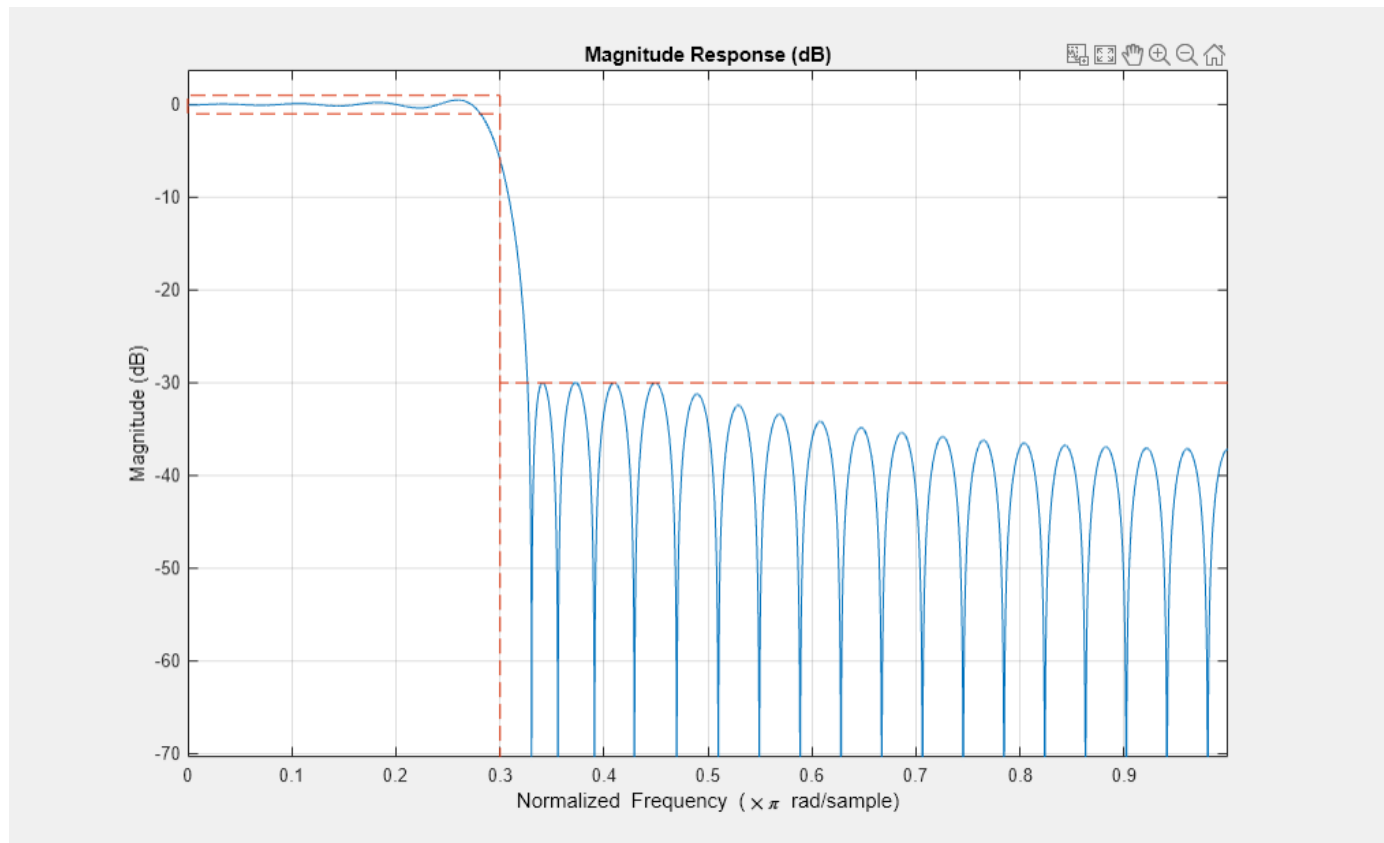
```
help(d,'fircls')
```

## Examples

### Create an FIR Constrained Least Squares (CLS) Filter

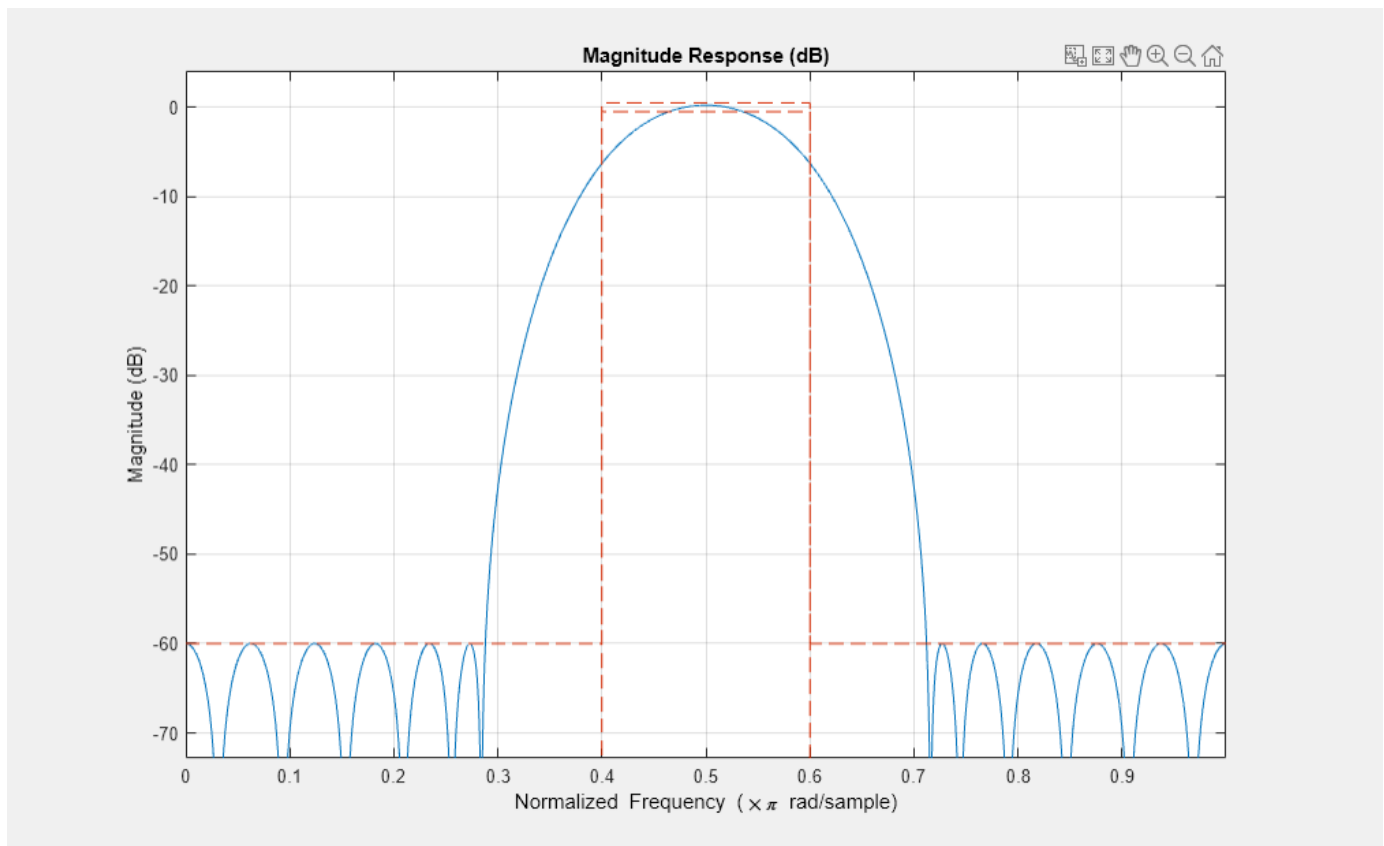
The following example designs a constrained least-squares FIR lowpass filter.

```
h = fdesign.lowpass('n,fc,ap,ast', 50, 0.3, 2, 30);
firLPFilter = design(h, 'fircls','SystemObject',true);
fvtool(firLPFilter)
```



The following example constructs a constrained least-squares FIR bandpass filter.

```
d = fdesign.bandpass('N,Fc1,Fc2,Ast1,Ap,Ast2',...
30,0.4,0.6,60,1,60);
firBPFfilter = design(d,'fircls','SystemObject',true);
fvtool(firBPFfilter)
```



## Version History

Introduced in R2011a

### See Also

[cheby1](#) | [cheby2](#) | [ellip](#)

## firgr

Parks-McClellan FIR filter

### Syntax

```

b = firgr(n,f,a,w)
b = firgr(n,f,a,ftype)
b = firgr(m,f,a,r),
b = firgr({m,ni},f,a,r)
b = firgr(n,f,a,w,e)
b = firgr(n,f,a,s)
b = firgr(n,f,a,s,w,e)
b = firgr(____,'l')
b = firgr(____,'minphase')
b = firgr(____,'check')
b = firgr(____,{lgrid}),
[b,err] = firgr(____)
[b,err,res] = firgr(____)
b = firgr(n,f,fresp,w)
b = firgr(n,f,{fresp,p1,p2,...},w)
b = firgr(n,f,a,w)
b = firgr(____,'h')
```

### Description

`b = firgr(n,f,a,w)` uses a minimax filter design algorithm to design a real linear phase FIR filter `b` of length `n+1`. The filter has the best approximation to the desired frequency response described by `f` and `a` in the minimax sense. `w` is a vector of weights, one per band. When you omit `w`, all bands are weighted equally. For more information on the input arguments, see `firpm`.

`b = firgr(n,f,a,ftype)` design a filter of type specified by `ftype`

`b = firgr(m,f,a,r)`, designs filters repeatedly until the minimum order filter as specified in `m` and that meets the specifications `f,a` is found. `m` can be one of 'minorder', 'mineven' or 'minodd'. `r` is a vector containing the peak ripple per frequency band. You must specify `r`. When you specify 'mineven' or 'minodd', the minimum even or odd order filter is found.

`b = firgr({m,ni},f,a,r)` where `m` is one of 'minorder', 'mineven' or 'minodd', uses `ni` as the initial estimate of the filter order. `ni` is optional for common filter designs, but it must be specified for designs in which `firpmord` cannot be used, such as while designing differentiators or Hilbert transformers.

`b = firgr(n,f,a,w,e)` specifies independent approximation errors for different bands. Use this syntax to design extra ripple or maximal ripple filters. These filters have interesting properties such as having the minimum transition width. `e` is a cell array of character vectors specifying the approximation errors to use. Its length must equal the number of bands.

`b = firgr(n,f,a,s)` is used to design filters with special properties `s` at certain frequency points.

`b = firgr(n,f,a,s,w,e)` specifies weights and independent approximation errors for filters with special properties. The weights and properties are included in vectors `w` and `e`. Sometimes, you may



need to use independent approximation errors to get designs with forced values to converge. For example,

```
b = firgr(82,[0 0.055 0.06 0.1 0.15 1], [0 0 0 0 1 1],...
{'n' 'i' 'f' 'n' 'n' 'n'}, [10 1 1] ,{'e1' 'e2' 'e3'});
```

`b = firgr( ____, '1' )` designs a type 1 filter (even-order symmetric). You can specify type 2 (odd-order symmetric), type 3 (even-order antisymmetric), and type 4 (odd-order antisymmetric) filters as well. Note that restrictions apply to `a` at `f = 0` or `f = 1` for FIR filter types 2, 3, and 4.

`b = firgr( ____, 'minphase' )` designs a minimum-phase FIR filter. You can use the argument `'maxphase'` to design a maximum phase FIR filter.

`b = firgr( ____, 'check' )` returns a warning when there are potential transition-region anomalies.

`b = firgr( ____, {lgrid} )`, where `{lgrid}` is a scalar cell array. The value of the scalar controls the density of the frequency grid by setting the number of samples used along the frequency axis.

`[b,err] = firgr( ____ )` returns the unweighted approximation error magnitudes. `err` contains one element for each independent approximation error returned by the function.

`[b,err,res] = firgr( ____ )` returns the structure `res` comprising optional results computed by `firgr`.

`b = firgr(n,f,fresp,w)` returns a length `n + 1` FIR filter which has the best approximation to the desired frequency response as returned by the user-defined function `fresp`.

`firgr` includes a predefined frequency response function named `'firpmfrf2'`. You can write your own based on the simpler `'firpmfrf'`. See the help for `private/firpmfrf` for more information.

`b = firgr(n,f,{fresp,p1,p2,...},w)` specifies optional arguments `p1`, `p2`,..., `pn` to be passed to the response function `fresp`.

`b = firgr(n,f,a,w)` is a synonym for `b = firgr(n,f,{'firpmfrf2',a},w)`, where `a` is a vector containing your specified response amplitudes at each band edge in `f`. By default, `firgr` designs symmetric (even) FIR filters. `'firpmfrf2'` is the predefined frequency response function. If you do not specify your own frequency response function (the `fresp` variable), `firgr` uses `'firpmfrf2'`.

`b = firgr( ____, 'h' )` and `b = firgr( ____, 'd' )` design antisymmetric (odd) filters. When you omit the `'h'` or `'d'` arguments from the `firgr` command syntax, each frequency response function `fresp` can tell `firgr` to design either an even or odd filter. Use the command syntax `sym = fresp('defaults',{n,f,[],w,p1,p2,...})`.

`firgr` expects `fresp` to return `sym = 'even'` or `sym = 'odd'`. If `fresp` does not support this call, `firgr` assumes even symmetry.

For more information about the input arguments to `firgr`, refer to `firpm`.

## Examples

### Design a Filter Using firgr

Design an FIR filter with two single-band notches at 0.25 and 0.55.

```
b1 = firgr(42,[0 0.2 0.25 0.3 0.5 0.55 0.6 1],[1 1 0 1 1 0 1 1],...
{'n' 'n' 's' 'n' 'n' 's' 'n' 'n'});
```

Design a highpass filter whose gain at 0.06 is forced to be zero. The gain at 0.055 is indeterminate since it should be about the band.

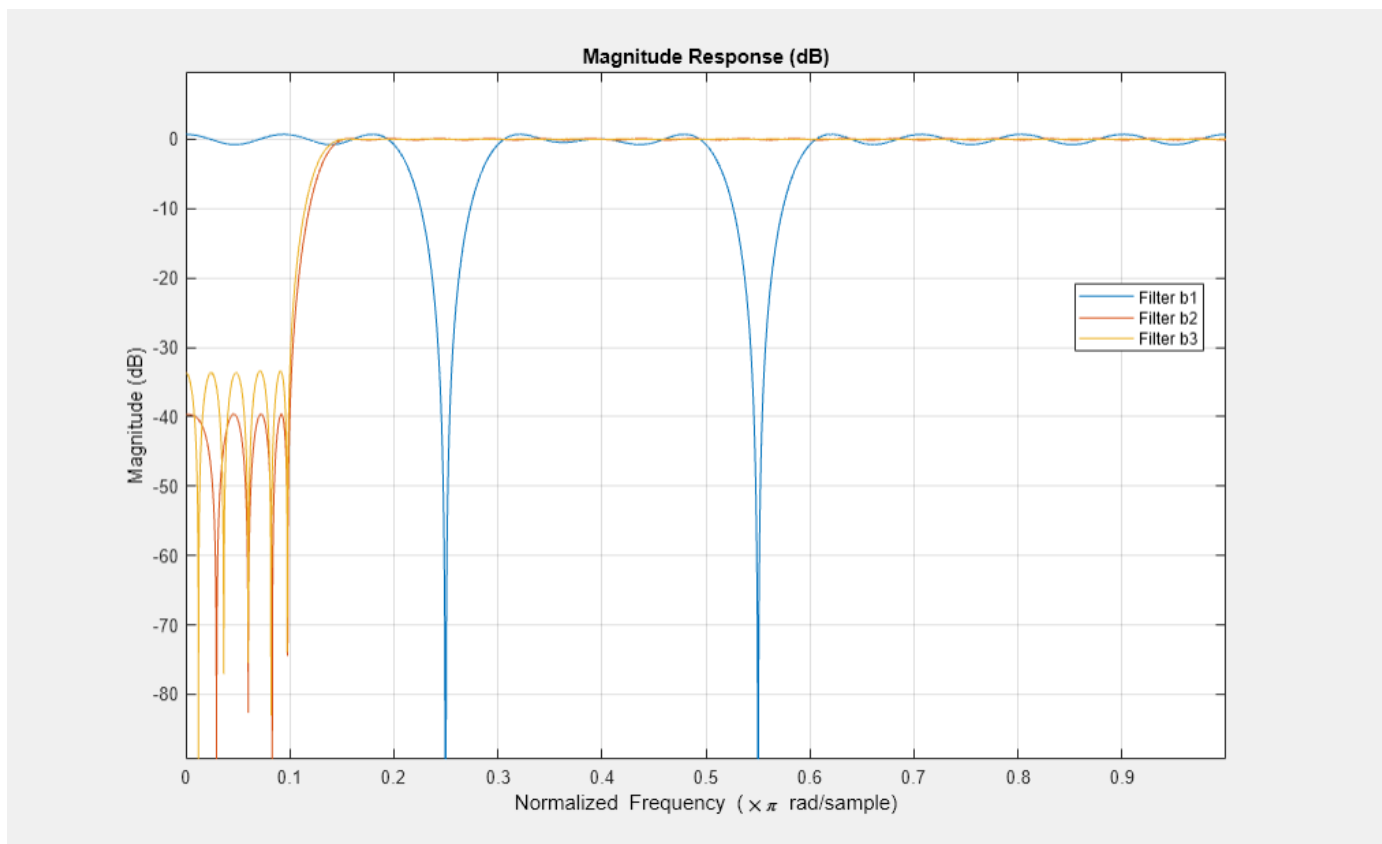
```
b2 = firgr(82,[0 0.055 0.06 0.1 0.15 1],[0 0 0 0 1 1],...
{'n' 'i' 'f' 'n' 'n' 'n'});
```

Design a second highpass filter with forced values and independent approximation errors.

```
b3 = firgr(82,[0 0.055 0.06 0.1 0.15 1], [0 0 0 0 1 1], ...
{'n' 'i' 'f' 'n' 'n' 'n'}, [10 1 1], {'e1' 'e2' 'e3'});
```

Use the filter visualization tool to view the results of the filters.

```
hvft = fvtool(b1,1,b2,1,b3,1);
legend(hvft,'Filter b1','Filter b2','Filter b3')
```



### Input Arguments

#### **n** — Filter order

integer greater than or equal to 3

Filter order, specified as an integer greater than or equal to 3.

### **f — Normalized frequency points**

real-valued vector

Normalized frequency points, specified as a real-valued vector. The argument must be in the range  $[0, 1]$ , where 1 corresponds to the Nyquist frequency. The length of **f** must be even. The frequencies must be in increasing order.

### **a — Desired amplitude**

vector

Desired amplitudes at the points specified in **f**, specified as a vector. **f** and **a** must be of the same length. The length of **a** must be even.

### **w — Weights**

real-valued vector

Weights used to adjust the fit in each frequency band, specified as a real-valued vector. The length of **w** is half the length of **f** and **a**, so there is exactly one weight per band.

### **f type — Filter type**

'hilbert' | 'differentiator'

Filter type for linear-phase filters with odd symmetry (type III and type IV), specified as either 'hilbert' or 'differentiator':

- 'hilbert' — The output coefficients in **b** obey the relation  $b(k) = -b(n + 2 - k)$ ,  $k = 1, \dots, n + 1$ . This class of filters includes the Hilbert transformer, which has a desired amplitude of 1 across the entire band.

For example,

```
b = firgr(30,[0.1 0.9],[1 1],'hilbert');
```

designs an approximate FIR Hilbert transformer of length 31.

- 'differentiator' — For nonzero amplitude bands, the filter weighs the error by a factor of  $1/f$  so that the error at low frequencies is much smaller than at high frequencies. For FIR differentiators, which have an amplitude characteristic proportional to frequency, these filters minimize the maximum relative error (the maximum of the ratio of the error to the desired amplitude).

### **m — Type of minimum order filter**

'minorder' | 'mineven' | 'minodd'

Type of minimum order filter, specified as 'minorder', 'mineven', or 'minodd'. When you specify 'mineven' or 'minodd', the function designs a minimum even order or minimum odd order filter.

### **r — Peak ripple per frequency band**

real-valued vector

Peak ripple per frequency band, specified as a real-valued vector. The length of **r** is half the length of **f** and **a**, so there is exactly one weight per band.

**ni — Initial estimate of filter order**

integer greater than or equal to 3

Initial estimate of the filter order, specified as an integer greater than or equal to 3. The function accepts this value in the `b = firgr({m,ni},f,a,r)` syntax. `ni` is optional for common filter designs, but it must be specified for designs in which `firpmord` cannot be used, such as while designing differentiators or Hilbert transformers.

**e — Approximation errors**

cell array of character vectors | array of strings

Independent approximation errors to use for different bands, specified as a cell array of character vectors or an array of strings. The length of `e` must equal the number of bands. Entries of `e` must be in the form `'e#'` or `"e#"` where `#` indicates which approximation error to use for the corresponding band. For example, when `e = ["e1", "e2", "e1"]`, the first and third bands use the same approximation error `"e1"` and the second band uses a different one `"e2"`. When all bands use the same approximation error, such as `["e1", "e1", "e1", ...]`, it is equivalent to omitting `e`, as in `b = firgr(n,f,a,w)`.

Data Types: char | string

**s — Special properties at certain frequency points**

cell array of character vectors | array of strings

Special properties at certain frequency points, specified as a cell array of character vectors or an array of strings. `s` must have the same length as `f` and `a`. Entries of `s` must be one of:

- `'n'` or `"n"` — Normal frequency point.
- `'s'` or `"s"` — Single-point band. The frequency “band” is given by a single point. Specify the corresponding gain at this frequency point in `a`.
- `'f'` or `"f"` — Forced frequency point. Forces the gain at the specified frequency band to be the value specified.
- `'i'` or `"i"` — Indeterminate frequency point. Use this argument when adjacent bands abut one another (no transition region).

For example, the following command designs a bandstop filter with zero-valued single-point stopbands (notches) at 0.25 and 0.55.

```
b = firgr(42,[0 0.2 0.25 0.3 0.5 0.55 0.6 1],...
[1 1 0 1 1 0 1 1],[ "n" "n" "s" "n" "n" "s" "n" "n"])
```

`b = firgr(82,[0 0.055 0.06 0.1 0.15 1],[0 0 0 0 1 1],[ "n" "i" "f" "n" "n" "n"])` designs a highpass filter with the gain at 0.06 forced to be zero. The band edge at 0.055 is indeterminate since the first two bands actually touch. The other band edges are normal.

Data Types: char | string

**lgrid — Density of frequency grid**

16 (default) | 1-by-1 cell array with integer value

Control the density of the frequency grid, which has roughly  $(lgrid*n)/(2*bw)$  frequency points, where `bw` is the fraction of the total frequency band interval `[0,1]` covered by `f`. Increasing `lgrid` often results in filters that more exactly match an equiripple filter, but that take longer to compute. The default value of 16 is the minimum value that should be specified for `lgrid`.

**fresp — Frequency response**

function handle

Frequency response, specified as a function handle. The function is called from within `firgr` with this syntax:

```
[dh,dw] = fresp(n,f,gf,w)
```

where:

- `fresp` identifies the function that you use to define your desired filter frequency response.
- `n` is the filter order.
- `f` is the vector of frequency band edges which must appear monotonically between 0 and 1, where 1 is one-half of the sampling frequency. The frequency bands span  $f(k)$  to  $f(k+1)$  for  $k$  odd. The intervals  $f(k+1)$  to  $f(k+2)$  for  $k$  odd are “transition bands” or “don't care” regions during optimization.
- `gf` is a vector of grid points that have been chosen over each specified frequency band by `firgr`, and determines the frequencies at which `firgr` evaluates the response function.
- `w` is a vector of real, positive weights, one per band, for use during optimization. `w` is optional in the call to `firgr`. If you do not specify `w`, it is set to unity weighting before being passed to `fresp`.
- `dh` and `dw` are the desired frequency response and optimization weight vectors, evaluated at each frequency in grid `gf`.

**Output Arguments****b — Filter coefficients**

row vector

Filter coefficients, returned as a row vector of length  $n + 1$ .

The `firgr` uses the minimax filter design algorithm to design the following types of real FIR filters:

- Types 1-4 linear phase:
  - Type 1 is even order, symmetric
  - Type 2 is odd order, symmetric
  - Type 3 is even order, antisymmetric
  - Type 4 is odd order, antisymmetric
- Minimum phase
- Maximum phase
- Minimum order (even or odd)
- Extra ripple
- Maximal ripple
- Constrained ripple
- Single-point band (notching and peaking)
- Forced gain

- Arbitrary shape frequency response curve filters

### **err** — Unweighted approximation error magnitude

scalar | vector

Unweighted approximation error magnitudes, returned as a scalar or a vector. `err` contains one element for each independent approximation error returned by the function.

### **res** — Frequency response characteristics

structure

Frequency response characteristics, returned as a structure. The structure `res` has the following fields:

| Structure Field             | Contents  |
|-----------------------------|---|
| <code>res.fgrid</code>      | Vector containing the frequency grid used in the filter design optimization   |
| <code>res.des</code>        | Desired response on <code>fgrid</code>  |
| <code>res.wt</code>         | Weights on <code>fgrid</code>   |
| <code>res.h</code>          | Actual frequency response on the frequency grid   |
| <code>res.error</code>      | Error at each point (desired response - actual response) on the frequency grid  |
| <code>res.iextr</code>      | Vector of indices into <code>fgrid</code> of external frequencies   |
| <code>res.fextr</code>      | Vector of external frequencies  |
| <code>res.order</code>      | Filter order  |
| <code>res.edgecheck</code>  | Transition-region anomaly check. One element per band edge. Element values have the following meanings: 1 = OK, 0 = probable transition-region anomaly, -1 = edge not checked. Computed when you specify the 'check' input option in the function syntax. |
| <code>res.iterations</code> | Number of <code>s</code> iterations for the optimization  |
| <code>res.evals</code>      | Number of function evaluations for the optimization   |

## Version History

Introduced in R2011a

### **R2022b: Approximation errors and special properties support array of strings**

Starting in R2022b, you can specify the approximation errors (`e`) and the special properties (`s`) as an array of strings.

```
b = firgr(82,[0 0.055 0.06 0.1 0.15 1], [0 0 0 0 1 1],...
["n" "i" "f" "n" "n" "n"],[10 1 1],["e1" "e2" "e3"]);
```

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- All inputs must be constant. Expressions or variables are allowed if their values do not change.
- Does not support syntaxes that have cell array input.

**See Also**

butter | cheby1 | cheby2 | ellip | freqz | filter | firls | fircls | firpm

## firhalfband

Halfband FIR filter design

### Syntax

```
b = firhalfband(n,fp)
b = firhalfband(n,win)
b = firhalfband(n,dev,'dev')
b = firhalfband('minorder',fp,dev)
b = firhalfband('minorder',fp,dev,'kaiser')
b = firhalfband( ____, 'high')
b = firhalfband( ____, 'minphase')
```

### Description

`b = firhalfband(n,fp)` designs a lowpass FIR halfband filter of order  $n$  with an equiripple characteristic.  $n$  must be an even integer. `fp` determines the passband edge frequency such that  $0 < fp < 1/2$ , where  $1/2$  corresponds to  $\pi/2$  rad/sample.

`b = firhalfband(n,win)` designs a lowpass  $n^{\text{th}}$ -order filter using the truncated windowed-impulse response method instead of the equiripple method. `win` should be an  $n+1$  length vector. The function truncates the ideal response to length  $n+1$ , then multiplies it point-by-point with the window specified in `win`.

`b = firhalfband(n,dev,'dev')` designs an  $n^{\text{th}}$ -order lowpass halfband filter with an equiripple characteristic. `dev` sets the value for the maximum passband and stopband ripple.

`b = firhalfband('minorder',fp,dev)` designs a lowpass minimum-order filter with passband edge `fp`. The peak ripple is constrained by the scalar `dev`. This design uses the equiripple method.

`b = firhalfband('minorder',fp,dev,'kaiser')` designs a lowpass minimum-order filter with passband edge `fp`. The peak ripple is constrained by the scalar `dev`. This design uses the Kaiser window method.

`b = firhalfband( ____, 'high')` returns a highpass halfband FIR filter.

`b = firhalfband( ____, 'minphase')` designs a minimum-phase FIR filter such that the filter is a spectral factor of a halfband filter. Recall that `h = conv(b,flip1r(b))` is a halfband filter. This can be useful for designing perfect reconstruction two-channel FIR filter banks. The **'minphase'** option is not available for window-based halfband filter designs such as `b = firhalfband(n,win)` and `b = firhalfband('minorder',fp,dev,'kaiser')`.

In the minimum phase case, the filter order  $n$  must be odd.

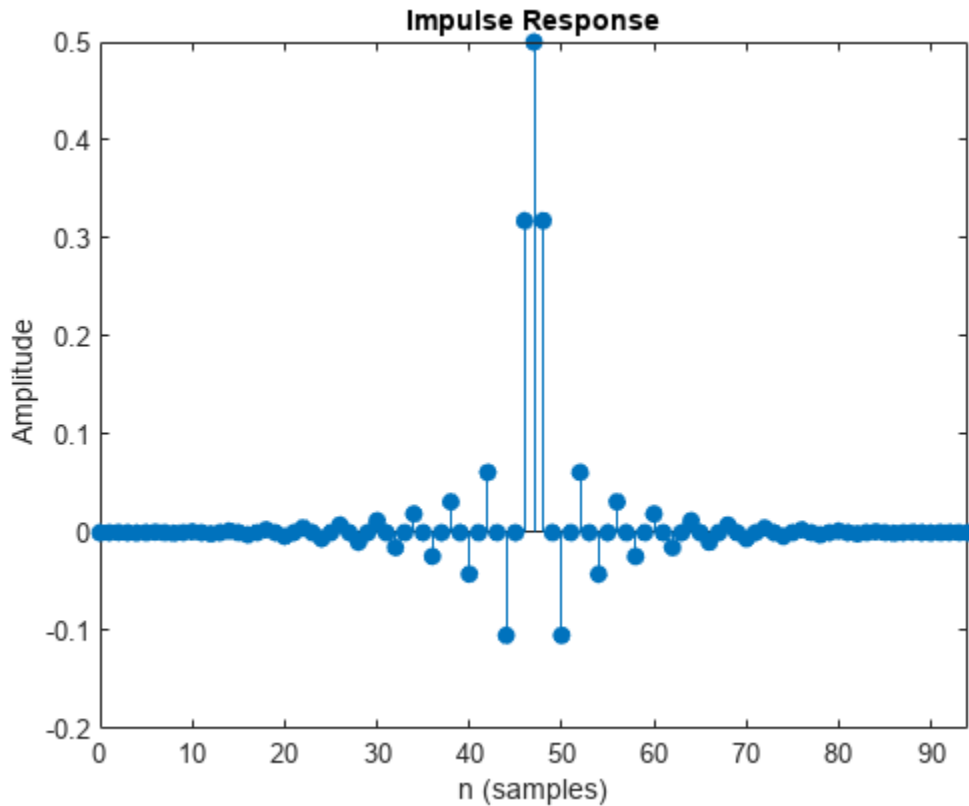
### Examples

#### Design Halfband Filter using firhalfband

Design a minimum order halfband filter with maximum ripple set to 0.0001.



```
b = firhalfband('minorder',.45,0.0001);
impz(b)
```



The impulse response is zero for every alternate sample.

## Input Arguments

### **n** – FIR halfband filter order

positive integer

FIR halfband filter order, specified as an integer

If you specify the `firhalfband` function to design a minimum phase filter using the `'minphase'` argument, the filter order must be odd. In all other cases, the filter order must be even.

Data Types: `single` | `double`

### **fp** – Passband edge frequency

scalar in the range  $(0, \frac{1}{2})$

Passband edge frequency, specified as a scalar in the range  $(0, \frac{1}{2})$ , where  $\frac{1}{2}$  corresponds to  $\pi/2$  rad/sample.

Data Types: `single` | `double`

**win — Window to apply to filter***n*+1 length vector

Window to apply to the filter, specified as an *n*+1 length vector. The function truncates the ideal response to length *n*+1, then multiplies it point-by-point with the window specified in *win*.

Data Types: `single` | `double`

Complex Number Support: Yes

**dev — Maximum passband and stopband ripple**

scalar

Maximum passband and stopband ripple in the halfband filter, specified as a scalar.

Data Types: `single` | `double`**Output Arguments****b — FIR halfband filter coefficients**

vector

FIR halfband filter coefficients, returned as a vector.

Data Types: `single` | `double`

Complex Number Support: Yes

**More About****Halfband Filters**

The ideal lowpass halfband filter is given by

$$h(n) = \frac{1}{2\pi} \int_{-\pi/2}^{\pi/2} e^{j\omega n} d\omega = \frac{\sin(\frac{\pi}{2}n)}{\pi n}.$$

The ideal filter is not realizable because the impulse response is noncausal and not absolutely summable. However, the impulse response of the ideal lowpass filter possesses some important properties that are required of a realizable approximation. Specifically, the ideal lowpass halfband filter's impulse response is:

- Equal to 0 for all even-indexed samples.
- Equal to 1/2 at *n*=0. You can see this by using L'Hopital's rule on the continuous-valued equivalent of the discrete-time impulse response.

The ideal highpass halfband filter is given by

$$g(n) = \frac{1}{2\pi} \int_{-\pi}^{-\pi/2} e^{j\omega n} d\omega + \frac{1}{2\pi} \int_{\pi/2}^{\pi} e^{j\omega n} d\omega.$$

Evaluating the preceding integral gives the following impulse response

$$g(n) = \frac{\sin(\pi n)}{\pi n} - \frac{\sin(\frac{\pi}{2}n)}{\pi n}.$$

The ideal highpass halfband filter's impulse is:

- Equal to 0 for all even-indexed samples
- Equal to 1/2 at  $n=0$

The `firhalfband` function uses a causal FIR approximation to the ideal halfband response, which is based on minimizing the  $\ell^\infty$  norm of the error (minimax).

### Kaiser Window

The coefficients of a Kaiser window are computed using this equation:

$$w(n) = \frac{I_0\left(\beta\sqrt{1 - \left(\frac{n - N/2}{N/2}\right)^2}\right)}{I_0(\beta)}, \quad 0 \leq n \leq N,$$

where  $I_0$  is the zeroth-order modified Bessel function of the first kind.

To obtain a Kaiser window that represents an FIR filter with stopband attenuation of  $\alpha$  dB, use this  $\beta$ .

$$\beta = \begin{cases} 0.1102(\alpha - 8.7), & \alpha > 50 \\ 0.5842(\alpha - 21)^{0.4} + 0.07886(\alpha - 21), & 50 \geq \alpha \geq 21 \\ 0, & \alpha < 21 \end{cases}$$

The filter order  $n$  is given by:

$$n = \frac{\alpha - 7.95}{2.285(\Delta\omega)}$$

where  $\Delta\omega$  is the transition width.

For more details, see "Algorithms" on page 5-683.

## Algorithms

The `firhalfband` function uses the equiripple or the Kaiser window method to design the FIR halfband filter. You can also specify a custom window using the `win` argument.

### Halfband Equiripple Design

In the equiripple method, the algorithm uses a minimax (minimize the maximum error) FIR design to design a fullband linear phase filter with the desired specifications. The algorithm upsamples a fullband filter to replace the even-indexed samples of the filter with zeros and creates a halfband filter. It then sets the filter tap corresponding to the group delay of the filter in samples to 1/2. This yields a causal linear-phase FIR filter approximation to the ideal halfband filter defined in "Halfband Filters" on page 5-682. See [2] for a description of this filter design method using the Remez exchange algorithm. As you can design a filter using this approximation method with a constant ripple both in the passband and stopband, the filter is also known as the equiripple filter.

### Window-based Design

In the window-based design method, the algorithm first truncates the ideal halfband filter defined in "Halfband Filters" on page 5-682, then it applies the user-specified window. This yields a causal

linear-phase FIR filter approximation to the ideal halfband filter. If you provide the 'kaiser' argument, the function calculates the window as mentioned in "Kaiser Window" on page 5-683.

For more information on designing FIR halfband filters, see "FIR Halfband Filter Design".

## Version History

### Introduced in R2011a

#### **R2022b: Iterative filter design**

*Behavior change in future release*

Starting R2022b, when you provide the 'kaiser' argument, if the designed filter does not meet the required specifications, the `firhalfband` function designs the filter by iterating on the filter order obtained using the empirical formula in "Kaiser Window" on page 5-683.

## References

- [1] Saramaki, T, "Finite Impulse Response Filter Design," *Handbook for Digital Signal Processing*. S.K. Mitra and J.F. Kaiser Eds. Wiley-Interscience, N.Y., 1993, Chapter 4.
- [2] Harris, F.J. *Multirate Signal Processing for Communication Systems*, Prentice Hall, 2004, pp. 208-209.

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

All inputs must be constant. Expressions or variables are allowed if their values do not change.

## See Also

### **Functions**

`firnyquist` | `firgr` | `fir1` | `firls` | `firpm`

## **fir2lp**

Transform FIR Type I lowpass to FIR Type 1 lowpass with inverse bandwidth

### **Syntax**

```
g = fir2lp(b)
```

### **Description**

`g = fir2lp(b)` transforms the Type I lowpass FIR filter `b` with zero-phase response  $H_r(w)$  to a Type I lowpass FIR filter `g` with zero-phase response  $[1 - H_r(\pi-w)]$ .

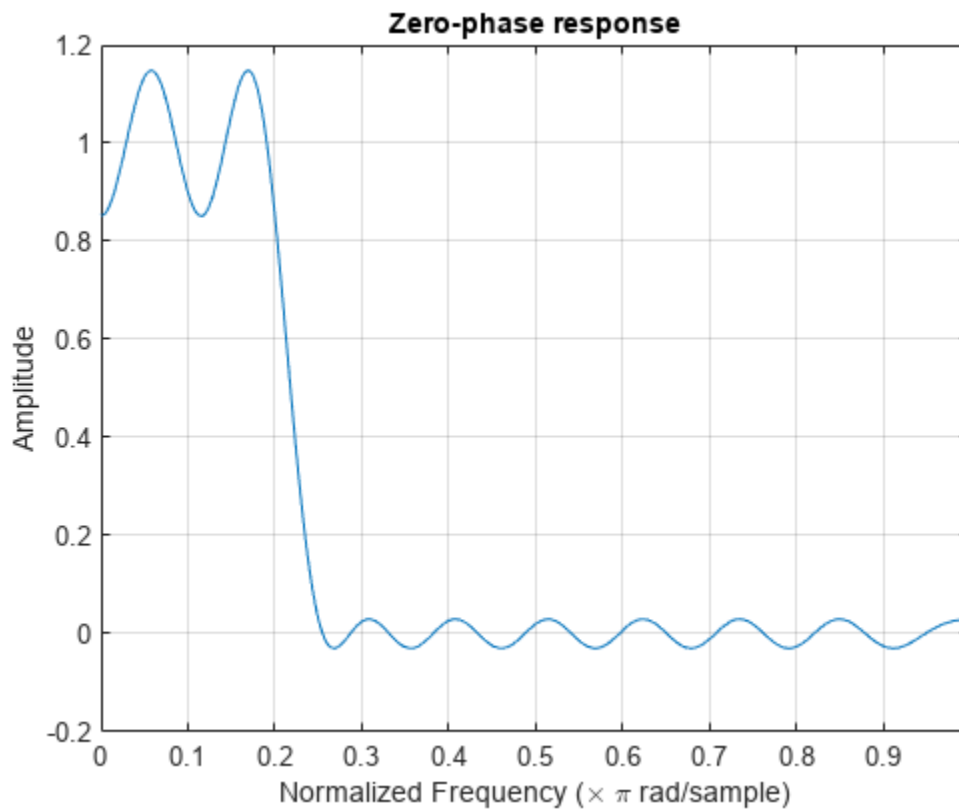
When `b` is a narrowband filter, `g` will be a wideband filter and vice versa. The passband and stopband ripples of `g` will be equal to the stopband and passband ripples of `b`, respectively.

### **Examples**

#### **Convert Narrowband Lowpass Filter to Wideband Lowpass Filter**

Create a narrowband lowpass filter to use as a prototype. Display its zero-phase response.

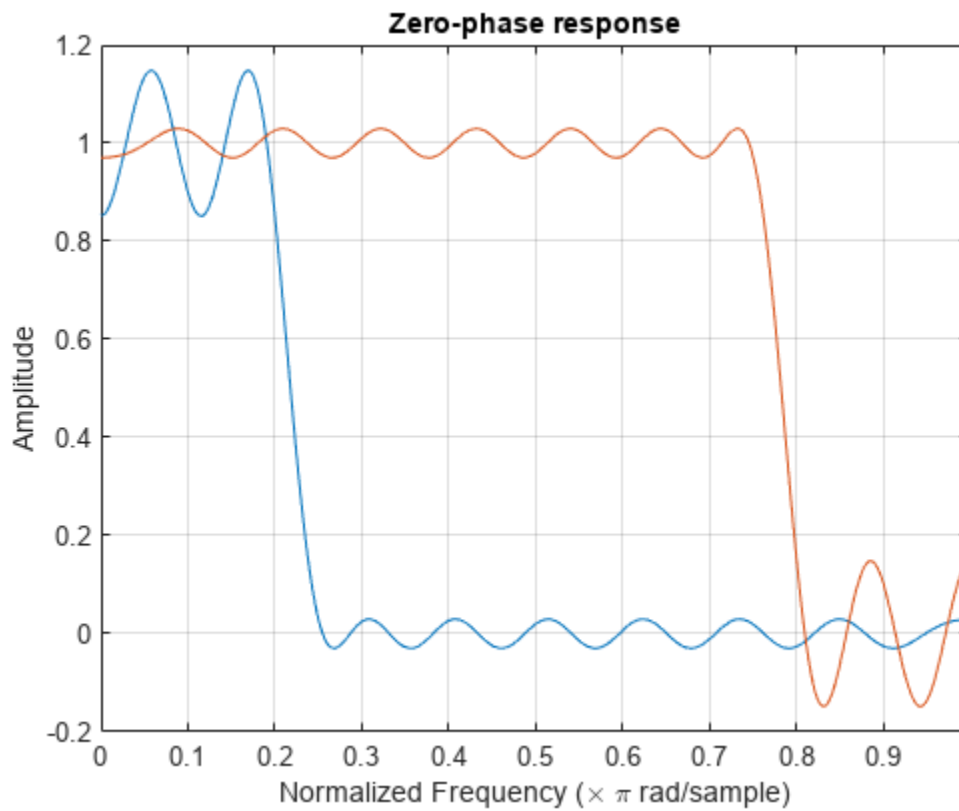
```
b = firgr(36,[0 0.2 0.25 1],[1 1 0 0],[1 5]);  
zerophase(b)
```



Convert the prototype filter to a wideband lowpass filter. Add to the plot the zero-phase response of the new filter.

```
h = fir1p2lp(b);
```

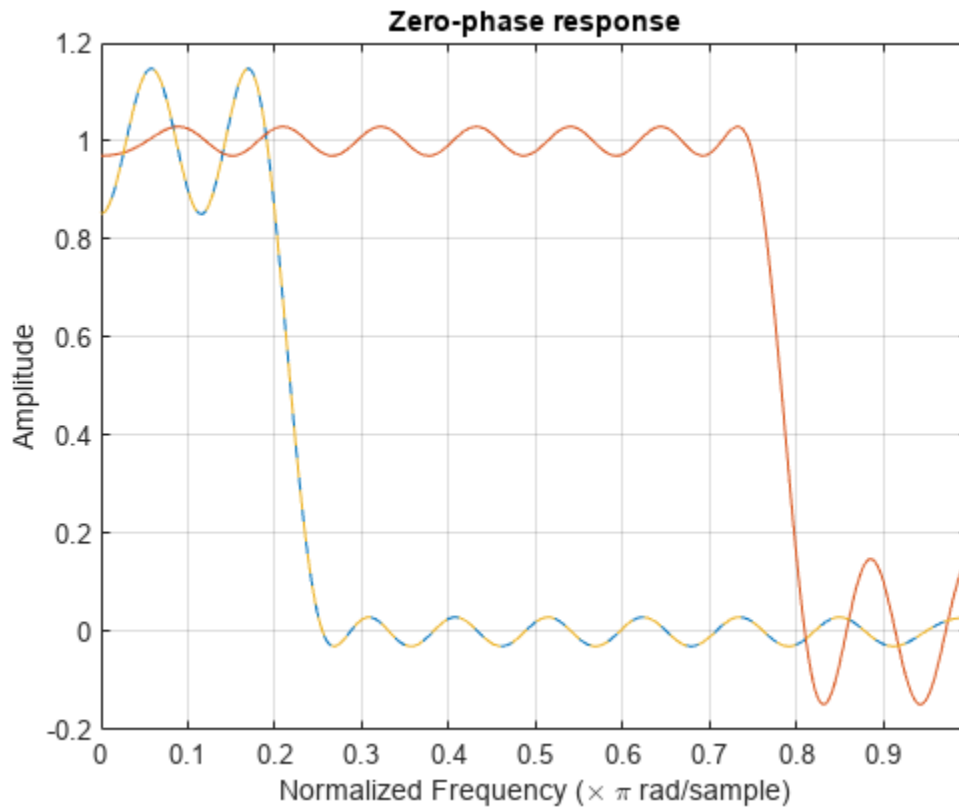
```
hold on  
zerophase(h)
```



Convert the previous filter back to a narrowband lowpass filter. Add to the plot the zero-phase response of the new filter.

```
g = fir2lp(h);
```

```
[gr,w] = zerophas(g);  
plot(w/pi,gr,'--')  
hold off
```



## Input Arguments

### **b** — Lowpass FIR filter coefficients

row vector

Type 1 lowpass FIR filter coefficients with zero-phase response  $H_r(w)$ , specified as a row vector.

Data Types: single | double

## Output Arguments

### **g** — Transformed lowpass FIR filter coefficients

row vector

Transformed Type 1 lowpass FIR filter coefficients with zero-phase response  $1-H_r(\pi-w)$ , specified as a row vector.

Data Types: single | double

## Version History

Introduced in R2011a



## References

- [1] Mitra, Sanjit Kumar, and James F. Kaiser, eds. *Handbook for Digital Signal Processing*. New York: Wiley, 1993.

## See Also

firlp2hp | zerophase

## firlp2hp

Transform FIR lowpass filter to Type I FIR highpass filter

### Syntax

```
g = firlp2hp(b)
g = firlp2hp(b,'narrow')
g = firlp2hp(b,'wide')
```

### Description

`g = firlp2hp(b)` transforms the lowpass FIR filter `b` with zero-phase response  $H_r(w)$  into a Type I highpass FIR filter `g` with zero-phase response  $H_r(\pi-w)$ . Filter `b` can be any FIR filter, including a nonlinear-phase filter.

The passband and stopband ripples of `g` will be equal to the passband and stopband ripples of `b`, respectively.

`g = firlp2hp(b,'narrow')` transforms the lowpass FIR filter `b` into a Type I narrowband highpass FIR filter `g` with zero-phase response  $H_r(\pi-w)$ . `b` can be any FIR filter, including a nonlinear-phase filter.

`g = firlp2hp(b,'wide')` transforms the Type I lowpass FIR filter `b` with zero-phase response  $H_r(w)$  into a Type I wideband highpass FIR filter `g` with zero-phase response  $1 - H_r(w)$ .

---

**Note** The `b` must be a Type I linear-phase filter.

---

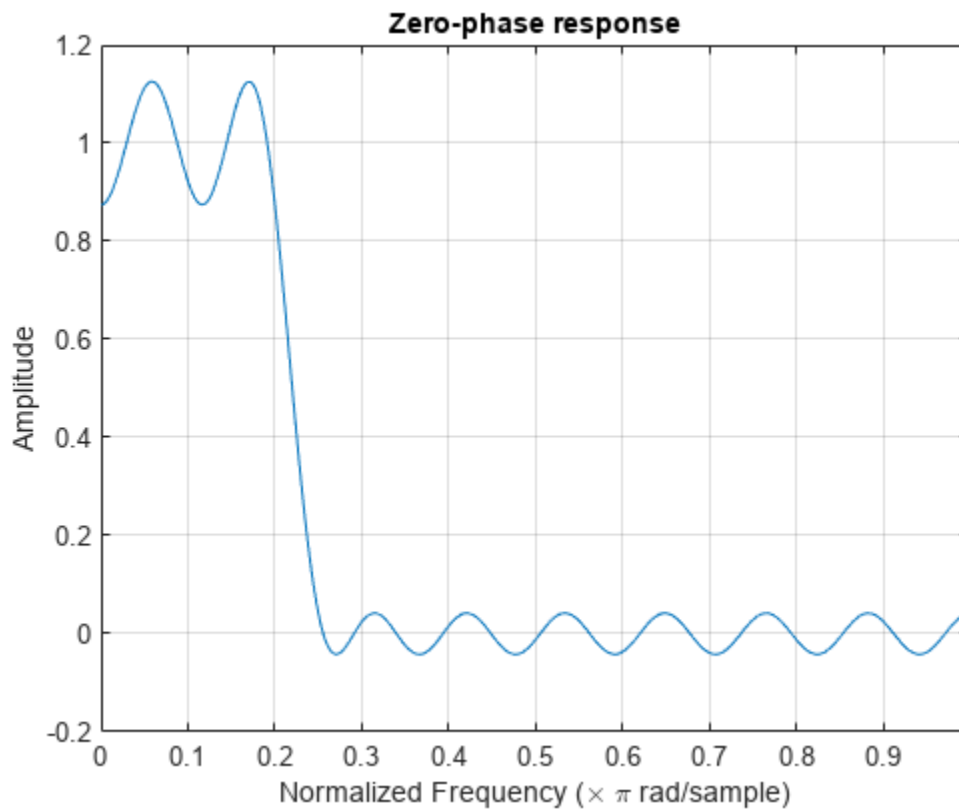
For this case, the passband and stopband ripples of `g` will be equal to the stopband and passband ripples of `b`.

### Examples

#### Convert Narrowband Lowpass Filter to Highpass Filter

Create a narrowband lowpass filter to use as a prototype. Display its zero-phase response.

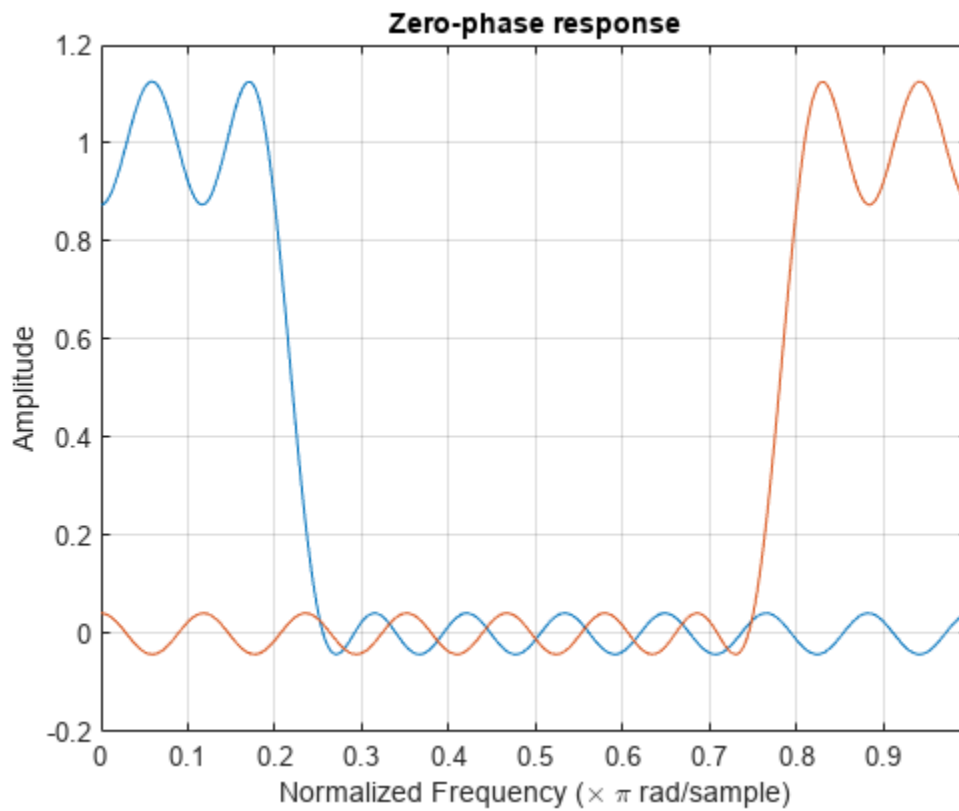
```
b = firgr(36,[0 0.2 0.25 1],[1 1 0 0],[1 3]);
zerophase(b)
```



Convert the prototype filter to a narrowband highpass filter. Add to the plot the zero-phase response of the new filter.

```
h = fir1p2hp(b);
```

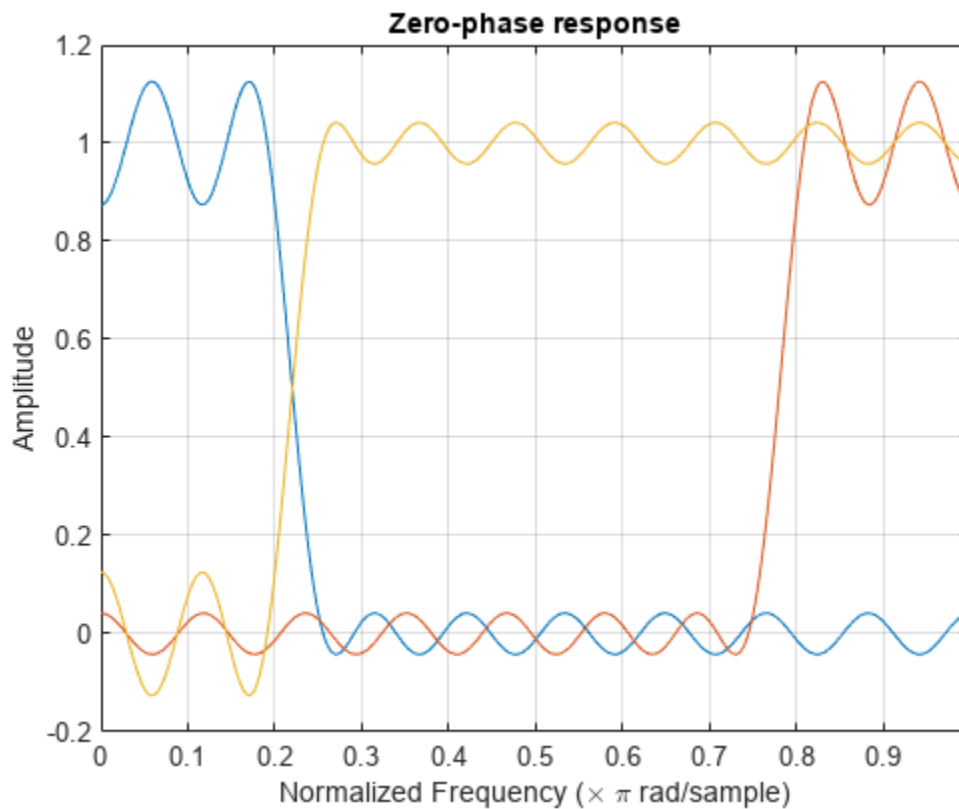
```
hold on  
zerophase(h)
```



Convert the prototype filter to a wideband highpass filter. Add to the plot the zero-phase response of the new filter.

```
g = fir1p2hp(b, 'wide');
```

```
zerophase(g)  
hold off
```



## Input Arguments

### **b** — Lowpass FIR filter coefficients

row vector

Lowpass FIR filter coefficients, specified as a row vector.

Data Types: `single` | `double`

## Output Arguments

### **g** — Type I highpass FIR filter coefficients

row vector

Type I highpass FIR filter coefficients, returned as a row vector.

Data Types: `single` | `double`

## Version History

Introduced in R2011a

## References

[1] Mitra, Sanjit Kumar, and James F. Kaiser, eds. *Handbook for Digital Signal Processing*. New York: Wiley, 1993.

## See Also

firlp2lp | zerophase

# firlpnorm

Least P-norm optimal FIR filter

## Syntax

```
b = firlpnorm(n,f,edges,a)
b = firlpnorm(n,f,edges,a,w)
b = firlpnorm(n,f,edges,a,w,p)
b = firlpnorm(n,f,edges,a,w,p,dens)
b = firlpnorm(n,f,edges,a,w,p,dens,initnum)
b = firlpnorm(...,'minphase')
[b,err] = firlpnorm(...)
```

## Description

`b = firlpnorm(n,f,edges,a)` returns a filter of numerator order `n` which represents the best approximation to the frequency response described by `f` and `a` in the least-Pth norm sense. `P` is set to 128 by default, which is essentially equivalent to the infinity norm. Vector `edges` specifies the band-edge frequencies for multiband designs. `firlpnorm` uses an unconstrained quasi-Newton algorithm to design the specified filter.

`f` and `a` must have the same number of elements, which can exceed the number of elements in `edges`. This lets you specify filters with any gain contour within each band. However, the frequencies in `edges` must also be in vector `f`. Always use `freqz` to check the resulting filter.

---

**Note** `firlpnorm` uses a nonlinear optimization routine that may not converge in some filter design cases. Furthermore the algorithm is not well-suited for certain large-order (order > 100) filter designs.

---

`b = firlpnorm(n,f,edges,a,w)` uses the weights in `w` to weight the error. `w` has one entry per frequency point (the same length as `f` and `a`) which tells `firlpnorm` how much emphasis to put on minimizing the error in the vicinity of each frequency point relative to the other points. For example,

```
b = firlpnorm(20,[0 .15 .4 .5 1],[0 .4 .5 1],...
[1 1.6 1 0 0],[1 1 1 10 10])
```

designs a lowpass filter with a peak of 1.6 within the passband, and with emphasis placed on minimizing the error in the stopband.

`b = firlpnorm(n,f,edges,a,w,p)` where `p` is a two-element vector [`pmin pmax`] lets you specify the minimum and maximum values of `p` used in the least-pth algorithm. Default is [2 128] which essentially yields the L-infinity, or Chebyshev, norm. `pmin` and `pmax` should be even numbers. The design algorithm starts optimizing the filter with `pmin` and moves toward an optimal filter in the `pmax` sense. When `p` is set to **'inspect'**, `firlpnorm` does not optimize the resulting filter. You might use this feature to inspect the initial zero placement.

`b = firlpnorm(n,f,edges,a,w,p,dens)` specifies the grid density `dens` used in the optimization. The number of grid points is [`dens*(n+1)`]. The default is 20. You can specify `dens` as a single-element cell array. The grid is equally spaced.

`b = firlnorm(n,f,edges,a,w,p,dens,initnum)` lets you determine the initial estimate of the filter numerator coefficients in vector `initnum`. This can prove helpful for difficult optimization problems. The pole-zero editor in Signal Processing Toolbox software can be used for generating `initnum`.

`b = firlnorm(...,'minphase')` where `'minphase'` is the last argument in the argument list generates a minimum-phase FIR filter. By default, `firlnorm` design mixed-phase filters. Specifying input option `'minphase'` causes `firlnorm` to use a different optimization method to design the minimum-phase filter. As a result of the different optimization used, the minimum-phase filter can yield slightly different results.

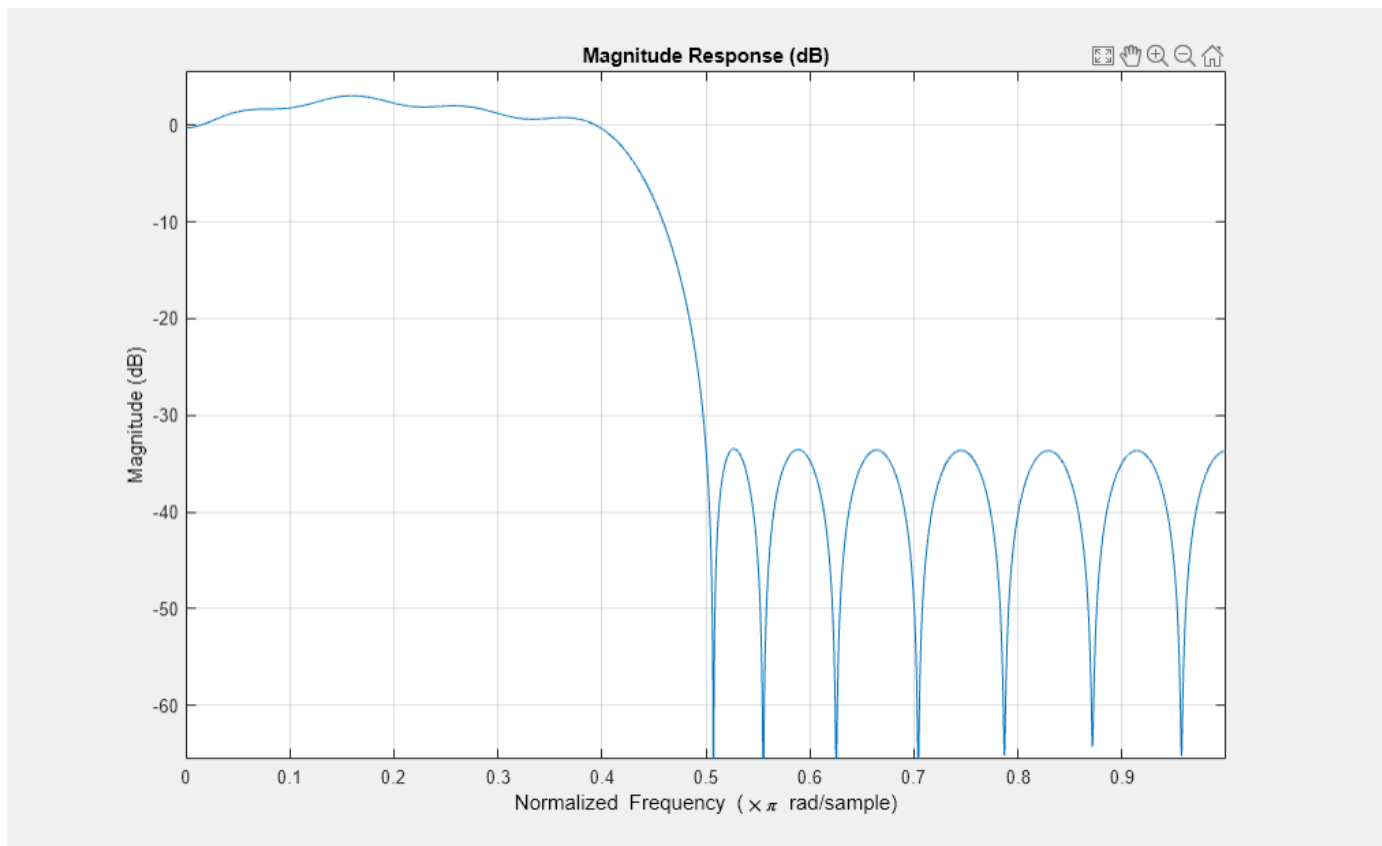
`[b,err] = firlnorm(...)` returns the least-pth approximation error `err`.

## Examples

### Design a Lowpass and Highpass Filter Using `firlnorm`

Lowpass filter with a peak of 1.4 in the passband.

```
b = firlnorm(22,[0 .15 .4 .5 1],[0 .4 .5 1],[1 1.4 1 0 0],...
[1 1 1 2 2]);
fvtool(b)
```

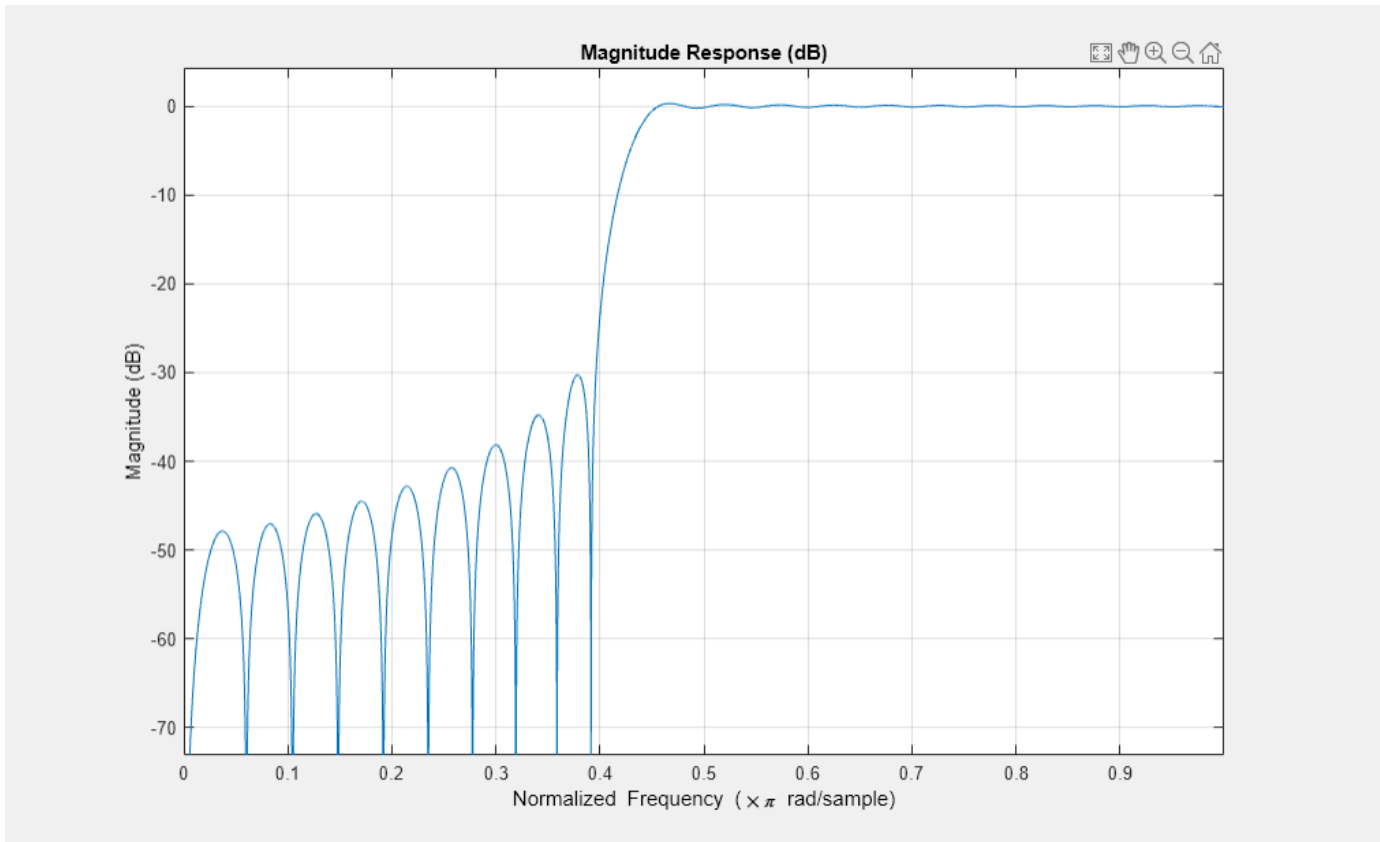


The resulting filter is lowpass, with the desired 1.4 peak in the passband (notice the 1.4 specified in vector `a`).



Highpass minimum-phase filter optimized for the 4-norm.

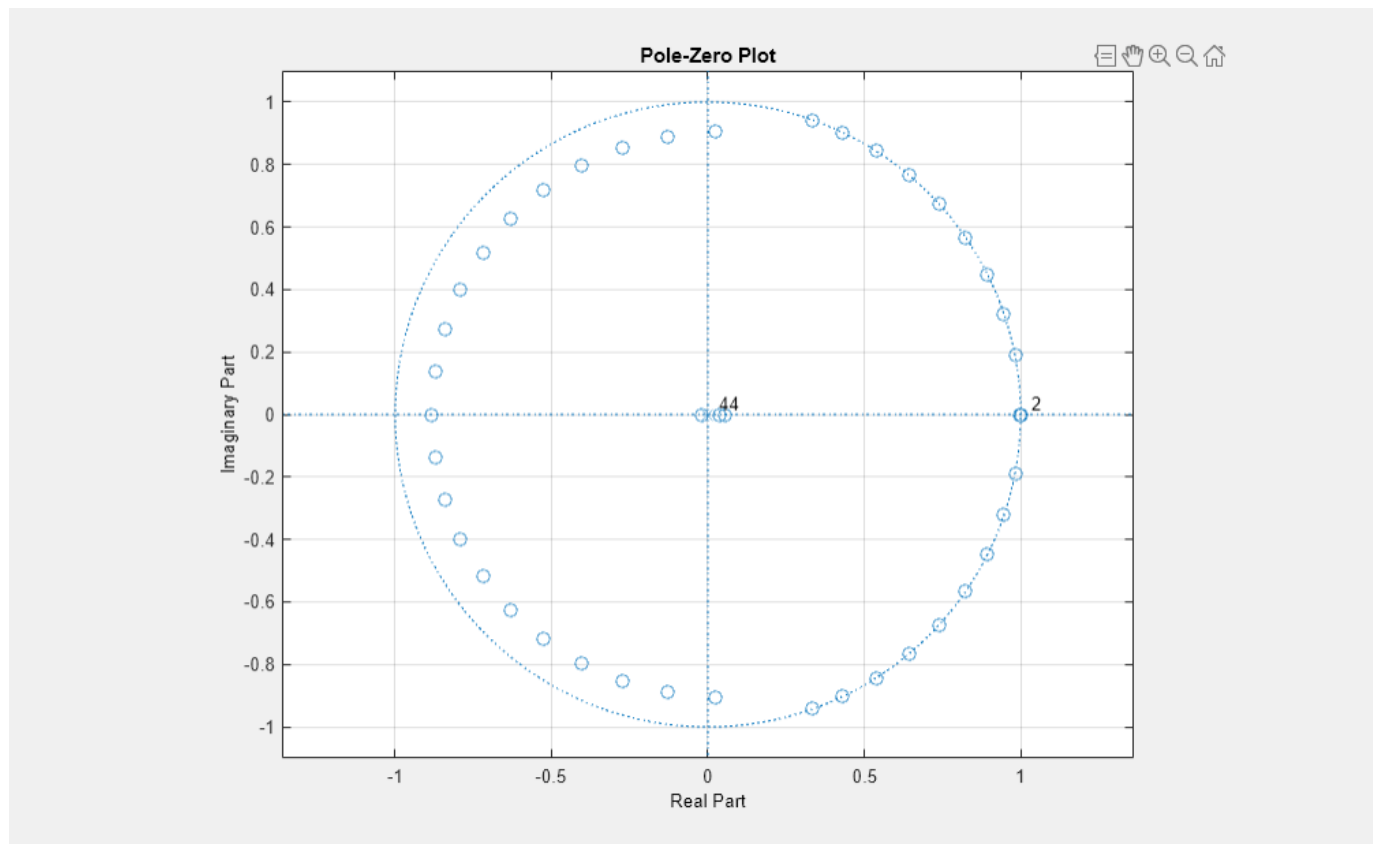
```
b = firlpnorm(44,[0 .4 .45 1],[0 .4 .45 1],[0 0 1 1],[5 1 1 1],...  
[2 4], 'minphase');  
fvtool(b)
```



This is a minimum-phase, highpass filter.

The zero-pole plot shows the minimum phase nature more clearly.

```
fvtool(b, 'polezero')
```



## Version History

Introduced in R2011a

## References

Saramaki, T, Finite Impulse Response Filter Design, *Handbook for Digital Signal Processing* Mitra, S.K. and J.F. Kaiser Eds. Wiley-Interscience, N.Y., 1993, Chapter 4.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- All inputs must be constant. Expressions or variables are allowed if their values do not change.
- Does not support syntaxes that have cell array input.

## See Also

### Functions

[firgr](#) | [iirgrpdelay](#) | [iirlpnorm](#) | [iirlpnormc](#) | [filter](#) | [freqz](#) | [zplane](#)

### Tools

[FVTool](#)

## firls

Least-square linear-phase FIR filter design

### Syntax

```
b = firls(n,f,a)
b = firls(n,f,a,w)
b = firls(n,f,a,ftype)
b = firls(n,f,a,w,ftype)
```

### Description

`b = firls(n,f,a)` returns row vector `b` containing the  $n+1$  coefficients of the order  $n$  FIR filter. This filter has frequency-amplitude characteristics approximately matching those given by vectors, `f` and `a`.

`b = firls(n,f,a,w)` uses the weights in vector `w`, to weigh the error.

`b = firls(n,f,a,ftype)` specifies a filter type where `ftype` is:

- 'hilbert'
- 'differentiator'

`b = firls(n,f,a,w,ftype)` uses the weights in vector `w` to weigh the error. It also specifies a filter type where `ftype` is:

- 'hilbert'
- 'differentiator'

### Examples

#### Design a Lowpass Filter with Transition Band

The following illustrates how to design a lowpass filter of order 225 with transition band.

Create the frequency and amplitude vectors, `f` and `a`.

```
f = [0 0.25 0.3 1]
```

```
f = 1×4
```

```
0    0.2500    0.3000    1.0000
```

```
a = [1 1 0 0]
```

```
a = 1×4
```

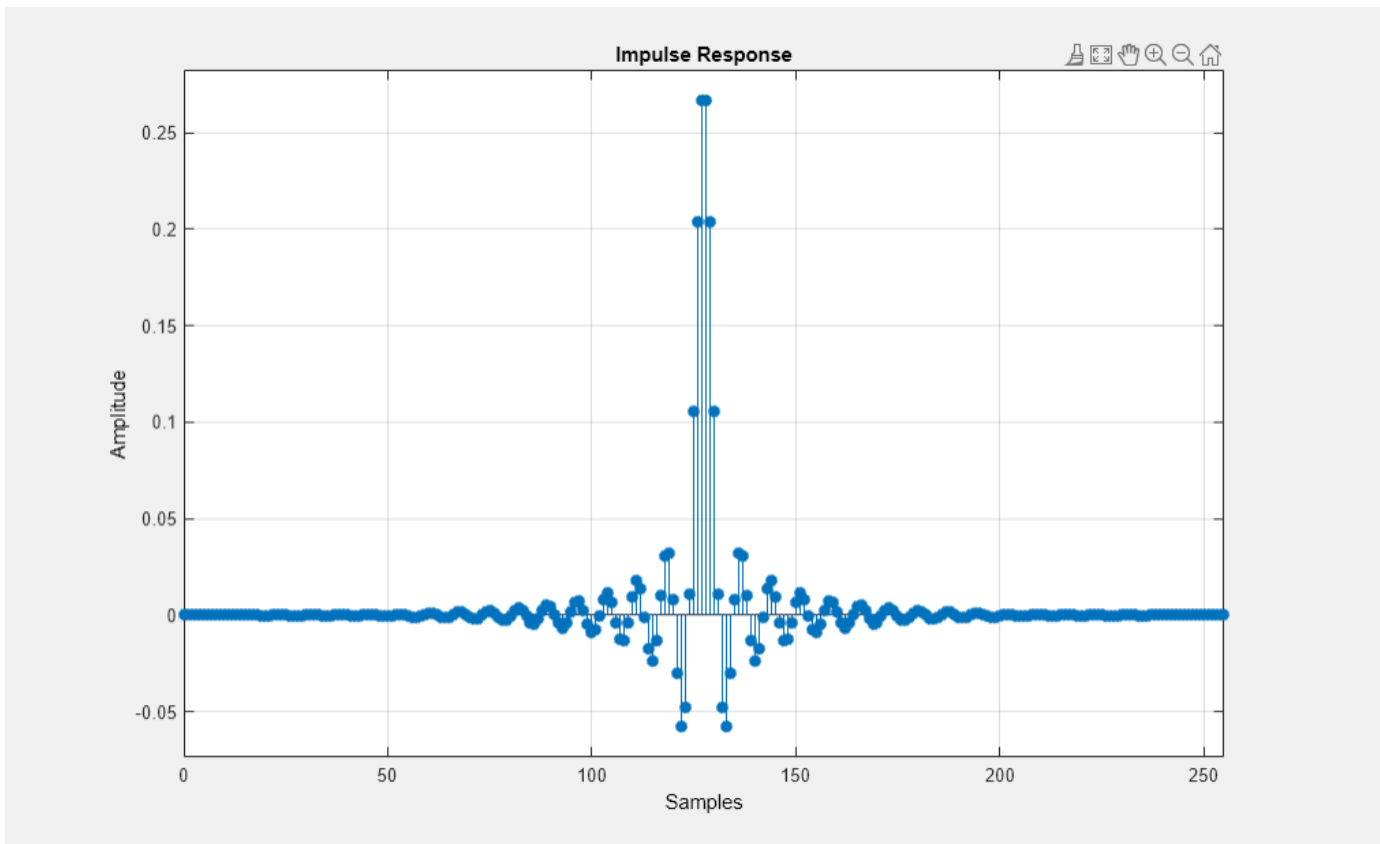
```
1    1    0    0
```

Use `firls` to obtain the  $n+1$  coefficients of the order  $n$  lowpass FIR filter.

```
b = firls(255,f,a);
```

Show the impulse response of the filter

```
fvtool(b,'impulse')
```



### Design an Antisymmetric Filter with Piecewise Linear Passbands

The following shows how to design a 24th-order anti-symmetric filter with piecewise linear passbands, and plot the desired and actual amplitude responses.

Create the frequency and amplitude vectors, `f` and `a`.

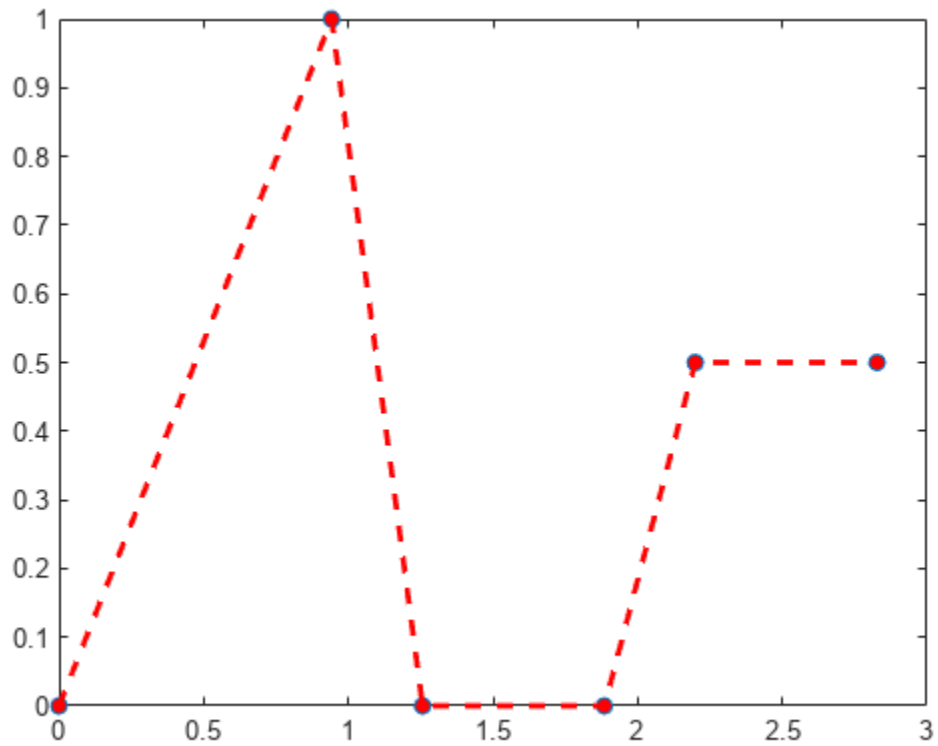
```
f = [0 0.3 0.4 0.6 0.7 0.9];
a = [0 1 0 0 0.5 0.5];
```

Use `firls` to obtain the 25 coefficients of the filter.

```
b = firls(24,f,a,'hilbert');
```

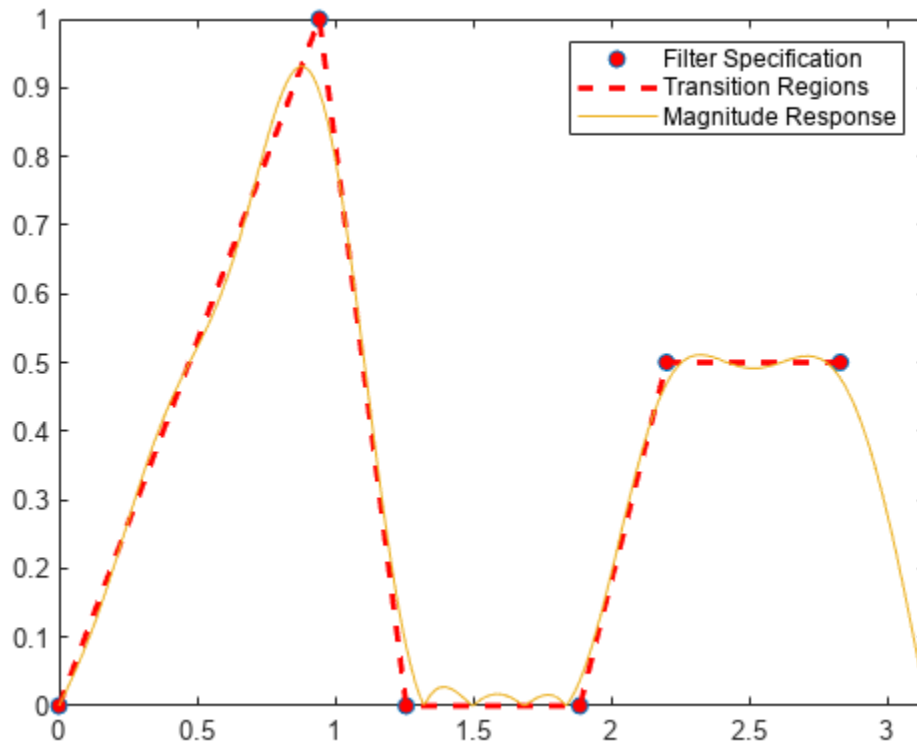
Plot the ideal amplitude response along with the transition regions.

```
plot(f.*pi,a,'o','markerfacecolor',[1 0 0]);  
hold on;  
plot(f.*pi,a,'r--','linewidth',2);
```



Use `freqz` to obtain the frequency response of the designed filter and plot the magnitude response of the filter.

```
[H,F] = freqz(b,1);  
plot(F,abs(H));  
set(gca,'xlim',[0 pi])  
legend('Filter Specification','Transition Regions','Magnitude Response')
```



## Input Arguments

### **n** – Filter order

integer scalar

Order of the filter, specified as an integer scalar. For odd orders, the frequency response at the Nyquist frequency is necessarily 0. For this reason, `firls` always uses an even filter order for configurations with a passband at the Nyquist frequency. If you specify an odd-valued `n`, `firls` increments it by 1.

Example: 8

Data Types: `int8` | `int16` | `int32` | `int64`

### **f** – Pairs of frequency points

vector of numeric values

Pairs of frequency points, specified as a vector of values ranging between 0 and 1, where 1 corresponds to the Nyquist frequency. The frequencies must be in increasing order, and duplicate frequency points are allowed. You can use duplicate frequency points to design filters exactly like those returned by the `fir1` and `fir2` functions with a rectangular (`rectwin`) window.

`f` and `a` are the same length. This length must be an even number.

Example: `[0 0.3 0.4 1]`

Data Types: double | single

### **a — Amplitude values**

vector of numeric values

Amplitude values of the function at each frequency point, specified as a vector of the same length as *f*. This length must be an even number.

The desired amplitude at frequencies between pairs of points ( $f(k)$ ,  $f(k+1)$ ) for  $k$  odd, is the line segment connecting the points ( $f(k)$ ,  $a(k)$ ) and ( $f(k+1)$ ,  $a(k+1)$ ).

The desired amplitude at frequencies between pairs of points ( $f(k)$ ,  $f(k+1)$ ) for  $k$  even is unspecified. These are transition or “don't care” regions.

Example: [1 1 0 0]

Data Types: double | single

### **w — Weights**

vector of numeric values

Weights to weigh the fit for each frequency band, specified as a vector of length half the length of *f* and *a*, so there is exactly one weight per band. *w* indicates how much emphasis to put on minimizing the integral squared error in each band, relative to the other bands.

Example: [0.5 1]

Data Types: double | single

### **ftype — Filter type**

'hilbert' and 'differentiator'

Filter type, specified as either 'hilbert' or 'differentiator'.

Example: 'hilbert'

Data Types: char

## **Output Arguments**

### **b — Filter coefficients**

vector of numeric values

Filter coefficients, returned as a numeric vector of  $n+1$  values, where  $n$  is the filter order.

$b = \text{firls}(n, f, a)$  designs a linear-phase filter of type I ( $n$  odd) and type II ( $n$ ). The output coefficients, or “taps,” in *b* obey the relation:

$$b(k) = b(n+2-k), k = 1, \dots, n + 1$$

$b = \text{firls}(n, f, a, \text{'hilbert'})$  designs a linear-phase filter with odd symmetry (type III and type IV). The output coefficients, or “taps,” in *b* obey the relation:

$$b(k) = -b(n+2-k), k = 1, \dots, n + 1$$

$b = \text{firls}(n, f, a, \text{'differentiator'})$  designs type III and type IV filters, using a special weighting technique. For nonzero amplitude bands, the integrated squared error has a weight of  $(1/$



$f)^2$ . This weighting causes the error at low frequencies to be much smaller than at high frequencies. For FIR differentiators, which have an amplitude characteristic proportional to frequency, the filters minimize the relative integrated squared error. This value is the integral of the square of the ratio of the error to the desired amplitude.

## More About

### Diagnostics

Error and warning messages

One of the following diagnostic messages is displayed when an incorrect argument is used:

```
F must be even length.
F and A must be equal lengths.
Requires symmetry to be 'hilbert' or 'differentiator'.
Requires one weight per band.
Frequencies in F must be nondecreasing.
Frequencies in F must be in range [0,1].
```

A more serious warning message is

```
Warning: Matrix is close to singular or badly scaled.
```

This tends to happen when the product of the filter length and transition width grows large. In this case, the filter coefficients  $b$  might not represent the desired filter. You can check the filter by looking at its frequency response.

## Algorithms

`firls` designs a linear-phase FIR filter. This filter minimizes the weighted, integrated squared error between an ideal piecewise linear function and the magnitude response of the filter over a set of desired frequency bands.

Reference [1] describes the theoretical approach behind `firls`. The function solves a system of linear equations involving an inner product matrix of size roughly  $n/2$  using the MATLAB `\` operator.

This function designs type I, II, III, and IV linear-phase filters. Type I and II are the defaults for  $n$  even and odd respectively. The 'hilbert' and 'differentiator' flags produce type III ( $n$  even) and IV ( $n$  odd) filters. The various filter types have different symmetries and constraints on their frequency responses (see [2] for details).

| Linear Phase Filter Type | Filter Order | Symmetry of Coefficients               | Response $H(f)$ , $f = 0$ | Response $H(f)$ , $f = 1$ (Nyquist) |
|--------------------------|--------------|--|---------------------------|-------------------------------------|
| Type I                   | Even         | $b(k) = b(n+2-k)$ , $k=1, \dots, n+1$  | No restriction            | No restriction                      |
| Type II                  | Odd          | $b(k) = b(n+2-k)$ , $k=1, \dots, n+1$  | No restriction            | $H(1) = 0$                          |
| Type III                 | Even         | $b(k) = -b(n+2-k)$ , $k=1, \dots, n+1$ | $H(0) = 0$                | $H(1) = 0$                          |
| Type IV                  | Odd          | $b(k) = -b(n+2-k)$ , $k=1, \dots, n+1$ | $H(0) = 0$                | No restriction                      |

## Version History

Introduced in R2011a

## References

- [1] Parks, T.W., and C.S. Burrus, *Digital Filter Design*, John Wiley & Sons, 1987, pp. 54-83.
- [2] Oppenheim, A.V., and R.W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989, pp. 256-266.

## See Also

`fir1` | `fir2` | `rcosdesign` | `firpm`

# firminphase

Minimum-phase FIR spectral factor

## Syntax

```
h = firminphase(b)
h = firminphase(b,nz)
```

## Description

`h = firminphase(b)` computes the minimum-phase FIR spectral factor `h` of a linear-phase FIR filter `b`.

You can find the maximum-phase spectral factor, `g`, by reversing `h` such that `g = flip1r(h)`, and `b = conv(h, g)`.

`h = firminphase(b, nz)` specifies the number of zeros, `nz`, of `b` that lie on the unit circle.

## Examples

### Design Constrained Least Squares Filter

Design a constrained least squares filter with a nonnegative zero-phase response, and then use the `firminphase` function to compute the minimum-phase spectral factor.

```
f = [0 0.4 0.8 1];
a = [0 1 0];
up = [0.02 1.02 0.01];
lo = [0 0.98 0]; % The zeros ensure nonnegative zero-phase resp.
n = 32;
b = fircls(n, f, a, up, lo);
h = firminphase(b)
```

```
h = 1×17
```

```
0.2397 -0.1556 -0.2834 0.3866 0.0415 -0.2529 0.0584 -0.0028 0.0868 0.0
```

## Input Arguments

### **b** — Lowpass FIR filter coefficients

row vector

Lowpass FIR filter coefficients, specified as a row vector. Elements of `b` must be real, of even order, and have nonnegative zero-phase response.

Data Types: `single` | `double`

### **nz** — Number of zeros

positive integer

Number of zeros of **b** that lie on the unit circle, specified as a positive integer. You must specify **nz** as an even number to compute the minimum-phase spectral factor because every root on the unit circle must have even multiplicity. Including **nz** can help the `firminphase` function calculate the required FIR spectral factor. Zeros with multiplicity greater than two on the unit circle cause problems in the spectral factor determination.

Data Types: `single` | `double` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `int64` | `uint64`

## Output Arguments

### **h** — Minimum-phase FIR spectral factor coefficients

row vector

Minimum-phase FIR spectral factor coefficients of a linear-phase FIR filter **b**, returned as a row vector.

Data Types: `single` | `double`

## Version History

Introduced in R2011a

## References

[1] Mitra, Sanjit Kumar, and James F. Kaiser, eds. *Handbook for Digital Signal Processing*. New York: Wiley, 1993.

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

All inputs must be constant. Expressions or variables are allowed if their values do not change.

### **See Also**

`firgr` | `fircls` | `zerophase`

# firnyquist

Lowpass Nyquist (Lth-band) FIR filter

## Syntax

```
b = firnyquist(n,l,r)
b = firnyquist('minorder',l,r,dev)
b = firnyquist(n,l,r,decay)
b = firnyquist(n,l,r,'nonnegative')
b = firnyquist(n,l,r,'minphase')
```

## Description

`b = firnyquist(n,l,r)` designs an Nth order, Lth band, Nyquist FIR filter with a rolloff factor  $r$  and an equiripple characteristic.

The rolloff factor  $r$  is related to the normalized transition width  $tw$  by  $tw = 2\pi(r/l)$  (rad/sample). The order,  $n$ , must be even.  $l$  must be an integer greater than one. If  $l$  is not specified, it defaults to 4.  $r$  must satisfy  $0 < r < 1$ . If  $r$  is not specified, it defaults to 0.5.

`b = firnyquist('minorder',l,r,dev)` designs a minimum-order, Lth band Nyquist FIR filter with a rolloff factor  $r$  using the Kaiser window. The peak ripple is constrained by the scalar `dev`.

`b = firnyquist(n,l,r,decay)` designs an Nth order ( $n$ ), Lth band ( $l$ ), Nyquist FIR filter where the scalar `decay`, specifies the rate of decay in the stopband. `decay` must be nonnegative. If you omit or leave it empty, `decay` defaults to 0 which yields an equiripple stopband. A nonequiripple stopband (`decay`  $\neq$  0) may be desirable for decimation purposes.

`b = firnyquist(n,l,r,'nonnegative')` returns an FIR filter with nonnegative zero-phase response. This filter can be spectrally factored into minimum-phase and maximum-phase “square-root” filters. This allows you to use the spectral factors in applications such as matched-filtering.

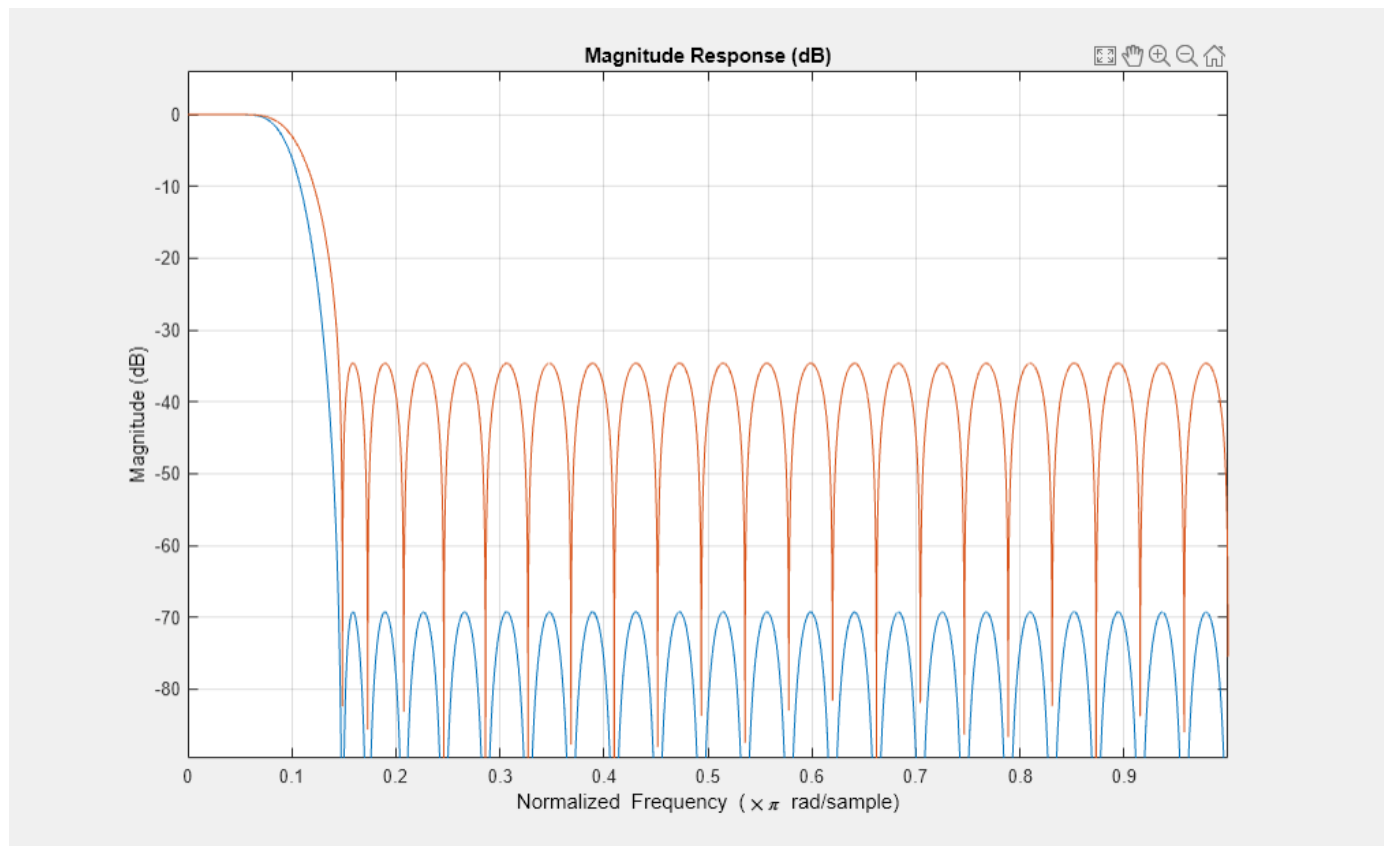
`b = firnyquist(n,l,r,'minphase')` returns the minimum-phase spectral factor `bmin` of order  $n$ . `bmin` meets the condition `b=conv(bmin,bmax)` so that `b` is an Lth band FIR Nyquist filter of order  $2n$  with filter rolloff factor  $r$ . Obtain `bmax`, the maximum phase spectral factor by reversing the coefficients of `bmin`. For example, `bmax = bmin(end:-1:1)`.

## Examples

### Design a Nyquist Filter Using firnyquist

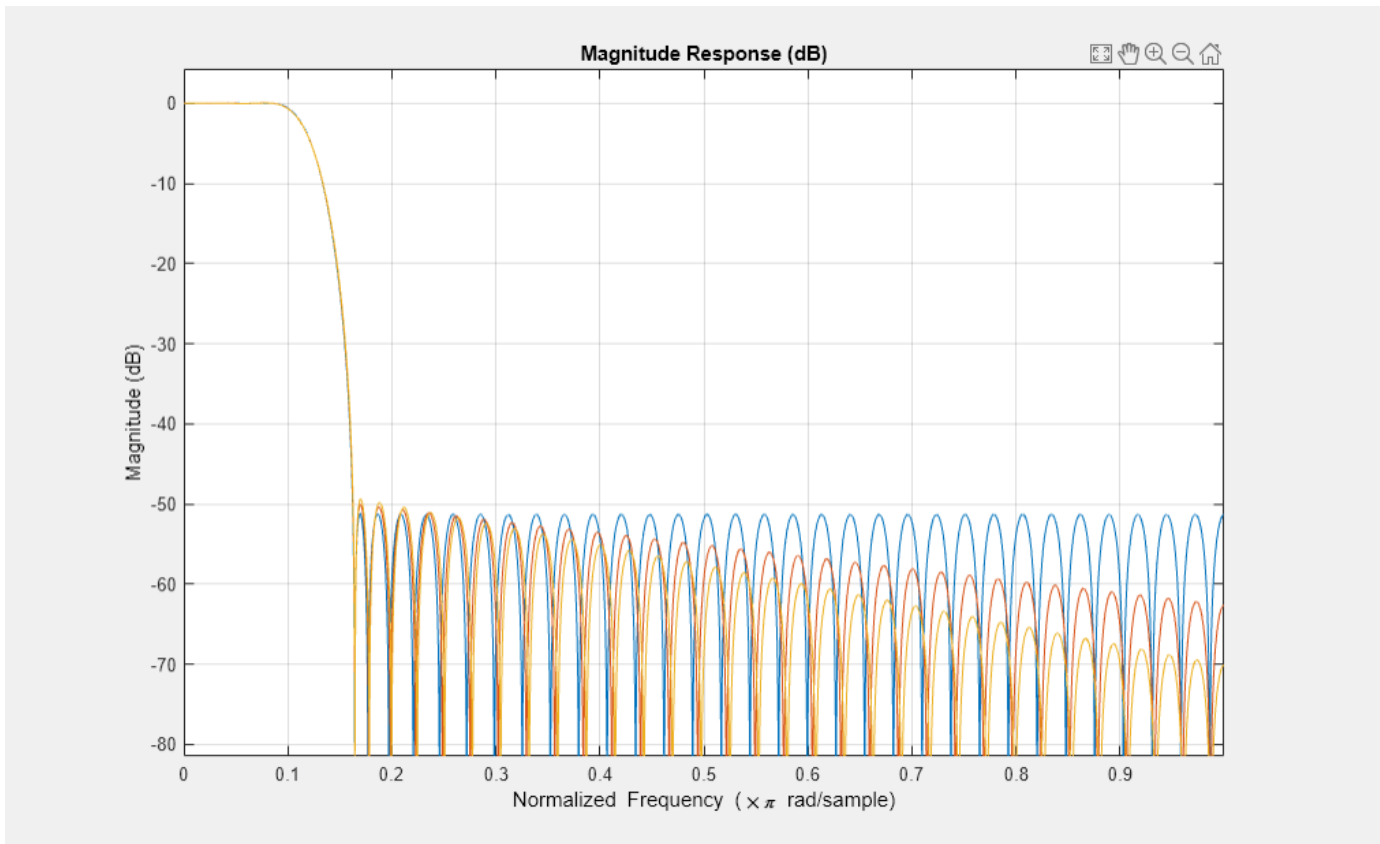
This example designs a minimum phase factor of a Nyquist filter.

```
bmin = firnyquist(47,10,.45,'minphase');
b = firnyquist(2*47,10,.45,'nonnegative');
[h,w,s] = freqz(b); hmin = freqz(bmin);
fvtool(b,1,bmin,1);
```



This example compares filters with different decay rates.

```
b1 = firnyquist(72,8,.3,0); % Equiripple  
b2 = firnyquist(72,8,.3,15);  
b3 = firnyquist(72,8,.3,25);  
fvtool(b1,1,b2,1,b3,1);
```



## Version History

Introduced in R2011a

## References

T. Saramaki, Finite Impulse Response Filter Design, *Handbook for Digital Signal Processing*, Mitra, S.K. and J.F. Kaiser Eds. Wiley-Interscience, N.Y., 1993, Chapter 4.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

All inputs must be constant. Expressions or variables are allowed if their values do not change.

## See Also

`firhalfband` | `firgr` | `firls` | `firminphase` | `rcosdesign` | `firls`

## firpr2chfb

Two-channel FIR filter bank for perfect reconstruction

### Syntax

```
[h0,h1,g0,g1] = firpr2chfb(n,fp)
[h0,h1,g0,g1] = firpr2chfb(n,dev,'dev')
[h0,h1,g0,g1] = firpr2chfb('minorder',fp,dev)
```

### Description

`[h0,h1,g0,g1] = firpr2chfb(n,fp)` designs four FIR filters for the analysis sections (`h0` and `h1`) and synthesis section is (`g0` and `g1`) of a two-channel perfect reconstruction filter bank. The design corresponds to the orthogonal filter banks also known as power-symmetric filter banks.

`n` is the order of all four filters. It must be an odd integer. `fp` is the passband-edge for the lowpass filters `h0` and `g0`. The passband-edge argument `fp` must be less than 0.5. `h1` and `g1` are highpass filters with the passband-edge given by  $(1-fp)$ .

`[h0,h1,g0,g1] = firpr2chfb(n,dev,'dev')` designs the four filters such that the maximum stopband ripple of `h0` is given by the scalar `dev`. Specify `dev` in linear units, not decibels. The stopband-ripple of `h1` is also be given by `dev`, while the maximum stopband-ripple for both `g0` and `g1` is  $(2*dev)$ .

`[h0,h1,g0,g1] = firpr2chfb('minorder',fp,dev)` designs the four filters such that `h0` meets the passband-edge specification `fp` and the stopband-ripple `dev` using minimum order filters to meet the specification.

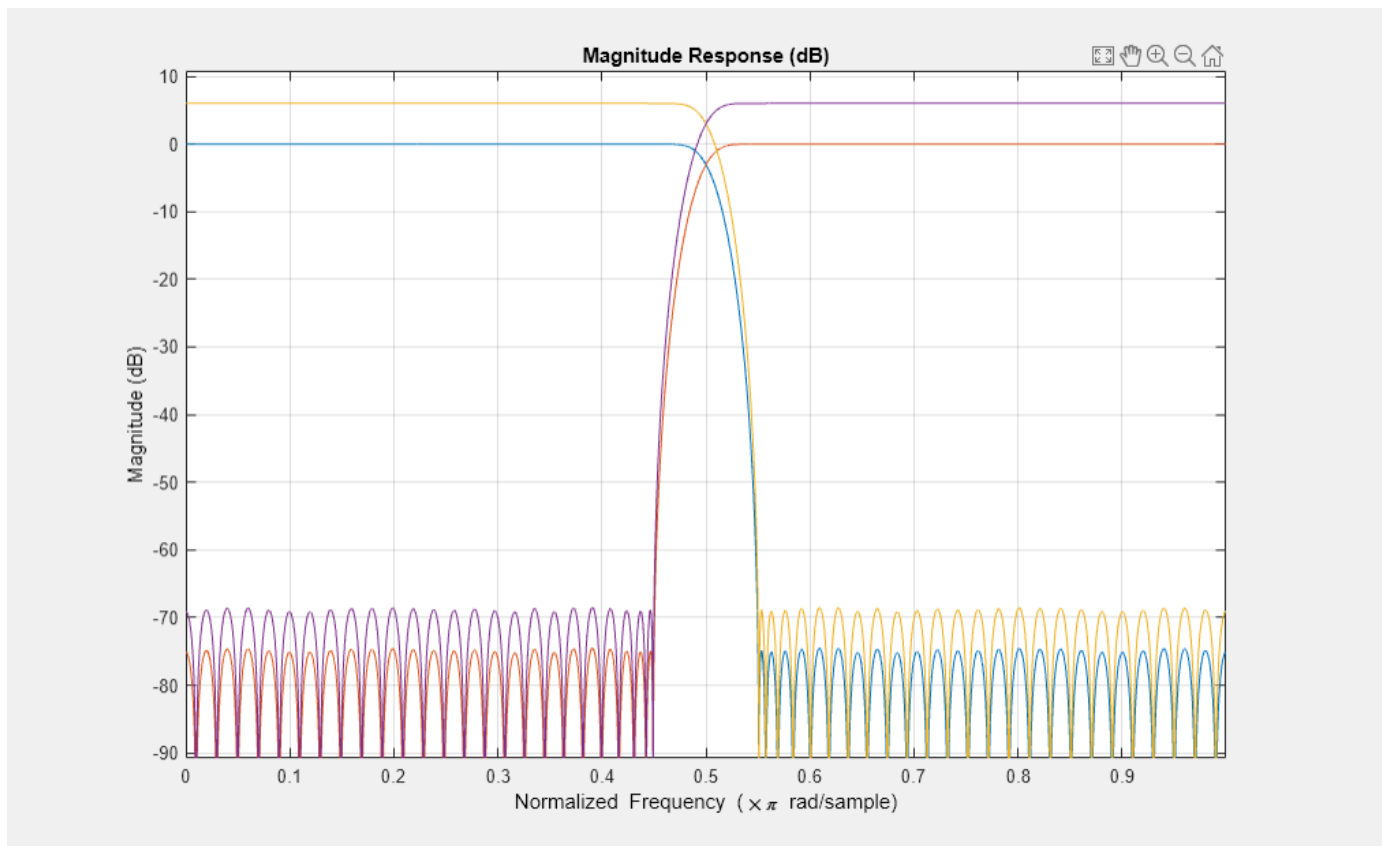
### Examples

#### Design a Filter Bank Using `firpr2chfb`

Design a filter bank with filters of order `n` equal to 99 and passband edges of 0.45 and 0.55.

```
n = 99;
[h0,h1,g0,g1] = firpr2chfb(n,.45);
fvtool(h0,1,h1,1,g0,1,g1,1);
```

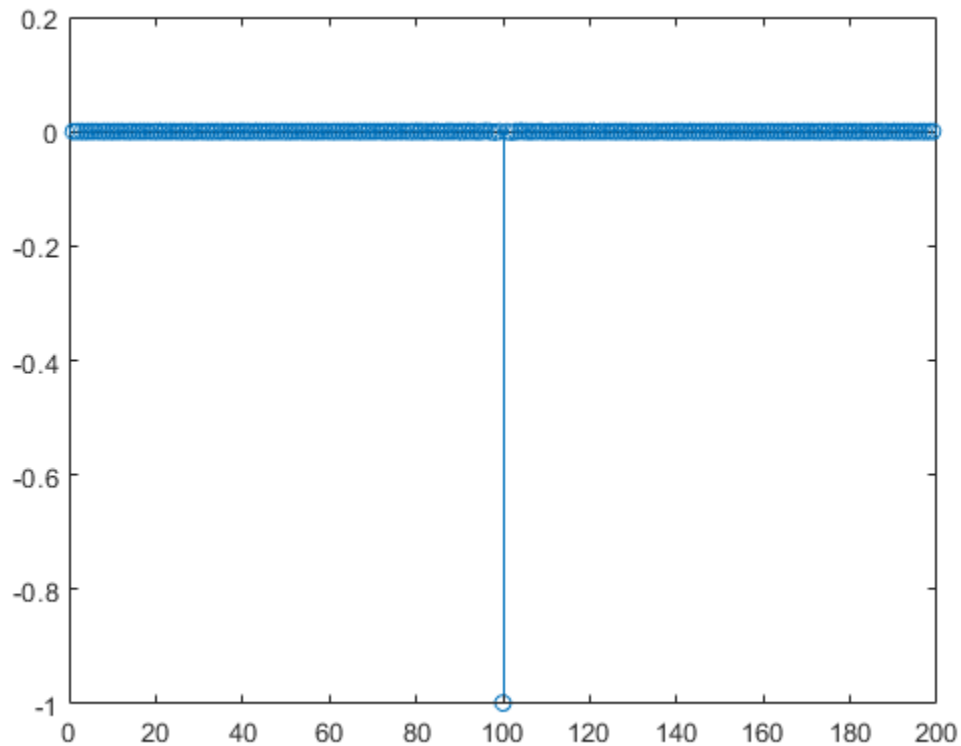




Here are the filters, showing clearly the passband edges.

Use the following stem plots to verify perfect reconstruction using the filter bank created by `firpr2chfb`.

```
stem(1/2*conv(g0,h0)+1/2*conv(g1,h1))
n=0:n;
stem(1/2*conv((-1).^n.*h0,g0)+1/2*conv((-1).^n.*h1,g1))
stem(1/2*conv((-1).^n.*g0,h0)+1/2*conv((-1).^n.*g1,h1))
stem(1/2*conv((-1).^n.*g0,(-1).^n.*h0)+...
1/2*conv((-1).^n.*g1,(-1).^n.*h1))
stem(conv((-1).^n.*h1,h0)-conv((-1).^n.*h0,h1))
```



## Version History

Introduced in R2011a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

All inputs must be constant. Expressions or variables are allowed if their values do not change.

## See Also

`firceqrip` | `firgr` | `firhalfband` | `firnyquist`

# firtype

**Package:** dsp

Determine type (1-4) of linear phase FIR filter System object

## Syntax

```
type = firtype(sysobj)
```

## Description

`type = firtype(sysobj)` determines the type (1 through 4) of the FIR filter System object

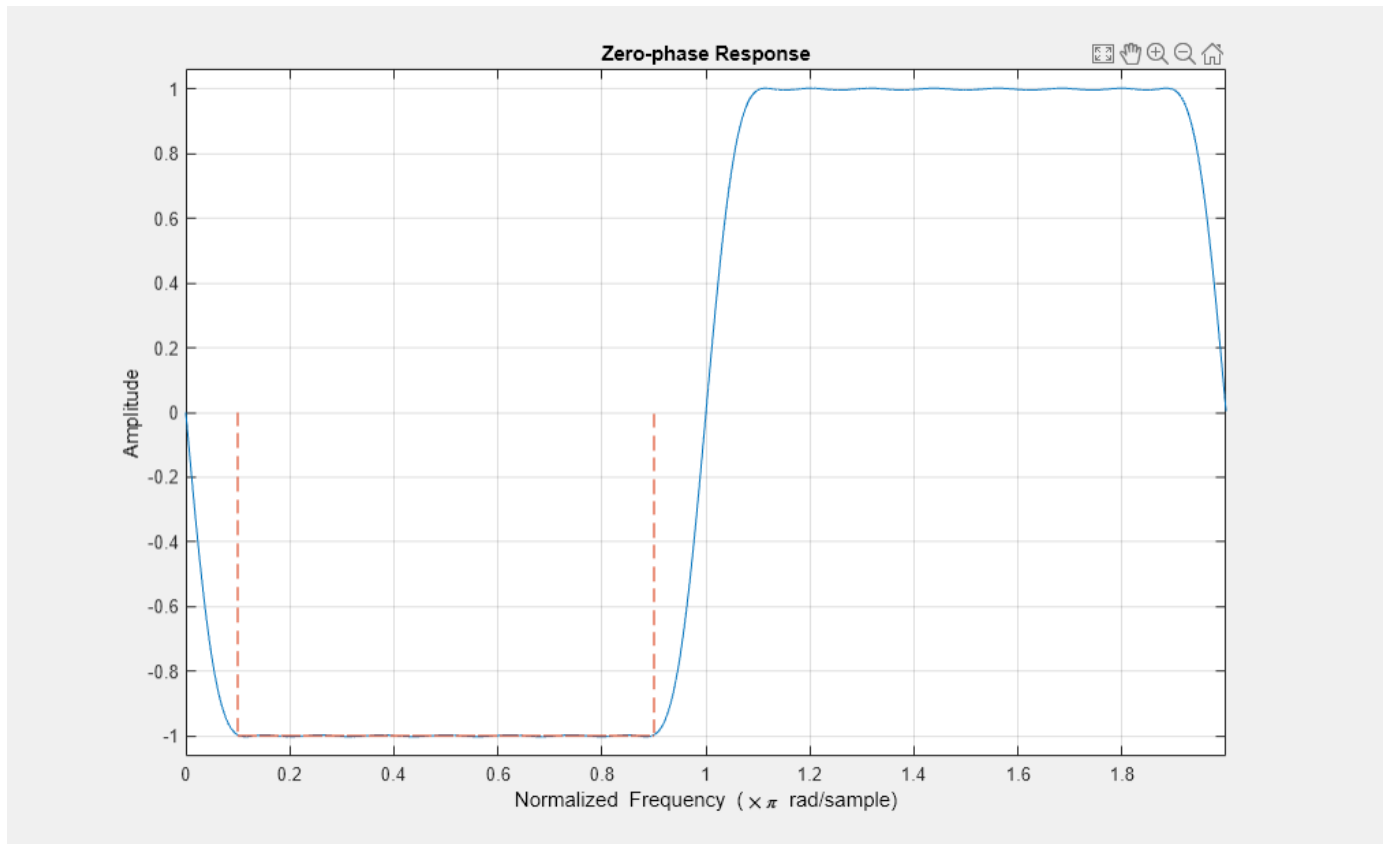
For more input options, see `firtype` in Signal Processing Toolbox.

## Examples

### Hilbert Transformers

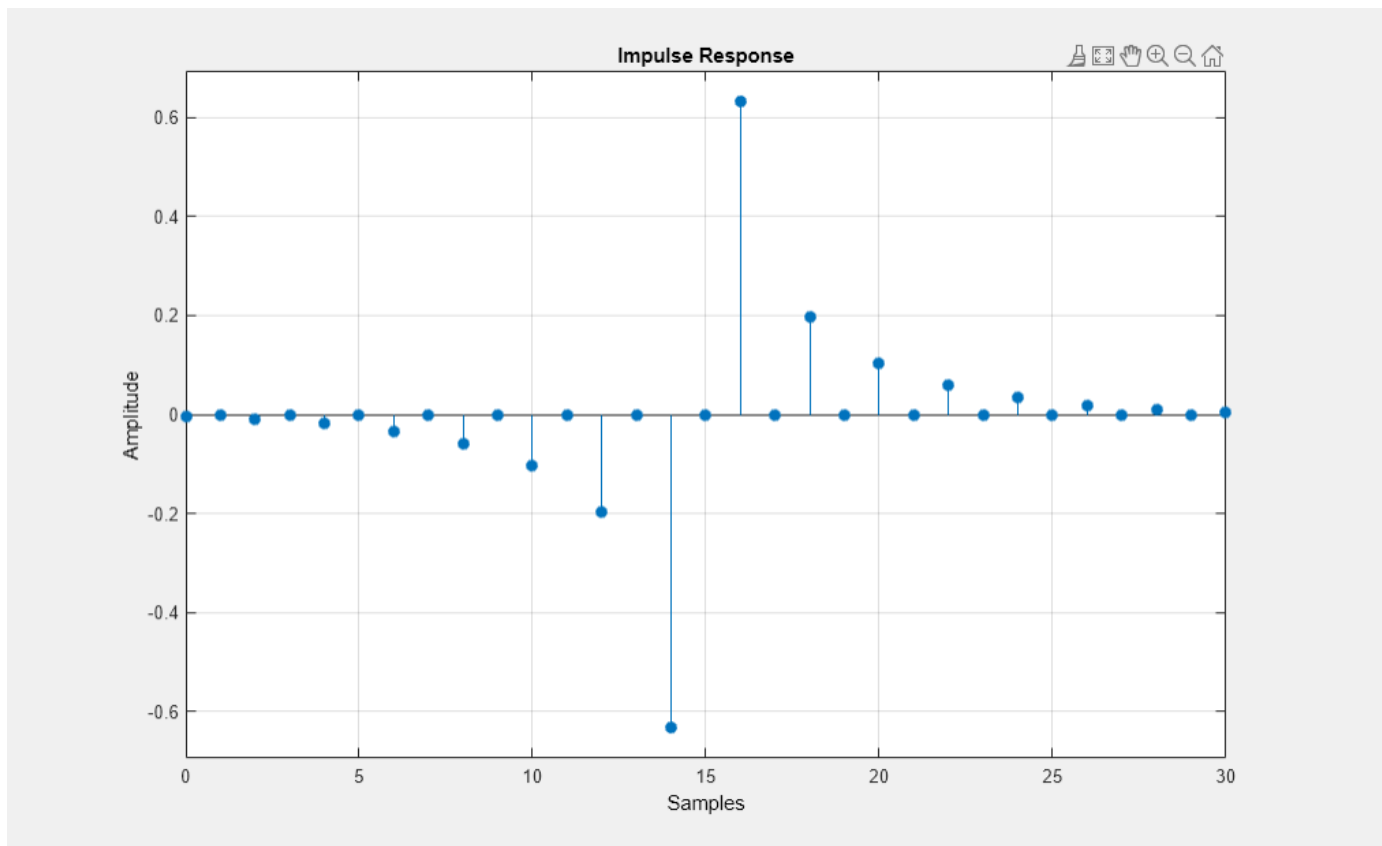
Design a Hilbert transformer of order 30 with a transition width of  $0.2\pi$  rad/sample. Use least-squares minimization to obtain an equiripple linear-phase FIR filter. Plot the zero-phase response in the interval  $[-\pi, \pi)$ .

```
d = fdesign.hilbert('N,TW',30,0.2);  
Hd = design(d,'equiripple','SystemObject',true);  
zerophase(Hd,'whole')
```



The impulse response of this even-order type-3 filter is antisymmetric.

`impz(Hd)`

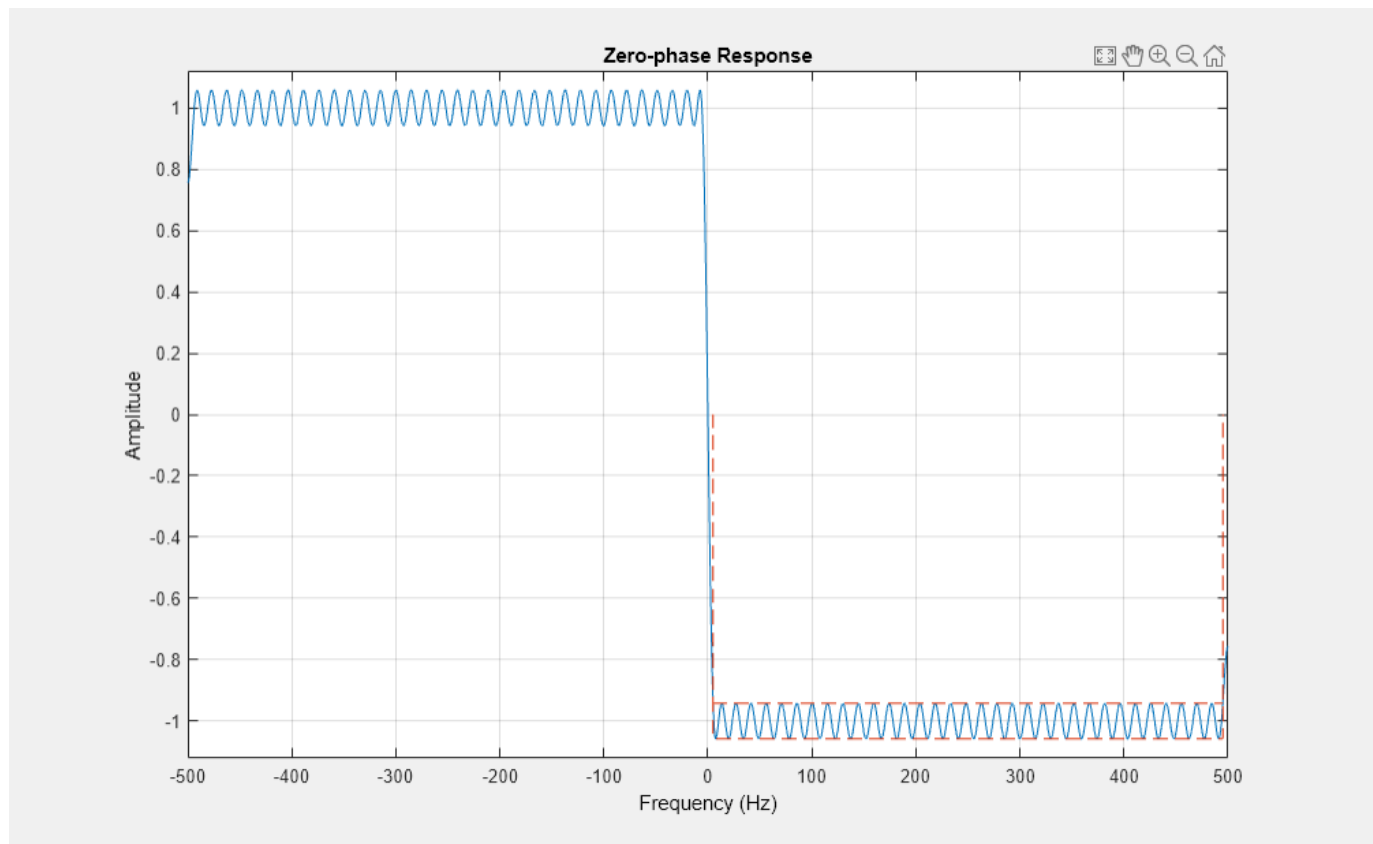


```
ftype = firtype(Hd)
```

```
ftype = 3
```

Design a minimum-order Hilbert transformer that has a sample rate of 1 kHz. Specify the width of the transition region as 10 Hz and the passband ripple as 1 dB. Display the zero-phase response of the filter.

```
fs = 1e3;
d = fdesign.hilbert('TW,Ap',10,1,fs);
hd = design(d,'equiripple','SystemObject',true);
zerophase(hd,-fs/2:0.1:fs/2,fs)
```



## Input Arguments

### **sysobj** — Input FIR filter

FIR filter System object

Input FIR filter with real and linear phase, specified as one of the following filter System objects:

- `dsp.CICDecimator`
- `dsp.CICInterpolator`
- `dsp.CICCompensationDecimator`
- `dsp.CICCompensationInterpolator`
- `dsp.Differentiator`
- `dsp.HighpassFilter`
- `dsp.FilterCascade`
- `dsp.FIRFilter`
- `dsp.FIRDecimator`
- `dsp.FIRInterpolator`
- `dsp.FIRRateConverter`
- `dsp.FIRHalfbandDecimator`

- `dsp.FIRHalfbandInterpolator`
- `dsp.LowpassFilter`
- `dsp.VariableBandwidthFIRFilter`

To check if a filter has linear phase, use the `islinphase` function. To check if a filter has real coefficients, use the `isreal` function.

## Output Arguments

**type** — FIR filter type

1 | 2 | 3 | 4

FIR filter type, defined as one of the following:

- 1 -- Type 1 filter with even order symmetric coefficients.
- 2 -- Type 2 filter with odd order symmetric coefficients.
- 3 -- Type 3 filter with even order antisymmetric coefficients.
- 4 -- Type 4 filter with odd order antisymmetric coefficients.

## Version History

Introduced in R2013a

## See Also

`firtype` | `islinphase` | `isreal`

# freqrespest

**Package:** dsp

Frequency response estimate via filtering

## Syntax

```
[h,w] = freqrespest(sysobj)
[h,w] = freqrespest(sysobj,L)
[h,w] = freqrespest(sysobj,L,Name,Value)
[h,w] = freqrespest(sysobj,L,opts)
[h,w] = freqrespest(sysobj,'Arithmetic',arithType)
freqrespest(sysobj, ___)
```

## Description

`[h,w] = freqrespest(sysobj)` computes the frequency response estimate of the input filter System object. The function estimates by running input data made up from sinusoids with uniformly distributed random frequencies through the filter and forming the ratio between output data and input data.

The `freqrespest` function computes the frequency response estimate of the input filter object that is close to the frequency response obtained by using only the quantized coefficients as long as:

- No overflow occurs when performing the fixed-point filtering
- There isn't significant quantization happening in the fixed-point additions and multiplications

`freqrespest` can be used as a diagnostic tool for fixed-point implementations. If `freqrespest` differs significantly from `freqz` (which only takes into account the quantization of the coefficients), it is an indication that at least one of the two conditions might be true.

`[h,w] = freqrespest(sysobj,L)` computes the frequency response estimate of the filter object by specifying `L` number of trials.

`[h,w] = freqrespest(sysobj,L,Name,Value)` uses the `Name,Value` pairs as input arguments to specify optional parameters for the test. Enter the parameter name as an input argument in single quotation marks followed by the possible value of the parameter.

`[h,w] = freqrespest(sysobj,L,opts)` uses an options object to specify the optional input parameters in lieu of specifying name-value pairs.

`[h,w] = freqrespest(sysobj,'Arithmetic',arithType)` analyzes the filter System object based on the arithmetic specified in the `arithType` input.

`freqrespest(sysobj, ___)` with no output argument launches FVTool and shows the magnitude response estimate of the filter object.

## Examples



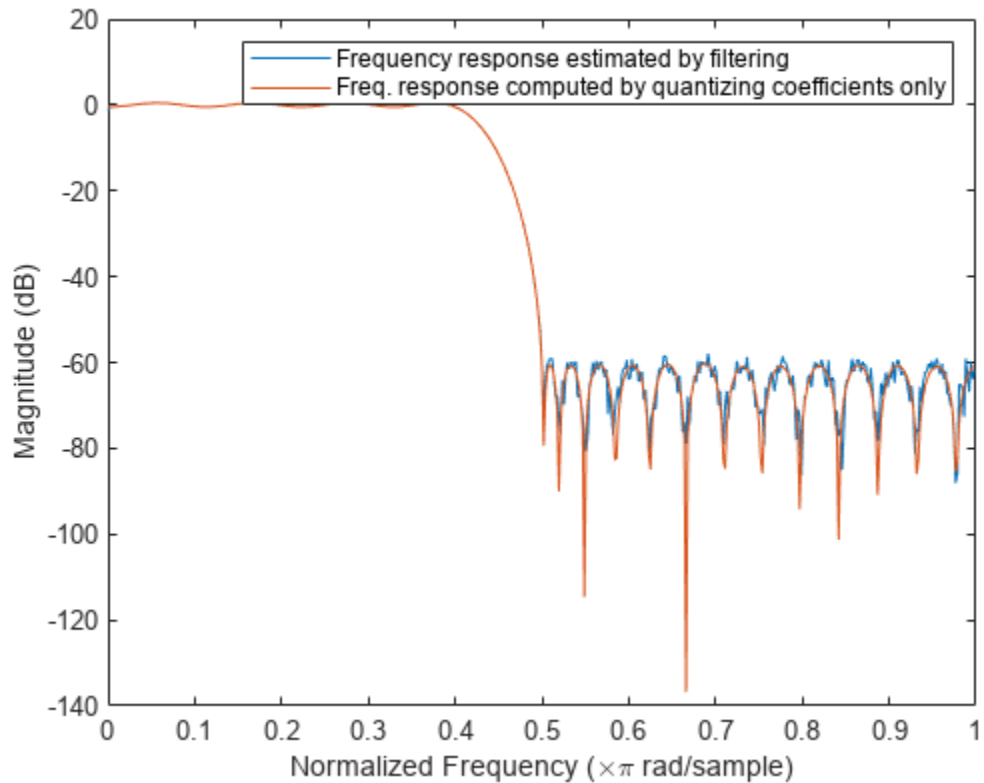
## Estimate the Frequency Response of Fixed-Point FIR Filter

Estimate the frequency response of a fixed-point FIR filter.

```
firFilt = design(fdesign.lowpass(.4,.5,1,60),'equiripple','Systemobject',true);
dataIn = fi(randn(16,15),1,16,15);
dataOut = firFilt(dataIn); %#ok
[h,w] = freqrespest(firFilt); %#ok % This should be about the same as freqz.
release(firFilt);
```

Continuing with the filter object `firFilt`, change the `FullPrecisionOverride` property to `false` and then specify the word lengths and precision (the fraction lengths) applied to the output from internal addition and multiplication operations. After you set the word and fraction lengths, use the `freqrespest` function to compute the frequency response estimate for the fixed-point filter.

```
firFilt.FullPrecisionOverride = false;
firFilt.ProductDataType = 'Custom';
firFilt.CustomProductDataType = numerictype(1,16,15);
firFilt.AccumulatorDataType = 'Custom';
firFilt.CustomAccumulatorDataType = numerictype(1,16,15);
firFilt.OutputDataType = 'Same as accumulator';
dataOut = firFilt(dataIn);
[h,w] = freqrespest(firFilt,2);
[h2,w2] = freqz(firFilt,512);
plot(w/pi,20*log10(abs([h,h2])))
legend('Frequency response estimated by filtering',...
'Freq. response computed by quantizing coefficients only');
xlabel('Normalized Frequency (\times\pi rad/sample)')
ylabel('Magnitude (dB)')
```



## Input Arguments

### **sysobj** — Input filter

filter System object

Input filter, specified as one of the following filter System objects:

- `dsp.AllpassFilter`
- `dsp.AllpoleFilter`
- `dsp.BiquadFilter`
- `dsp.CoupledAllpassFilter`
- `dsp.Differentiator`
- `dsp.FIRFilter`
- `dsp.FourthOrderSectionFilter`
- `dsp.HighpassFilter`
- `dsp.IIRFilter`
- `dsp.LowpassFilter`
- `dsp.NotchPeakFilter`
- `dsp.SOSFilter`

- `dsp.VariableBandwidthFIRFilter`
- `dsp.VariableBandwidthIIRFilter`

### **L — Number of trials**

10 | positive integer

Number of trials used to compute the estimate, specified as a positive integer. If you do not specify this value, L defaults to 10. More the number of trials, the greater is the accuracy in the estimate of the response. However, when L is large, the function requires more time to compute the estimate.

Data Types: `single` | `double`

### **opts — Options object**

options object

Options object to specify the optional input parameters in lieu of specifying name-value pairs.

Create the `opts` object using the `freqrespopts` function.

```
opts = freqrespopts(sysobj);
```

Because `opts` is an object, you use the `set` function to change the property values in `opts` before you use it with `freqrespest`. For example, you could specify a new sample rate with:

```
set(opts, 'fs', 48e3); % Same as opts.fs = 48e3
```

### **arithType — Arithmetic type**

'double' | 'single' | 'fixed'

Analyze the filter System object, based on the arithmetic specified in the `arithType` input. `arithType` can be set to 'double', 'single', or 'fixed'. The analysis tool assumes a double-precision filter when the arithmetic input is not specified and the filter System object is in an unlocked state.

`freqrespest` requires knowledge of the input data type. Analysis cannot be performed if the input data type is not available. If you do not specify the `arithType` parameter, that is, use the syntax `[h,w] = freqrespest(sysobj)`, then these rules apply.

- The System object state is `Unlocked` — `freqrespest` performs double precision analysis.
- The System object state is `Locked` — `freqrespest` performs analysis based on the locked input data type.

When you do specify the `arithType` parameter, that is, use the syntax `[h,w] = freqrespest(sysobj, 'Arithmetic', arithType)`, the following rules apply:

| Value                             | System Object State   | Rule   |
|-----------------------------------|-----------------------|--|
| <code>arithType = 'double'</code> | <code>Unlocked</code> | <code>freqrespest</code> performs double-precision analysis. |
|                                   | <code>Locked</code>   | <code>freqrespest</code> performs double-precision analysis. |
| <code>arithType = 'single'</code> | <code>Unlocked</code> | <code>freqrespest</code> performs single-precision analysis. |

| Value                            | System Object State | Rule  |
|----------------------------------|---------------------|---|
|                                  | Locked              | <code>freqrespest</code> performs single-precision analysis.  |
| <code>arithType = 'fixed'</code> | Unlocked            | <code>freqrespest</code> produces an error because the fixed-point input data type is unknown.  |
|                                  | Locked              | If the input data type is double or single, then <code>freqrespest</code> produces an error because the fixed-point input data type is unknown. |
|                                  |                     | When the input data is of fixed-point type, <code>freqrespest</code> performs analysis based on the locked input data type.                     |

### Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

```
Example: firFilt =
design(fdesign.lowpass(.4,.5,1,60),'equiripple','Systemobject',true); [h,w] =
freqrespest(firFilt,10,'NFFT',1024);
```

### NFFT — Number of FFT points

512 (default) | positive integer

Number of FFT points used in computing the frequency response estimate, specified as a positive integer. This value determines the length of the frequency response vector `h` and the length of the frequency vector `w`. When not specified, `NFFT` defaults to 512.

Data Types: `single` | `double`

### NormalizedFrequency — Use normalized frequency

`true` (default) | `false`

Indicate whether to use normalized frequency or linear frequency, specified as either:

- `true` -- Use normalized frequency. When not specified, the function defaults to `true`.
- `false` -- Use linear frequency. When you specify `false`, you must supply the sampling frequency `Fs`.

### Fs — Sampling frequency

positive scalar

Sampling frequency to be specified when `NormalizedFrequency` is set to `false`. No default value. You must set `NormalizedFrequency` to `false` before setting a value for `Fs`.

Data Types: `single` | `double`

**SpectrumRange — Spectrum range**`half (default) | whole`

Spectrum range to be used while computing the frequency response estimate, specified as either:

- `half`
- `whole`

**CenterDC — Set center of spectrum to DC**`false (default) | true`

Specify whether to set the center of the spectrum to the DC value in the output plot. If you select `true`, both the negative and positive values appear in the plot. If you select `false`, DC appears at the origin of the axes.

**Output Arguments****h — Estimate of complex frequency response**

vector

Estimate of the complex frequency response, returned as a vector. The length of the vector equals the NFFT value. By default, this vector is of length 512.

Data Types: `double`

Complex Number Support: Yes

**w — Frequencies**

vector

Frequencies at which the complex frequency response `h` is estimated, returned as a vector. The length of the vector equals the NFFT value. By default, this vector is of length 512.

Data Types: `double`**Version History**

Introduced in R2011a

**See Also****Functions**`freqrespopts` | `freqz` | `noisepsd` | `scale`

## freqrespopts

**Package:** dsp

Create an options object for frequency response estimate

### Syntax

```
opts = freqrespopts(sysobj)
```

### Description

`opts = freqrespopts(sysobj)` uses the current settings in the filter System object to create an options object. This object contains options for frequency response estimation. You pass the `opts` object as an input argument to the `freqrespest` function to specify values for the input parameters.

`freqrespopts` allows you to use the same settings for `freqrespest` with multiple filters without specifying all of the parameters as input arguments to `freqrespest`.

### Examples

#### Set Frequency Response Options

This example uses the `freqrespopts` function to set options which are used by the `freqrespest` function. `hd` and `hd2` are bandpass filters that use different design methods. The `opts` object makes it easier to set the same conditions for the frequency response estimate when using the `freqrespest` function with different filter objects..

```
d = fdesign.bandpass('fst1,fp1,fp2,fst2,ast1,ap,ast2',...
0.25,0.3,0.45,0.5,60,0.1,60);
hd = design(d,'butter','SystemObject',true)
```

```
hd =
```

```
  dsp.SOSFilter with properties:
```

```
      Structure: 'Direct form II'
CoefficientSource: 'Property'
      Numerator: [18x3 double]
      Denominator: [18x3 double]
HasScaleValues: true
      ScaleValues: [0.2535 0.2535 0.2437 0.2437 0.2353 0.2353 0.2283 0.2283 0.2226 0.2226 0.2163 0.2163 0.2103 0.2103 0.2043 0.2043 0.1983 0.1983 0.1923 0.1923 0.1863 0.1863 0.1803 0.1803 0.1743 0.1743 0.1683 0.1683 0.1623 0.1623 0.1563 0.1563 0.1503 0.1503 0.1443 0.1443 0.1383 0.1383 0.1323 0.1323 0.1263 0.1263 0.1203 0.1203 0.1143 0.1143 0.1083 0.1083 0.1023 0.1023 0.0963 0.0963 0.0903 0.0903 0.0843 0.0843 0.0783 0.0783 0.0723 0.0723 0.0663 0.0663 0.0603 0.0603 0.0543 0.0543 0.0483 0.0483 0.0423 0.0423 0.0363 0.0363 0.0303 0.0303 0.0243 0.0243 0.0183 0.0183 0.0123 0.0123 0.0063 0.0063 0.0003 0.0003]
```

```
Show all properties
```

```
hd2 = design(d,'cheby2','SystemObject',true)
```

```
hd2 =
```

```
  dsp.SOSFilter with properties:
```

```
      Structure: 'Direct form II'
CoefficientSource: 'Property'
```

```

        Numerator: [9x3 double]
        Denominator: [9x3 double]
        HasScaleValues: true
        ScaleValues: [0.7226 0.7226 0.6446 0.6446 0.5277 0.5277 0.3810 0.3810 0.2932 1]

```

Show all properties

```
opts = freqrespopts(hd)
```

```
opts =
```

```
struct with fields:
```

```

        FreqPoints: 'All'
            NFFT: 512
    NormalizedFrequency: true
            Fs: 'Normalized'
    SpectrumRange: 'Half'
        CenterDC: false

```

```
opts.NFFT = 256; % Same as set(opts,'nfft',256).
```

```
opts.NormalizedFrequency = false;
```

```
opts.fs = 1.5e3;
```

```
opts.CenterDC = true
```

```
opts =
```

```
struct with fields:
```

```

        FreqPoints: 'All'
            NFFT: 256
    NormalizedFrequency: false
            Fs: 1500
    SpectrumRange: 'Whole'
        CenterDC: true

```

With the `opts` object configured as needed, use it as an input argument for the `freqrespest` function.

```
[h2,w2] = freqrespest(hd2,20,opts);
```

```
[h1,w1] = freqrespest(hd,20,opts);
```

## Input Arguments

### **sysobj** — Input filter

filter System object

Input filter, specified as one of the following filter System objects:

- `dsp.AllpassFilter`
- `dsp.AllpoleFilter`
- `dsp.BiquadFilter`
- `dsp.CoupledAllpassFilter`

- `dsp.Differentiator`
- `dsp.FIRFilter`
- `dsp.FourthOrderSectionFilter`
- `dsp.HighpassFilter`
- `dsp.IIRFilter`
- `dsp.LowpassFilter`
- `dsp.NotchPeakFilter`
- `dsp.SOSFilter`
- `dsp.VariableBandwidthFIRFilter`
- `dsp.VariableBandwidthIIRFilter`

## Output Arguments

### **opts** — Options object

`pseudospectrum` object

Options object which contains the options for frequency response estimation, returned as a `pseudospectrum` object. You pass the `opts` object as an input argument to the `freqrespest` function to specify values for input parameters.

## Version History

**Introduced in R2011a**

### **See Also**

`freqrespest` | `noisepsd` | `noisepsdopts` | `scale`



# freqsamp

Real or complex frequency-sampled FIR filter from specification object

## Syntax

```
hd = design(d,'freqsamp','SystemObject',true)
hd = design(...,'filterstructure',structure,'SystemObject',true)
hd = design(...,'window',window,'SystemObject',true)
```

## Description

`hd = design(d,'freqsamp','SystemObject',true)` designs a frequency-sampled filter specified by the filter specifications object `d`.

`hd = design(...,'filterstructure',structure,'SystemObject',true)` returns a filter with the filter structure you specify by the `structure` input argument. `structure` is `dffir` by default and can be any one of the following filter structures.

| Structure              | Description of Resulting Filter Structure |
|------------------------|---|
| <code>dffir</code>     | Direct-form FIR filter                    |
| <code>dffirt</code>    | Transposed direct-form FIR filter         |
| <code>dfsymfir</code>  | Symmetrical direct-form FIR filter        |
| <code>dfasymfir</code> | Asymmetrical direct-form FIR filter       |

`hd = design(...,'window',window,'SystemObject',true)` designs filters using the window specified by `window`. Provide the input argument `window` as

- A character vector for the window type. For example, use `'bartlett'`, or `'hamming'`. See `window` for the full list of windows available.
- A function handle that references the window function. When the window function requires more than one input, use a cell array to hold the required arguments. The first example shows a cell array input argument.
- The window vector itself.

## Examples

### Design a Frequency Sampled FIR Filter

These examples design FIR filters that have arbitrary magnitude responses. In the first filter, the response has three distinct sections and the resulting filter is real.

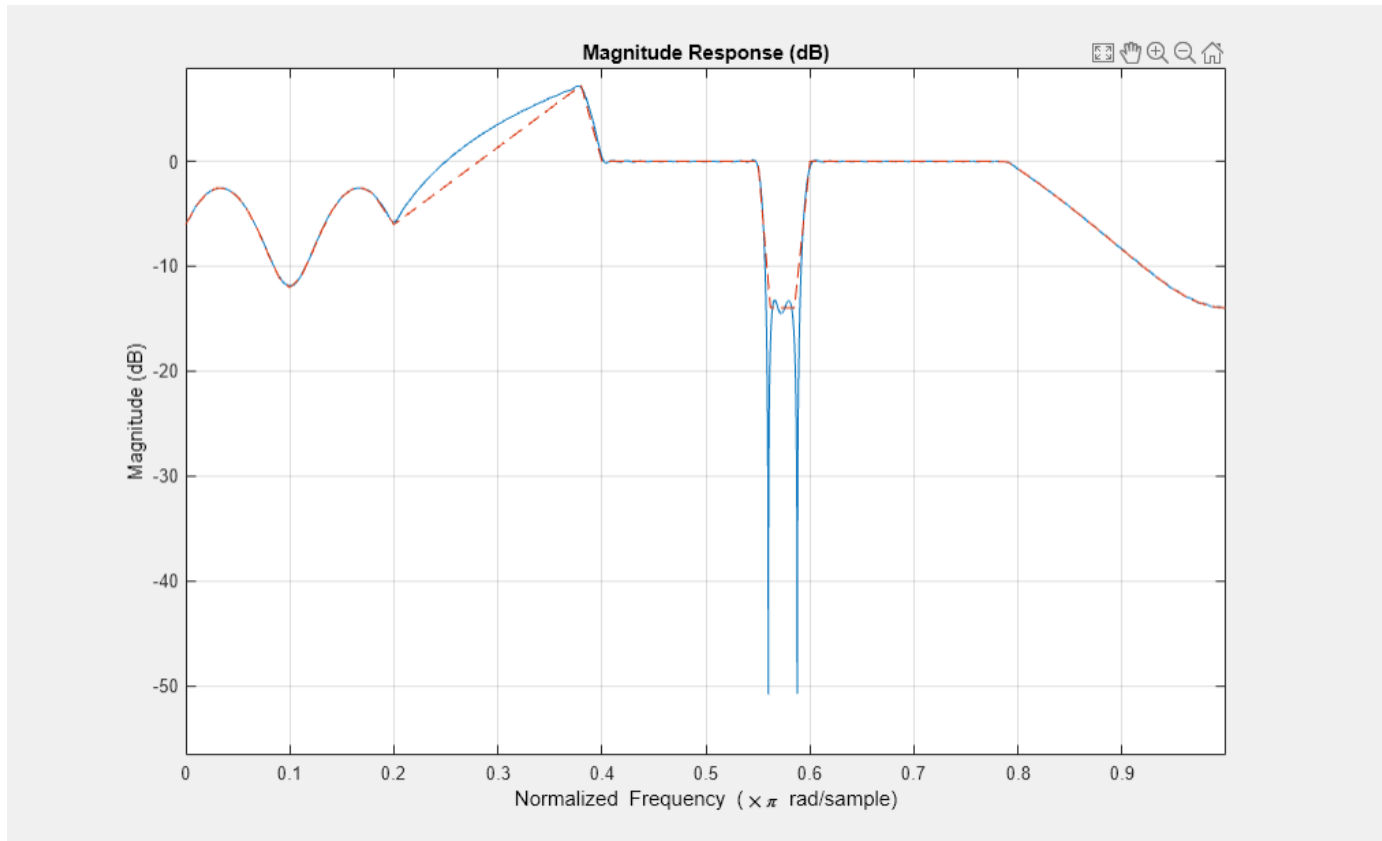
The second example creates a complex filter.

```
b1 = 0:0.01:0.18;
b2 = [.2 .38 .4 .55 .562 .585 .6 .78];
b3 = 0.79:0.01:1;
```

```

a1 = .5+sin(2*pi*7.5*b1)/4; % Sinusoidal response section.
a2 = [.5 2.3 1 1 -.2 -.2 1 1]; % Piecewise linear response section.
a3 = .2+18*(1-b3).^2; % Quadratic response section.
f = [b1 b2 b3];
a = [a1 a2 a3];
n = 300;
d = fdesign.arbmag('n,f,a',n,f,a); % First specifications object.
hd = design(d,'freqsamp','window',{@kaiser,.5},...
'SystemObject',true); % Filter.
fvtool(hd)

```

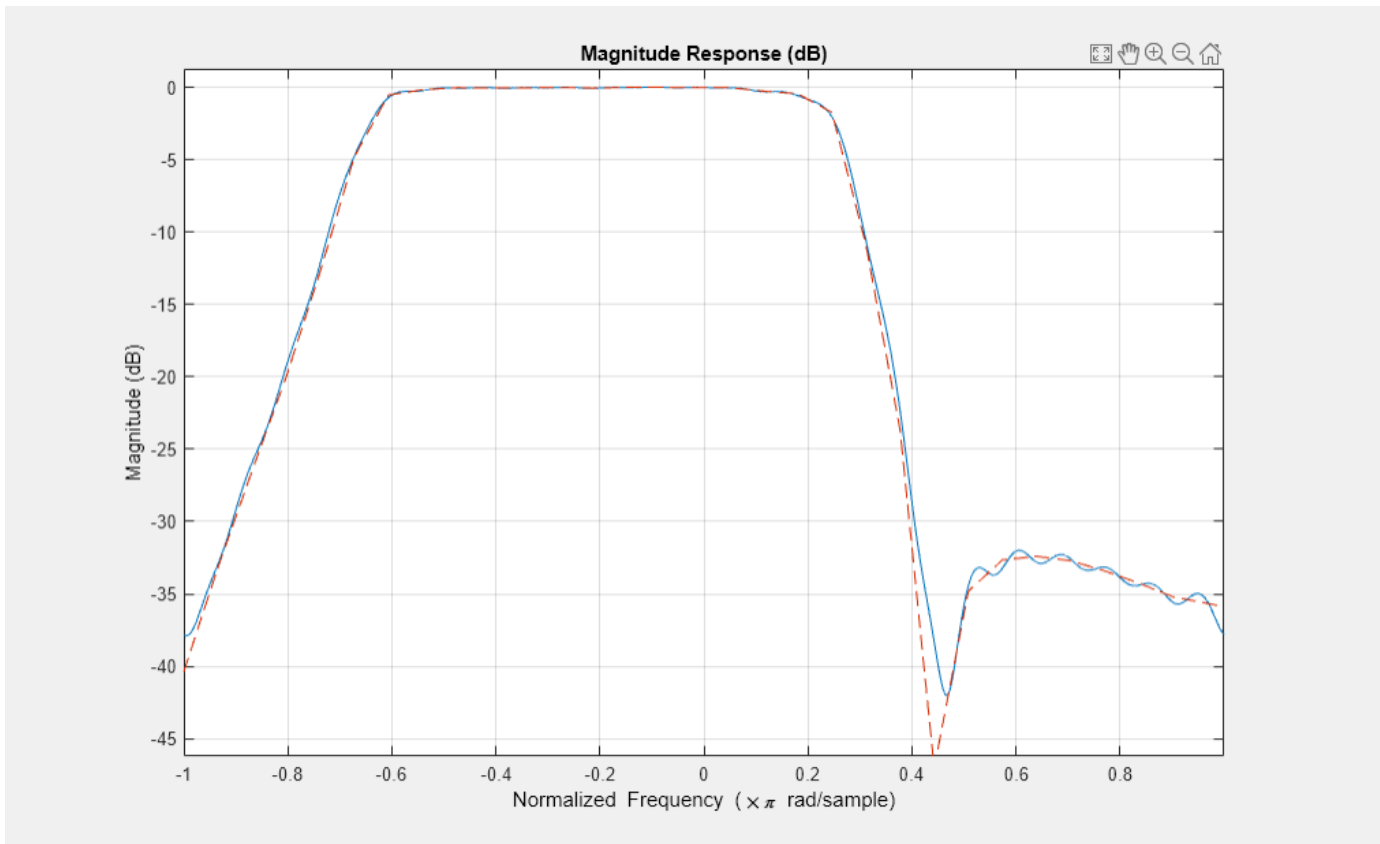


Now design the arbitrary-magnitude complex FIR filter. Recall that vector  $f$  contains frequency locations and vector  $a$  contains the desired filter response values at the locations specified in  $f$ .

```

f = [-1 -.93443 -.86885 -.80328 -.7377 -.67213 -.60656 -.54098 ...
-.47541,-.40984 -.34426 -.27869 -.21311 -.14754 -.081967 ...
-.016393 .04918 .11475,.18033 .2459 .31148 .37705 .44262 ...
.5082 .57377 .63934 .70492 .77049,.83607 .90164 1];
a = [.0095848 .021972 .047249 .099869 .23119 .57569 .94032 ...
.98084 .99707,.99565 .9958 .99899 .99402 .99978 .99995 .99733 ...
.99731 .96979 .94936,.8196 .28502 .065469 .0044517 .018164 ...
.023305 .02397 .023141 .021341,.019364 .017379 .016061];
n = 48;
d = fdesign.arbmag('n,f,a',n,f,a); % Second spec. object.
hdc = design(d,'freqsamp','window','rectwin','SystemObject',true); % Filter.
fvtool(hdc)

```



fvtool shows you the response for hdc from -1 to 1 in normalized frequency. `design(d,...)` returns a complex filter for hdc because the frequency vector includes negative frequency values.

## Version History

Introduced in R2011a

### See Also

[design](#) | [designmethods](#) | [fdesign.arbmag](#) | [help](#) | [window](#)

# freqz

**Package:** dsp

Frequency response of discrete-time filter System object

## Syntax

```
[h,w] = freqz(sysobj)
[h,w] = freqz(sysobj,n)
[h,w] = freqz(sysobj,'Arithmetic',arithType)
freqz(sysobj)
```

## Description

`[h,w] = freqz(sysobj)` returns the complex frequency response  $h$  of the filter System object, `sysobj`. The vector  $w$  contains the frequencies (in radians/sample) at which the function evaluates the frequency response. The frequency response is evaluated at 8192 points equally spaced around the upper half of the unit circle.

`[h,w] = freqz(sysobj,n)` returns the complex frequency response of the filter System object and the corresponding frequencies at  $n$  points equally spaced around the upper half of the unit circle.

`freqz` uses the transfer function associated with the filter to calculate the frequency response of the filter with the current coefficient values.

`[h,w] = freqz(sysobj,'Arithmetic',arithType)` analyzes the filter System object, based on the arithmetic specified in `arithType`, using either of the previous syntaxes.

`freqz(sysobj)` uses `fvtool` to plot the magnitude and unwrapped phase of the frequency response of the filter System object `sysobj`.

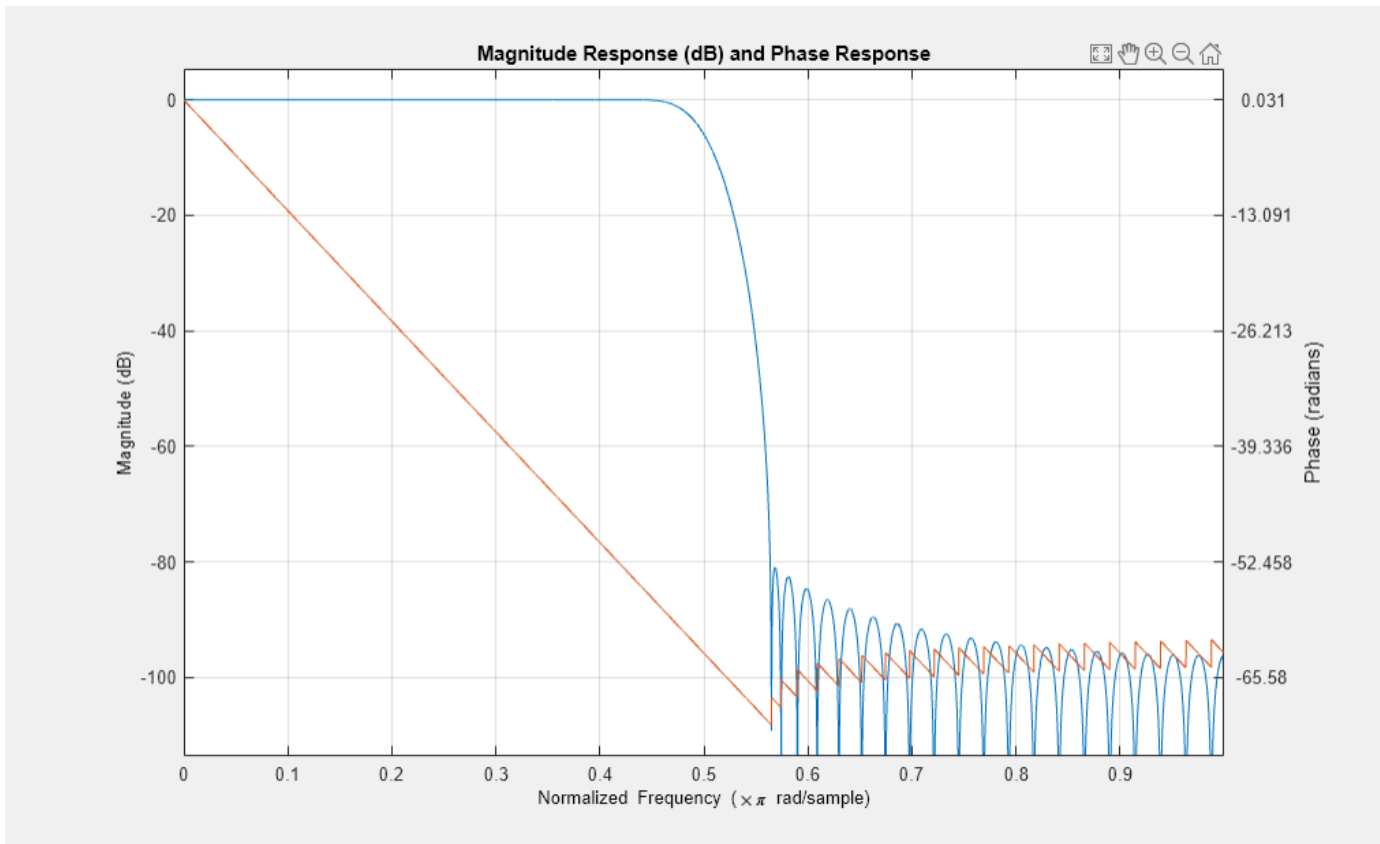
For more input options, see `freqz` in Signal Processing Toolbox.

## Examples

### Frequency Response of the Filter

This examples plot the frequency response of the lowpass FIR filter using `freqz`.

```
b = fir1(80,0.5,kaiser(81,8));
firFilt = dsp.FIRFilter('Numerator',b);
freqz(firFilt);
```



## Input Arguments

### sysobj — Input filter

filter System object

Input filter, specified as one of the following filter System objects:

- `dsp.AllpassFilter`
- `dsp.AllpoleFilter`
- `dsp.BiquadFilter`
- `dsp.CICCompensationDecimator`
- `dsp.CICCompensationInterpolator`
- `dsp.CICDecimator`
- `dsp.CICInterpolator`
- `dsp.CoupledAllpassFilter`
- `dsp.Differentiator`
- `dsp.FarrowRateConverter`
- `dsp.FilterCascade`
- `dsp.FIRDecimator`

- `dsp.FIRFilter`
- `dsp.FIRHalfbandDecimator`
- `dsp.FIRHalfbandInterpolator`
- `dsp.FIRInterpolator`
- `dsp.FIRRateConverter`
- `dsp.FourthOrderSectionFilter`
- `dsp.HighpassFilter`
- `dsp.IIRFilter`
- `dsp.IIRHalfbandDecimator`
- `dsp.IIRHalfbandInterpolator`
- `dsp.LowpassFilter`
- `dsp.NotchPeakFilter`
- `dsp.SOSFilter`
- `dsp.VariableBandwidthFIRFilter`
- `dsp.VariableBandwidthIIRFilter`

**n — Number of points over which the frequency response is computed**

8192 (default) | positive integer

Number of points over which the frequency response is computed. For an FIR filter where *n* is a power of two, the computation is done faster using FFTs.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**arithType — Arithmetic type**

'double' (default) | 'single' | 'Fixed'

Arithmetic used in the filter analysis, specified as 'double', 'single', or 'Fixed'. When the arithmetic input is not specified and the filter System object is unlocked, the analysis tool assumes a double-precision filter. When the arithmetic input is not specified and the System object is locked, the function performs the analysis based on the data type of the locked input.

The 'Fixed' value applies to filter System objects with fixed-point properties only.

When the 'Arithmetic' input argument is specified as 'Fixed' and the filter object has the data type of the coefficients set to 'Same word length as input', the arithmetic analysis depends on whether the System object is unlocked or locked.

- **unlocked** -- The analysis object function cannot determine the coefficients data type. The function assumes that the coefficients data type is signed, has a 16-bit word length, and is auto scaled. The function performs fixed-point analysis based on this assumption.
- **locked** -- When the input data type is 'double' or 'single', the analysis object function cannot determine the coefficients data type. The function assumes that the data type of the coefficients is signed, has a 16-bit word length, and is auto scaled. The function performs fixed-point analysis based on this assumption.

To check if the System object is locked or unlocked, use the `isLocked` function.

When the arithmetic input is specified as 'Fixed' and the filter object has the data type of the coefficients set to a custom numeric type, the object function performs fixed-point analysis based on the custom numeric data type.

## Output Arguments

### **h** — Frequency response

vector

Complex  $n$ -element frequency response vector. If  $n$  is not specified, the function uses a default value of 8192. The frequency response is evaluated at  $n$  points equally spaced around the upper half of the unit circle.

Data Types: double

### **w** — frequencies

vector

Frequency vector of length  $n$ , in radians/sample.  $w$  consists of  $n$  points equally spaced around the upper half of the unit circle (from 0 to  $\pi$  radians/sample). If  $n$  is not specified, the function uses a default value of 8192.

Data Types: double

## Tips

There are several ways of analyzing the frequency response of filters. `freqz` accounts for quantization effects in the filter coefficients, but does not account for quantization effects in filtering arithmetic. To account for the quantization effects in filtering arithmetic, refer to function `noisepsd`.

## Algorithms

`freqz` calculates the frequency response for a filter from the filter transfer function  $Hq(z)$ . The complex-valued frequency response is calculated by evaluating  $Hq(e^{j\omega})$  at discrete values of  $w$  specified by the syntax you use. The integer input argument  $n$  determines the number of equally-spaced points around the upper half of the unit circle at which `freqz` evaluates the frequency response. The frequency ranges from 0 to  $\pi$  radians per sample when you do not supply a sampling frequency as an input argument. When you supply the scalar sampling frequency `fs` as an input argument to `freqz`, the frequency ranges from 0 to  $fs/2$  Hz.

## Version History

Introduced in R2011a

## See Also

### Functions

`freqz`

### Topics

“Analysis Methods for Filter System Objects” on page 3-2

# freqz

**Package:** dsp

Frequency response of filters in channelizer

## Syntax

```
[H,w] = freqz(obj)
[H,w] = freqz(obj,ind)
[H,f] = freqz(obj,ind,Name,Value)
```

## Description

`[H,w] = freqz(obj)` computes a matrix of complex frequency responses for each filter in the `dsp.Channelizer` System object. Each column of `H` corresponds to the frequency response for one of the filters in the channelizer. `w` is a vector of normalized frequencies at which the rows of `H` are computed.

`[H,w] = freqz(obj,ind)` computes the frequency response of the filters with indices corresponding to the elements in the vector `ind`. `ind` is a row vector of indices between 1 and `obj.NumFrequencyBands`. By default, this vector is `[1:N]`, where `N` is the number of frequency bands.

For example, to compute the frequency response of the first 4 filters, set `ind` to `[1:4]`.

```
channelizer = dsp.Channelizer;
[H,w] = freqz(channelizer,[1:4]);
```

`[H,f] = freqz(obj,ind,Name,Value)` computes the frequency response of the filters with additional options specified by one or more `Name,Value` pair arguments.

For example, to specify a sampling rate of 44100 Hz, set `'Fs'` to 44100. To compute the frequency response using 1024 frequency points, set `'NFFT'` to 1024. In addition, to compute the sum of the frequency response of the filters, set `'overall'` to `true`.

```
channelizer = dsp.Channelizer;
[H,f] = freqz(channelizer,[1:4],'Fs',44100,'NFFT',1024,'overall',true);
```

## Examples

### Frequency Response of Channelizer Filter Bank

Compute the frequency response of the filters in the channelizer using the `freqz` function.

Design a channelizer with the number of frequency bands or polyphase branches set to 8, the number of taps or coefficients per band set to 12, and stopband attenuation set to 80 dB. Compute the frequency response matrix, `H`, and the corresponding vector of frequency points, `w`.



```
channelizer = dsp.Channelizer;
[H,w] = freqz(channelizer); %#ok
whos H
```

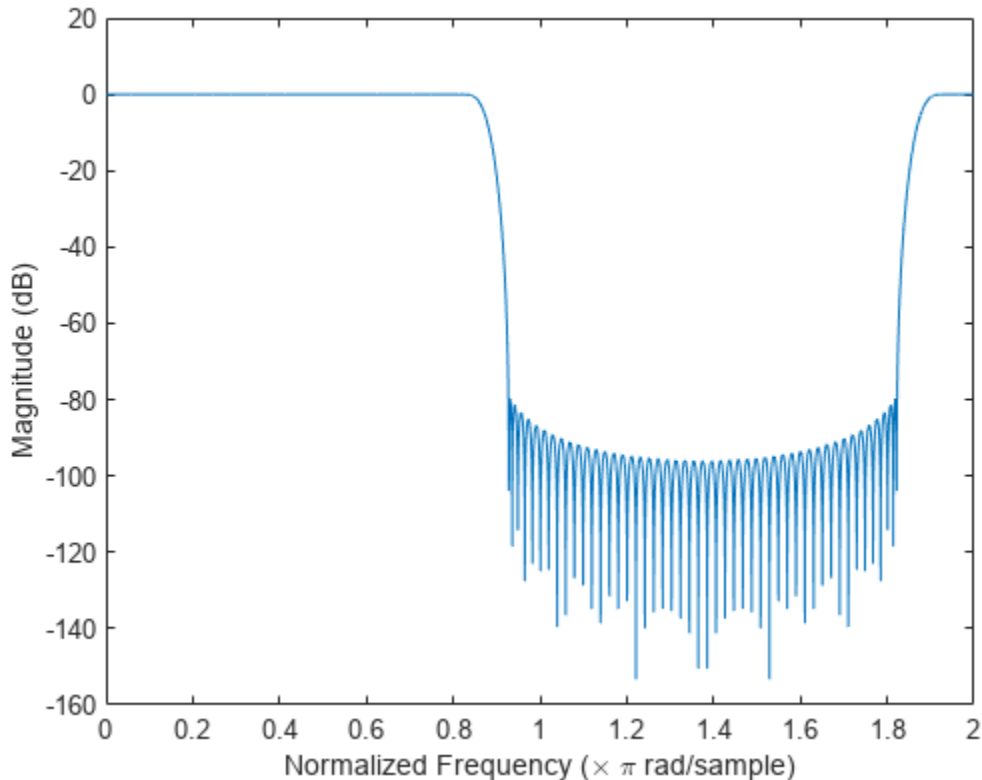
| Name | Size   | Bytes   | Class  | Attributes |
|------|--------|---------|--------|------------|
| H    | 8192x8 | 1048576 | double | complex    |

The number of rows in  $H$  corresponds to the number of frequency points, and the number of columns in  $H$  corresponds to the number of frequency bands. To view only a portion of the filter bank, specify the indices.

```
[H,w] = freqz(channelizer,(1:4)); %#ok
```

Specifying the filter indices as  $[1:4]$  computes the individual frequency response of the first 4 filters. You can alternatively view the sum of the filter responses by setting the 'overall' to true.

```
[H,w] = freqz(channelizer,1:4,'overall',true);
plot(w/pi,20*log10(abs(H)))
xlabel('Normalized Frequency (\times \pi rad/sample)')
ylabel('Magnitude (dB)')
```



You can also compute the frequencies in Hz by passing a sampling frequency. Frequency in Hz,  $f$ , equals  $(w/2 * \pi) * F_s$ , where  $w$  is frequency in radians, and  $F_s$  is the sampling rate.

```
[H,f] = freqz(channelizer,'Fs',44100); %#ok
```

Specify the number of frequency points using the 'NFFT' argument.

```
[H,f] = freqz(channelizer,'Fs',44100,'NFFT',1024);
```

## Input Arguments

### **obj** — Input filter System object

`dsp.Channelizer`

Input filter, specified as a `dsp.Channelizer` System object.

Example: `[H,w] = freqz(channelizer);`

### **ind** — Filter indices

row vector

Filter indices, specified as a row vector in the range `[1 obj.NumFrequencyBands]`. By default, `ind` is set to `1:N`, where `N` is the number of frequency bands specified through the `obj.NumFrequencyBands` property.

Example: `freqz(channelizer,[1:4]);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

## Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, . . . , NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: `[H,f] = freqz(channelizer,[1:4],'Fs',44100,'NFFT',1024,'overall',true);`

### **Fs** — Sampling rate

scalar

Sampling rate, specified as a scalar. This value determines the frequencies in Hz at which `freqz` computes the frequency response.

Example: 44100

Example: 22050

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **NFFT** — Number of frequency points used to compute frequency response

8192 (default) | positive scalar

Number of frequency points used to compute the frequency response, specified as a positive scalar.

Example: 8192

Example: 1024

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **overall** — Type of filter response

`false` (default) | `true`

The type of filter response, specified as either:

- `true` -- `freqz` computes the sum of the filter responses.
- `false` -- `freqz` computes the individual filter responses.

Data Types: `logical`

## Output Arguments

### **H** — Frequency response of filter

matrix (default) | vector

Complex frequency response of the filters in the channelizer. The dimensions of the output depend on the value of the `'overall'` argument:

- When the `'overall'` argument is `true`, the frequency response vector contains the sum of the frequency responses of all the filters. The vector is of size `[nfft 1]`, where `nfft` is the number of frequency points. For example, if `nfft` is 8192, `H` is a matrix of size `[8192 1]`.
- When the `'overall'` argument is `false`, the frequency response is a matrix of size `[nfft nFilters]`, where `nfft` is the number of frequency points and `nFilters` is the number of filters whose frequency response is computed. Suppose `nfft` is 8192 and `ind` is `[2:5]`, `H` is a matrix of size `[8192 4]`.

Data Types: `double`

### **w** — Normalized frequencies

vector

Normalized frequencies, specified in rad/sample, at which the frequency response is computed. The vector is of size `[nfft 1]`.

Data Types: `double`

### **f** — Frequencies

vector

Frequencies, specified in Hz, at which the frequency response is computed. The vector is of size `[nfft 1]`.

Data Types: `double`

## Version History

Introduced in R2017b

## See Also

### Functions

`coeffs` | `tf` | `polyphase` | `fvtool` | `bandedgeFrequencies` | `centerFrequencies` | `getFilters`

### Objects

`dsp.Channelizer`

## freqz

**Package:** dsp

Frequency response of the multirate multistage filter

### Syntax

```
[h, f] = freqz(sysobj, n, range)
[h, f] = freqz(sysobj, f)
```

### Description

`[h, f] = freqz(sysobj, n, range)` returns the complex frequency response, `h`, of the multirate multistage filter System object and the frequency vector `f` at which `h` is computed. `n` is the number of frequency points, and `range` is the frequency range over which the response is computed.

For the sample rate converter object, the sample rate is the larger of `InputSampleRate` and `OutputSampleRate`.

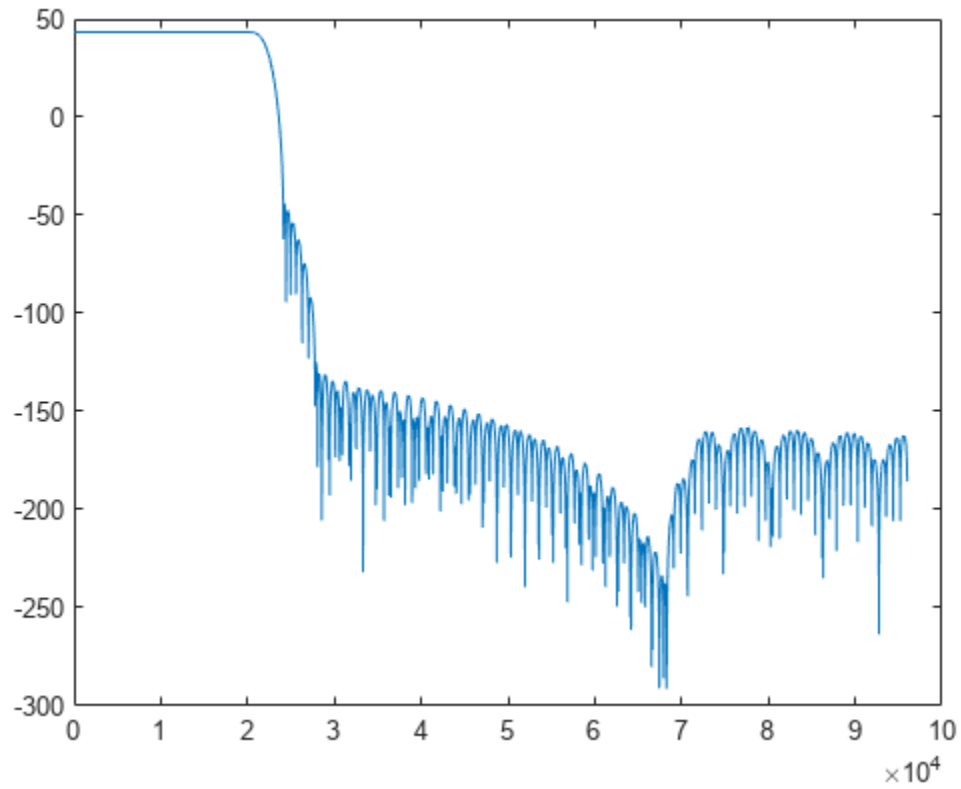
`[h, f] = freqz(sysobj, f)` returns the complex frequency response `h` computed at the frequency points specified by vector `f`. The input vector `f` is in Hz.

### Examples

#### Frequency Response of Default Converter

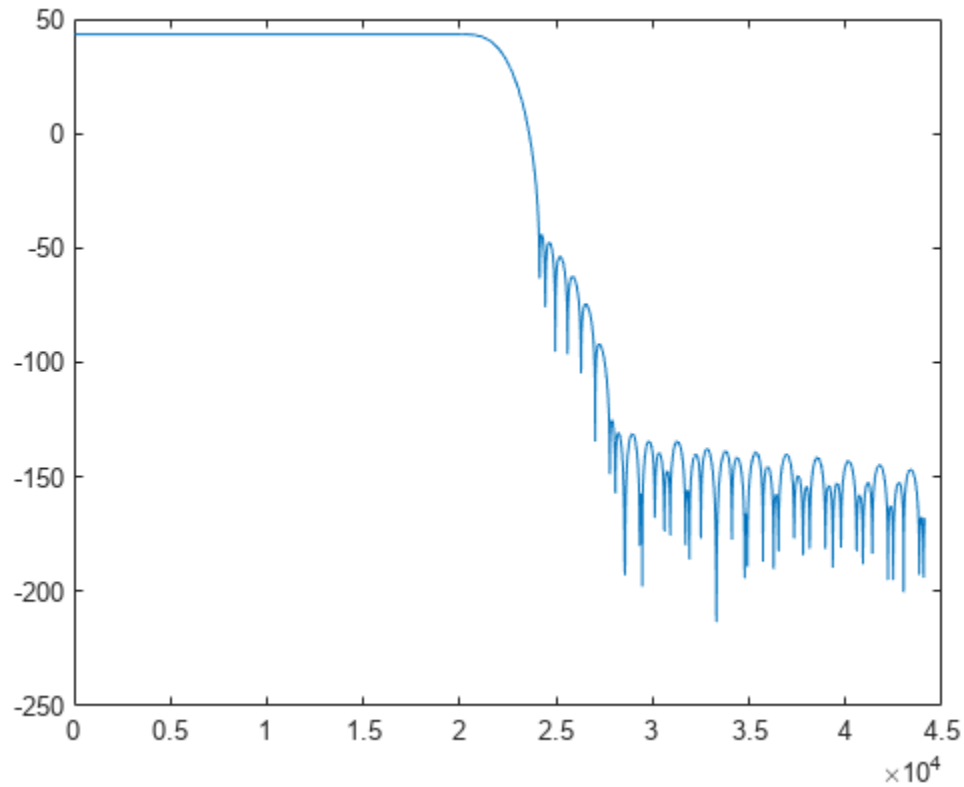
Create a multistage sample rate converter with default properties, corresponding to the combined three filter stages used to convert from 192 kHz to 44.1 kHz. Compute and display the frequency response.

```
src = dsp.SampleRateConverter;
[H, f] = freqz(src);
plot(f, 20*log10(abs(H)))
```



Compute and display the frequency response over the range between 20 Hz and 44.1 kHz.

```
f = 20:10:44.1e3;  
[H,f] = freqz(src,f);  
plot(f,20*log10(abs(H)))
```

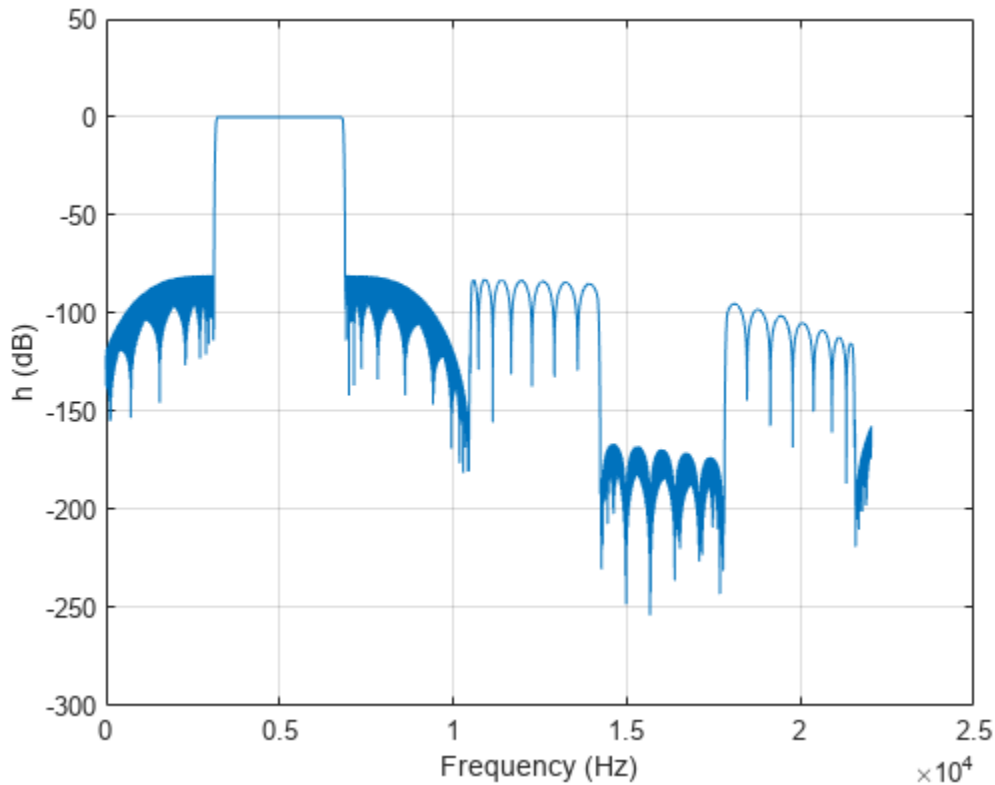


### Compute Frequency Response of Complex Bandpass Decimator

Compute the complex frequency response of a complex bandpass decimator using the `freqz` function.

Create a `dsp.ComplexBandpassDecimator` object. Set the `DecimationFactor` to 12, the `CenterFrequency` to 5000 Hz, and the `SampleRate` to 44100 Hz. Compute and display the frequency response.

```
cbp = dsp.ComplexBandpassDecimator(12,5000,44100);  
[h,f] = freqz(cbp);  
plot(f,20*log10(abs(h)))  
grid on  
xlabel('Frequency (Hz)')  
ylabel('h (dB)')
```



## Input Arguments

### **sysobj** — Filter System object

`dsp.ComplexBandpassDecimator` | `dsp.SampleRateConverter`

Filter System object, specified as a `dsp.ComplexBandpassDecimator` or a `dsp.SampleRateConverter` System object.

### **n** — Number of evaluation points

8192 (default) | positive integer

Number of frequency points for response evaluation, specified as a positive integer scalar. If `n` is not specified, the default is 8192.

Data Types: `single` | `double`

### **range** — Range of frequencies

'half' (default) | 'whole'

Range considered when computing the frequency response, specified as either 'half' (from 0 to  $\pi$ ) or 'whole' (from 0 to  $2\pi$ ). If range is not specified, the default is 'half'.

### **f** — Frequencies in Hz at which response is computed

vector

Frequencies in Hz at which the response is computed, specified as a vector.

Data Types: double

## Output Arguments

### **h** — Complex frequency response

vector

Complex frequency response, returned as a vector.

Data Types: double

### **f** — Frequencies at which response is computed

vector

Frequencies at which the response is computed, returned as a vector.

Data Types: double

## Version History

Introduced in R2018a

## See Also

### Functions

`cost` | `info` | `visualizeFilterStages` | `cost` | `getActualOutputRate` | `getFilters` | `info` | `getRateChangeFactors`

### Objects

`dsp.ComplexBandpassDecimator` | `dsp.SampleRateConverter`



# info

**Package:** dsp

Characteristic information about generated signal

## Syntax

```
S = info(nco)
```

## Description

`S = info(nco)` returns a structure containing the characteristic information, `S`, about the `dsp.NCO` System object, `nco`.

## Examples

### Obtain the Characteristic Information of the NCO Object

The characteristic information of the NCO object is defined by the following fields:

- `NumPointsLUT` — Number of data points in the lookup table.
- `SineLUTSize` — Quarter-wave sine lookup table size in bytes.
- `TheoreticalSFDR` — Theoretical spurious free dynamic range (SFDR) in dBc.
- `FrequencyResolution` — Frequency resolution of the NCO.

To obtain the above characteristics for a specific NCO object, call the `info` function on the object.

```
nco = dsp.NCO
```

```
nco =
    dsp.NCO with properties:
        PhaseIncrementSource: 'Input port'
        PhaseOffsetSource: 'Property'
        PhaseOffset: 0
        Dither: true
        NumDitherBits: 4
        PhaseQuantization: true
        NumQuantizerAccumulatorBits: 12
        PhaseQuantizationErrorOutputPort: false
        Waveform: 'Sine'
        SamplesPerFrame: 1
        OutputDataType: 'Custom'
```

Show all properties

```
info(nco)
```

```
ans = struct with fields:
    NumPointsLUT: 1025
```

```

        SineLUTSize: 2050
    TheoreticalSFDR: 84
    FrequencyResolution: 1.5259e-05

```

The fields and their corresponding values change depending on the settings of the object. For instance, if the `PhaseQuantization` property is set to `false`, the `TheoreticalSFDR` field does not appear.

```

nco.PhaseQuantization = false;
info(nco)

ans = struct with fields:
    NumPointsLUT: 16385
    SineLUTSize: 32770
    FrequencyResolution: 1.5259e-05

```

## Input Arguments

### **nco** — Numerically controlled oscillator

`dsp.NCO` System object

Numerically controlled oscillator, specified as a `dsp.NCO` System object.

## Output Arguments

### **S** — Characteristic information about signal

structure

Characteristic information about the `dsp.NCO` System object, returned as a structure, `S`. The number of fields of `S` and their values vary depending on the property value settings of `nco`. The possible fields and their values are:

| Field                            | Value   |
|----------------------------------|---|
| <code>NumPointsLUT</code>        | Number of data points for lookup table. The lookup table is implemented as a quarter-wave sine table.   |
| <code>SineLUTSize</code>         | Quarter-wave sine lookup table size in bytes.   |
| <code>TheoreticalSFDR</code>     | Theoretical spurious free dynamic range (SFDR) in dBc. This field applies when you set the <code>PhaseQuantization</code> property to <code>true</code> . |
| <code>FrequencyResolution</code> | Frequency resolution of the NCO in Hz. The sample time of the output signal is assumed to be 1 second.  |

Data Types: `struct`

## Version History

Introduced in R2012a

## **See Also**

**Objects**  
dsp.NCO

## fvtool

**Package:** dsp

Visualize frequency response of DSP filters

### Syntax

```
fvtool(sysobj)
fvtool(sysobj,options)
fvtool(____,Name,Value)
```

### Description

`fvtool(sysobj)` displays the magnitude response of the filter System object.

`fvtool(sysobj,options)` displays the response that is specified by the options.

For example, to visualize the impulse response of an FIR filter System object, set options to 'impulse'.

```
Fs = 96e3; filtSpecs = fdesign.lowpass(20e3,22.05e3,1,80,Fs);
    fir1p2 = design(filtSpecs,'equiripple','SystemObject',true);
fvtool(fir1p2,'impulse');
```

`fvtool(____,Name,Value)` visualizes the response of the filter with each specified property set to the specified value.

For more input options, see **FVTool** in Signal Processing Toolbox.

### Examples

#### Impulse and Frequency Response of Halfband Decimation Filter

Create two lowpass halfband decimation filters for data sampled at 44.1 kHz. The design method in the first filter is set to "Equiripple" and in the second filter is set to "Kaiser".

The output data rate is 1/2 the input sample rate, or 22.05 kHz. Specify the filter order to be 52 with a transition width of 4.1 kHz.

```
Fs = 44.1e3;
filterspec = "Filter order and transition width";
Order = 52;
TW = 4.1e3;
firhalfbanddecimEqui = dsp.FIRHalfbandDecimator(...
    Specification=filterspec,...
    FilterOrder=Order,...
    TransitionWidth=TW,...
    DesignMethod="Equiripple",...
    SampleRate=Fs);
firhalfbanddecimKaiser = dsp.FIRHalfbandDecimator(...
```

```

Specification=filterspec,...
FilterOrder=Order,...
TransitionWidth=TW,...
DesignMethod="Kaiser",...
SampleRate=Fs);

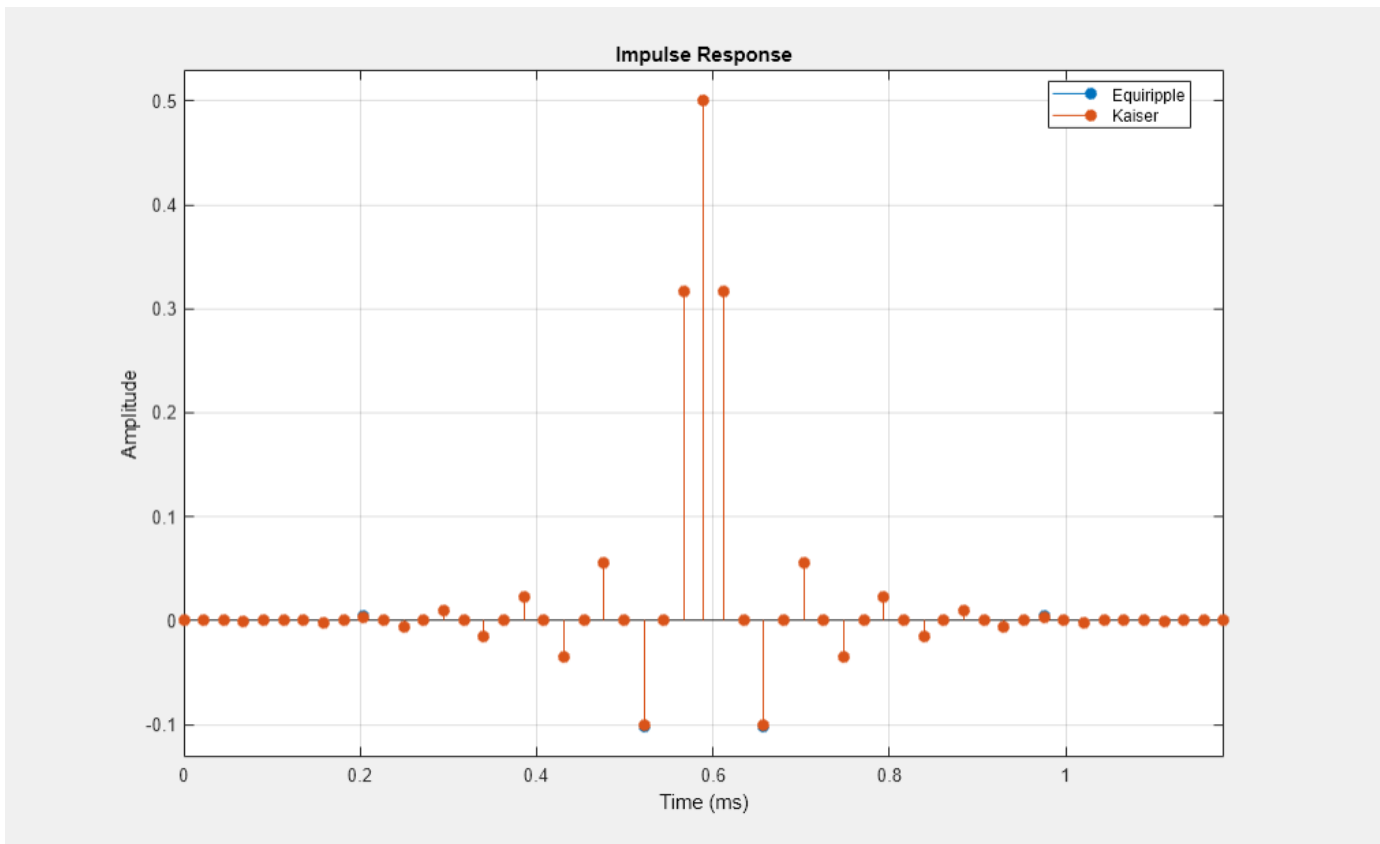
```

Plot the impulse response of both the filters. The zeroth-order coefficient is delayed 26 samples, which is equal to the group delay of the filter. This yields a causal halfband filter.

```

hfvt = fvtool(firhalfbanddecimEqui,firhalfbanddecimKaiser,...
    Analysis="impulse");
legend(hfvt,{'Equiripple','Kaiser'})

```



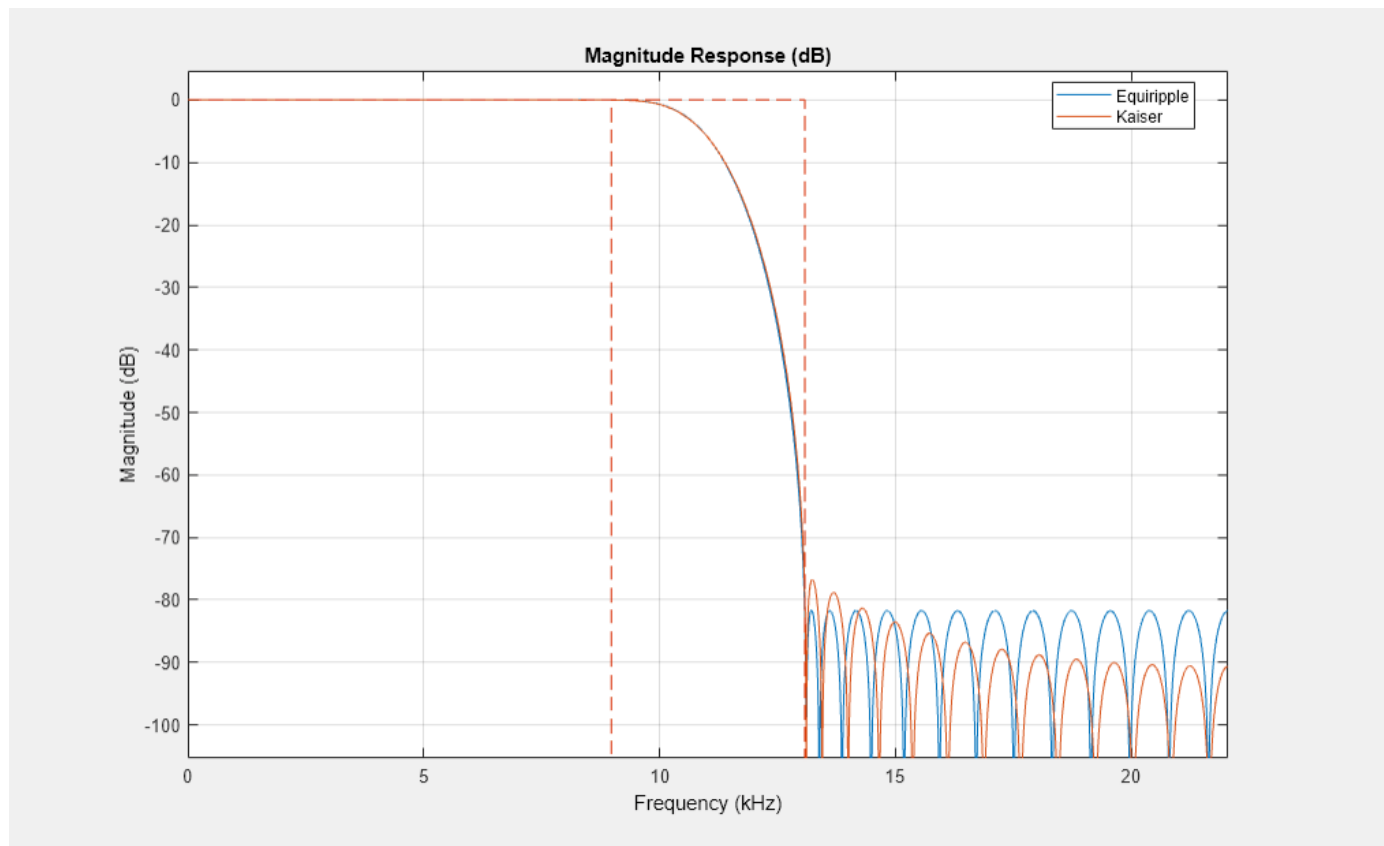
Plot the magnitude and phase response.

If the filter specifications are tight, say a very high filter order with a very narrow transition width, the filter designed using the "Kaiser" method converges more effectively.

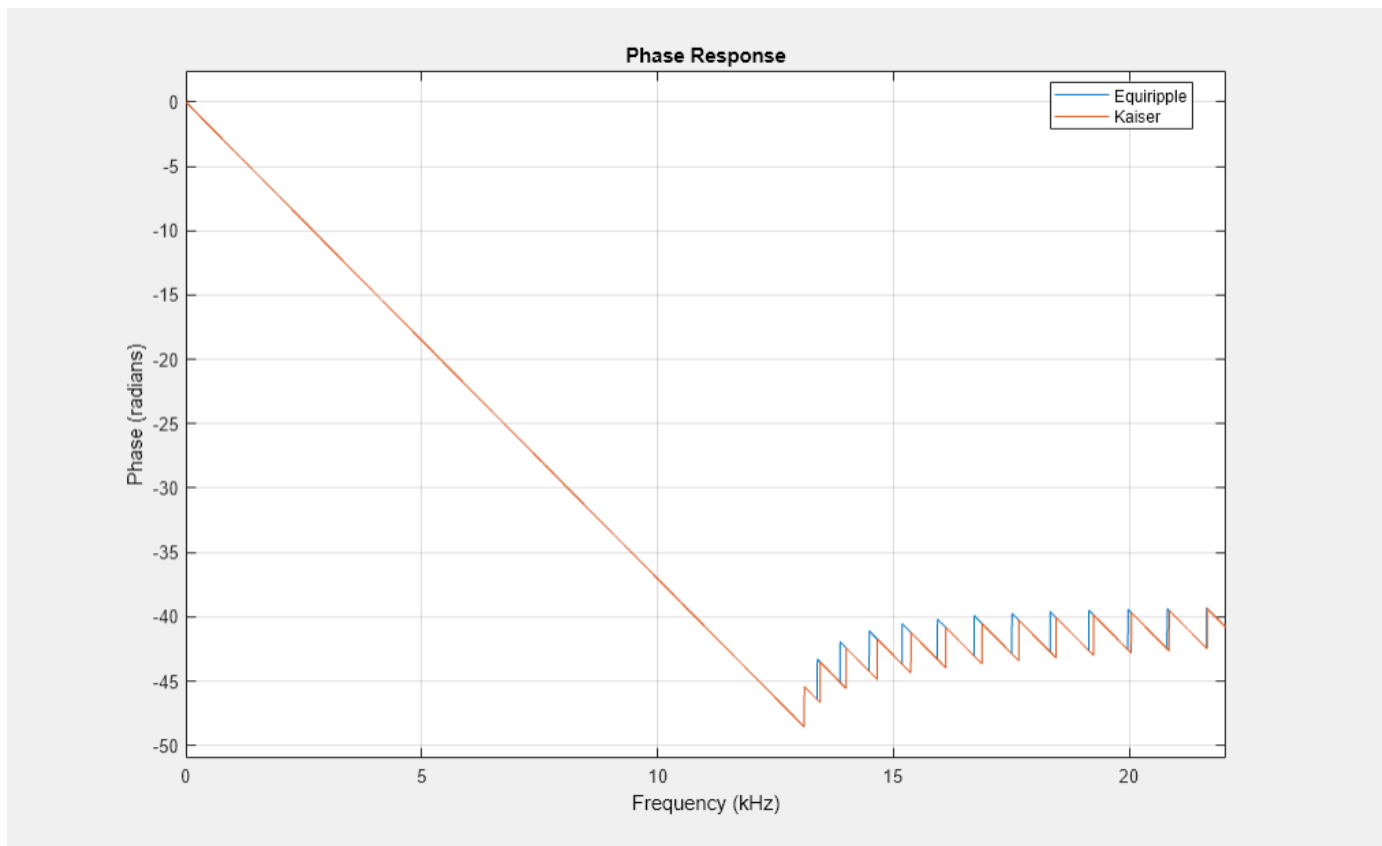
```

hvftMag = fvtool(firhalfbanddecimEqui,firhalfbanddecimKaiser,...
    Analysis="Magnitude");
legend(hvftMag,{'Equiripple','Kaiser'})

```



```
hvftPhase = fvtool(firhalfbanddecimEqui, firhalfbanddecimKaiser, ...  
    Analysis="Phase");  
legend(hvftPhase, {'Equiripple', 'Kaiser'})
```



### Impulse and Frequency Response of FIR and IIR Lowpass Filters

Create a minimum-order FIR lowpass filter for data sampled at 44.1 kHz. Specify a passband frequency of 8 kHz, a stopband frequency of 12 kHz, a passband ripple of 0.1 dB, and a stopband attenuation of 80 dB.

```

Fs = 44.1e3;
filtertype = 'FIR';
Fpass = 8e3;
Fstop = 12e3;
Rp = 0.1;
Astop = 80;
FIRLPF = dsp.LowpassFilter(SampleRate=Fs,...
    FilterType=filtertype,...
    PassbandFrequency=Fpass,...
    StopbandFrequency=Fstop,...
    PassbandRipple=Rp,...
    StopbandAttenuation=Astop);

```

Design a minimum-order IIR lowpass filter with the same properties as the FIR lowpass filter. Change the `FilterType` property of the cloned filter to IIR.

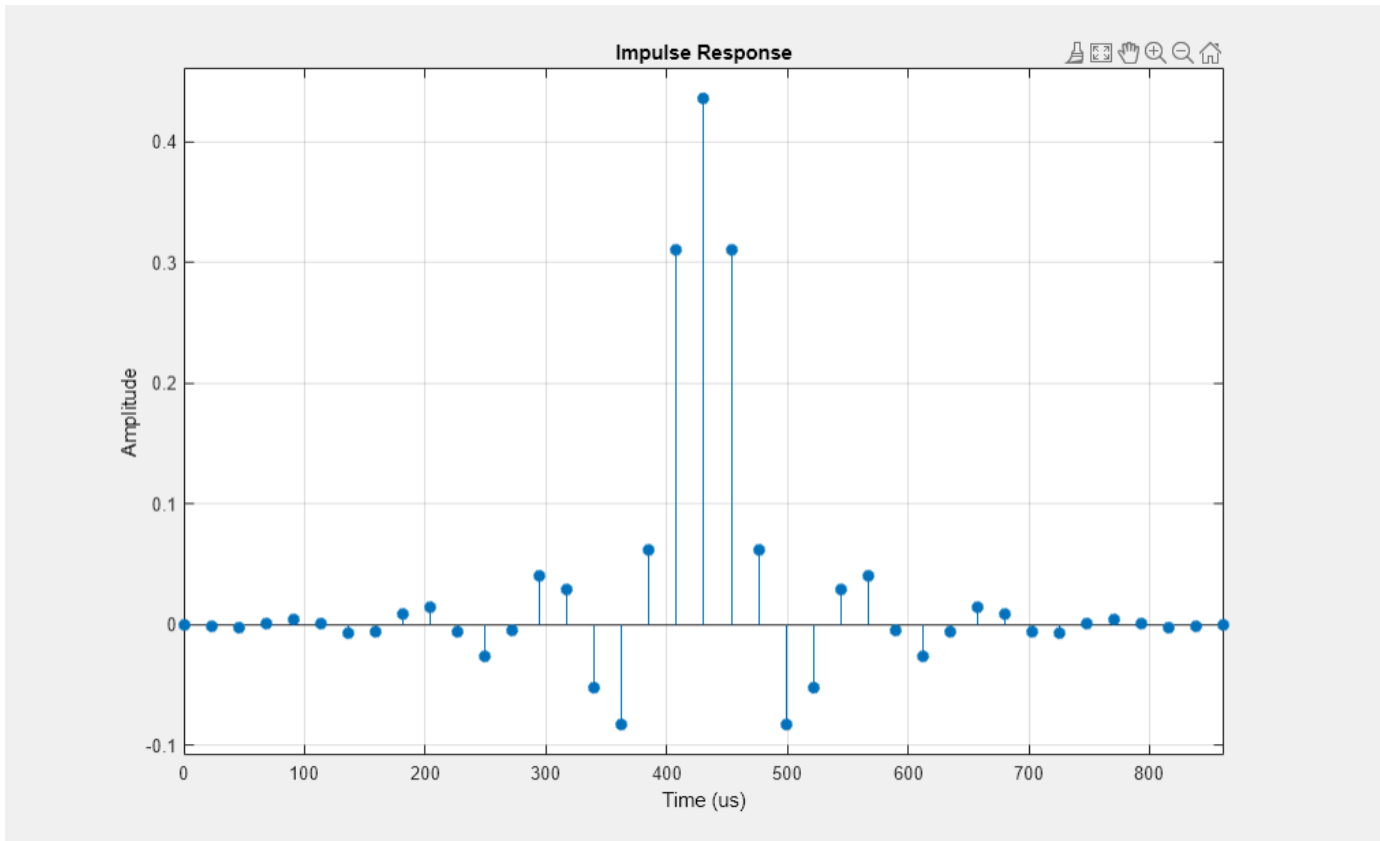
```

IIRLPF = clone(FIRLPF);
IIRLPF.FilterType = 'IIR';

```

Plot the impulse response of the FIR lowpass filter. The zeroth-order coefficient is delayed by 19 samples, which is equal to the group delay of the filter. The FIR lowpass filter is a causal FIR filter.

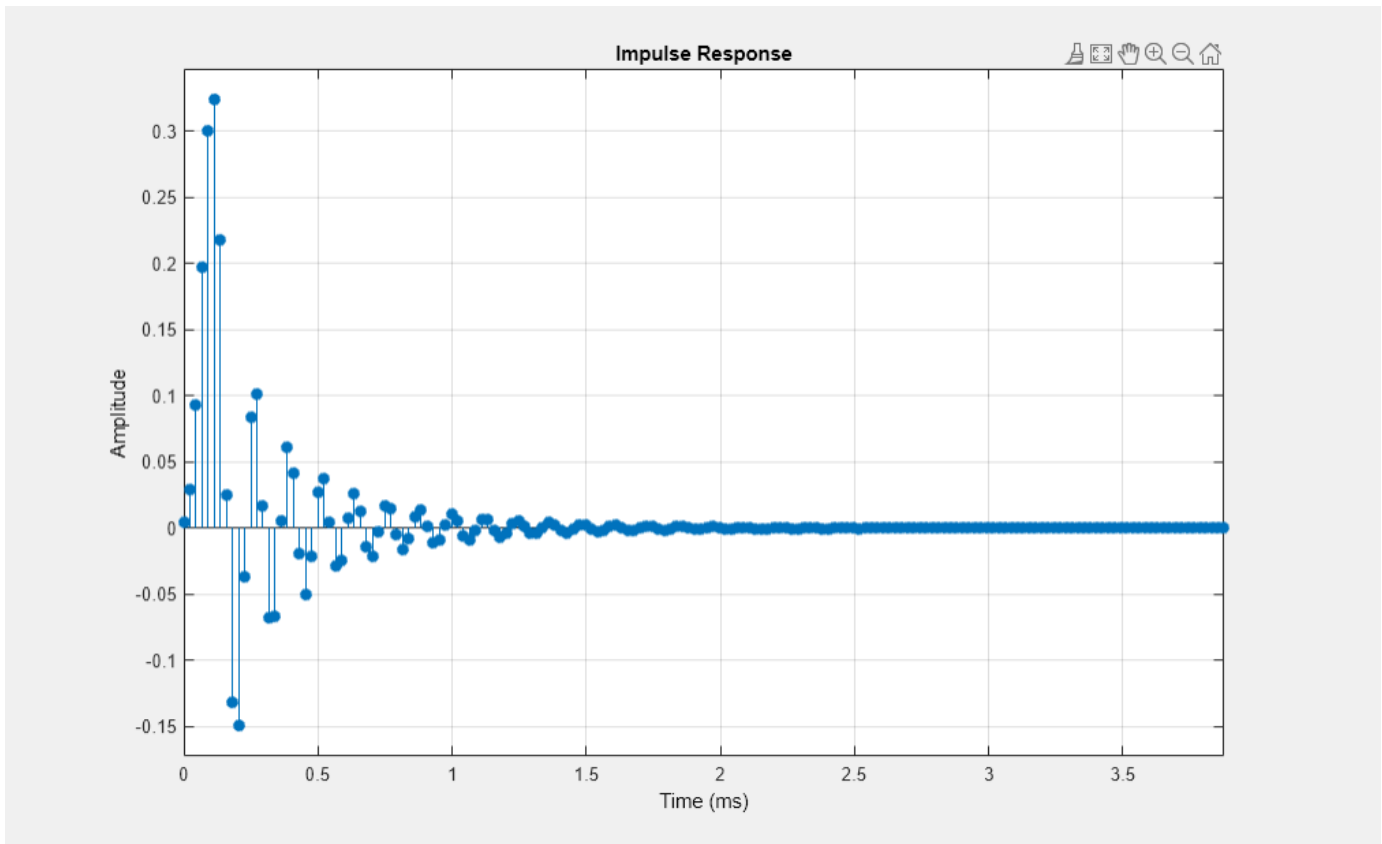
```
fvtool(FIRLPF,Analysis='impulse')
```



Plot the impulse response of the IIR lowpass filter.

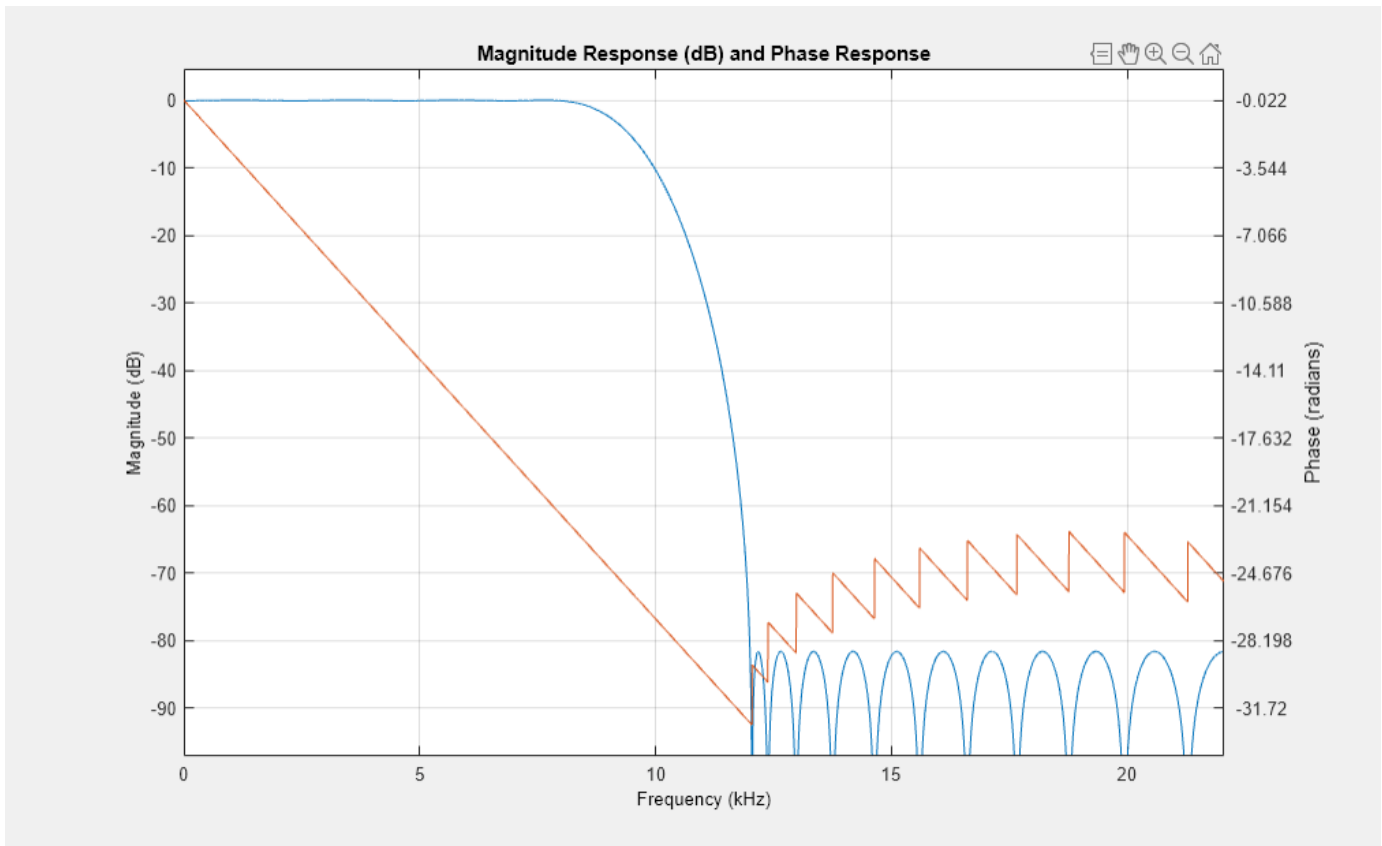
```
fvtool(IIRLPF,Analysis='impulse')
```





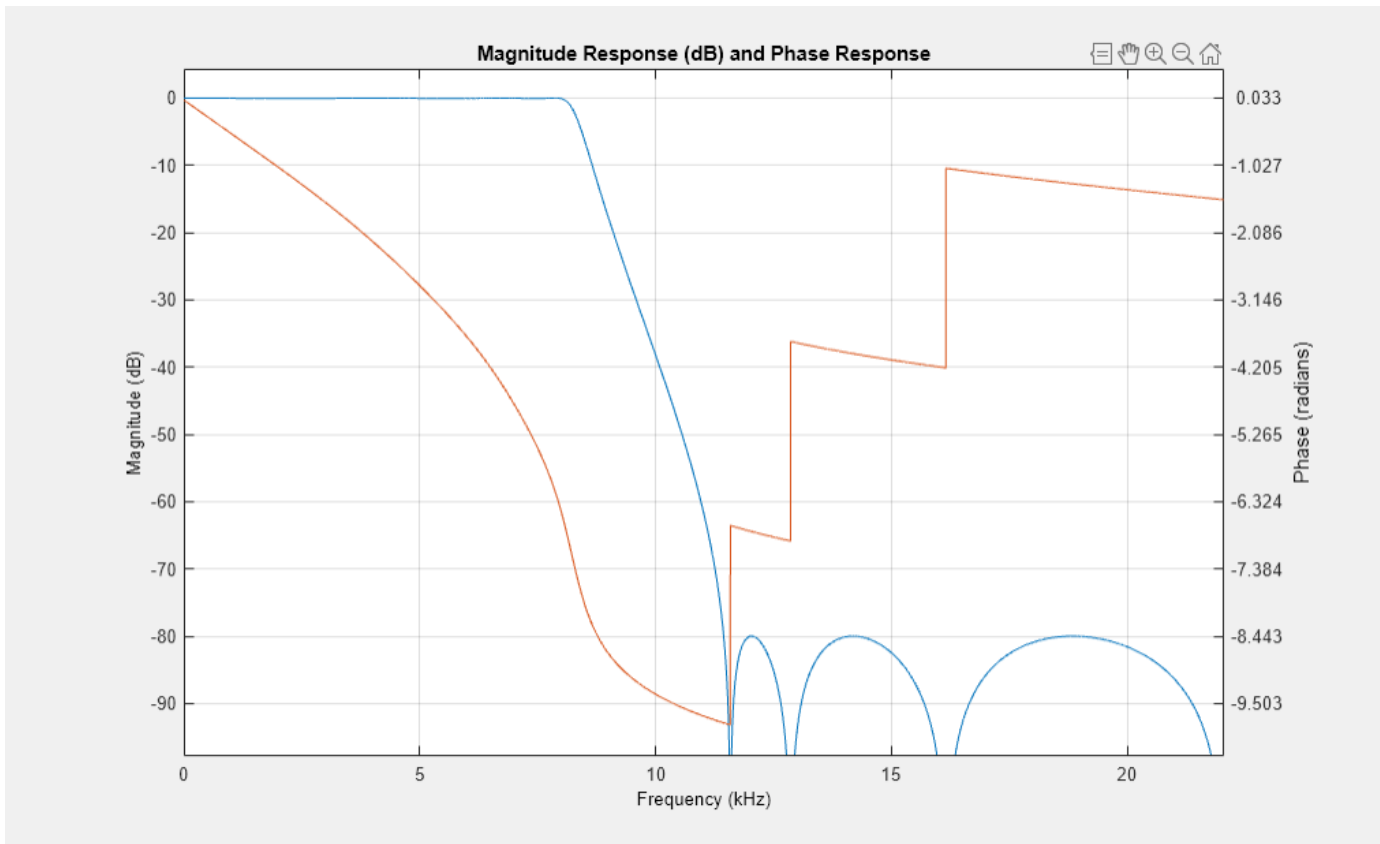
Plot the magnitude and phase response of the FIR lowpass filter.

```
fvtool(FIRLPF,Analysis='freq')
```



Plot the magnitude and phase response of the IIR lowpass filter.

```
fvtool(IIRLPF,Analysis='freq')
```



Calculate the cost of implementing the FIR lowpass filter.

```
cost(FIRLPF)
```

```
ans = struct with fields:
    NumCoefficients: 39
    NumStates: 38
    MultiplicationsPerInputSample: 39
    AdditionsPerInputSample: 38
```

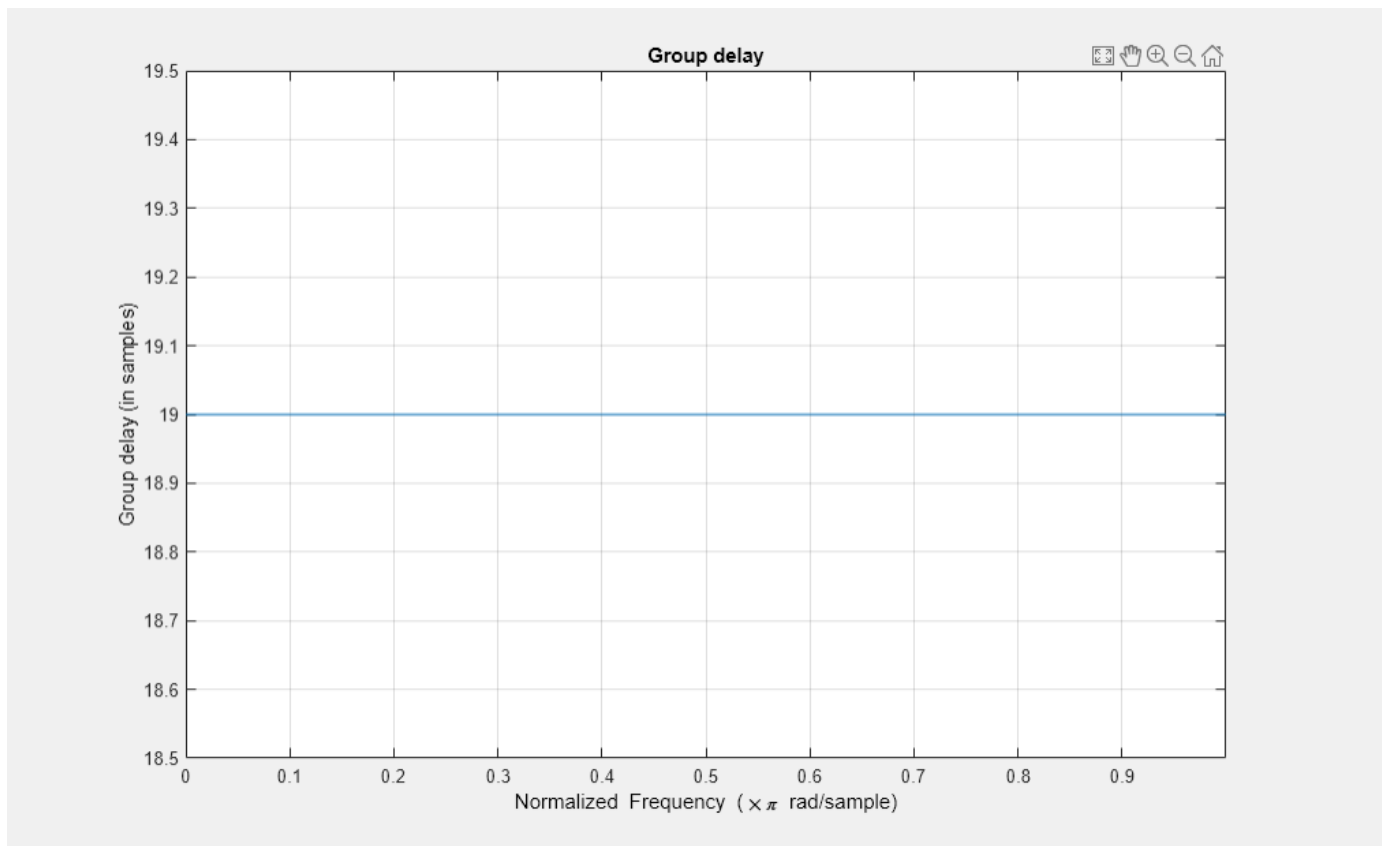
Calculate the cost of implementing the IIR lowpass filter. The IIR filter is more efficient to implement than the FIR filter.

```
cost(IIRLPF)
```

```
ans = struct with fields:
    NumCoefficients: 18
    NumStates: 14
    MultiplicationsPerInputSample: 18
    AdditionsPerInputSample: 14
```

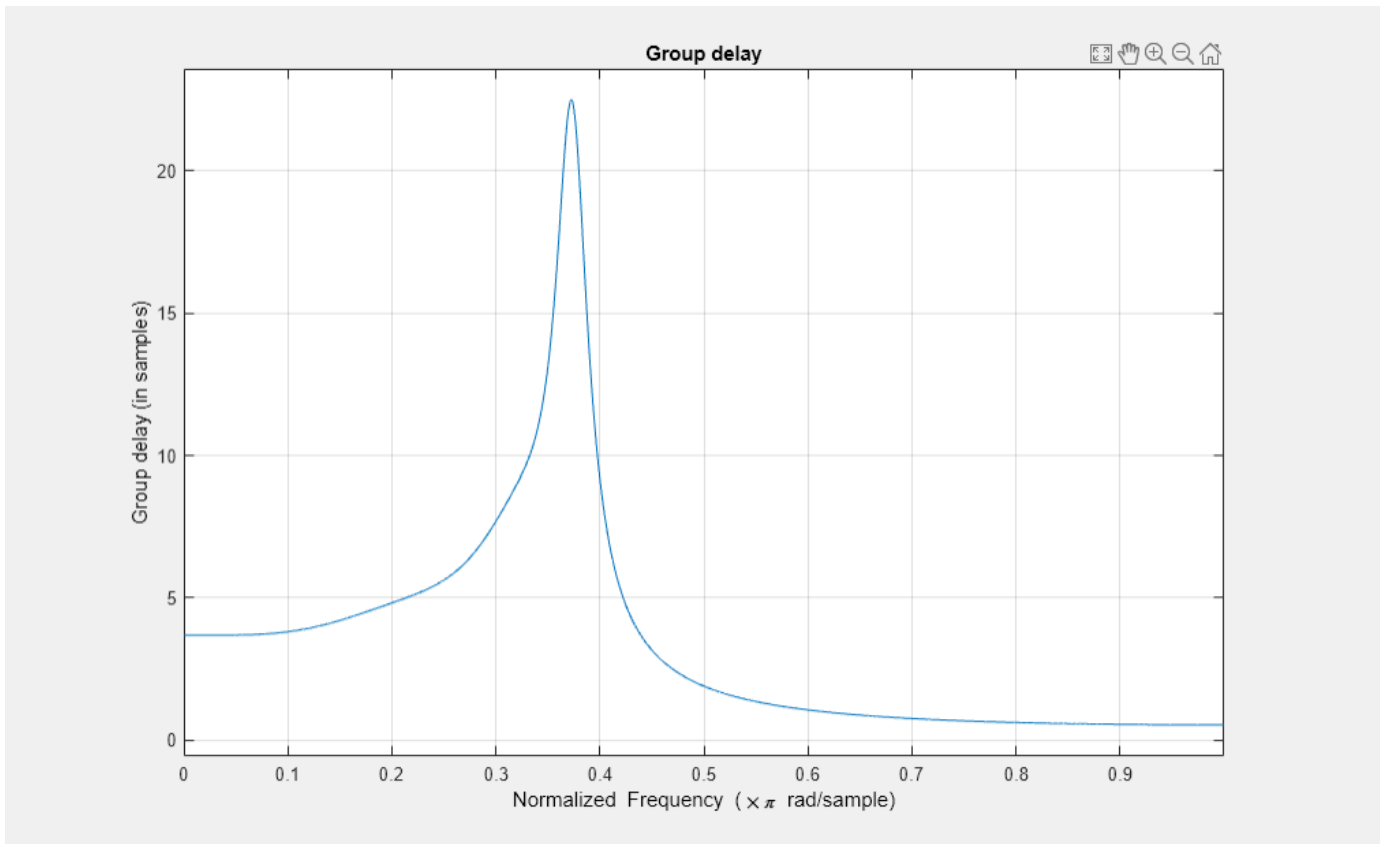
Calculate the group delay of the FIR lowpass filter.

```
grpdelay(FIRLPF)
```



Calculate the group delay of the IIR lowpass filter. The FIR filter has a constant group delay (linear phase), while its IIR counterpart does not.

```
grpdelay(IIRLPF)
```



## Input Arguments

### sysobj — Input filter

filter System object

Input filter, specified as one of the following filter System objects:

- `dsp.AllpassFilter`
- `dsp.AllpoleFilter`
- `dsp.BiquadFilter`
- `dsp.CICCompensationDecimator`
- `dsp.CICCompensationInterpolator`
- `dsp.CICDecimator`
- `dsp.CICInterpolator`
- `dsp.Differentiator`
- `dsp.FarrowRateConverter`
- `dsp.FilterCascade`
- `dsp.FIRDecimator`
- `dsp.FIRFilter`

- `dsp.FIRHalfbandDecimator`
- `dsp.FIRHalfbandInterpolator`
- `dsp.FIRInterpolator`
- `dsp.FIRRateConverter`
- `dsp.FourthOrderSectionFilter`
- `dsp.HighpassFilter`
- `dsp.IIRFilter`
- `dsp.IIRHalfbandDecimator`
- `dsp.IIRHalfbandInterpolator`
- `dsp.LowpassFilter`
- `dsp.NotchPeakFilter`
- `dsp.SOSFilter`
- `dsp.VariableBandwidthFIRFilter`
- `dsp.VariableBandwidthIIRFilter`

```
Example: firFilt = dsp.FIRFilter('Numerator', fir1(130, 2000/(8000/2)));
fvtool(firFilt)
```

### options — Filter analysis options

'magnitude' (default) | 'phase' | 'freq' | 'grpdelay' | 'phasedelay' | 'impulse' | 'step' | 'polezero' | 'coefficients' | 'info' | 'magestimate' | 'noisepower'

Filter analysis options, specified as one of the following:

- 'magnitude' -- Magnitude response
- 'phase' -- Phase response
- 'freq' -- Frequency response
- 'grpdelay' -- Group delay
- 'phasedelay' -- Phase delay
- 'impulse' -- Impulse response
- 'step' -- Step response
- 'polezero' -- Pole zero plot
- 'coefficients' -- Coefficients vector
- 'info' -- Filter information
- 'magestimate' -- Magnitude response estimate
- 'noisepower' -- Round-off noise power spectrum

```
Example: fvtool(firFilt, 'freq')
```

### Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

```
Example: firFilt = dsp.FIRFilter('Numerator', fir1(130, 2000/(8000/2)));  
fvtool(firFilt, 'Arithmetic', 'single')
```

### **Fs — Sampling rate**

scalar

Sampling rate, specified as a scalar. This value determines the Nyquist interval  $[-Fs/2 \ Fs/2]$  in which the fvtool shows the frequency response of the filters in the channelizer.

Data Types: `single` | `double`

### **Arithmetic — Arithmetic type**

`'double'` (default) | `'single'` | `'Fixed'`

Specify the arithmetic used during analysis. The analysis tool assumes a double-precision filter when the arithmetic input is not specified and the filter System object is unlocked. The `'Arithmetic'` property set to `'Fixed'` applies only to filter System objects with fixed-point properties.

When the `'Arithmetic'` property is set to `'Fixed'`, the tool shows both the double-precision reference filter and the quantized version of the filter. The `CoefficientsDataType` property in the respective filter System object is used in creating the quantized version of the filter for all the analyses options except for the two below:

- `'magestimate'` -- Magnitude response estimate.
- `'noisepower'` -- Round-off noise power spectrum

For these two analyses options, all the fixed-point settings are used in analyzing the quantized version of the filter.

## **Version History**

Introduced before R2006a

## **See Also**

### **Tools**

FVTool

### **Topics**

“Analysis Methods for Filter System Objects” on page 3-2

## fvtool

**Package:** dsp

Visualize the filters in the channelizer

### Syntax

```
fvtool(obj)
fvtool(obj,ind)
fvtool(obj,ind,Name,Value)
```

### Description

`fvtool(obj)` visualizes the filters in the `dsp.Channelizer System` object using the Filter Visualization Tool (FVTool).

`fvtool(obj,ind)` visualizes the filters corresponding to the indices in the vector `ind`. `ind` is a row vector of indices between 1 and `obj.NumFrequencyBands`. By default, this vector is `[1:N]`, where `N` is the smallest of `obj.NumFrequencyBands` and 64.

For example, to visualize the first 4 filters, set `ind` to `[1:4]`.

```
channelizer = dsp.Channelizer;
fvtool(channelizer,[1:4]);
```

`fvtool(obj,ind,Name,Value)` visualizes the filters with additional options specified by one or more `Name,Value` pair arguments.

For example, to visualize the first 4 filters in the channelizer over the Nyquist interval  $[-44100/2, 44100/2]$  Hz, set `'Fs'` to 44100. To compute the frequency response using 1024 frequency points, set `'NFFT'` to 1024. In addition, to visualize sum of the filter responses, set `'overall'` to `true`.

```
channelizer = dsp.Channelizer;
fvtool(channelizer,[1:4],'Fs',44100,'NFFT',1024,'overall',true);
```

### Examples

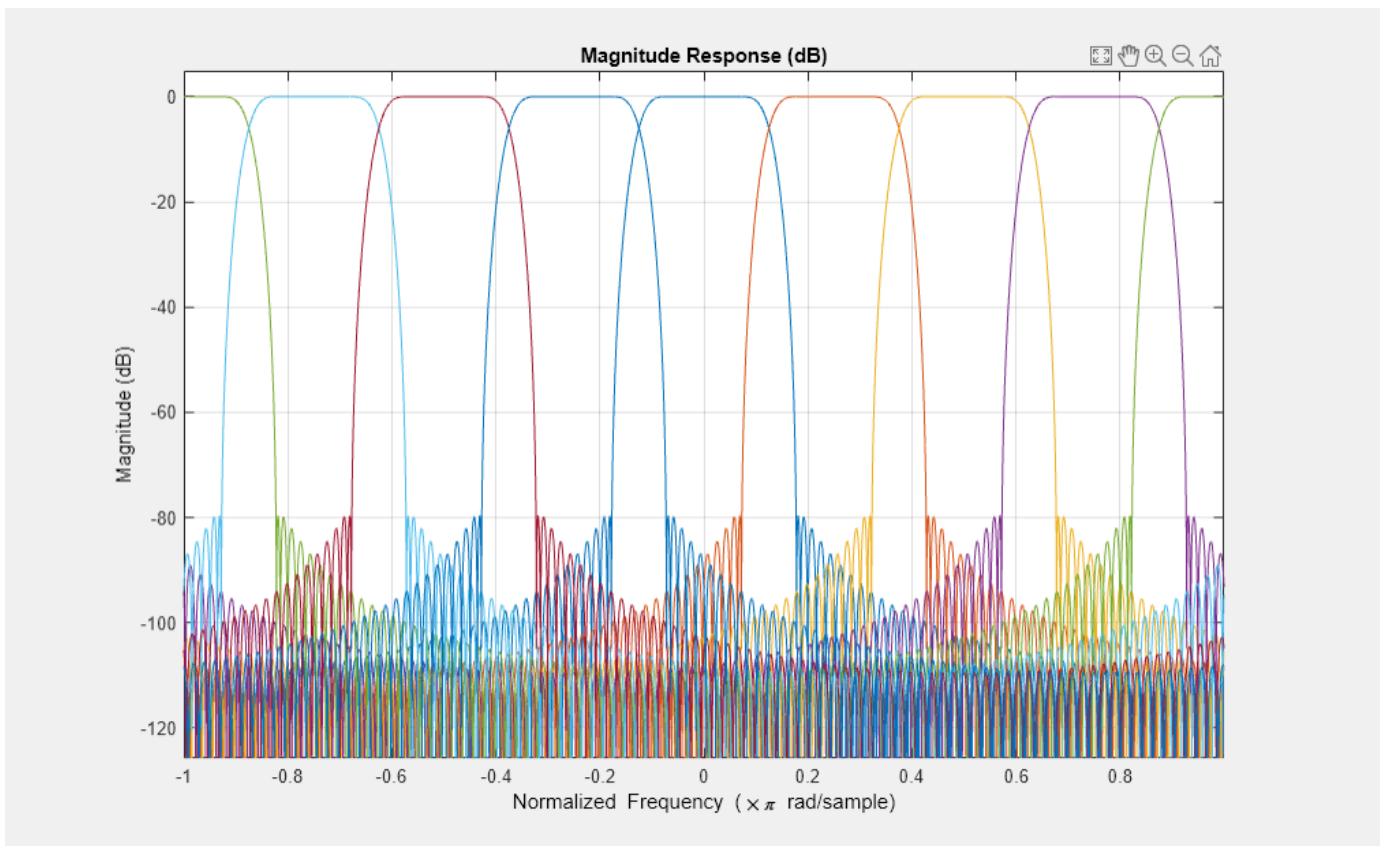
#### Visualize Filters in Channelizer

Using the `fvtool` function, you can visualize the individual filter responses or sum of all the filter responses in the channelizer filter bank.

Design a channelizer with the number of frequency bands or polyphase branches set to 8, the number of taps or coefficients per band set to 12, and stopband attenuation set to 80 dB. View the response of the filter bank.

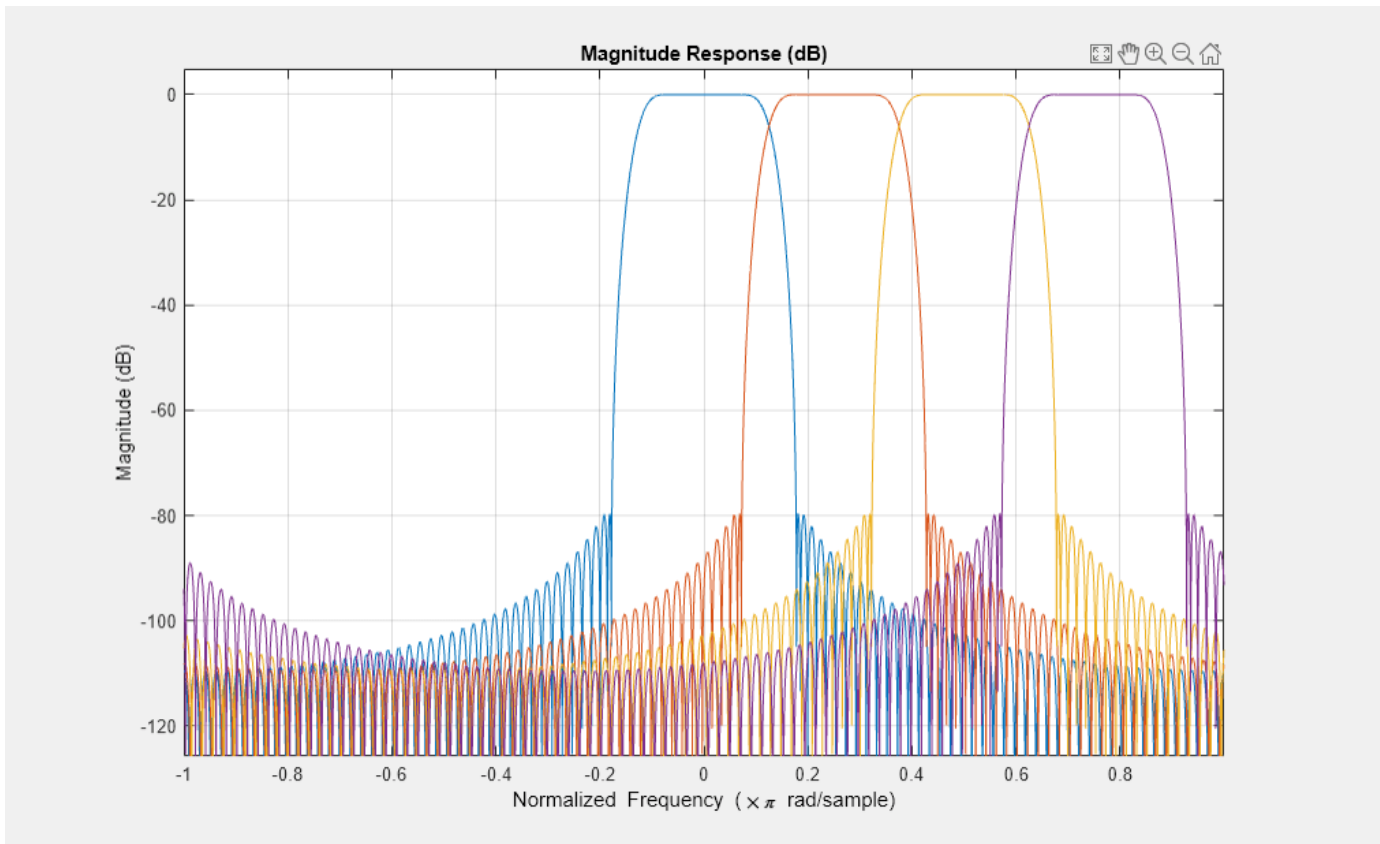
```
channelizer = dsp.Channelizer;
fvtool(channelizer)
```





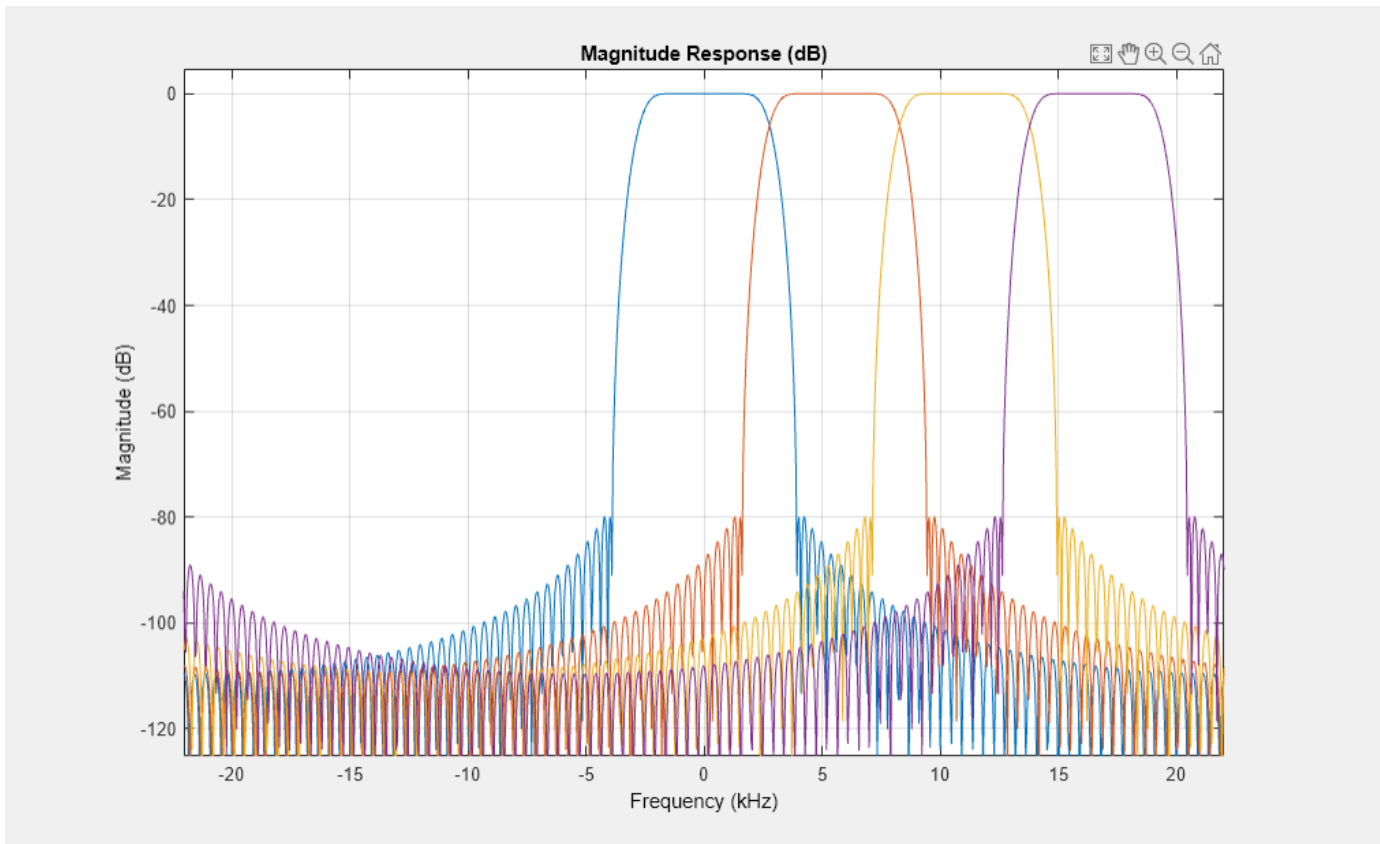
The `fvtool` shows the response of the lowpass prototype filter and all the modulated filters. To view only a portion of the filter bank, specify the indices in `ind`. To view the response of the first 4 filters, set `ind` to `[1:4]`.

```
fvtool(channelizer,(1:4));
```



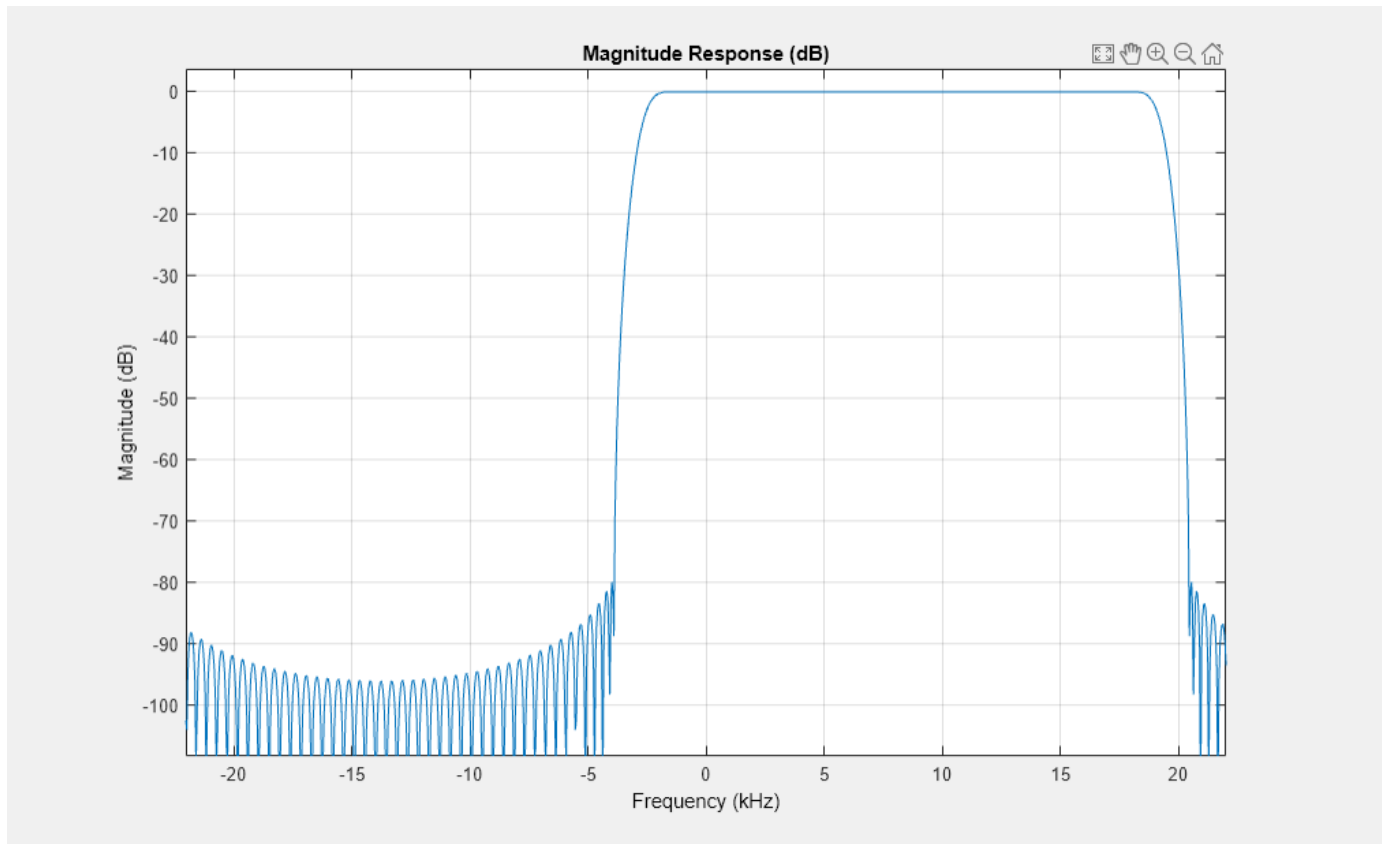
You can change the Nyquist interval to [-22,050 22,050] Hz and the number of frequency points to 1024.

```
fvtool(channelizer,(1:4),'Fs',44100,'NFFT',1024);
```



To see the sum of the responses of all 4 filters, set the 'overall' argument to true.

```
fvtool(channelizer,(1:4),'Fs',44100,'NFFT',1024,'overall',true);
```



## Input Arguments

### **obj** — Input filter System object

`dsp.Channelizer`

Input filter, specified as a `dsp.Channelizer` System object.

Example: `channelizer = dsp.Channelizer; fvtool(channelizer);`

### **ind** — Filter indices

row vector

Filter indices, specified as a row vector in the range `[1 obj.NumFrequencyBands]`. By default, `ind` is set to `1:N`, where  $N$  is the smallest of `obj.NumFrequencyBands` and 64.

Example: `fvtool(channelizer,[1:4]);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: `fvtool(channelizer,'Fs',44100,'NFFT',1024,'overall',true);`

### **Fs — Sampling rate**

scalar

Sampling rate, specified as a scalar. This value determines the Nyquist interval  $[-Fs/2 \ Fs/2]$  in which the `fvtool` shows the frequency response of the filters in the channelizer.

Example: 44100

Example: 22050

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **NFFT — Number of frequency points used to compute the frequency response**

8192 (default) | positive scalar

Number of frequency points used to compute the frequency response, specified as a positive scalar.

Example: 8192

Example: 1024

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **overall — Type of filter response**

`false` (default) | `true`

Type of filter response, specified as either:

- `true` -- `fvtool` shows the sum of the filter responses.
- `false` -- `fvtool` shows the individual filter responses.

Data Types: `logical`

## **Version History**

**Introduced in R2017b**

### **See Also**

#### **Functions**

`polyphase` | `coeffs` | `tf` | `freqz` | `bandedgeFrequencies` | `centerFrequencies` | `getFilters`

#### **Objects**

`dsp.Channelizer`

## fvtool

**Package:** dsp

Visualize frequency response of coupled allpass filter

### Syntax

```
fvtool(caf)
fvtool(caf,options)
fvtool(____,Name,Value)
```

### Description

`fvtool(caf)` displays the magnitude response of the coupled allpass filter System object.

`fvtool(caf,options)` displays the response that is specified by the options. For example, to visualize the impulse response of a coupled allpass filter System object, set options to 'impulse'.

```
caf = dsp.CoupledAllpassFilter;
fvtool(caf,'impulse');
```

`fvtool(____,Name,Value)` visualizes the response of the filter with each specified property set to the specified value.

For more input options, see **FVTool**.

### Examples

#### View Power Complementary Output of Coupled Allpass Filter

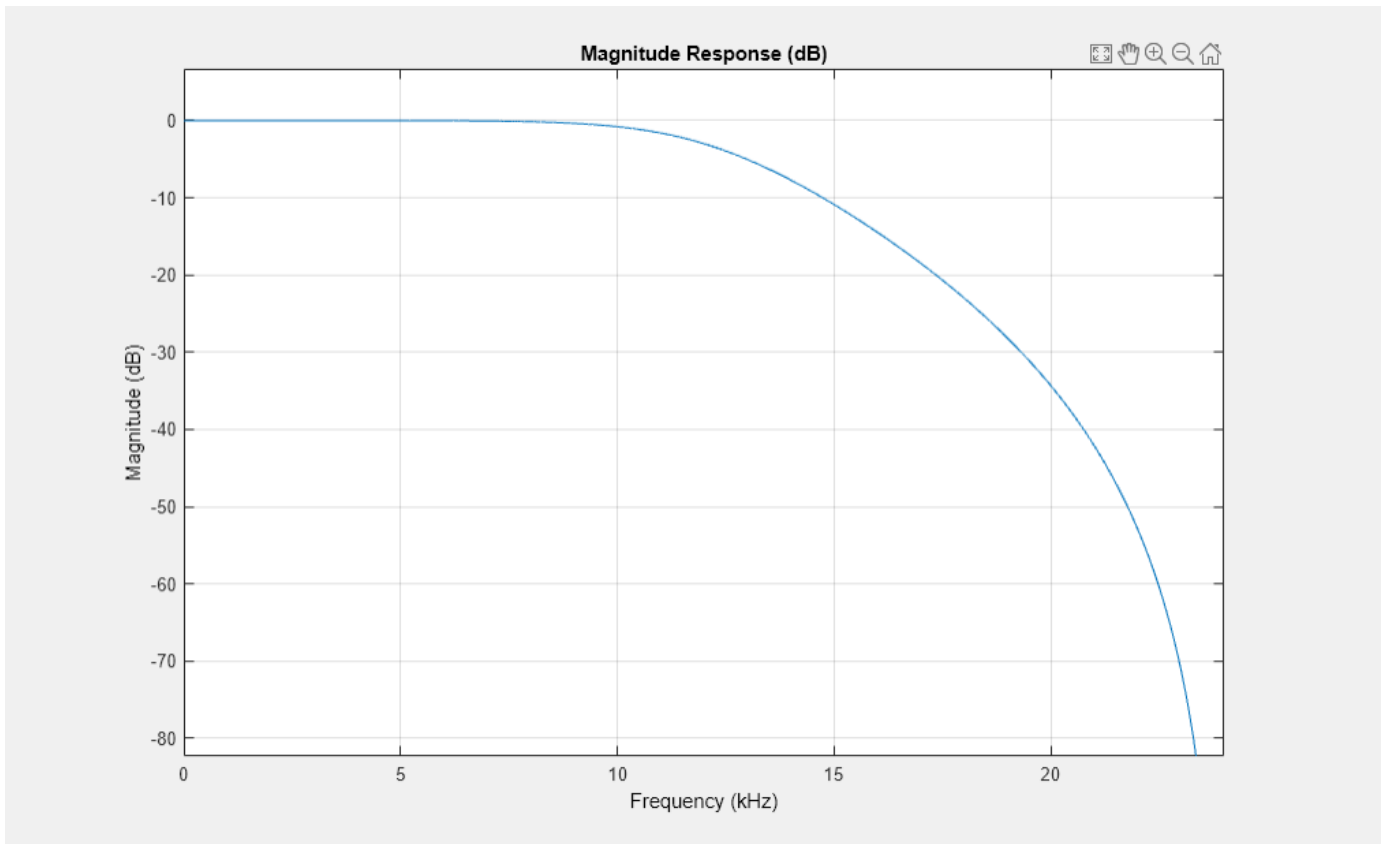
Design a Butterworth lowpass filter of order 3. Use a coupled allpass structure with inner minimum multiplier structure.

```
Fs = 48000;    % in Hz
Fc = 12000;   % in Hz
frameLength = 1024;
[b,a] = butter(3,2*Fc/Fs);
AExp = [freqz(b,a,frameLength/2); NaN];
[c1,c2] = tf2ca(b,a);
caf = dsp.CoupledAllpassFilter(c1(2:end),c2(2:end));
```

Using the 'SubbandView' option of the `dsp.CoupledAllpassFilter`, you can visualize the lowpass filter output, the power complementary highpass filter output, or both using the `fvtool`.

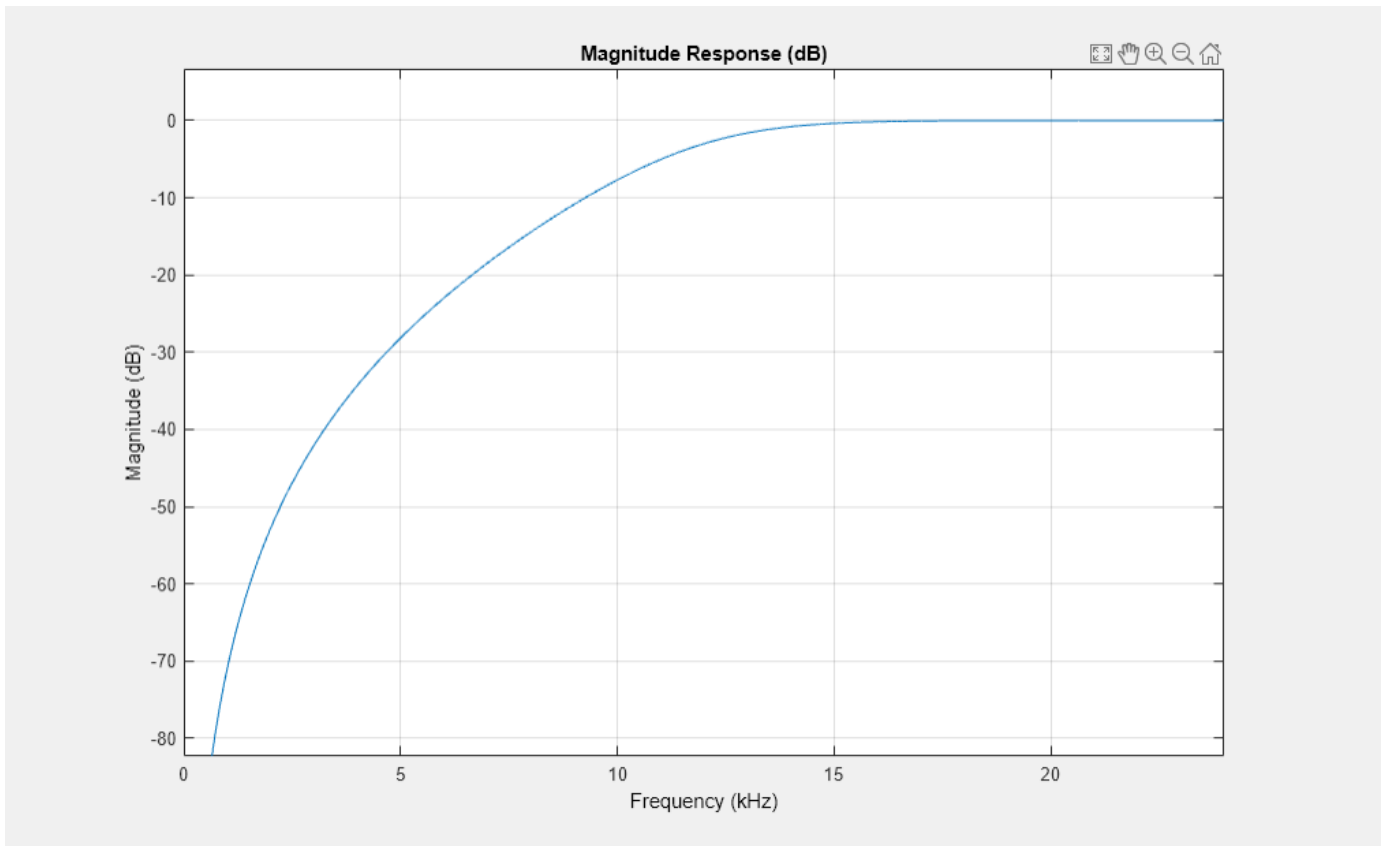
To view the lowpass filter output, set 'SubbandView' to 1.

```
fvtool(caf,'SubbandView',1,'Fs',Fs)
```



To view the highpass filter output, set 'SubbandView' to 2.

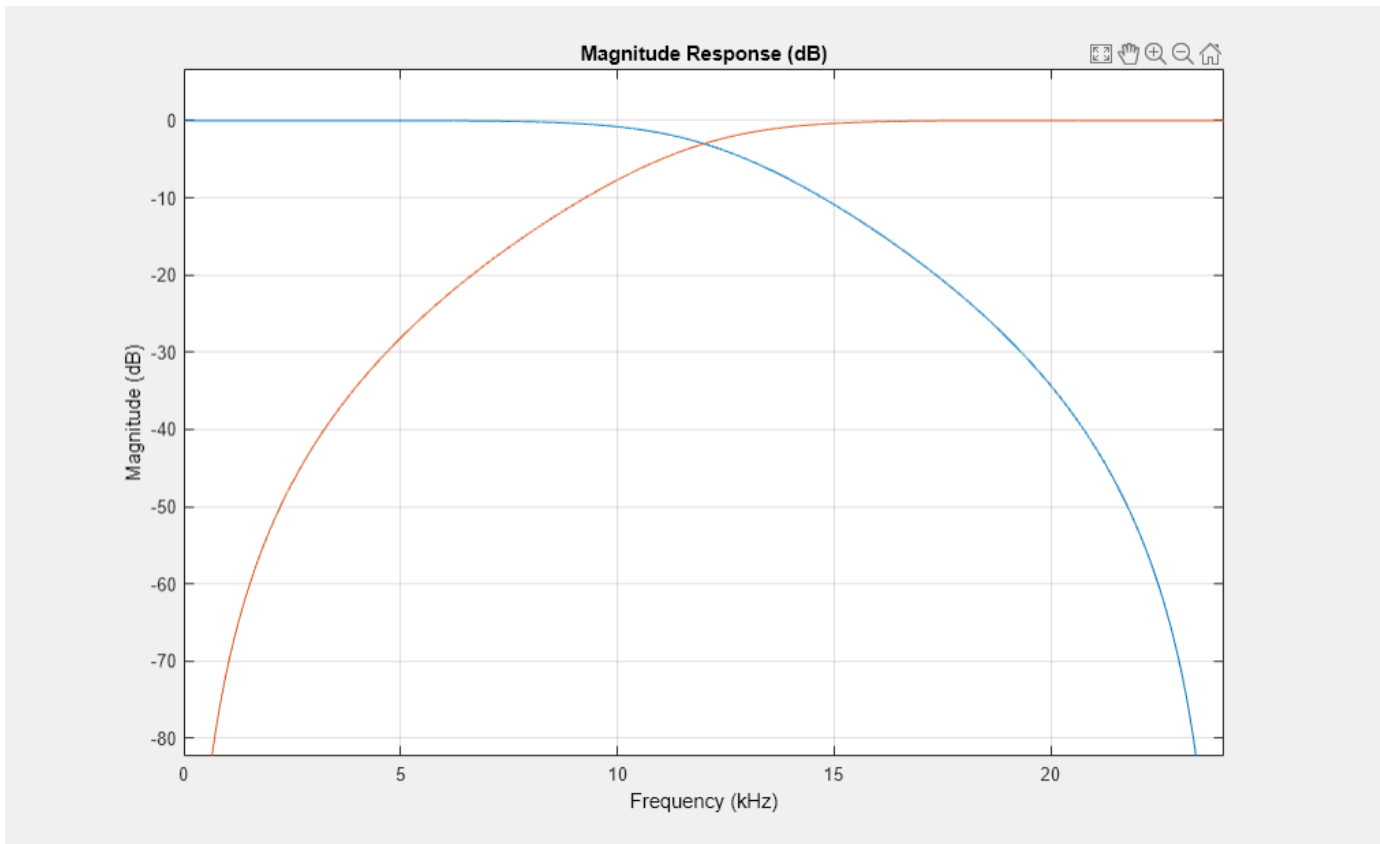
```
fvtool(caf, 'SubbandView', 2, 'Fs', Fs)
```



To view both the outputs, set 'SubbandView' to 'all', [1 2] or [1;2].

```
fvtool(caf, 'SubbandView', 'all', 'Fs', Fs);
```





## Input Arguments

### caf — Input filter

filter System object

Input filter, specified as a `dsp.CoupledAllpassFilter` System object.

Example: `caf = dsp.CoupledAllpassFilter; fvtool(caf)`

### options — Filter analysis options

'magnitude' (default) | 'phase' | 'freq' | 'grpdelay' | 'phasedelay' | 'impulse' | 'step' | 'polezero' | 'coefficients' | 'info' | 'magestimate' | 'noisepower'

Filter analysis options, specified as one of the following:

- 'magnitude' -- Magnitude response
- 'phase' -- Phase response
- 'freq' -- Frequency response
- 'grpdelay' -- Group delay
- 'phasedelay' -- Phase delay
- 'impulse' -- Impulse response
- 'step' -- Step response

- 'polezero' -- Pole zero plot
- 'coefficients' -- Coefficients vector
- 'info' -- Filter information
- 'magemestimate' -- Magnitude response estimate
- 'noisepower' -- Round-off noise power spectrum

Example: `fvtool(caf, 'freq')`

### Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: `caf = dsp.CoupledAllpassFilter;`  
`fvtool(caf, 'SubbandView', 'all', 'Arithmetic', 'single')`

### SubbandView — Subband view

1 (default) | 2 | 'all' | [1 2] | [1;2]

Specify the subband to be viewed. You can set this property to one of the following:

- 1 -- Display the lowpass filter output.
- 2 -- Display the power complimentary highpass filter output.
- 'all', [1 2], [1;2] -- Display both the outputs.

### Fs — Sampling rate

scalar

Sampling rate, specified as a scalar. This value determines the Nyquist interval  $[-Fs/2 \ Fs/2]$  in which FVTool shows the frequency response of the filters in the channelizer.

Data Types: `single` | `double`

### Arithmetic — Arithmetic type

'double' (default) | 'single'

Specify the arithmetic used during analysis. The analysis tool assumes a double-precision filter when the arithmetic input is not specified and the filter System object is unlocked.

## Version History

Introduced in R2013b

### See Also

#### Functions

`getBranches`

#### Tools

**FVTool**

**Objects**`dsp.CoupledAllpassFilter`

## gain

**Package:** dsp

Gain of CIC filter System object

### Syntax

```
g = gain(sysobj)
g = gain(sysobj,j)
```

### Description

`g = gain(sysobj)` returns the gain of the CIC filter System object.

When `sysobj` is a `dsp.CICDecimator` object, the `gain` function returns the gain for the overall CIC decimator.

When `sysobj` is a `dsp.CICInterpolator` object, the `gain` function returns the gain of the  $2N^{\text{th}}$  stage of the CIC interpolation filter, where  $N$  is the number of filter sections. For more details, see `g`.

`g = gain(sysobj,j)` returns the gain of the  $j^{\text{th}}$  section of a CIC interpolation filter. When you omit `j`, the function assumes that `j` is  $2N$ , and returns the gain of the last section of the filter. This syntax does not apply when `sysobj` is a `dsp.CICDecimator` object.

### Examples

#### Compare Performance of Interpolation Filters

To compare the performance of two interpolators, one a CIC filter and the other an FIR filter, use the `gain` function to adjust the CIC filter output amplitude to match the FIR filter output amplitude.

Start by creating an input data set, a sinusoidal signal `x`.

```
fs = 1000;           % Input sampling frequency.
t = (0:1/fs:1.5)';  % Signal length = 1501 samples.
x = sin(2*pi*10*t); % Amplitude = 1 sinusoid.
```

Design a cascade of two `dsp.FIRInterpolator` objects with an overall interpolation factor of 4.

```
l = 4; % Interpolation factor for FIR filter.
firInterp = designMultistageInterpolator(l)
```

```
firInterp =
  dsp.FilterCascade with properties:
    Stage1: [1x1 dsp.FIRInterpolator]
    Stage2: [1x1 dsp.FIRInterpolator]
    CloneStages: false
```

Run the data through the interpolator.

```
yfir = firInterp(x);
```

Design a `dsp.CICInterpolator` object with an interpolation factor set 4, differential delay set to 1, and the number of sections set to 4.

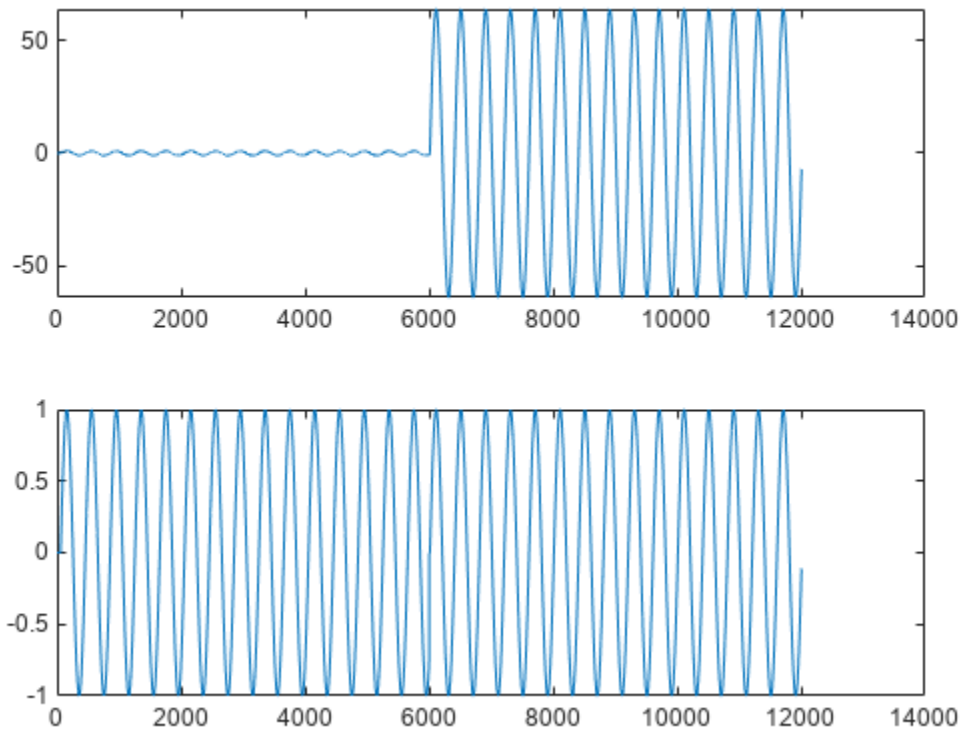
```
r = 4; % Interpolation factor for the CIC filter.
cicInterp = dsp.CICInterpolator(r,1,4);
```

Run the same data through the filter.

```
ycic = cicInterp(x);
```

Use the `gain` function to adjust the CIC filter output amplitude to match the FIR filter output amplitude.

```
gaincic = gain(cicInterp);
subplot(211);
plot([yfir; double(ycic)]);
subplot(212)
plot([yfir; double(ycic)/gain(cicInterp)]);
```

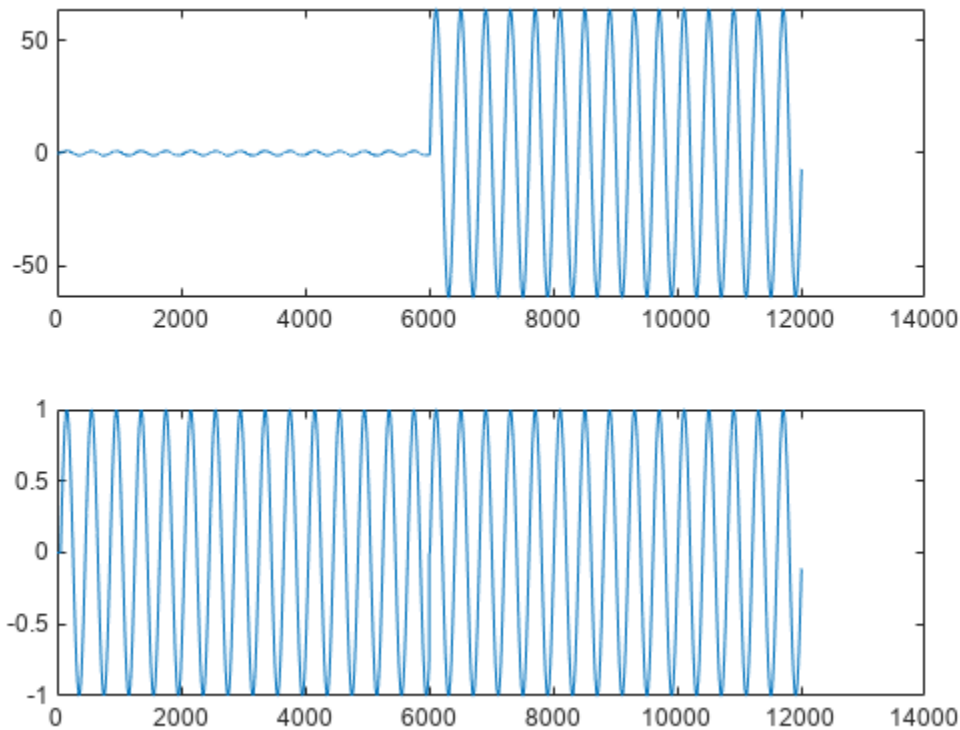


After correcting for the gain induced by the CIC interpolator, in the second subplot you can see that the FIR filter and the CIC filter provide nearly identical interpolation.

This gain equals the gain of the last section of the CIC filter. To confirm, correct the FIR filter amplitude using `gain(cicInterp,2N)`. If  $N$  is the number of integrator and comb sections of the CIC filter, then  $2N$  is the last section of the CIC filter.  $N$  is given by `cicInterp.NumSections`.

The second subplot shows that the FIR filter and CIC filter provide nearly identical interpolation when the correction gain equals the gain of the last section of the CIC filter.

```
subplot(212);
plot([yfir; double(ycic)/gain(cicInterp,2*cicInterp.NumSections)]);
```



## Input Arguments

### **sysobj** — Input CIC filter

CIC filter System object

Input CIC filter, specified as one of the following filter System objects:

- `dsp.CICDecimator`
- `dsp.CICInterpolator`

### **j** — Index of the CIC interpolator stage

positive scalar

Index of the CIC interpolator stage for which the gain is computed, specified as a positive scalar.

Data Types: `single` | `double`

## Output Arguments

### **g** — Gain of CIC filter

scalar

Gain of the CIC filter, returned a scalar. When the input `sysobj` is:

- `dsp.CICDecimator` -- The gain function returns the gain for the overall CIC decimator.
- `dsp.CICInterpolator` -- The CIC interpolator inserts zeros into the input data stream, reducing the filter overall gain by  $1/R$ , where  $R$  is the interpolation factor, to account for the added zero-valued samples. Therefore, the gain of a CIC interpolator is  $(RM)^N/R$ , where  $N$  is the number of filter sections and  $M$  is the filter differential delay. The gain function returns this value.

Data Types: `single` | `double`

## Version History

Introduced in R2011a

## See Also

### Functions

`getFixedPointInfo` | `generatehdl` | `impz` | `freqz` | `phasez` | `fvtool`

# generatehdl

**Package:** dsp

Generate HDL code for quantized DSP filter (requires Filter Design HDL Coder)

## Syntax

```
generatehdl(filtS0, 'InputDataType', nt)
generatehdl(filtS0, 'InputDataType', nt, 'FractionalDelayDataType', fd)
generatehdl(filterObj)

generatehdl( ____, Name, Value)
```

## Description

`generatehdl(filtS0, 'InputDataType', nt)` generates HDL code for the specified filter System object and the input data type, `nt`.

The generated file is a single source file that includes the entity declaration and architecture code. You can find this file in your current working folder, inside the `hdlsrc` subfolder.

`generatehdl(filtS0, 'InputDataType', nt, 'FractionalDelayDataType', fd)` generates HDL code for a `dsp.VariableFractionalDelay` filter System object. Specify the input data type, `nt`, and the fractional delay data type, `fd`.

`generatehdl(filterObj)` generates HDL code for the specified `dfilt` filter object using default settings.

`generatehdl( ____, Name, Value)` uses optional name-value arguments, in addition to the input arguments in previous syntaxes. Use these properties to override default HDL code generation settings.

For more details, see the corresponding properties in the Filter Design HDL Coder documentation:

- To customize filter name, destination folder, and to specify target language, see Fundamental HDL Code Generation Properties.
- To configure coefficients, complex input ports, and optional ports for specific filter types, see HDL Filter Configuration Properties.
- To optimize the speed or area of generated HDL code, see HDL Optimization Properties.
- To customize ports, identifiers, and comments, see HDL Ports and Identifiers Properties.
- To customize HDL constructs, see HDL Constructs Properties.
- To generate and customize test bench, see HDL Test Bench Properties.
- To integrate third-party EDA tools into the filter design workflow, see Synthesis and Workflow Automation Properties.

## Examples



## Generate HDL Code for FIR Equiripple Filter

Design a direct form symmetric equiripple filter with these specifications:

- Normalized passband frequency of 0.2
- Normalized stopband frequency of 0.22
- Passband ripple of 1 dB
- Stopband attenuation of 60 dB

The design function returns a `dsp.FIRFilter` System object™ that implements the specification.

```
filtSpecs = fdesign.lowpass('Fp,Fst,Ap,Ast',0.2,0.22,1,60);
FIRe = design(filtSpecs,'equiripple','FilterStructure','dfsymfir','SystemObject',true)

FIRe =
    dsp.FIRFilter with properties:

        Structure: 'Direct form symmetric'
    NumeratorSource: 'Property'
        Numerator: [-0.0011 -0.0017 -0.0025 -0.0031 -0.0031 -0.0024 -9.7703e-04 0.0010 0.0031]
    InitialConditions: 0

    Show all properties
```

To generate HDL code, call the `generatehdl` function. When the filter is a System object, you must specify a fixed-point data type for the input using the “InputDataType” (Filter Design HDL Coder) property. The coder generates the file `MyFilter.vhd` in the default target folder, `hdlsrc`.

```
generatehdl(FIRe,'InputDataType',numeric(1,16,15),'Name','MyFilter');

### Starting VHDL code generation process for filter: MyFilter
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex48836167\hdlsrc\
### Starting generation of MyFilter VHDL entity
### Starting generation of MyFilter VHDL architecture
### Successful completion of VHDL code generation process for filter: MyFilter
### HDL latency is 2 samples
```

## Generate HDL Code and Test Bench for FIR Equiripple Filter

Design a direct form symmetric equiripple filter with these specifications:

- Normalized passband frequency of 0.2
- Normalized stopband frequency of 0.22
- Passband ripple of 1 dB
- Stopband attenuation of 60 dB

The design function returns a `dsp.FIRFilter` System object™ that implements the specification.

```
filtSpecs = fdesign.lowpass('Fp,Fst,Ap,Ast',0.2,0.22,1,60);
FIRe = design(filtSpecs,'equiripple','FilterStructure','dfsymfir','SystemObject',true)

FIRe =
    dsp.FIRFilter with properties:
```

```

        Structure: 'Direct form symmetric'
    NumeratorSource: 'Property'
        Numerator: [-0.0011 -0.0017 -0.0025 -0.0031 -0.0031 -0.0024 -9.7703e-04 0.0010 0.0031]
    InitialConditions: 0

```

Show all properties

Generate VHDL code and a VHDL test bench for the FIR equiripple filter. When the filter is a System object, you must specify a fixed-point data type for the input data type. The coder generates the files `MyFilter.vhd` and `MyFilterTB.vhd` in the default target folder, `hdlsrc`.

```

generatehdl(FIR,'InputDataType',numerictype(1,16,15),'Name','MyFilter',...
    'GenerateHDLTestbench','on','TestBenchName','MyFilterTB')

### Starting VHDL code generation process for filter: MyFilter
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex63281302\hdlsrc\
### Starting generation of MyFilter VHDL entity
### Starting generation of MyFilter VHDL architecture
### Successful completion of VHDL code generation process for filter: MyFilter
### HDL latency is 2 samples
### Starting generation of VHDL Test Bench.
### Generating input stimulus
### Done generating input stimulus; length 4486 samples.
### Generating Test bench: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex63281302\hdlsrc\
### Creating stimulus vectors ...
### Done generating VHDL Test Bench.

```

### Generate HDL Code for Fully Parallel FIR Filter with Programmable Coefficients

Design a direct form symmetric equiripple filter with fully parallel (default) architecture and programmable coefficients. The `design` function returns a `dsp.FIRFilter` System object™ with default lowpass filter specification.

```
firfilt = design(fdesign.lowpass,'equiripple','FilterStructure','dfsymfir','SystemObject',true)
```

```

firfilt =
    dsp.FIRFilter with properties:
        Structure: 'Direct form symmetric'
    NumeratorSource: 'Property'
        Numerator: [-0.0024 -0.0021 0.0068 0.0167 0.0111 -0.0062 -0.0084 0.0093 0.0130 -0.0130]
    InitialConditions: 0

```

Show all properties

To generate HDL code, call the `generatehdl` function. When the filter is a System object, you must specify a fixed-point data type for the input data. To generate a processor interface for the coefficients, you must specify an additional name-value pair argument.

```

generatehdl(firfilt,'InputDataType',numerictype(1,16,15),'CoefficientSource','ProcessorInterface')

### Starting VHDL code generation process for filter: firfilt
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex74213987\hdlsrc\

```

```

### Starting generation of firfilt VHDL entity
### Starting generation of firfilt VHDL architecture
### Successful completion of VHDL code generation process for filter: firfilt
### HDL latency is 2 samples

```

The coder generates this VHDL entity for the filter object.

```

ENTITY firfilt IS
  PORT( clk      : IN      std_logic;
        clk_enable : IN      std_logic;
        reset     : IN      std_logic;
        filter_in  : IN      std_logic_vector(15 DOWNTO 0); -- sfix16_En15
        write_enable : IN      std_logic;
        write_done  : IN      std_logic;
        write_address : IN      std_logic_vector(4 DOWNTO 0); -- ufix5
        coeffs_in   : IN      std_logic_vector(15 DOWNTO 0); -- sfix16_En16
        filter_out  : OUT     std_logic_vector(36 DOWNTO 0) -- sfix37_En31
        );
END firfilt;

```

### Generate Partly Serial FIR Filter with Programmable Coefficients

Create a direct form antisymmetric filter with coefficients:

```

coeffs = fir1(22,0.45);
firfilt = dsp.FIRFilter('Numerator',coeffs,'Structure','Direct form antisymmetric')

```

```

firfilt =
  dsp.FIRFilter with properties:
      Structure: 'Direct form antisymmetric'
      NumeratorSource: 'Property'
      Numerator: [3.6133e-04 0.0031 8.4473e-04 -0.0090 -0.0072 0.0203 0.0272 -0.0341 -0.07
      InitialConditions: 0

```

Show all properties

To generate HDL code, call the `generatehdl` function. When the filter is a System object, you must specify a fixed-point data type for the input data. To generate a partly serial architecture, specify a serial partition. To enable `CoefficientMemory` property, you must set `CoefficientSource` to `ProcessorInterface`.

```

generatehdl(firfilt,'InputDataType',numerictype(1,16,15), ...
  'SerialPartition',[7 4],'CoefficientMemory','DualPortRAMs', ...
  'CoefficientSource','ProcessorInterface')

```

```

### Starting VHDL code generation process for filter: firfilt
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex21465785\hdlsrc\
### Starting generation of firfilt VHDL entity
### Starting generation of firfilt VHDL architecture
### Clock rate is 7 times the input sample rate for this architecture.

```

```
### Successful completion of VHDL code generation process for filter: firfilt
### HDL latency is 3 samples
```

The generated code includes a dual-port RAM interface for the programmable coefficients.

```
ENTITY firfilt IS
  PORT ( clk          : IN      std_logic;
         clk_enable   : IN      std_logic;
         reset        : IN      std_logic;
         filter_in    : IN      std_logic_vector(15 DOWNT0 0); -- sfix16_En15
         write_enable  : IN      std_logic;
         write_done    : IN      std_logic;
         write_address : IN      std_logic_vector(3 DOWNT0 0); -- ufix4
         coeffs_in    : IN      std_logic_vector(15 DOWNT0 0); -- sfix16_En16
         filter_out    : OUT     std_logic_vector(35 DOWNT0 0) -- sfix36_En31
        );
END firfilt;
```

### Generate Serial Partitions for FIR Filter

Explore clock rate and latency for different serial implementations of the same filter. Using a symmetric structure also allows the filter logic to share multipliers for symmetric coefficients.

Create a direct form symmetric FIR filter with these specifications:

- Filter order 13
- Normalized cut-off frequency of 0.4 for the 6-dB point

The design function returns a `dsp.FIRFilter` System object™ that implements the specification.

```
FIR = design(fdesign.lowpass('N,Fc',13,.4), 'FilterStructure', 'dfsymfir', 'SystemObject', true)
```

```
FIR =
```

```
  dsp.FIRFilter with properties:
```

```
      Structure: 'Direct form symmetric'
      NumeratorSource: 'Property'
      Numerator: [0.0037 0.0045 -0.0115 -0.0417 1.0911e-17 0.1776 0.3674 0.3674 0.1776 1.0911e-17 -0.0417 -0.0115 0.0045 0.0037]
      InitialConditions: 0
```

```
Show all properties
```

To generate HDL code, call the `generatehdl` function. When the filter is a System object, you must specify a fixed-point data type for the input data.

For a baseline comparison, first generate a default fully parallel architecture.

```
generatehdl(FIR, 'Name', 'FullyParallel', ...
            'InputDataType', numerictype(1,16,15))

### Starting VHDL code generation process for filter: FullyParallel
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex94948885\hdlsrc\
```

```
### Starting generation of FullyParallel VHDL entity
### Starting generation of FullyParallel VHDL architecture
### Successful completion of VHDL code generation process for filter: FullyParallel
### HDL latency is 2 samples
```

Generate a fully serial architecture by setting the partition size to the effective filter length. The system clock rate is six times the input sample rate. The reported HDL latency is one sample greater than the default parallel implementation.

```
generatehdl(FIR, 'SerialPartition', 6, 'Name', 'FullySerial', ...
            'InputDataType', numeric_type(1, 16, 15))
```

```
### Starting VHDL code generation process for filter: FullySerial
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex94948885\hdlsrc\
### Starting generation of FullySerial VHDL entity
### Starting generation of FullySerial VHDL architecture
### Clock rate is 6 times the input sample rate for this architecture.
### Successful completion of VHDL code generation process for filter: FullySerial
### HDL latency is 3 samples
```

Generate a partly serial architecture with three equal partitions. This architecture uses three multipliers. The clock rate is two times the input rate, and the latency is the same as the default parallel implementation.

```
generatehdl(FIR, 'SerialPartition', [2 2 2], 'Name', 'PartlySerial', ...
            'InputDataType', numeric_type(1, 16, 15))
```

```
### Starting VHDL code generation process for filter: PartlySerial
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex94948885\hdlsrc\
### Starting generation of PartlySerial VHDL entity
### Starting generation of PartlySerial VHDL architecture
### Clock rate is 2 times the input sample rate for this architecture.
### Successful completion of VHDL code generation process for filter: PartlySerial
### HDL latency is 3 samples
```

Generate a cascade-serial architecture by enabling accumulator reuse. Specify the three partitions in descending order of size. Notice that the clock rate is higher than the rate in the partly serial (without accumulator reuse) example.

```
generatehdl(FIR, 'SerialPartition', [3 2 1], 'ReuseAccum', 'on', 'Name', 'CascadeSerial', ...
            'InputDataType', numeric_type(1, 16, 15))
```

```
### Starting VHDL code generation process for filter: CascadeSerial
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex94948885\hdlsrc\
### Starting generation of CascadeSerial VHDL entity
### Starting generation of CascadeSerial VHDL architecture
### Clock rate is 4 times the input sample rate for this architecture.
### Successful completion of VHDL code generation process for filter: CascadeSerial
### HDL latency is 3 samples
```

You can also generate a cascade-serial architecture without specifying the partitions explicitly. The coder automatically selects partition sizes.

```
generatehdl(FIR, 'ReuseAccum', 'on', 'Name', 'CascadeSerial', ...
            'InputDataType', numeric_type(1, 16, 15))
```

```
### Starting VHDL code generation process for filter: CascadeSerial
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex94948885\hdlsrc\
```

```

### Starting generation of CascadeSerial VHDL entity
### Starting generation of CascadeSerial VHDL architecture
### Clock rate is 4 times the input sample rate for this architecture.
### Serial partition # 1 has 3 inputs.
### Serial partition # 2 has 3 inputs.
### Successful completion of VHDL code generation process for filter: CascadeSerial
### HDL latency is 3 samples

```

### Generate Serial Partitions of Cascaded Filter

Create a two-stage cascaded filter with these specifications for each filter stage:

- Direct form symmetric FIR filter
- Filter order 8
- Normalized cut-off frequency of 0.4 for the 6-dB point

Each call of the `design` function returns a `dsp.FIRFilter` System object™ that implements the specification. The `cascade` function returns a two-stage cascaded filter.

```

lp = design(fdesign.lowpass('N,Fc',8,.4),'FilterStructure','dfsymfir','SystemObject',true)
lp =
    dsp.FIRFilter with properties:
        Structure: 'Direct form symmetric'
        NumeratorSource: 'Property'
        Numerator: [-0.0061 -0.0136 0.0512 0.2657 0.4057 0.2657 0.0512 -0.0136 -0.0061]
        InitialConditions: 0
    Show all properties

hp = design(fdesign.highpass('N,Fc',8,.4),'FilterStructure','dfsymfir','SystemObject',true)
hp =
    dsp.FIRFilter with properties:
        Structure: 'Direct form symmetric'
        NumeratorSource: 'Property'
        Numerator: [0.0060 0.0133 -0.0501 -0.2598 0.5951 -0.2598 -0.0501 0.0133 0.0060]
        InitialConditions: 0
    Show all properties

casc = cascade(lp, hp)
casc =
    dsp.FilterCascade with properties:
        Stage1: [1x1 dsp.FIRFilter]
        Stage2: [1x1 dsp.FIRFilter]
        CloneStages: true

```

To generate HDL code, call the `generatehdl` function for the cascaded filter. When the filter is a System object, you must specify a fixed-point data type for the input data.

Specify different partitions for each cascade stage as a cell array.

```
generatehdl(casc, 'InputDataType', numerictype(1,16,15), 'SerialPartition', {[3 2],[4 1]})
```

```
### Starting VHDL code generation process for filter: casfilt
### Cascade stage # 1
### Starting VHDL code generation process for filter: casfilt_stage1
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex16715237\hdlsrc\
### Starting generation of casfilt_stage1 VHDL entity
### Starting generation of casfilt_stage1 VHDL architecture
### Clock rate is 3 times the input sample rate for this architecture.
### Successful completion of VHDL code generation process for filter: casfilt_stage1
### Cascade stage # 2
### Starting VHDL code generation process for filter: casfilt_stage2
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex16715237\hdlsrc\
### Starting generation of casfilt_stage2 VHDL entity
### Starting generation of casfilt_stage2 VHDL architecture
### Clock rate is 4 times the input sample rate for this architecture.
### Successful completion of VHDL code generation process for filter: casfilt_stage2
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex16715237\hdlsrc\
### Starting generation of casfilt VHDL entity
### Starting generation of casfilt VHDL architecture
### Successful completion of VHDL code generation process for filter: casfilt
### HDL latency is 2 samples
```

To explore the effective filter length and partitioning options for each filter stage of a cascade, call the `hdlfilterserialinfo` function. The function returns a partition vector corresponding to a desired number of multipliers. Request serial partition possibilities for the first stage, and choose a number of multipliers.

```
hdlfilterserialinfo(casc.Stage1, 'InputDataType', numerictype(1,16,15))
```

| Total Coefficients | Zeros | A/Symm | Effective |
|--------------------|-------|--------|-----------|
| 9                  | 0     | 4      | 5         |

Effective filter length for SerialPartition value is 5.

Table of 'SerialPartition' values with corresponding values of folding factor and number of multipliers for the given filter.

| Folding Factor | Multipliers | SerialPartition |
|----------------|-------------|-----------------|
| 1              | 5           | [1 1 1 1 1]     |
| 2              | 3           | [2 2 1]         |
| 3              | 2           | [3 2]           |
| 4              | 2           | [4 1]           |
| 5              | 1           | [5]             |

Select a serial partition vector for a target of two multipliers, and pass the vectors to the `generatehdl` function. Calling the function this way returns the first possible partition vector, but there are multiple partition vectors that achieve a two-multiplier architecture. Each stage uses a different clock rate based on the number of multipliers. The coder generates a timing controller to derive these clocks.

```

sp1 = hdlfilterserialinfo(casc.Stage1, 'InputDataType', numerictype(1,16,15), 'Multiplier', 2)
sp1 = 1×2
      3      2

sp2 = hdlfilterserialinfo(casc.Stage2, 'InputDataType', numerictype(1,16,15), 'Multiplier', 3)
sp2 = 1×3
      2      2      1

generatehdl(casc, 'InputDataType', numerictype(1,16,15), 'SerialPartition', {sp1,sp2})
### Starting VHDL code generation process for filter: casfilt
### Cascade stage # 1
### Starting VHDL code generation process for filter: casfilt_stage1
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex16715237\hdlsrc\
### Starting generation of casfilt_stage1 VHDL entity
### Starting generation of casfilt_stage1 VHDL architecture
### Clock rate is 3 times the input sample rate for this architecture.
### Successful completion of VHDL code generation process for filter: casfilt_stage1
### Cascade stage # 2
### Starting VHDL code generation process for filter: casfilt_stage2
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex16715237\hdlsrc\
### Starting generation of casfilt_stage2 VHDL entity
### Starting generation of casfilt_stage2 VHDL architecture
### Clock rate is 2 times the input sample rate for this architecture.
### Successful completion of VHDL code generation process for filter: casfilt_stage2
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex16715237\hdlsrc\
### Starting generation of casfilt VHDL entity
### Starting generation of casfilt VHDL architecture
### Successful completion of VHDL code generation process for filter: casfilt
### HDL latency is 2 samples

```

### Generate Serial Architectures for IIR Filter

Create a direct form I SOS filter with these specifications:

- Sampling frequency of 48 kHz
- Filter order 5
- Cut-off frequency of 10.8 kHz for the 3 dB point

The design function returns a `dsp.BiquadFilter` System object™ that implements the specification. The custom accumulator data type avoids quantization error.

```

Fs = 48e3;
Fc = 10.8e3;
N = 5;
lp = design(fdesign.lowpass('n,f3db',N,Fc,Fs), 'butter', ...
    'FilterStructure', 'df1sos', 'SystemObject', true, 'UseLegacyBiquadFilter', true)
lp =
    dsp.BiquadFilter with properties:

```



```

        Structure: 'Direct form I'
        SOSMatrixSource: 'Property'
        SOSMatrix: [3x6 double]
        ScaleValues: [4x1 double]
        NumeratorInitialConditions: 0
        DenominatorInitialConditions: 0
        OptimizeUnityScaleValues: true

```

Show all properties

```

nt_accum = numericitytype('Signedness','auto','WordLength',20, ...
    'FractionLength',15);
nt_input = numericitytype(1,16,15);
lp.NumeratorAccumulatorDataType = 'Custom';
lp.CustomNumeratorAccumulatorDataType = nt_accum;
lp.DenominatorAccumulatorDataType = 'Custom';
lp.CustomDenominatorAccumulatorDataType = nt_accum;

```

To list all possible serial architecture specifications for this filter, call the `hdlfilterserialinfo` function. When the filter is a System object, you must specify a fixed-point data type for the input data.

```
hdlfilterserialinfo(lp,'InputDataType',nt_input)
```

Table of folding factors with corresponding number of multipliers for the given filter.

| Folding factor | Multipliers |
|----------------|-------------|
| 6              | 3           |
| 9              | 2           |
| 18             | 1           |

To generate HDL code, call the `generatehdl` function with one of the serial architectures. Specify either the `NumMultipliers` or `FoldingFactor` property, but not both. For instance, using the `NumMultipliers` property:

```
generatehdl(lp,'NumMultipliers',2,'InputDataType',nt_input)
```

```

### Starting VHDL code generation process for filter: lp
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex90334139\hdlsrc\
### Starting generation of lp VHDL entity
### Starting generation of lp VHDL architecture
### Successful completion of VHDL code generation process for filter: lp
### HDL latency is 2 samples

```

Alternatively, specify the same architecture with the `FoldingFactor` property.

```
generatehdl(lp,'FoldingFactor',9,'InputDataType',nt_input)
```

```

### Starting VHDL code generation process for filter: lp
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex90334139\hdlsrc\
### Starting generation of lp VHDL entity
### Starting generation of lp VHDL architecture
### Successful completion of VHDL code generation process for filter: lp
### HDL latency is 2 samples

```

Both these commands generate a filter that uses a total of two multipliers, with a latency of nine clock cycles. This architecture uses less area than the parallel implementation, at the expense of latency.

### Distributed Arithmetic for Single Rate Filters

Use distributed arithmetic options to reduce the number of multipliers in the filter implementation.

Create a direct-form FIR filter and calculate the filter length, FL.

```
filtdes = fdesign.lowpass('N,Fc,Ap,Ast',30,0.4,0.05,0.03,'linear');
firfilt = design(filtdes,'FilterStructure','dffir','SystemObject',true);
FL = length(find(firfilt.Numerator ~= 0))
```

```
FL = 31
```

Specify a set of partitions such that the partition sizes add up to the filter length. This is just one partition option, you can specify other combinations of sizes.

```
generatehdl(firfilt,'InputDataType',numeric(1,16,15), ...
'DALUTPartition',[8 8 8 7])
```

```
### Structure fir has symmetric coefficients, consider converting to structure symmetricfir for
### Starting VHDL code generation process for filter: firfilt
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex00198568\hdlsrc\
### Starting generation of firfilt VHDL entity
### Starting generation of firfilt VHDL architecture
### Clock rate is 16 times the input sample rate for this architecture.
### Successful completion of VHDL code generation process for filter: firfilt
### HDL latency is 3 samples
```

For comparison, create a direct-form symmetric FIR filter. The filter length is smaller in the symmetric case.

```
filtdes = fdesign.lowpass('N,Fc,Ap,Ast',30,0.4,0.05,0.03,'linear');
firfilt = design(filtdes,'FilterStructure','dfsymfir','SystemObject',true);
FL = ceil(length(find(firfilt.Numerator ~= 0))/2)
```

```
FL = 16
```

Specify a set of partitions such that the partition sizes add up to the filter length. This is just one partition option, you can specify other combinations of sizes. **Tip:** Use the `hdlfilterdainfo` function to display the effective filter length, LUT partitioning options, and possible DARadix values for a filter.

```
generatehdl(firfilt,'InputDataType',numeric(1,16,15), ...
'DALUTPartition',[8 8])
```

```
### Starting VHDL code generation process for filter: firfilt
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex00198568\hdlsrc\
### Starting generation of firfilt VHDL entity
### Starting generation of firfilt VHDL architecture
### Clock rate is 17 times the input sample rate for this architecture.
### Successful completion of VHDL code generation process for filter: firfilt
### HDL latency is 3 samples
```

## Distributed Arithmetic for Multirate Filters

Use distributed arithmetic options to reduce the number of multipliers in the filter implementation.

Create a direct-form FIR polyphase decimator, and calculate the filter length.

```
d = fdesign.decimator(4);
filt = design(d, 'SystemObject', true);
FL = size(polyphase(filt), 2)
```

```
FL = 27
```

Specify distributed arithmetic LUT partitions that add up to the filter size. When you specify partitions as a vector for a polyphase filter, each subfilter uses the same partitions.

```
generatehdl(filt, 'InputDataType', numerictype(1, 16, 15), ...
    'DALUTPartition', [8 8 8 3])
```

```
### Starting VHDL code generation process for filter: firdecim
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex51670151\hdlsrc\
### Starting generation of firdecim VHDL entity
### Starting generation of firdecim VHDL architecture
### Clock rate is 4 times the input and 16 times the output sample rate for this architecture.
### Successful completion of VHDL code generation process for filter: firdecim
### HDL latency is 16 samples
```

You can also specify unique partitions for each subfilter. For the same filter, specify subfilter partitioning as a matrix. The length of the first subfilter is 1, and the other subfilters have length 26.

**Tip:** Use the `hdlfilterdainfo` function to display the effective filter length, LUT partitioning options, and possible DARadix values for a filter.

```
d = fdesign.decimator(4);
filt = design(d, 'SystemObject', true);
generatehdl(filt, 'InputDataType', numerictype(1, 16, 15), ...
    'DALUTPartition', [1 0 0 0; 8 8 8 2; 8 8 6 4; 8 8 8 2])
```

```
### Starting VHDL code generation process for filter: firdecim
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex51670151\hdlsrc\
### Starting generation of firdecim VHDL entity
### Starting generation of firdecim VHDL architecture
### Clock rate is 4 times the input and 16 times the output sample rate for this architecture.
### Successful completion of VHDL code generation process for filter: firdecim
### HDL latency is 16 samples
```

## Distributed Arithmetic for Cascaded Filters

Use distributed arithmetic options to reduce the number of multipliers in the filter implementation.

### Create Cascaded Filter

Create a two-stage cascaded filter. Define different LUT partitions for each stage, and specify the partition vectors in a cell array.

```

lp = design(fdesign.lowpass('N,Fc',8,.4), 'filterstructure', 'dfsymfir', ...
  'SystemObject', true);
hp = design(fdesign.highpass('N,Fc',8,.4), 'filterstructure', 'dfsymfir', ...
  'SystemObject', true);
casc = cascade(lp, hp);
nt1 = numerictype(1,12,10);
generatehdl(casc, 'InputDataType', nt1, 'DALUTPartition', {[3 2],[2 2 1]})

### Starting VHDL code generation process for filter: casfilt
### Cascade stage # 1
### Starting VHDL code generation process for filter: casfilt_stage1
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex17169310\hdlsrc\
### Starting generation of casfilt_stage1 VHDL entity
### Starting generation of casfilt_stage1 VHDL architecture
### Clock rate is 13 times the input sample rate for this architecture.
### Successful completion of VHDL code generation process for filter: casfilt_stage1
### Cascade stage # 2
### Starting VHDL code generation process for filter: casfilt_stage2
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex17169310\hdlsrc\
### Starting generation of casfilt_stage2 VHDL entity
### Starting generation of casfilt_stage2 VHDL architecture
### Clock rate is 29 times the input sample rate for this architecture.
### Successful completion of VHDL code generation process for filter: casfilt_stage2
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex17169310\hdlsrc\
### Starting generation of casfilt VHDL entity
### Starting generation of casfilt VHDL architecture
### Successful completion of VHDL code generation process for filter: casfilt
### HDL latency is 4 samples

```

### Distributed Arithmetic Options

Use the `hdlfilterdainfo` function to display the effective filter length, LUT partitioning options, and possible DARadix values for each filter stage of a cascade. The function returns a LUT partition vector corresponding to a desired number of address bits.

Request LUT partition possibilities for the first stage.

```
hdlfilterdainfo(casc.Stage1, 'InputDataType', nt1);
```

| Total Coefficients | Zeros | A/Symm | Effective |
|--------------------|-------|--------|-----------|
| 9                  | 0     | 4      | 5         |

Effective filter length for SerialPartition value is 5.

Table of 'DARadix' values with corresponding values of folding factor and multiple for LUT sets for the given filter.

| Folding Factor | LUT-Sets Multiple | DARadix         |
|----------------|-------------------|-----------------|
| 1              | 12                | 2 <sup>12</sup> |
| 3              | 6                 | 2 <sup>6</sup>  |
| 4              | 4                 | 2 <sup>4</sup>  |
| 5              | 3                 | 2 <sup>3</sup>  |
| 7              | 2                 | 2 <sup>2</sup>  |
| 13             | 1                 | 2 <sup>1</sup>  |

Details of LUTs with corresponding 'DALUTPartition' values.

| Max Address Width | Size(bits) | LUT Details           | DALUTPartition |
|-------------------|------------|-----------------------|----------------|
| 5                 | 416        | 1x32x13               | [5]            |
| 4                 | 216        | 1x16x12, 1x2x12       | [4 1]          |
| 3                 | 124        | 1x4x13, 1x8x9         | [3 2]          |
| 2                 | 104        | 1x2x12, 1x4x12, 1x4x8 | [2 2 1]        |

Notes:

- LUT Details indicates number of LUTs with their sizes. e.g. 1x1024x18 implies 1 LUT of 1024 18-bit wide locations.

To request LUT partition possibilities for the second stage, you must first determine the input data type of the second stage.

```
y = casc.Stage1(fi(0,nt1));
nt2 = y.numericType;
hdlfilterdainfo(casc.Stage2,'InputDataType',nt2);
```

| Total Coefficients | Zeros | A/Symm | Effective |
|--------------------|-------|--------|-----------|
| 9                  | 0     | 4      | 5         |

Effective filter length for SerialPartition value is 5.

Table of 'DARadix' values with corresponding values of folding factor and multiple for LUT sets for the given filter.

| Folding Factor | LUT-Sets Multiple | DARadix         |
|----------------|-------------------|-----------------|
| 1              | 28                | 2 <sup>28</sup> |
| 3              | 14                | 2 <sup>14</sup> |
| 5              | 7                 | 2 <sup>7</sup>  |
| 8              | 4                 | 2 <sup>4</sup>  |
| 15             | 2                 | 2 <sup>2</sup>  |
| 29             | 1                 | 2 <sup>1</sup>  |

Details of LUTs with corresponding 'DALUTPartition' values.

| Max Address Width | Size(bits) | LUT Details            | DALUTPartition |
|-------------------|------------|------------------------|----------------|
| 5                 | 896        | 1x32x28                | [5]            |
| 4                 | 488        | 1x16x27, 1x2x28        | [4 1]          |
| 3                 | 304        | 1x4x28, 1x8x24         | [3 2]          |
| 2                 | 256        | 1x2x28, 1x4x23, 1x4x27 | [2 2 1]        |

Notes:

- LUT Details indicates number of LUTs with their sizes. e.g. 1x1024x18 implies 1 LUT of 1024 18-bit wide locations.

### Different LUT Partitions for Each Stage

Select address widths and folding factors to obtain LUT partition for each stage. The first stage uses LUTs with a maximum address size of five bits. The second stage uses LUTs with a maximum address size of three bits. They run at the same clock rate, and have different LUT partitions.

```
dp1 = hdlfilterdainfo(casc.Stage1,'InputDataType',nt1, ...
    'LUTInputs',5,'FoldingFactor',3);
```

```

dp2 = hdlfilterdainfo(casc.Stage2,'InputDataType',nt1, ...
    'LUTInputs',3,'FoldingFactor',5);
generatehdl(casc,'InputDataType',nt1,'DALUTPartition',{dp1,dp2});

### Starting VHDL code generation process for filter: casfilt
### Cascade stage # 1
### Starting VHDL code generation process for filter: casfilt_stage1
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex17169310\hdlsrc\
### Starting generation of casfilt_stage1 VHDL entity
### Starting generation of casfilt_stage1 VHDL architecture
### Clock rate is 13 times the input sample rate for this architecture.
### Successful completion of VHDL code generation process for filter: casfilt_stage1
### Cascade stage # 2
### Starting VHDL code generation process for filter: casfilt_stage2
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex17169310\hdlsrc\
### Starting generation of casfilt_stage2 VHDL entity
### Starting generation of casfilt_stage2 VHDL architecture
### Clock rate is 29 times the input sample rate for this architecture.
### Successful completion of VHDL code generation process for filter: casfilt_stage2
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex17169310\hdlsrc\
### Starting generation of casfilt VHDL entity
### Starting generation of casfilt VHDL architecture
### Successful completion of VHDL code generation process for filter: casfilt
### HDL latency is 4 samples

```

### Different DARadix Values for Each Stage

You can also specify different DARadix values for each filter in a cascade. You can only specify different cascade partitions on the command-line. When you specify partitions in the **Generate HDL** dialog box, all cascade stages use the same partitions. Inspect the results of `hdlfilterdainfo` to set DARadix values for each stage.

```

generatehdl(casc,'InputDataType',nt1, ...
    'DALUTPartition',{[3 2],[2 2 1]},'DARadix',{2^3,2^7})

### Starting VHDL code generation process for filter: casfilt
### Cascade stage # 1
### Starting VHDL code generation process for filter: casfilt_stage1
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex17169310\hdlsrc\
### Starting generation of casfilt_stage1 VHDL entity
### Starting generation of casfilt_stage1 VHDL architecture
### Clock rate is 5 times the input sample rate for this architecture.
### Successful completion of VHDL code generation process for filter: casfilt_stage1
### Cascade stage # 2
### Starting VHDL code generation process for filter: casfilt_stage2
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex17169310\hdlsrc\
### Starting generation of casfilt_stage2 VHDL entity
### Starting generation of casfilt_stage2 VHDL architecture
### Clock rate is 5 times the input sample rate for this architecture.
### Successful completion of VHDL code generation process for filter: casfilt_stage2
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex17169310\hdlsrc\
### Starting generation of casfilt VHDL entity
### Starting generation of casfilt VHDL architecture
### Successful completion of VHDL code generation process for filter: casfilt
### HDL latency is 4 samples

```

## Cascaded Filter with Multiple Architectures

Specify different filter architectures for the different stages of a cascaded filter. You can specify a mix of serial, distributed arithmetic (DA), and parallel architectures depending upon your hardware constraints.

### Create Cascaded Filter

Create a three-stage filter. Each stage is a different type.

```
h1 = dsp.FIRFilter('Numerator',[0.05 -.25 .88 0.9 .88 -.25 0.05]);
h2 = dsp.FIRFilter('Numerator',[-0.008 0.06 -0.44 0.44 -0.06 0.008], ...
    'Structure','Direct form antisymmetric');
h3 = dsp.FIRFilter('Numerator',[-0.008 0.06 0.44 0.44 0.06 -0.008], ...
    'Structure','Direct form symmetric');
casc = cascade(h1,h2,h3);
```

### Specify Architecture for Each Stage

Specify a DA architecture for the first stage, a serial architecture for the second stage, and a fully parallel (default) architecture for the third stage.

To obtain DARadix values for the first architecture, use `hdlfilterdainfo`, then pick a value from `dr`.

```
nt = numerictype(1,12,10);
[dp,dr,lutsize,ff] = hdlfilterdainfo(casc.Stage1, ...
    'InputDataType',numerictype(1,12,10));
dr

dr = 6x1 cell
    {'2^12'}
    {'2^6' }
    {'2^4' }
    {'2^3' }
    {'2^2' }
    {'2^1' }
```

Set the property values as cell arrays, where each cell applies to a stage. To disable a property for a particular stage, use default values (-1 for the partitions and 2 for DARadix).

```
generatehdl(casc,'InputDataType',nt, ...
    'SerialPartition',{-1,3,-1}, ...
    'DALUTPartition',{[4 3],-1,-1}, ...
    'DARadix',{2^6,2,2});

### Structure fir has symmetric coefficients, consider converting to structure symmetricfir for
### Starting VHDL code generation process for filter: casfilt
### Cascade stage # 1
### Starting VHDL code generation process for filter: casfilt_stage1
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex13094988\hdlsrc\
### Starting generation of casfilt_stage1 VHDL entity
### Starting generation of casfilt_stage1 VHDL architecture
### Clock rate is 2 times the input sample rate for this architecture.
### Successful completion of VHDL code generation process for filter: casfilt_stage1
### Cascade stage # 2
### Starting VHDL code generation process for filter: casfilt_stage2
```

```

### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex13094988\hdlsrc\
### Starting generation of casfilt_stage2 VHDL entity
### Starting generation of casfilt_stage2 VHDL architecture
### Clock rate is 3 times the input sample rate for this architecture.
### Successful completion of VHDL code generation process for filter: casfilt_stage2
### Cascade stage # 3
### Starting VHDL code generation process for filter: casfilt_stage3
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex13094988\hdlsrc\
### Starting generation of casfilt_stage3 VHDL entity
### Starting generation of casfilt_stage3 VHDL architecture
### Successful completion of VHDL code generation process for filter: casfilt_stage3
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex13094988\hdlsrc\
### Starting generation of casfilt VHDL entity
### Starting generation of casfilt VHDL architecture
### Successful completion of VHDL code generation process for filter: casfilt
### HDL latency is 3 samples

```

### Test Bench for FIR Filter with Programmable Coefficients

You can specify input coefficients to test a filter with programmable coefficients.

Create a direct-form symmetric FIR filter with a fully parallel (default) architecture. Define the coefficients for the filter object in the vector **b**. The coder generates test bench code to test the coefficient interface using a second set of coefficients, **c**. The coder trims **c** to the effective length of the filter.

```

b = [-0.01 0.1 0.8 0.1 -0.01];
c = [-0.03 0.5 0.7 0.5 -0.03];
c = c(1:ceil(length(c)/2));
filt = dsp.FIRFilter('Numerator',b,'Structure','Direct form symmetric');
generatehdl(filt,'InputDataType',numeric(1,16,15), ...
    'GenerateHDLTestbench','on', ...
    'CoefficientSource','ProcessorInterface','TestbenchCoeffStimulus',c)

### Starting VHDL code generation process for filter: firfilt
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex66247050\hdlsrc\
### Starting generation of firfilt VHDL entity
### Starting generation of firfilt VHDL architecture
### Successful completion of VHDL code generation process for filter: firfilt
### HDL latency is 2 samples
### Starting generation of VHDL Test Bench.
### Generating input stimulus
### Done generating input stimulus; length 3107 samples.
### Generating Test bench: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex662470
### Creating stimulus vectors ...
### Done generating VHDL Test Bench.

```

### IIR Filter with Programmable Coefficients

Create a filter specification. When you generate HDL code, specify a programmable interface for the coefficients.

```

Fs = 48e3;
Fc = 10.8e3;

```



```

N = 5;
f_lp = fdesign.lowpass('n,f3db',N,Fc,Fs);
filtiir = design(f_lp,'butter','FilterStructure','df2sos','SystemObject',true,'UseLegacyBiquadFi
filtiir.OptimizeUnityScaleValues = 0;
generatehdl(filtiir,'InputDataType',numeric(1,16,15), ...
    'CoefficientSource','ProcessorInterface')

### Starting VHDL code generation process for filter: filtiiir
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex92389569\hdlsrc\
### Starting generation of filtiiir VHDL entity
### Starting generation of filtiiir VHDL architecture
### Second-order section, # 1
### Second-order section, # 2
### First-order section, # 3
### Successful completion of VHDL code generation process for filter: filtiiir
### HDL latency is 2 samples

```

The coder generates this VHDL entity for the filter object.

```

ENTITY filtiiir IS
  PORT ( clk          : IN      std_logic;
         clk_enable   : IN      std_logic;
         reset        : IN      std_logic;
         filter_in    : IN      std_logic_vector(15 DOWNTO 0); -- sfix16_En15
         write_enable  : IN      std_logic;
         write_done    : IN      std_logic;
         write_address : IN      std_logic_vector(4 DOWNTO 0); -- ufix5
         coeffs_in    : IN      std_logic_vector(15 DOWNTO 0); -- sfix16
         filter_out    : OUT     std_logic_vector(15 DOWNTO 0)  -- sfix16_En15
        );
END filtiiir;

```

## Clock Ports for Multirate Filters

Explore various ways to specify clock ports for multirate filters.

### Default Setting

Create a polyphase sample rate converter. By default, the coder generates a single input clock (`clk`), an input clock enable (`clk_enable`), and a clock enable output signal named `ce_out`. The `ce_out` signal indicates when an output sample is ready. The `ce_in` output signal indicates when an input sample was accepted. You can use this signal to control the upstream data flow.

```

firrc = dsp.FIRRateConverter('InterpolationFactor',5,'DecimationFactor',3);
generatehdl(firrc,'InputDataType',numeric(1,16,15))

### Starting VHDL code generation process for filter: firrc
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex09049114\hdlsrc\
### Starting generation of firrc VHDL entity
### Starting generation of firrc VHDL architecture
### Successful completion of VHDL code generation process for filter: firrc
### HDL latency is 2 samples

```

The generated entity has the following signals:

```
ENTITY firrc IS
  PORT( clk           : IN    std_logic;
        clk_enable    : IN    std_logic;
        reset         : IN    std_logic;
        filter_in     : IN    std_logic_vector(15 DOWNTO 0); -- sfix16_En15
        filter_out    : OUT   std_logic_vector(35 DOWNTO 0); -- sfix36_En31
        ce_in         : OUT   std_logic;
        ce_out        : OUT   std_logic
        );
END firrc;
```

### Custom Clock Names

You can provide custom names for the input clock enable and the output clock enable signals. You cannot rename the ce\_in signal.

```
firrc = dsp.FIRRateConverter('InterpolationFactor',5,'DecimationFactor',3)
```

```
firrc =
  dsp.FIRRateConverter with properties:
```

```
  Main
```

```
  InterpolationFactor: 5
```

```
  DecimationFactor: 3
```

```
  NumeratorSource: 'Property'
```

```
  Numerator: [0 -6.6976e-05 -1.6044e-04 -2.2552e-04 -1.8884e-04 0 3.2095e-04 6.5785e-05]
```

```
  Show all properties
```

```
generatehdl(firrc,'InputDataType',numerictype(1,16,15),...
            'ClockEnableInputPort','clk_en1', ...
            'ClockEnableOutputPort','clk_en2')
```

```
### Starting VHDL code generation process for filter: firrc
```

```
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex09049114\hdlsrc\
```

```
### Starting generation of firrc VHDL entity
```

```
### Starting generation of firrc VHDL architecture
```

```
### Successful completion of VHDL code generation process for filter: firrc
```

```
### HDL latency is 2 samples
```

The generated entity has the following signals:

```

ENTITY firrc IS
  PORT( clk           : IN    std_logic;
        clk_en1      : IN    std_logic;
        reset        : IN    std_logic;
        filter_in    : IN    std_logic_vector(15 DOWNTO 0); -- sfix16_En15
        filter_out   : OUT   std_logic_vector(35 DOWNTO 0); -- sfix36_En31
        ce_in       : OUT   std_logic;
        clk_en2     : OUT   std_logic
        );

END firrc;

```

### Multiple Clock Inputs

To generate multiple clock input signals for a supported multirate filter, set the `ClockInputs` property to 'Multiple'. In this case, the coder does not generate any output clock enable ports.

```

decim = dsp.CICDecimator(7,1,4);
generatehdl(decim, 'InputDataType', numerictype(1,16,15), ...
    'ClockInputs', 'Multiple')

```

```

### Starting VHDL code generation process for filter: cicDec0rIntFilt
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex09049114\hdlsrc\
### Starting generation of cicDec0rIntFilt VHDL entity
### Starting generation of cicDec0rIntFilt VHDL architecture
### Section # 1 : Integrator
### Section # 2 : Integrator
### Section # 3 : Integrator
### Section # 4 : Integrator
### Section # 5 : Comb
### Section # 6 : Comb
### Section # 7 : Comb
### Section # 8 : Comb
### Successful completion of VHDL code generation process for filter: cicDec0rIntFilt
### HDL latency is 7 samples

```

The generated entity has the following signals:

```

ENTITY cicdecimfilt IS
  PORT( clk           : IN    std_logic;
        clk_enable   : IN    std_logic;
        reset        : IN    std_logic;
        filter_in    : IN    std_logic_vector(15 DOWNT0 0); -- sfix16_En15
        clk1         : IN    std_logic;
        clk_enable1  : IN    std_logic;
        reset1       : IN    std_logic;
        filter_out   : OUT   std_logic_vector(27 DOWNT0 0) -- sfix28_En15
        );

END cicdecimfilt;

```

### Generate Default Altera Quartus II Synthesis Script

Create a filter object. Then call `generatehdl`, and specify a synthesis tool.

```

lpf = fdesign.lowpass('fp,fst,ap,ast',0.45,0.55,1,60);
firfilt = design(lpf,'equiripple','FilterStructure','dfsymfir', ...
  'SystemObject',true);
generatehdl(firfilt,'InputDataType',numerictype(1,14,13), ...
  'HDLSynthTool','Quartus');

```

```

### Starting VHDL code generation process for filter: firfilt
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex92219095\hdlsrc\
### Starting generation of firfilt VHDL entity
### Starting generation of firfilt VHDL architecture
### Successful completion of VHDL code generation process for filter: firfilt
### HDL latency is 2 samples

```

The coder generates a script file named `firfilt_quartus.tcl`, using the default script properties for the Altera® Quartus II synthesis tool.

```
type hdlsrc/firfilt_quartus.tcl
```

```

load_package flow
set top_level firfilt
set src_dir "./hdlsrc"
set prj_dir "q2dir"
file mkdir ../$prj_dir
cd ../$prj_dir
project_new $top_level -revision $top_level -overwrite
set_global_assignment -name FAMILY "Stratix II"
set_global_assignment -name DEVICE EP2S60F484C3
set_global_assignment -name TOP_LEVEL_ENTITY $top_level
set_global_assignment -name vhdl_FILE "../$src_dir/firfilt.vhd"
execute_flow -compile
project_close

```

## Construct Customized Synthesis Script

You can set the script automation properties to dummy values to illustrate how the coder constructs the synthesis script from the properties.

Design a filter and generate HDL. Specify a synthesis tool and custom text to include in the synthesis script.

```
lpf = fdesign.lowpass('fp,fst,ap,ast',0.45,0.55,1,60);
firfilt = design(lpf,'equiripple','FilterStructure','dfsymfir', ...
    'Systemobject',true);
generatehdl(firfilt,'InputDataType',numerictype(1,14,13), ...
    'HDLSynthTool','ISE', ...
    'HDLSynthInit','init line 1 : module name is %s\ninit line 2\n', ...
    'HDLSynthCmd','command : HDL filename is %s\n', ...
    'HDLSynthTerm','term line 1\nterm line 2\n');

### Starting VHDL code generation process for filter: firfilt
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex64737676\hdlsrc\
### Starting generation of firfilt VHDL entity
### Starting generation of firfilt VHDL architecture
### Successful completion of VHDL code generation process for filter: firfilt
### HDL latency is 2 samples
```

The coder generates a script file named `firfilt_ise.tcl`. Note the locations of the custom text you specified. You can use this feature to add synthesis instructions to the generated script.

```
type hdlsrc/firfilt_ise.tcl

init line 1 : module name is firfilt
init line 2
command : HDL filename is firfilt.vhd
term line 1
term line 2
```

## Input Arguments

### `filtS0` — Filter

filter System object

Filter from which to generate HDL code, specified as a filter System object. To create a filter System object, use the `design` function or see the reference page of the object. You can use the following System objects from DSP System Toolbox:

### Single Rate Filters

- `dsp.FIRFilter`
- `dsp.BiquadFilter`
- `dsp.HighpassFilter`
- `dsp.LowpassFilter`
- `dsp.FilterCascade`
- `dsp.VariableFractionalDelay`

### Multirate Filters

- `dsp.FIRDecimator`
- `dsp.FIRInterpolator`
- `dsp.FIRRateConverter`
- `dsp.FarrowRateConverter`
- `dsp.CICDecimator`
- `dsp.CICInterpolator`
- `dsp.CICCompensationDecimator`
- `dsp.CICCompensationInterpolator`
- `dsp.FilterCascade`
- `dsp.DigitalDownConverter`
- `dsp.DigitalUpConverter`

#### **nt** — Input data type

`numericType` object

Input data type, specified as a `numericType` object. This argument applies only when the input filter is a System object. Call `numericType(s,w,f)`, where `s` is 1 for signed and 0 for unsigned, `w` is the word length in bits, and `f` is the number of fractional bits.

#### **fd** — Fractional delay data type

`numericType` object

Fractional delay data type, specified as a `numericType` object. This argument applies only when the input filter is a `dsp.VariableFractionalDelay` System object. Call `numericType(s,w,f)`, where `s` is 1 for signed and 0 for unsigned, `w` is the word length in bits, and `f` is the number of fractional bits.

#### **filterObj** — Filter

`dfilt` object

Filter from which to generate HDL code, specified as a `dfilt` object. You can create this object by using the `design` function. For an overview of supported filter features, see “Filter Configuration Options” (Filter Design HDL Coder).

### Alternatives

You can use the `fdhdltool` function to generate HDL code instead (requires Filter Design HDL Coder). Specify the input and fractional delay data types as arguments, and then set additional properties in the Generate HDL dialog box.

## Version History

Introduced before R2006a

### See Also

`generatetbstimulus` | `fdhdltool`

# getPolynomialCoefficients

**Package:** dsp

Get polynomial coefficients of farrow rate conversion filter

## Syntax

```
[C] = getPolynomialCoefficients(frc)
```

## Description

[C] = getPolynomialCoefficients(frc) returns the polynomial coefficients that the dsp.FarrowRateConverter System object, frc, uses to implement the specified sample rate conversion.

## Examples

### Return Polynomial Coefficients of dsp.FarrowRateConverter

Create a default dsp.FarrowRateConverter System object™ that converts a signal from 44.1 kHz to 48 kHz.

```
frc = dsp.FarrowRateConverter()
```

```
frc =
  dsp.FarrowRateConverter with properties:
```

```
  Main
    InputSampleRate: 44100
    OutputSampleRate: 48000
    OutputRateTolerance: 0
    Specification: 'Polynomial order'
    PolynomialOrder: 3
```

```
Show all properties
```

Return the self-designed polynomial coefficients that the object uses to implement the specified rate conversion.

```
c = getPolynomialCoefficients(frc)
```

```
c = 4x4

   -0.1667    0.5000   -0.3333         0
    0.5000   -1.0000   -0.5000    1.0000
   -0.5000    0.5000    1.0000         0
    0.1667         0   -0.1667         0
```

## Input Arguments

### **frc** — Polynomial sample rate conversion filter

`dsp.FarrowRateConverter` System object

Polynomial sample rate conversion filter, specified as a `dsp.FarrowRateConverter` System object.

## Output Arguments

### **C** — Polynomial coefficients of Farrow rate conversion filter

real-valued square matrix

Polynomial coefficients of Farrow rate conversion filter, returned as a  $M$ -by- $M$  matrix, where  $M$  is the polynomial order.

Data Types: `double`

## Version History

**Introduced in R2014b**

## See Also

### Functions

`getActualOutputRate` | `getRateChangeFactors`

### Objects

`dsp.FarrowRateConverter`



# getActualOutputRate

**Package:** dsp

Get actual output rate

## Syntax

```
fsout = getActualOutputRate(rateConverter)
```

## Description

`fsout = getActualOutputRate(rateConverter)` returns the actual output sample rate of the `rateConverter` object, taking into account the `OutputRateTolerance` property. The rate converter object can be a `dsp.FarrowRateConverter System` object or a `dsp.SampleRateConverter System` object.

## Examples

### Specify Tolerance and Confirm Output Sample Rate of `dsp.FarrowRateConverter`

Set the tolerance of the filter to 1%, then return the actual output sample rate for the default conversion between 44.1 kHz and 48 kHz.

```
frc = dsp.FarrowRateConverter();  
frc.OutputRateTolerance = 0.01;  
FsOut = getActualOutputRate(frc)
```

```
FsOut = 4.8109e+04
```

The actual output rate can differ from the requested `OutputSampleRate`, within the configured tolerance.

### Output Sample Rate with Given Tolerance

Get the actual output sample rate for conversion between 192 kHz and 44.1 kHz when given a tolerance of 1%.

```
src = dsp.SampleRateConverter;  
src.OutputRateTolerance = 0.01;  
FsOut = getActualOutputRate(src)
```

```
FsOut = 4.4308e+04
```

## Input Arguments

**rateConverter** — Sample rate conversion filter

`dsp.FarrowRateConverter` | `dsp.SampleRateConverter`

Polynomial sample rate conversion filter, specified as a `dsp.FarrowRateConverter` or a `dsp.SampleRateConverter` System object.

## Output Arguments

### **fsout** — Actual output sample rate

scalar

Actual output sample rate of the filter, returned as a scalar in Hz.

Data Types: `double`

## Version History

**Introduced in R2014b**

## See Also

### Functions

`getRateChangeFactors` | `getPolynomialCoefficients`

### Objects

`dsp.FarrowRateConverter` | `dsp.SampleRateConverter`

# getRateChangeFactors

**Package:** dsp

Get overall interpolation and decimation factors

## Syntax

```
[L,M] = getRateChangeFactors(rateConverter)
```

## Description

`[L,M] = getRateChangeFactors(rateConverter)` returns the overall interpolation factor, L, and the overall decimation factor, M, corresponding to the `rateConverter`. The rate converter object can be a `dsp.FarrowRateConverter` System object or a `dsp.SampleRateConverter` System object. The rate change factors computed take into account the `OutputRateTolerance` property.

In the case of `dsp.SampleRateConverter`, the overall decimation factor affects the allowable frame size of the input to the object. The row length of the input vector or matrix to the `dsp.SampleRateConverter` object must be an integer multiple of M.

This function supports C and C++ code generation.

## Examples

### Return Resampling Factors of `dsp.FarrowRateConverter`

Create a default `dsp.FarrowRateConverter` object that converts a signal from 44.1 kHz to 48 kHz.

```
frc = dsp.FarrowRateConverter
frc =
  dsp.FarrowRateConverter with properties:
    Main
      InputSampleRate: 44100
      OutputSampleRate: 48000
      OutputRateTolerance: 0
      Specification: 'Polynomial order'
      PolynomialOrder: 3
    Show all properties
```

Return the overall interpolation (L) and decimation (M) factors of the filter object.

```
[L,M] = getRateChangeFactors(frc)
```

```
L = 160
```

```
M = 147
```

### Default Resampling Factors

Create `src`, a multistage sample rate converter with default properties. `src` combines three filter stages to convert from 192 kHz to 44.1 kHz. Determine its overall interpolation and decimation factors.

```
src = dsp.SampleRateConverter;  
[L,M] = getRateChangeFactors(src)
```

```
L = 147
```

```
M = 640
```

### Input Arguments

#### **rateConverter** — Sample rate conversion filter

`dsp.FarrowRateConverter` | `dsp.SampleRateConverter`

Polynomial sample rate conversion filter, specified as a `dsp.FarrowRateConverter` or a `dsp.SampleRateConverter` System object.

### Output Arguments

#### **L** — Overall interpolation factor

scalar

Overall interpolation factor, returned as a scalar.

Data Types: `double`

#### **M** — Overall decimation factor

scalar

Overall decimation factor, returned as a scalar. In the case of `dsp.SampleRateConverter`, the overall decimation factor affects the allowable frame size of the input to the object. The row length of the input vector or matrix to the `dsp.SampleRateConverter` object must be an integer multiple of `M`.

Data Types: `double`

### Version History

Introduced in R2014b

### Extended Capabilities

#### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

## See Also

### Functions

[getActualOutputRate](#) | [getPolynomialCoefficients](#)

### Objects

[dsp.FarrowRateConverter](#) | [dsp.SampleRateConverter](#)

## getBandwidth

**Package:** dsp

Get 3 dB bandwidth

### Syntax

```
BW = getBandwidth(npFilter)
```

### Description

`BW = getBandwidth(npFilter)` returns the 3 dB bandwidth for the notch peak filter. If the `Specification` property is set to 'Quality factor and center frequency', the 3 dB bandwidth is determined from the quality factor value. If the `Specification` property is set to 'Coefficients', the 3 dB bandwidth is determined from the `BandwidthCoefficient` value and the sample rate.

### Examples

#### Get 3 dB Bandwidth of Notch Peak Filter

Create a `dsp.NotchPeakFilter` object with the `Specification` property set to 'Quality factor and center frequency'. The default quality factor  $Q$  is 5, and the center frequency  $F_c$  is 11,025 Hz.

```
np = dsp.NotchPeakFilter('Specification',...  
    'Quality factor and center frequency')
```

```
np =  
dsp.NotchPeakFilter with properties:  
  
    Specification: 'Quality factor and center frequency'  
    QualityFactor: 5  
    CenterFrequency: 11025  
    SampleRate: 44100
```

Compute the 3 dB bandwidth of the notch peak filter using the `getBandwidth` function. The bandwidth is computed as the ratio of the center frequency and the quality factor,  $\frac{F_c}{Q}$ .

```
getBandwidth(np)
```

```
ans = 2205
```

Visualize the filter response using `fvtool`.

```
fvtool(np)
```

## Input Arguments

**npFilter — Notch peak filter**  
`dsp.NotchPeakFilter`

Notch peak filter whose 3 dB bandwidth is measured, specified as a `dsp.NotchPeakFilter` object.

## Output Arguments

**BW — 3 dB bandwidth**  
scalar

3 dB bandwidth of the filter, returned as a scalar.

Data Types: `double`

## Version History

Introduced in R2014a

## **See Also**

### **Functions**

getCenterFrequency | getOctaveBandwidth | getQualityFactor | tf

### **Objects**

dsp.NotchPeakFilter



# getBranches

**Package:** dsp

Return internal allpass branches

## Syntax

```
s = getBranches(caf)
```

## Description

`s = getBranches(caf)` returns copies of the internal allpass branches as a two-field structure, `s`. Each branch is an instance of `dsp.AllpassFilter`.

## Examples

### Get Branches of `dsp.CoupledAllpassFilter`

Get the internal allpass branches of the `dsp.CoupledAllpassFilter` System object™.

#### Minimum Multiplier

Create a `dsp.CoupledAllpassFilter` object with filter structure set to 'Minimum multiplier'. Use the `getBranches` function to get the internal allpass filter objects.

```
caf = dsp.CoupledAllpassFilter
```

```
caf =
  dsp.CoupledAllpassFilter with properties:
      Structure: 'Minimum multiplier'
  PureDelayBranch: 0
  AllpassCoefficients1: {[0 0.5000]}
  AllpassCoefficients2: {[[]]}
      Gain1: '1'
      Gain2: '1'
```

```
s = getBranches(caf)
```

```
s = struct with fields:
  Branch1: [1x1 dsp.internal.LegacyAllpassFilter]
  Branch2: [1x1 dsp.internal.LegacyAllpassFilter]
```

```
s.Branch1
```

```
ans =
  dsp.internal.LegacyAllpassFilter with properties:
      AllpassCoefficients: {[0 0.5000]}
      Structure: 'Minimum multiplier'
```

```
InitialConditions: 0
```

```
s.Branch2
```

```
ans =  
dsp.internal.LegacyAllpassFilter with properties:  
  
AllpassCoefficients: {}  
    Structure: 'Minimum multiplier'  
InitialConditions: 0
```

### Wave Digital Filter

Change the filter structure to 'Wave Digital Filter'. The internal allpass filters display WDF coefficients.

```
caf.Structure = 'Wave Digital Filter'
```

```
caf =  
dsp.CoupledAllpassFilter with properties:  
  
    Structure: 'Wave Digital Filter'  
PureDelayBranch: 0  
WDFCoefficients1: {[0.5000 0]}  
WDFCoefficients2: {}  
    Gain1: '1'  
    Gain2: '1'
```

```
s = getBranches(caf)
```

```
s = struct with fields:  
Branch1: [1x1 dsp.internal.LegacyAllpassFilter]  
Branch2: [1x1 dsp.internal.LegacyAllpassFilter]
```

```
s.Branch1
```

```
ans =  
dsp.internal.LegacyAllpassFilter with properties:  
  
WDFCoefficients: {[0.5000 0]}  
    Structure: 'Wave Digital Filter'  
InitialConditions: 0
```

```
s.Branch2
```

```
ans =  
dsp.internal.LegacyAllpassFilter with properties:  
  
WDFCoefficients: {}  
    Structure: 'Wave Digital Filter'  
InitialConditions: 0
```

## Lattice

When the filter structure is set to 'Lattice', the internal allpass filters display the lattice coefficients.

```
caf.Structure = 'Lattice'
```

```
caf =
  dsp.CoupledAllpassFilter with properties:
      Structure: 'Lattice'
      PureDelayBranch: 0
      LatticeCoefficients1: {[0 0.5000]}
      LatticeCoefficients2: {[]}
      Beta: 1
      Gain1: '1'
      Gain2: '1'
      ComplexConjugateCoefficients: 0
```

```
s = getBranches(caf)
```

```
s = struct with fields:
  Branch1: [1x1 dsp.internal.LegacyAllpassFilter]
  Branch2: [1x1 dsp.internal.LegacyAllpassFilter]
```

```
s.Branch1
```

```
ans =
  dsp.internal.LegacyAllpassFilter with properties:
      LatticeCoefficients: {[0 0.5000]}
      Structure: 'Lattice'
      InitialConditions: 0
```

```
s.Branch2
```

```
ans =
  dsp.internal.LegacyAllpassFilter with properties:
      LatticeCoefficients: {[]}
      Structure: 'Lattice'
      InitialConditions: 0
```

## Input Arguments

### caf — Input filter object

dsp.CoupledAllpassFilter

Input filter object, specified as a dsp.CoupledAllpassFilter System object.

## Output Arguments

### s — Internal allpass branches

structure

Internal allpass branches, returned as a two-field structure. The two fields contain instances of the `dsp.AllpassFilter` System objects representing the two individual branches of the `dsp.CoupledAllpassFilter` object.

## **Version History**

**Introduced in R2013b**

### **See Also**

#### **Functions**

`fvtool`

#### **Objects**

`dsp.CoupledAllpassFilter`

# getCenterFrequency

**Package:** dsp

Get center frequency

## Syntax

```
CF = getCenterFrequency(npFilter)
```

## Description

`CF = getCenterFrequency(npFilter)` returns the center frequency of the notch peak filter. If the `Specification` property is set to `'Coefficients'`, the center frequency is determined from the `CenterFrequencyCoefficient` value and the sample rate.

## Examples

### Get Center Frequency of Notch Peak Filter

Create a `dsp.NotchPeakFilter` object with the `Specification` property set to `'Coefficients'`.

```
np = dsp.NotchPeakFilter('Specification','Coefficients')
```

```
np =
```

```
    dsp.NotchPeakFilter with properties:
```

```
        Specification: 'Coefficients'
    BandwidthCoefficient: 0.7265
    CenterFrequencyCoefficient: 0
        SampleRate: 44100
```

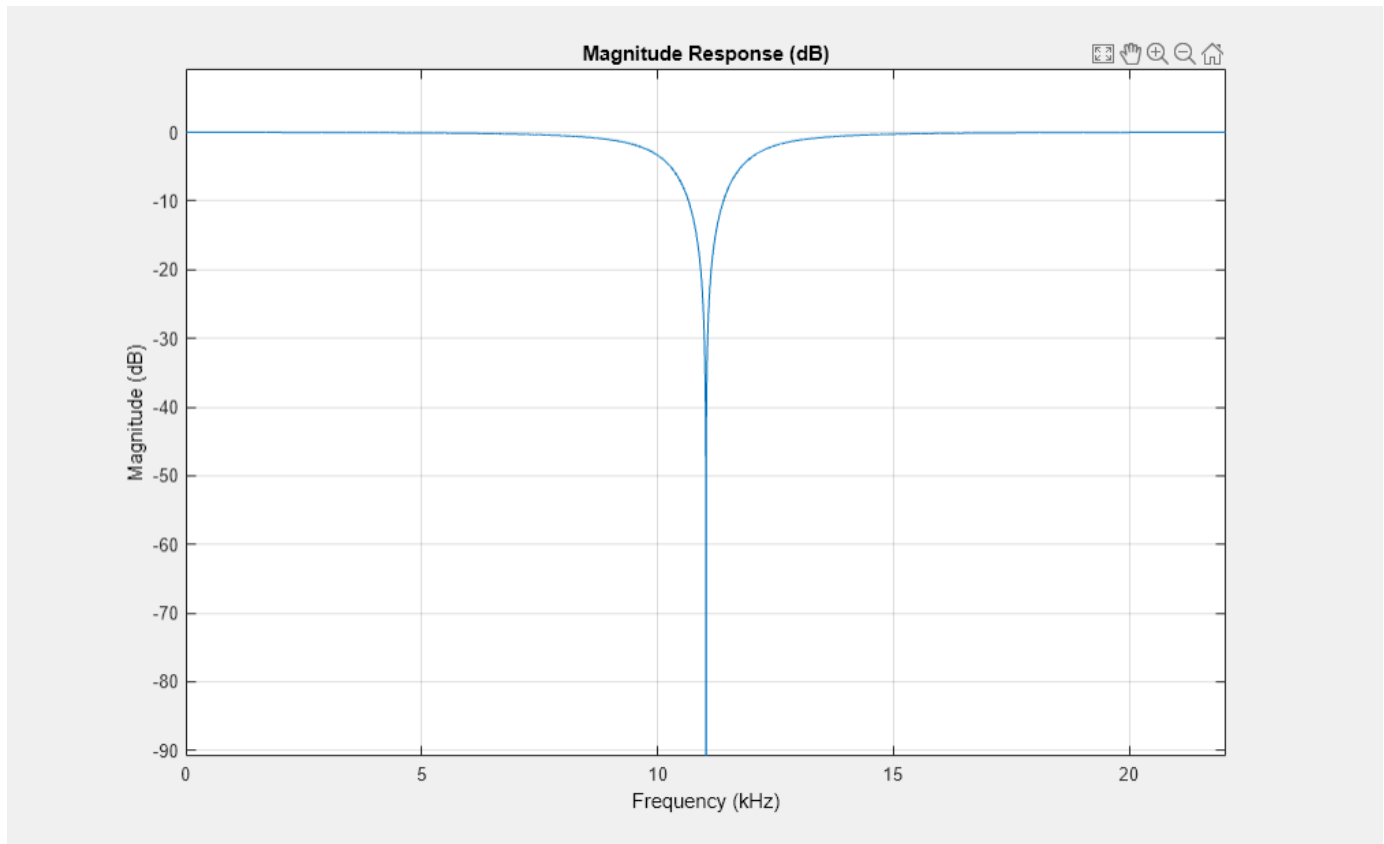
Determine the center frequency of the notch peak filter using the `getCenterFrequency` function. When the `Specification` is set to `'Coefficients'`, the center frequency is determined from the `CenterFrequencyCoefficient` value and the sample rate.

```
getCenterFrequency(np)
```

```
ans = 11025
```

Visualize the filter response using `fvtool`.

```
fvtool(np)
```



## Input Arguments

**npFilter** — Notch peak filter  
`dsp.NotchPeakFilter`

Notch peak filter whose center frequency is measured, specified as `dsp.NotchPeakFilter` object.

## Output Arguments

**CF** — Center frequency  
scalar

Center frequency of the filter, returned as a scalar.

Data Types: `double`

## Version History

Introduced in R2014a

## See Also

### Functions

getBandwidth | getOctaveBandwidth | getQualityFactor | tf

### Objects

dsp.NotchPeakFilter

## getCursorInfo

**Package:** dsp

Return settings for Logic Analyzer cursor

### Syntax

```
cursorInfo = getCursorInfo(scope, 'CursorTag', tag)
```

### Description

`cursorInfo = getCursorInfo(scope, 'CursorTag', tag)` returns the settings for the cursor or cursors specified by the input tag.

### Examples

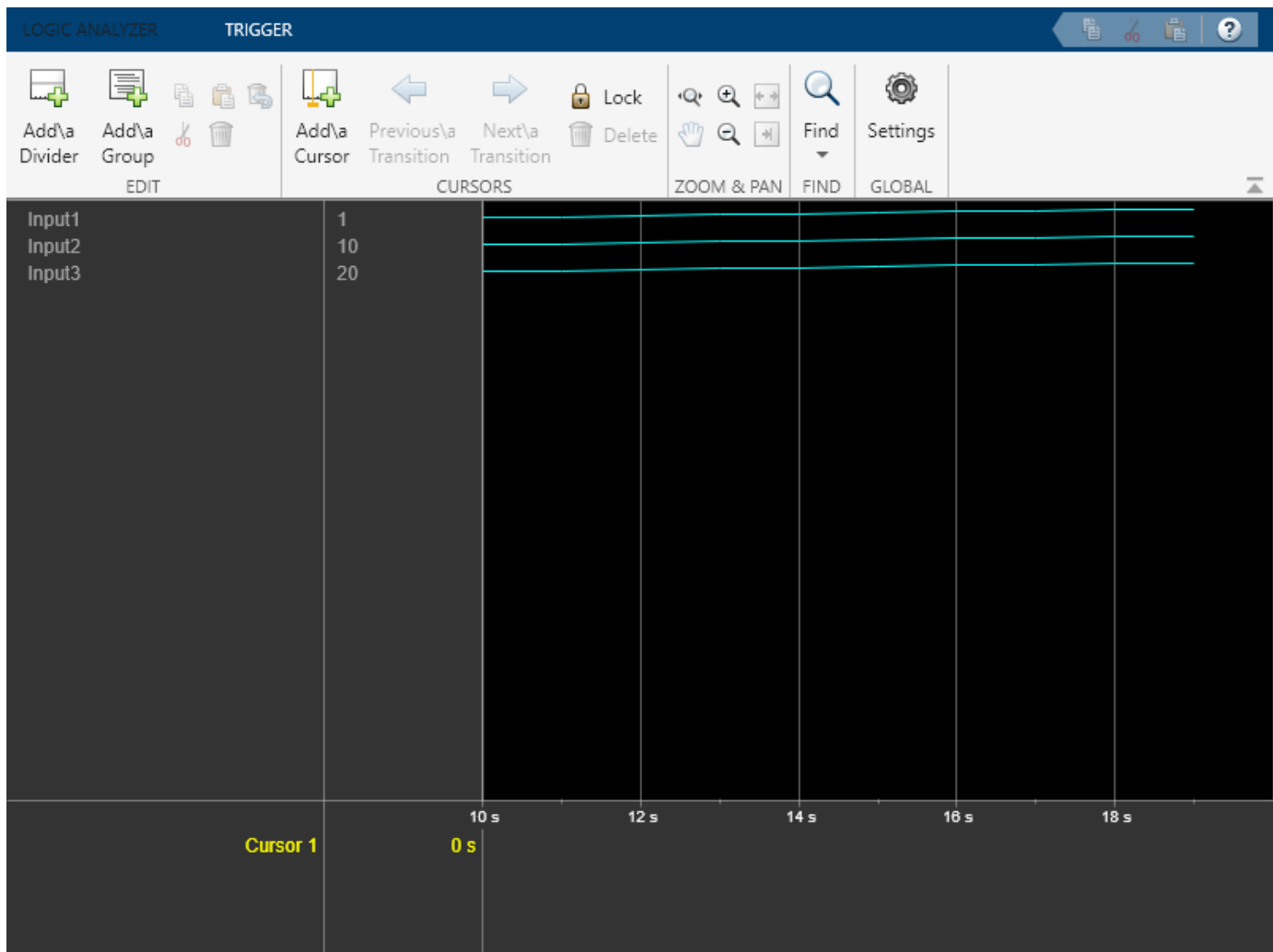
#### Modify Logic Analyzer Cursors Programmatically

This example shows how to use functions to create, manipulate, and delete cursors in a `dsp.LogicAnalyzer` object.

#### Create Logic Analyzer and Signals

```
scope = dsp.LogicAnalyzer('NumInputPorts',3);  
for ii = 1:20  
    scope(ii,10*ii,20*ii);  
end
```





### Add Cursor

Add a cursor at 15 seconds and show the cursor information.

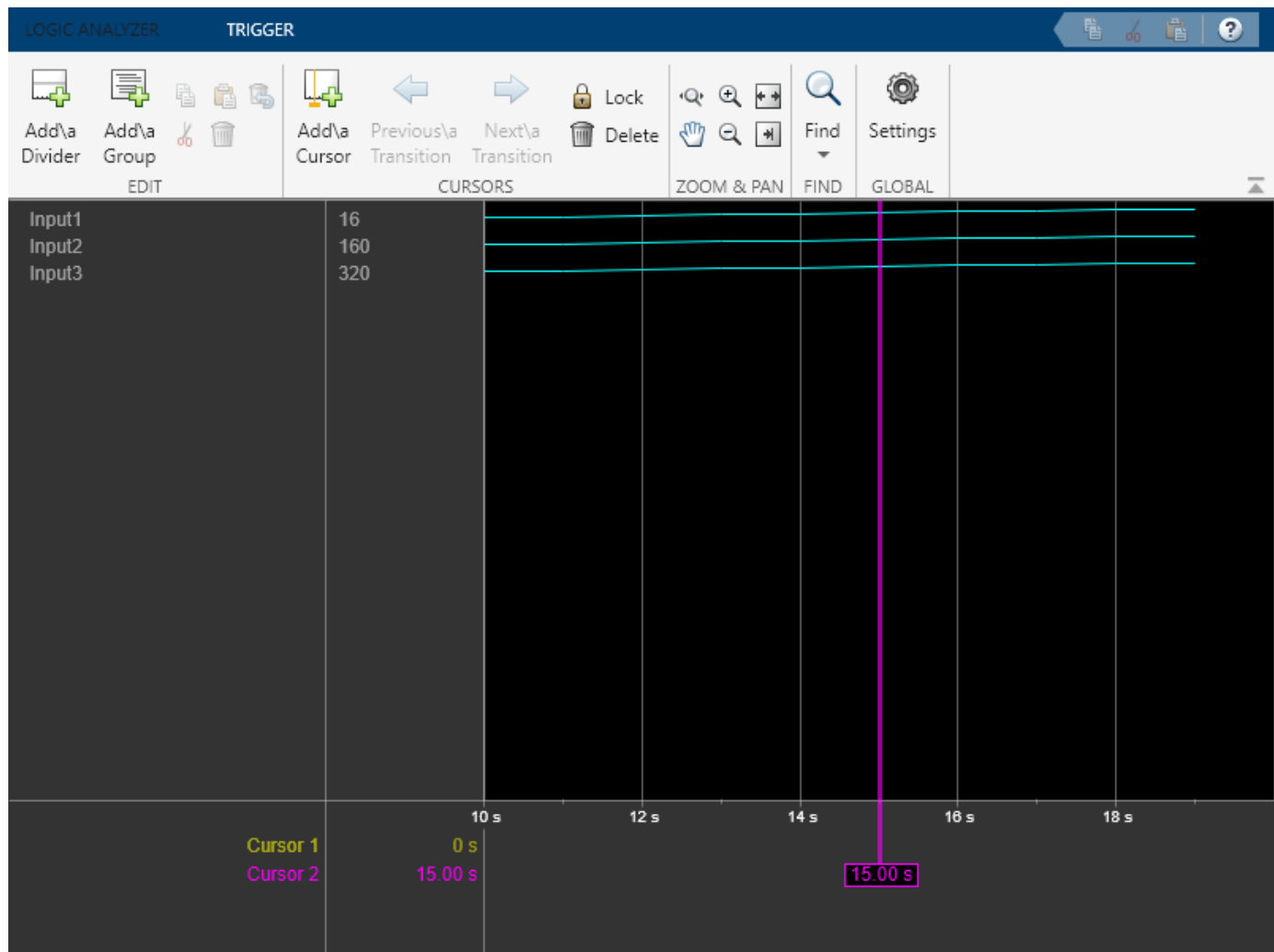
```
cursor = addCursor(scope, 'Location', 15, 'Color', 'Cyan');
getCursorInfo(scope, cursor)
```

```
ans = struct with fields:
  Location: 15
  Color: [0 1 1]
  Locked: 0
  Tag: 'C2'
```

### Modify Cursor

Change the cursor color to magenta.

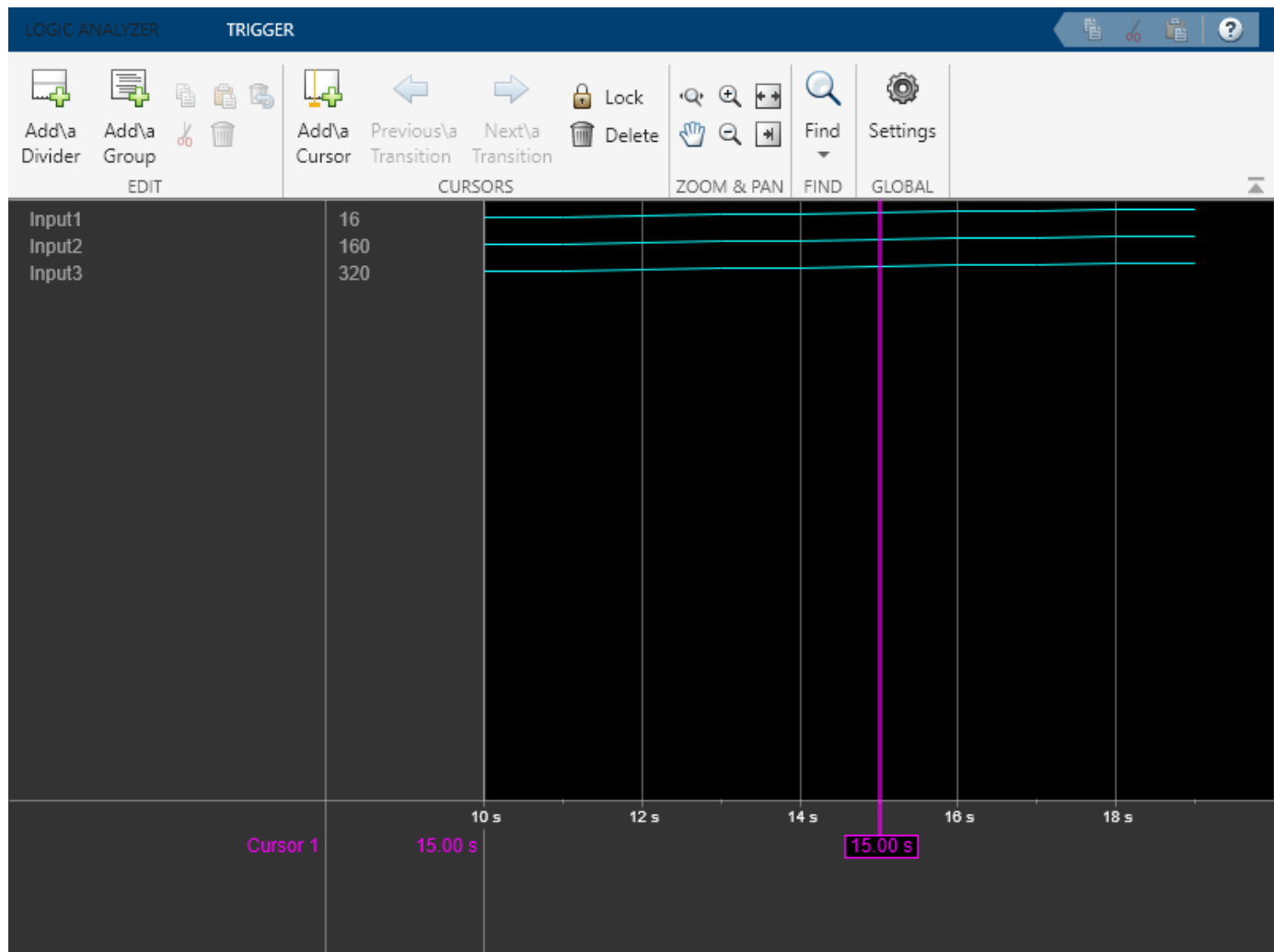
```
hide(scope)
modifyCursor(scope, cursor, 'Color', 'Magenta')
show(scope)
```



### Remove Cursor

Delete the yellow cursor at 0 seconds.

```
hide(scope)
tags = getCursorTags(scope);
deleteCursor(scope, tags{1});
show(scope)
```



## Input Arguments

**scope** — The Logic Analyzer object from which you want to return cursor settings

`dsp.LogicAnalyzer` object

The Logic Analyzer object from which you want to return cursor settings, specified as a handle to the `dsp.LogicAnalyzer` object.

**tag** — identifying tag or tags

character vector | string scalar | cell array of character vectors | string array

The tag or tags identifying the cursor or cursors about which to get information, specified as the randomly assigned cursor tag or tags.

Example: 'C5'

Example: {'C4', 'C5'}

Example: ["C4", "C5"]

Data Types: char | cell | string

## Output Arguments

**cursorInfo** — Information about settings for the cursor or cursors

struct

The `cursorInfo` struct contains the following fields:

- `Location` — Location of the cursors
- `Color` — Color of the cursors
- `Locked` — Locked status of the cursors
- `Tag` — Tag identifying cursors

## Version History

Introduced in R2013a

### See Also

`dsp.LogicAnalyzer` | `getCursorTags` | `modifyCursor` | `getDisplayChannelInfo`

# getCursorTags

**Package:** dsp

Return all Logic Analyzer cursor tags

## Syntax

```
cursorTags = getCursorTags(scope)
```

## Description

`cursorTags = getCursorTags(scope)` returns all the cursor tags for the Logic Analyzer. You can use these tags to get information about a cursor using the `getCursorInfo` method, to modify a cursor using the `modifyCursor` method, or to delete a cursor using the `deleteCursor` method.

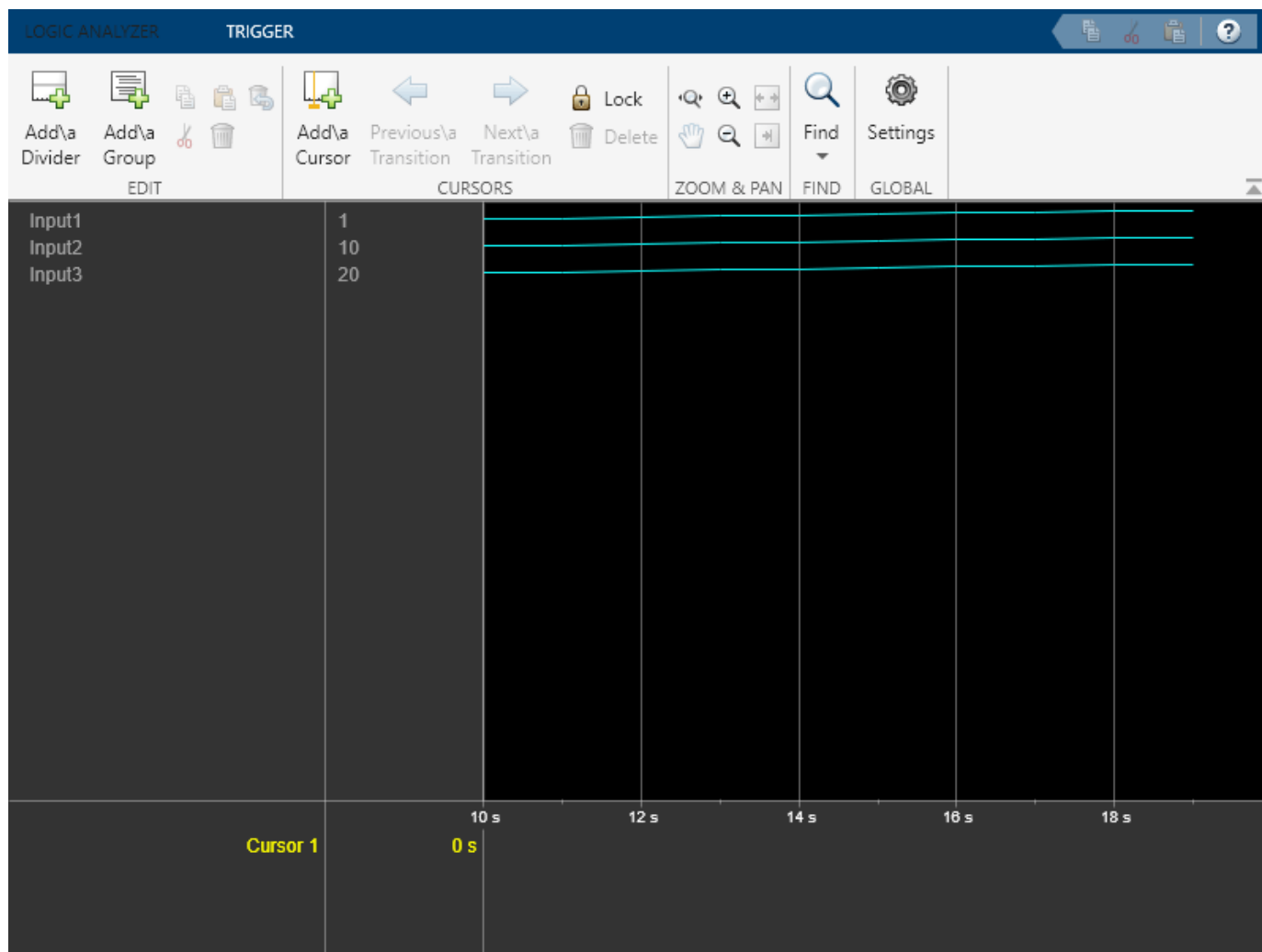
## Examples

### Modify Logic Analyzer Cursors Programmatically

This example shows how to use functions to create, manipulate, and delete cursors in a `dsp.LogicAnalyzer` object.

### Create Logic Analyzer and Signals

```
scope = dsp.LogicAnalyzer('NumInputPorts',3);  
for ii = 1:20  
    scope(ii,10*ii,20*ii);  
end
```



### Add Cursor

Add a cursor at 15 seconds and show the cursor information.

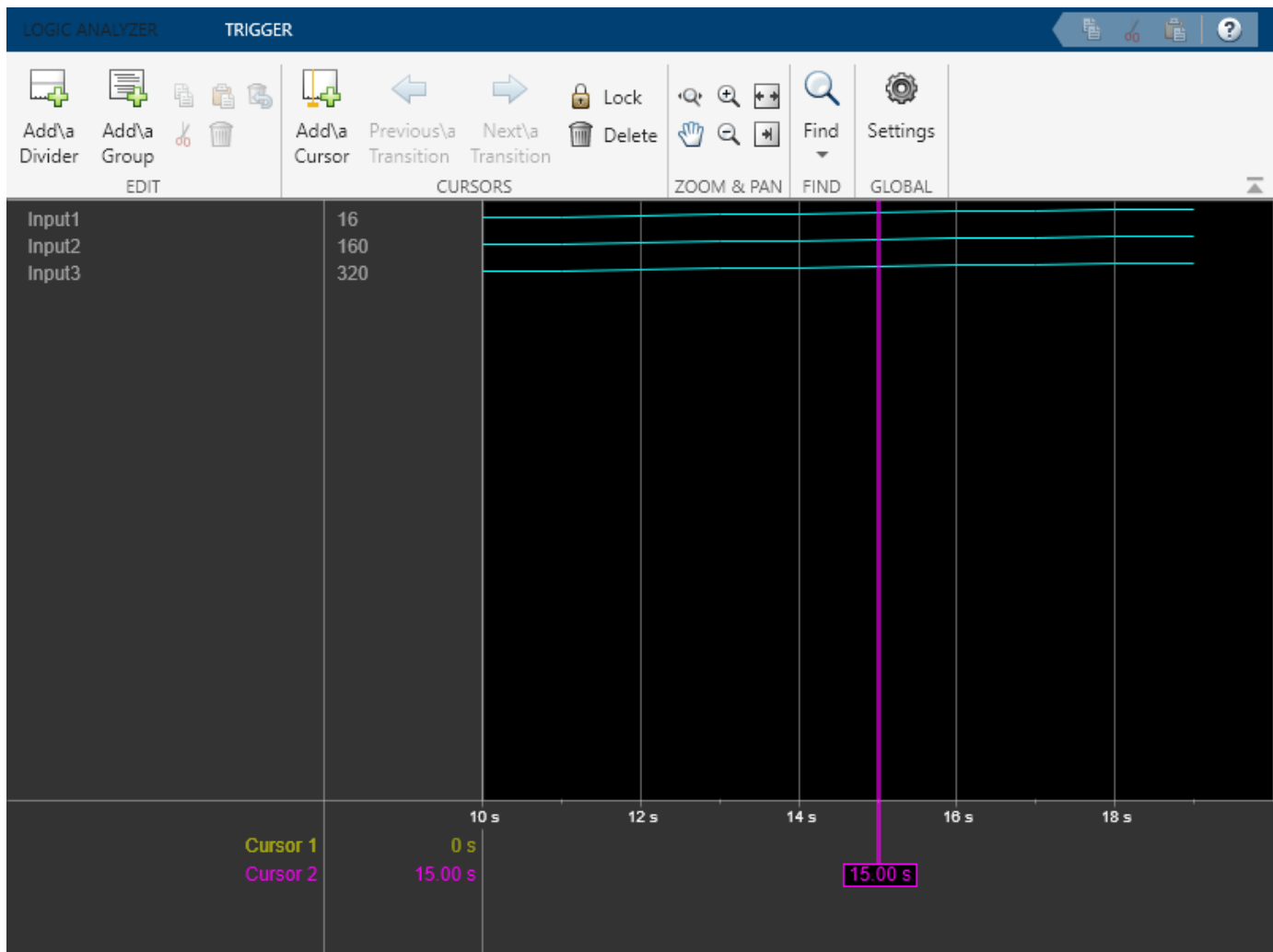
```
cursor = addCursor(scope, 'Location', 15, 'Color', 'Cyan');
getCursorInfo(scope, cursor)
```

```
ans = struct with fields:
    Location: 15
    Color: [0 1 1]
    Locked: 0
    Tag: 'C2'
```

### Modify Cursor

Change the cursor color to magenta.

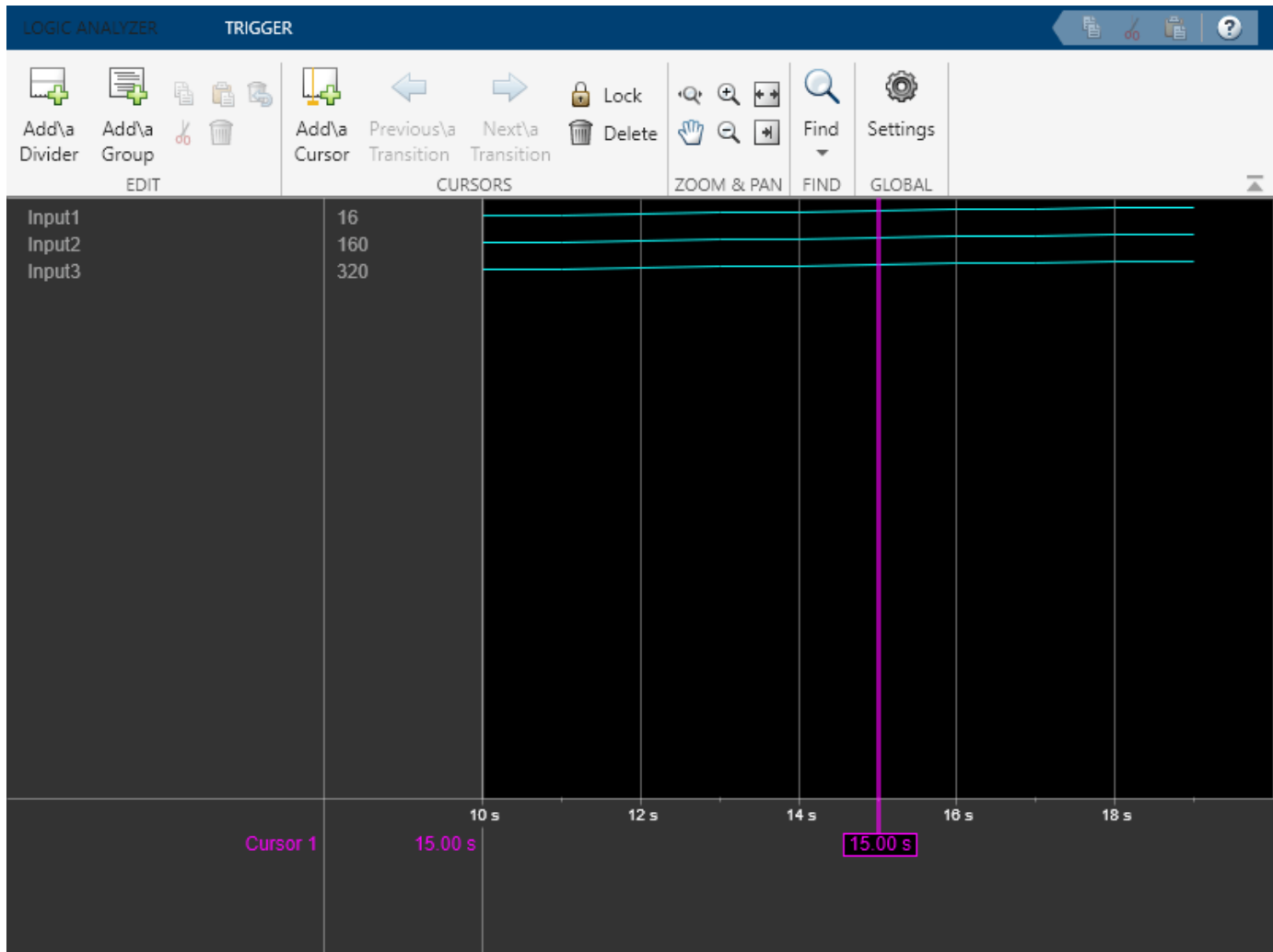
```
hide(scope)
modifyCursor(scope, cursor, 'Color', 'Magenta')
show(scope)
```



### Remove Cursor

Delete the yellow cursor at 0 seconds.

```
hide(scope)
tags = getCursorTags(scope);
deleteCursor(scope, tags{1});
show(scope)
```



## Input Arguments

### scope — Logic analyzer object

`dsp.LogicAnalyzer` object

The Logic Analyzer object from which you want to return all cursor tags, specified as a handle to the `dsp.LogicAnalyzer` object.

## Output Arguments

### cursorTags — All cursor tags

cell array of character vectors

The cursor tags, specified as a cell array of character vectors.

Example: `{ 'C1' }`

Example: `{ 'C1', 'C2', 'C3' }`

Data Types: `cell`



## **Version History**

**Introduced in R2013a**

### **See Also**

`dsp.LogicAnalyzer` | `getDisplayChannelTags` | `getCursorInfo` | `modifyCursor`

## getDecimationFactors

**Package:** dsp

Get decimation factors of each filter stage of a digital down converter

### Syntax

```
M = getDecimationFactors(dwnConv)
```

### Description

`M = getDecimationFactors(dwnConv)` returns a vector, `M`, with the decimation factors of each filter stage of the digital down converter, `dwnConv`. If the third filter stage is bypassed, then `M` is a 1-by-2 vector containing the decimation factors of the first and second filter stages. If the third filter stage is not bypassed, then `M` is a 1-by-3 vector containing the decimation factors of the first, second, and third filter stages.

### Examples

#### Get Decimation Factors

Get decimation factors of each filter stage of the `dsp.DigitalDownConverter` System object™.

Create a `dsp.DigitalDownConverter` System object with the default settings. Using the `getDecimationFactors` function, obtain the decimation factors of each stage of the object.

```
dwnConv = dsp.DigitalDownConverter

dwnConv =
    dsp.DigitalDownConverter with properties:

        DecimationFactor: 100
    MinimumOrderDesign: true
           Bandwidth: 200000
StopbandFrequencySource: 'Auto'
      PassbandRipple: 0.1000
StopbandAttenuation: 60
      Oscillator: 'Sine wave'
    CenterFrequency: 14000000
      SampleRate: 30000000

Show all properties

M = getDecimationFactors(dwnConv) %#ok

M = 1×3

    25     2     2
```

The `DecimationFactor` property of the object is set to 100. The output `M` is by default a 1-by-3 vector, where each element in the vector is a factor of the overall decimation factor.

When you set the `DecimationFactor` to a 1-by-2 vector, the object bypasses the third filter stage and sets the decimation factor of the first and second filtering stages to the values in the first and second vector elements respectively.

```
dwnConv.DecimationFactor = [10 10]
```

```
dwnConv =
  dsp.DigitalDownConverter with properties:

    DecimationFactor: [10 10]
    MinimumOrderDesign: true
    Bandwidth: 200000
    StopbandFrequencySource: 'Auto'
    PassbandRipple: 0.1000
    StopbandAttenuation: 60
    Oscillator: 'Sine wave'
    CenterFrequency: 14000000
    SampleRate: 30000000
```

```
Show all properties
```

```
M = getDecimationFactors(dwnConv)
```

```
M = 1×2
    10    10
```

The output of the `getDecimationFactors` function is now a 1-by-2 vector.

## Input Arguments

### **dwnConv** — Digital down converter

`dsp.DigitalDownConverter`

Digital down converter, specified as a `dsp.DigitalDownConverter` System object.

## Output Arguments

### **M** — Decimation factors

vector

Decimation factors of each filter stage, returned as a 1-by-2 or a 1-by-3 vector. If the third filter stage is bypassed, then `M` is a 1-by-2 vector containing the decimation factors of the first and second filter stages. If the third filter stage is not bypassed, then `M` is a 1-by-3 vector containing the decimation factors of the first, second, and third filter stages.

Data Types: `double`

## **Version History**

**Introduced in R2012a**

### **See Also**

#### **Functions**

[getFilterOrders](#) | [getFilters](#) | [fvtool](#) | [groupDelay](#) | [visualizeFilterStages](#) | [getInterpolationFactors](#)

#### **Objects**

[dsp.DigitalDownConverter](#)

# getDisplayChannelInfo

**Package:** dsp

Return settings for Logic Analyzer display channel

## Syntax

```
channelInfo = getDisplayChannelInfo(scope, 'DisplayChannelTag', tag)
```

## Description

`channelInfo = getDisplayChannelInfo(scope, 'DisplayChannelTag', tag)` returns the settings for the display channel or channels, specified by the input tag.

## Examples

### Manipulate Logic Analyzer Programmatically

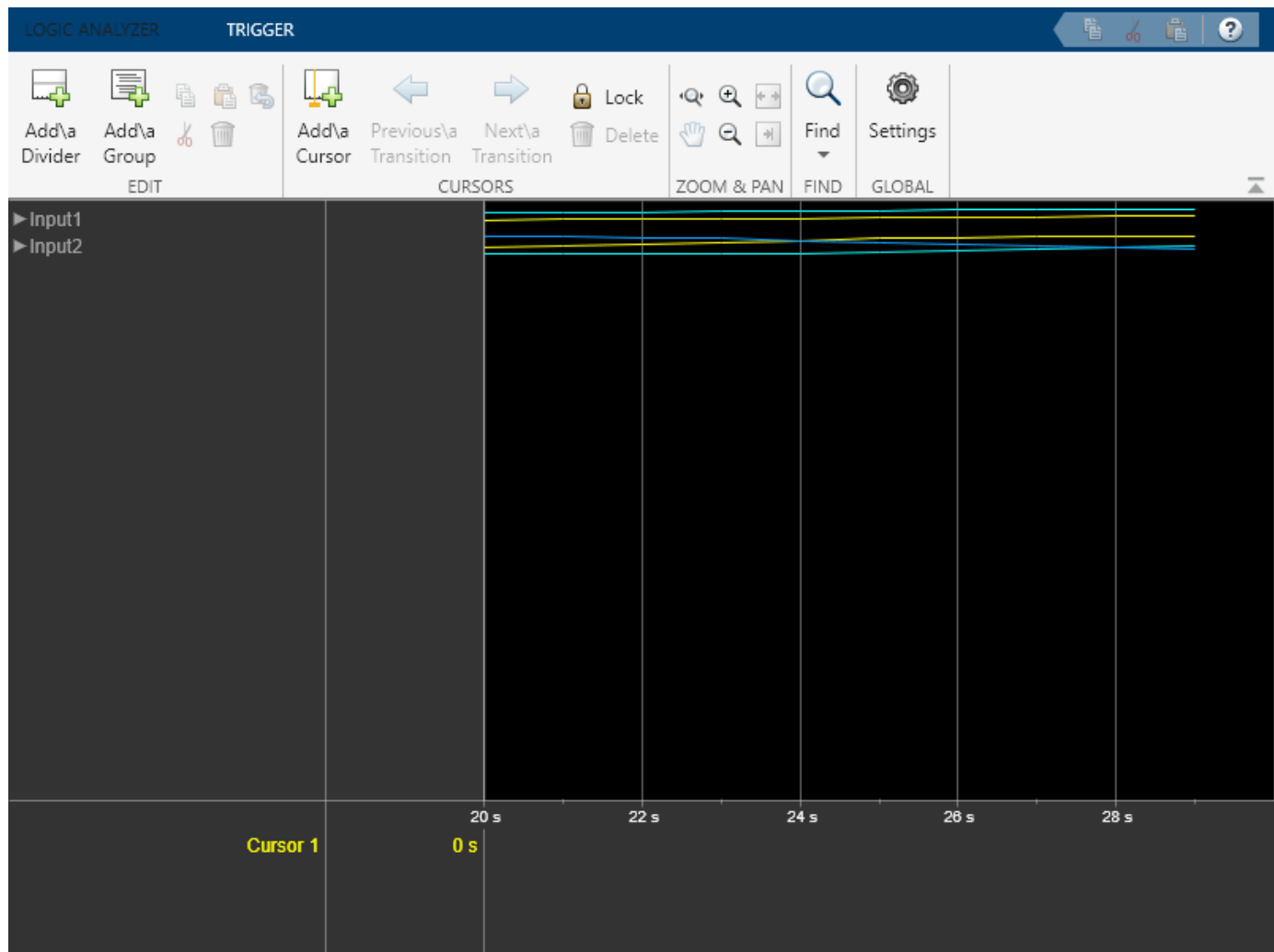
Use functions to construct and manipulate a `dsp.LogicAnalyzer` System object.

#### Display Waves on Logic Analyzer scope.

```
scope = dsp.LogicAnalyzer('NumInputPorts',2);

stop = 30;
for count = 1:stop
    sinValVec = sin(count/stop*2*pi);
    cosValVec = cos(count/stop*2*pi);
    cosValVecOffset = cos((count+10)/stop*2*pi);

    scope([count (count-(stop/2))],[sinValVec cosValVec cosValVecOffset])
end
```



### Reorganize Display

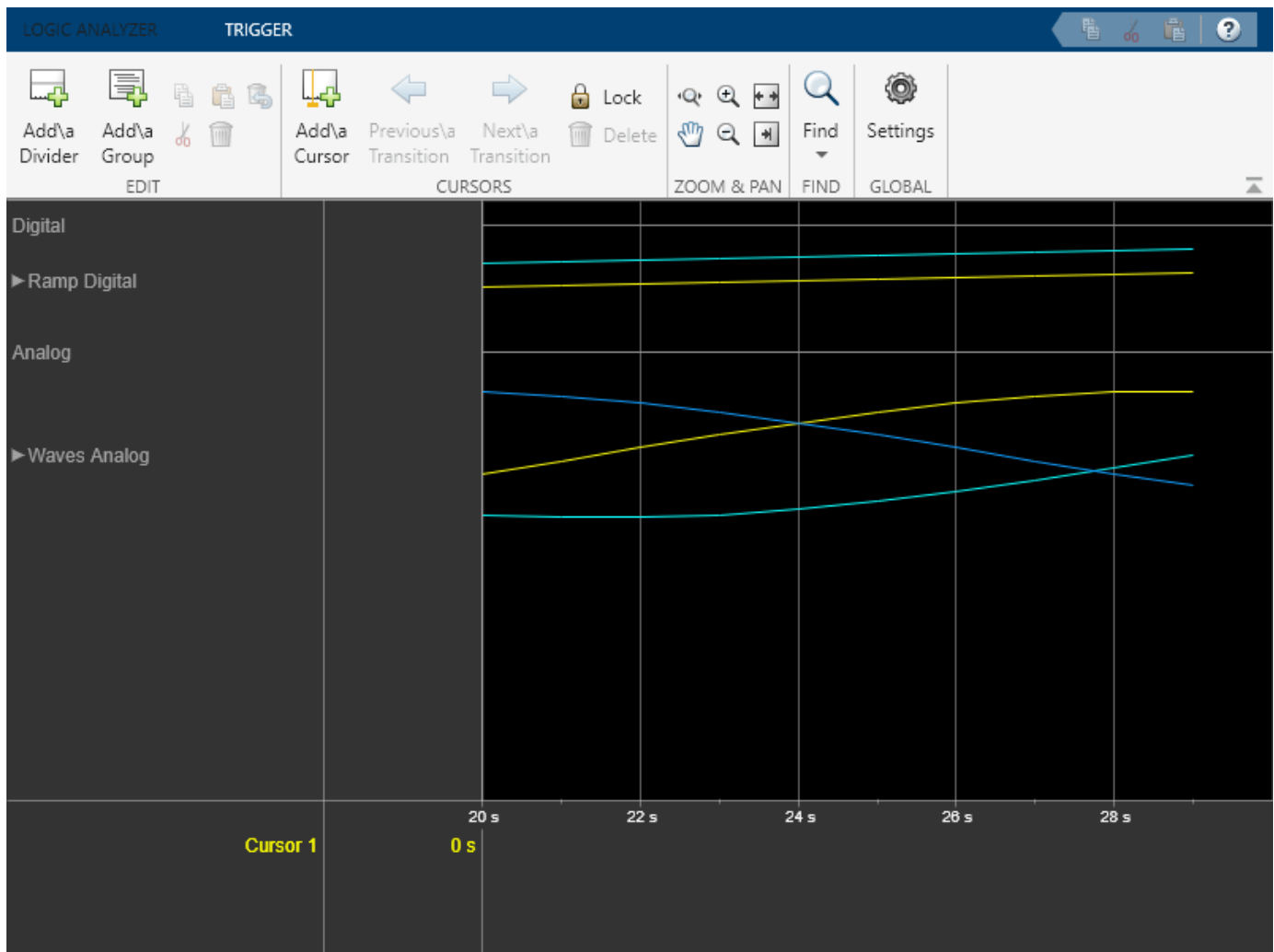
```
hide(scope)
digitalDividerTag = addDivider(scope, 'Name', 'Digital', 'Height', 20);
analogDividerTag = addDivider(scope, 'Name', 'Analog', 'Height', 40);

tags = getDisplayChannelTags(scope);

modifyDisplayChannel(scope, tags{1}, 'InputChannel', 1, ...
    'Name', 'Ramp Digital', 'Height', 40);
modifyDisplayChannel(scope, tags{2}, 'InputChannel', 2, ...
    'Name', 'Waves Analog', 'Format', 'Analog', 'Height', 80);

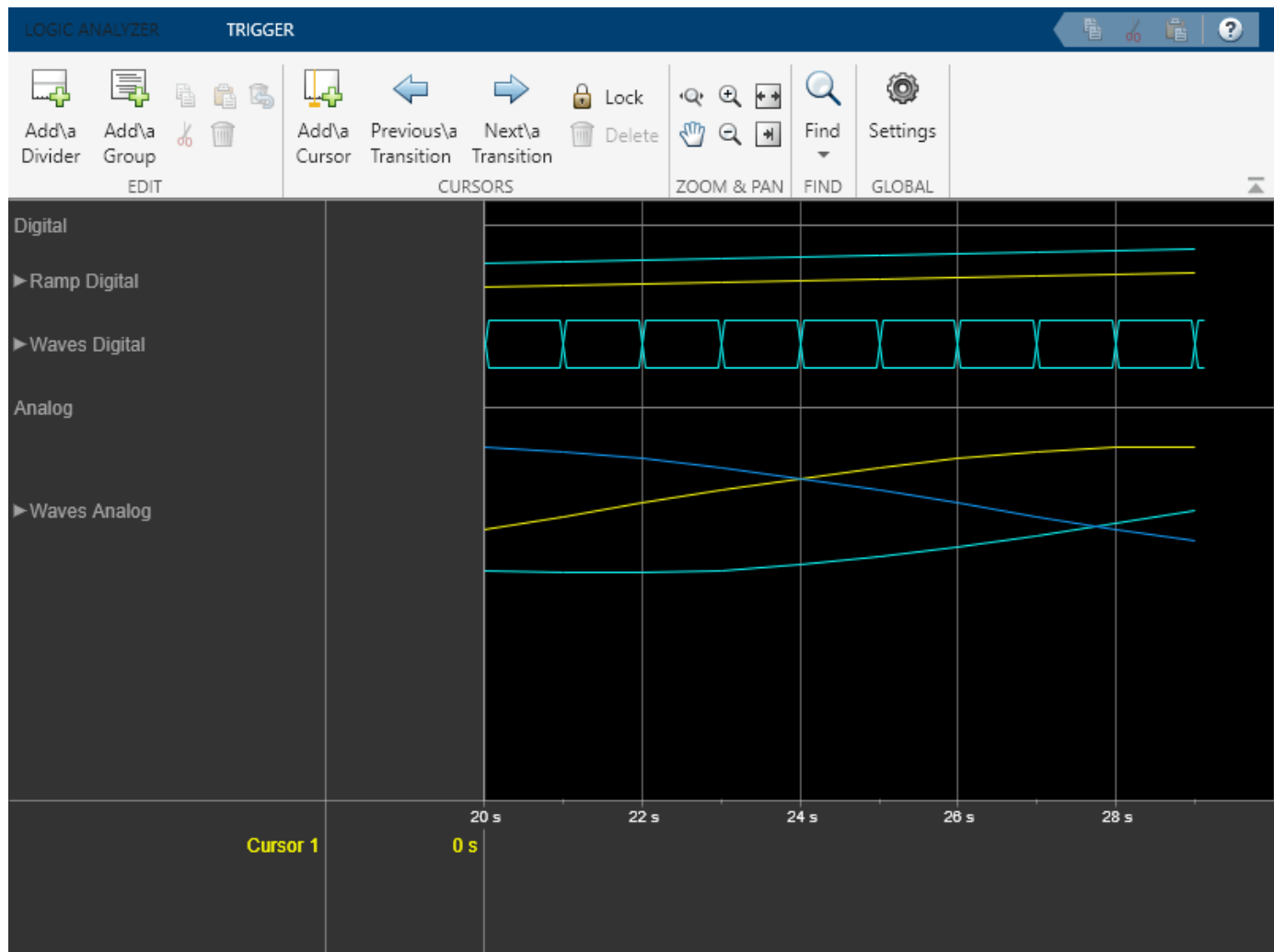
moveDisplayChannel(scope, digitalDividerTag, 'DisplayChannel', 1)
moveDisplayChannel(scope, tags{2}, 'DisplayChannel', length(tags))

show(scope)
```



### Duplicate Wave and Check Information

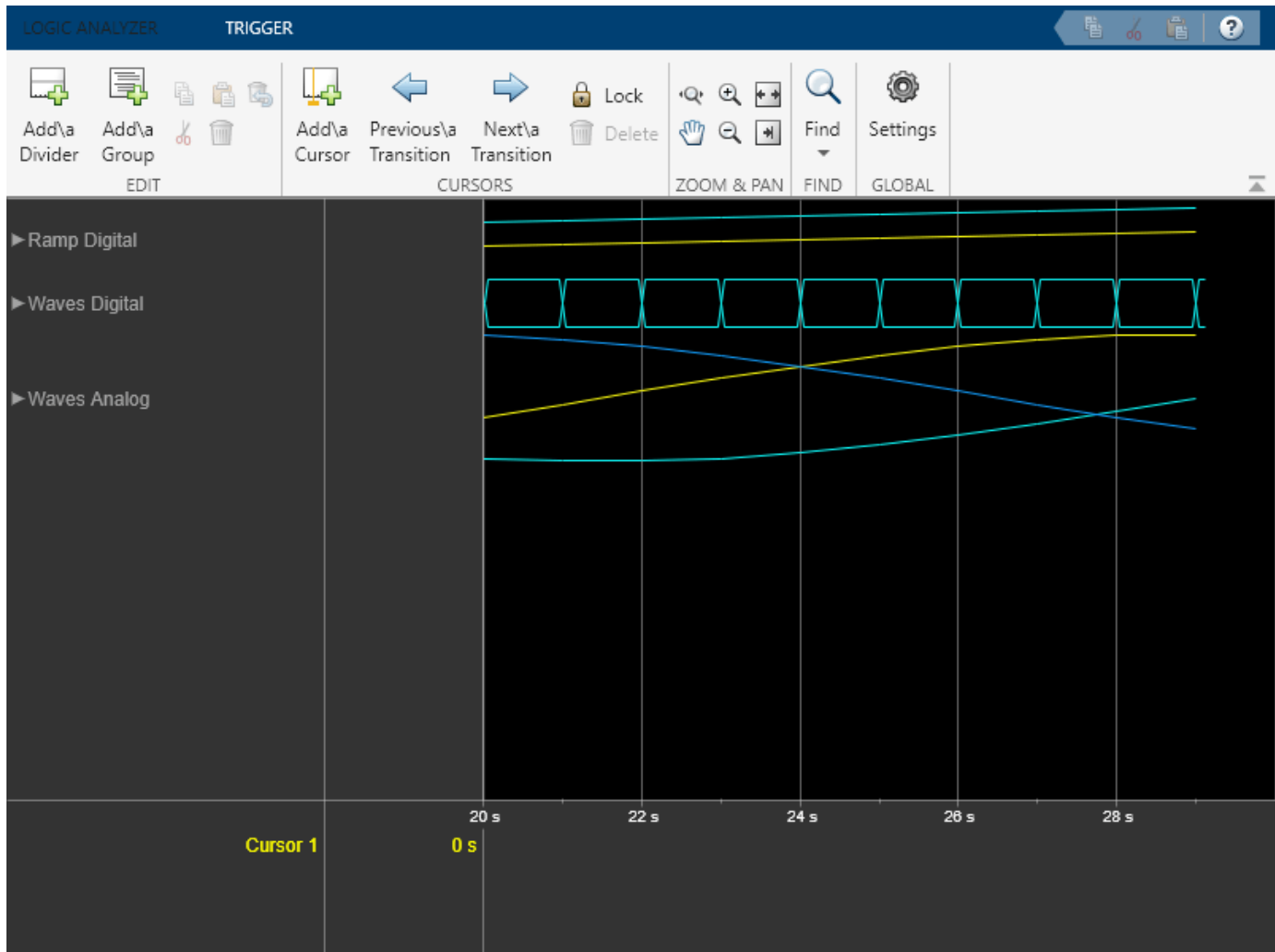
```
hide(scope)
addWave(scope, 'InputChannel', 2, 'Name', 'Waves Digital', 'Format', 'Digital', ...
'Height', 30, 'DisplayChannel', 3);
show(scope)
```



### Remove Dividers

```
hide(scope)
deleteDisplayChannel(scope,digitalDividerTag)
deleteDisplayChannel(scope,analogDividerTag)
show(scope)
```





### Clear variables

```
clear analogDividerTag cosValVec cosValVecOffset count digitalDividerTag duplicateWave scope sin
```

## Input Arguments

### scope — Logic Analyzer object

dsp.LogicAnalyzer object

The Logic Analyzer object from which you want to return display channel settings, specified as a handle to the dsp.LogicAnalyzer object.

### tag — Tag or tags identifying the display channel or channels about which to get information

character vector | string scalar | cell array of character vectors | string array

The tag or tags identifying the display channel or channels about which to get information.

Example: 'DisplayChannelTag', 'W5'

Example: 'DisplayChannelTag', {'W4', 'W5'}

Example: 'DisplayChannelTag', ["W4", "W5"]

Data Types: char | string

## Output Arguments

**channelInfo** — Information about settings for the display channel or channels  
struct

The `channelInfo` struct contains the following fields:

- `Color` — Color of the waves.
- `InputChannel` — Channel on the display that corresponds to the specified waves.
- `Radix` — Radix for the waves.
- `FontSize` — Font size for values in the waves. A value of 0 indicates that the waves inherit `FontSize` from the global `DisplayChannelColor` property.
- `Name` — The name or label for the waves.
- `Height` — Height of the wave. A value of 0 indicates that the waves inherit `Height` from the global `DisplayChannelHeight` property.
- `Tag` — Tag for the channel.

## Version History

Introduced in R2013a

### See Also

`dsp.LogicAnalyzer` | `getDisplayChannelTags` | `modifyDisplayChannel` | `deleteDisplayChannel` | `getCursorInfo`

# getDisplayChannelTags

**Package:** dsp

Return all Logic Analyzer display channel tags

## Syntax

```
displayChannelTags = getDisplayChannelTags(scope)
```

## Description

`displayChannelTags = getDisplayChannelTags(scope)` returns all the tags for waves or dividers in a Logic Analyzer. You use these tags to

- Get information about a wave or divider using `getDisplayChannelInfo`
- Modify a wave or divider using `modifyDisplayChannel`
- Delete a wave or divider using `deleteDisplayChannel`

## Examples

### Manipulate Logic Analyzer Programatically

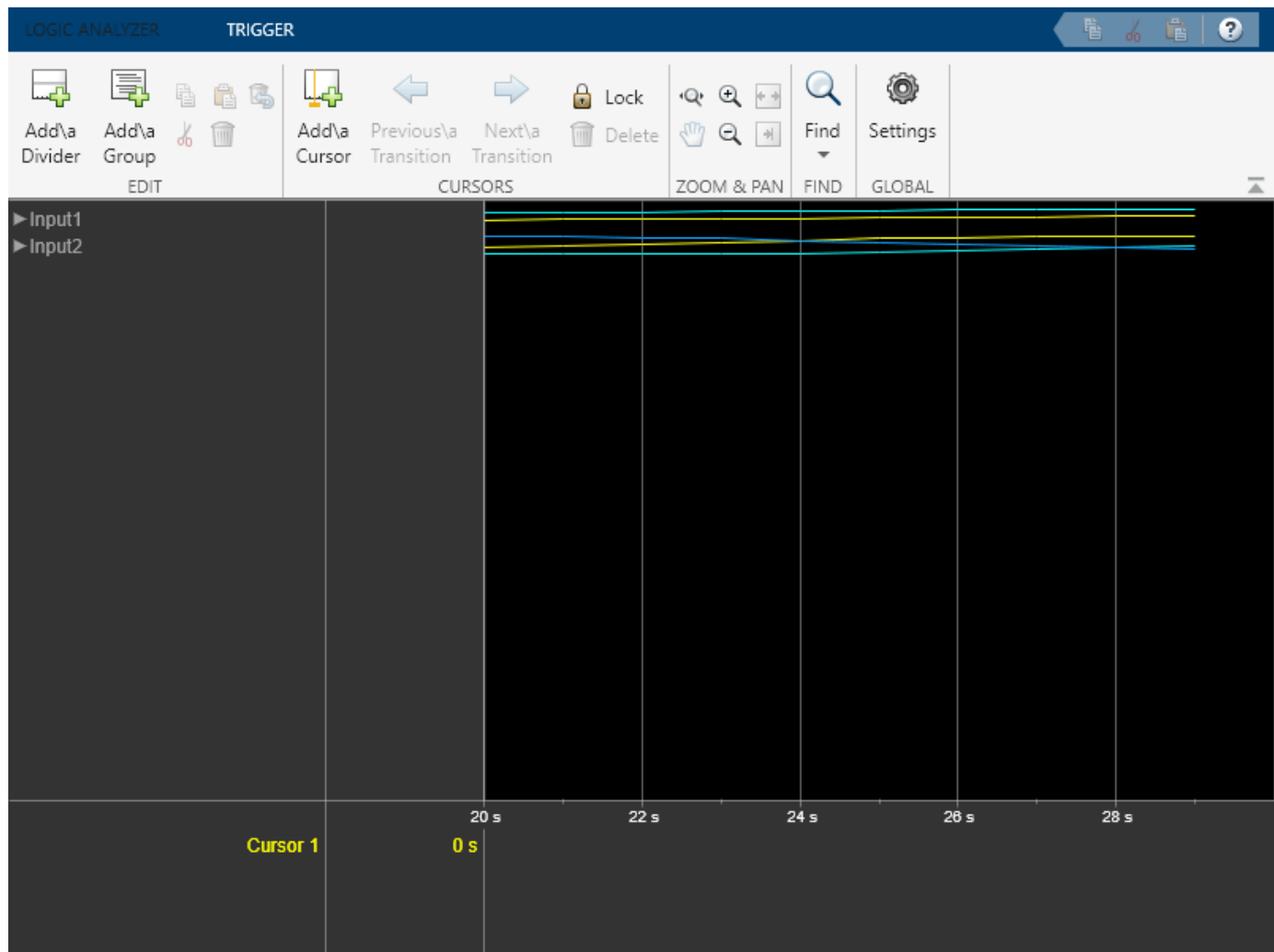
Use functions to construct and manipulate a `dsp.LogicAnalyzer` System object.

#### Display Waves on Logic Analyzer scope.

```
scope = dsp.LogicAnalyzer('NumInputPorts',2);

stop = 30;
for count = 1:stop
    sinValVec = sin(count/stop*2*pi);
    cosValVec = cos(count/stop*2*pi);
    cosValVecOffset = cos((count+10)/stop*2*pi);

    scope([count (count-(stop/2))],[sinValVec cosValVec cosValVecOffset])
end
```



### Reorganize Display

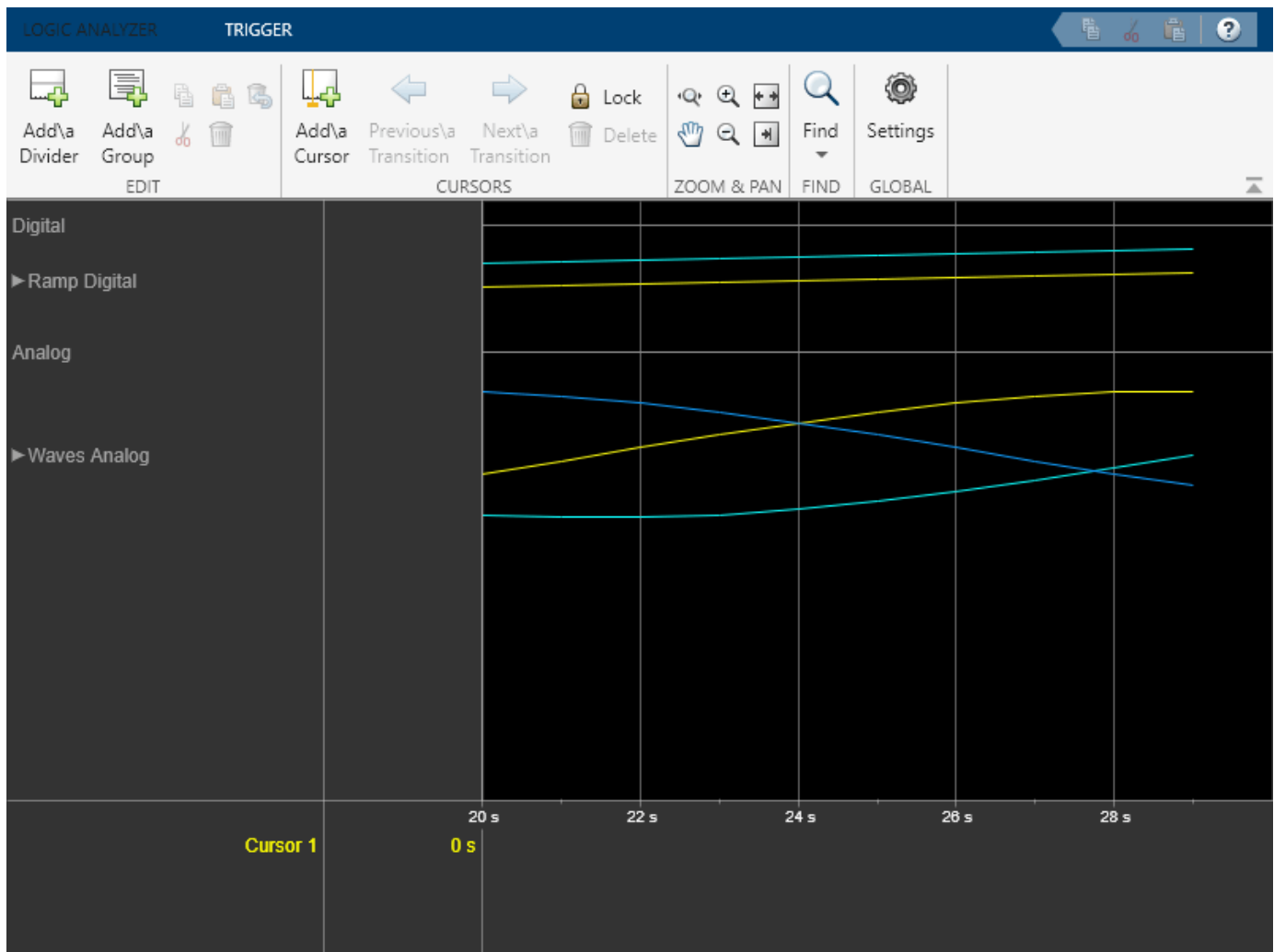
```
hide(scope)
digitalDividerTag = addDivider(scope, 'Name', 'Digital', 'Height', 20);
analogDividerTag = addDivider(scope, 'Name', 'Analog', 'Height', 40);

tags = getDisplayChannelTags(scope);

modifyDisplayChannel(scope, tags{1}, 'InputChannel', 1, ...
    'Name', 'Ramp Digital', 'Height', 40);
modifyDisplayChannel(scope, tags{2}, 'InputChannel', 2, ...
    'Name', 'Waves Analog', 'Format', 'Analog', 'Height', 80);

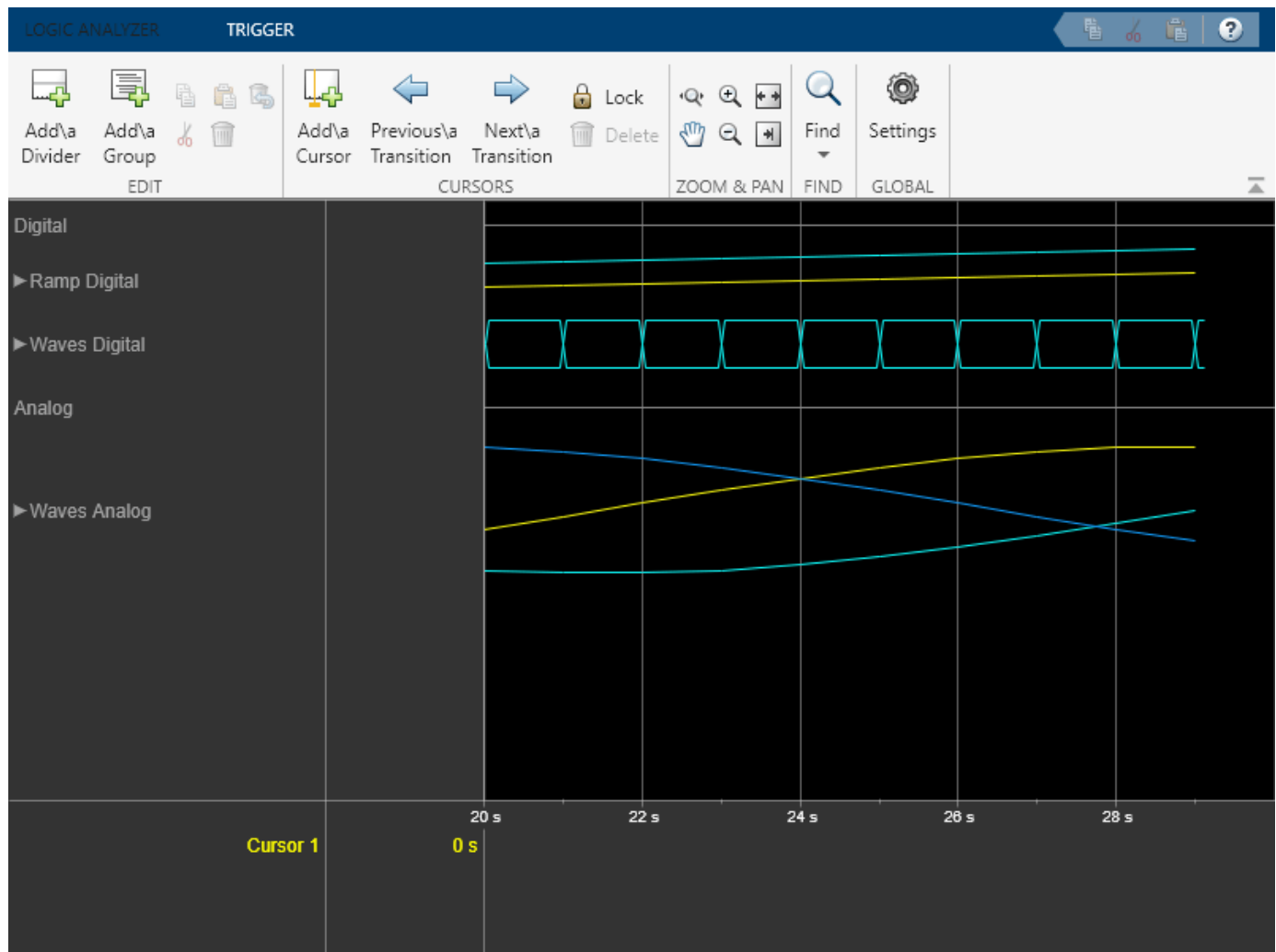
moveDisplayChannel(scope, digitalDividerTag, 'DisplayChannel', 1)
moveDisplayChannel(scope, tags{2}, 'DisplayChannel', length(tags))

show(scope)
```



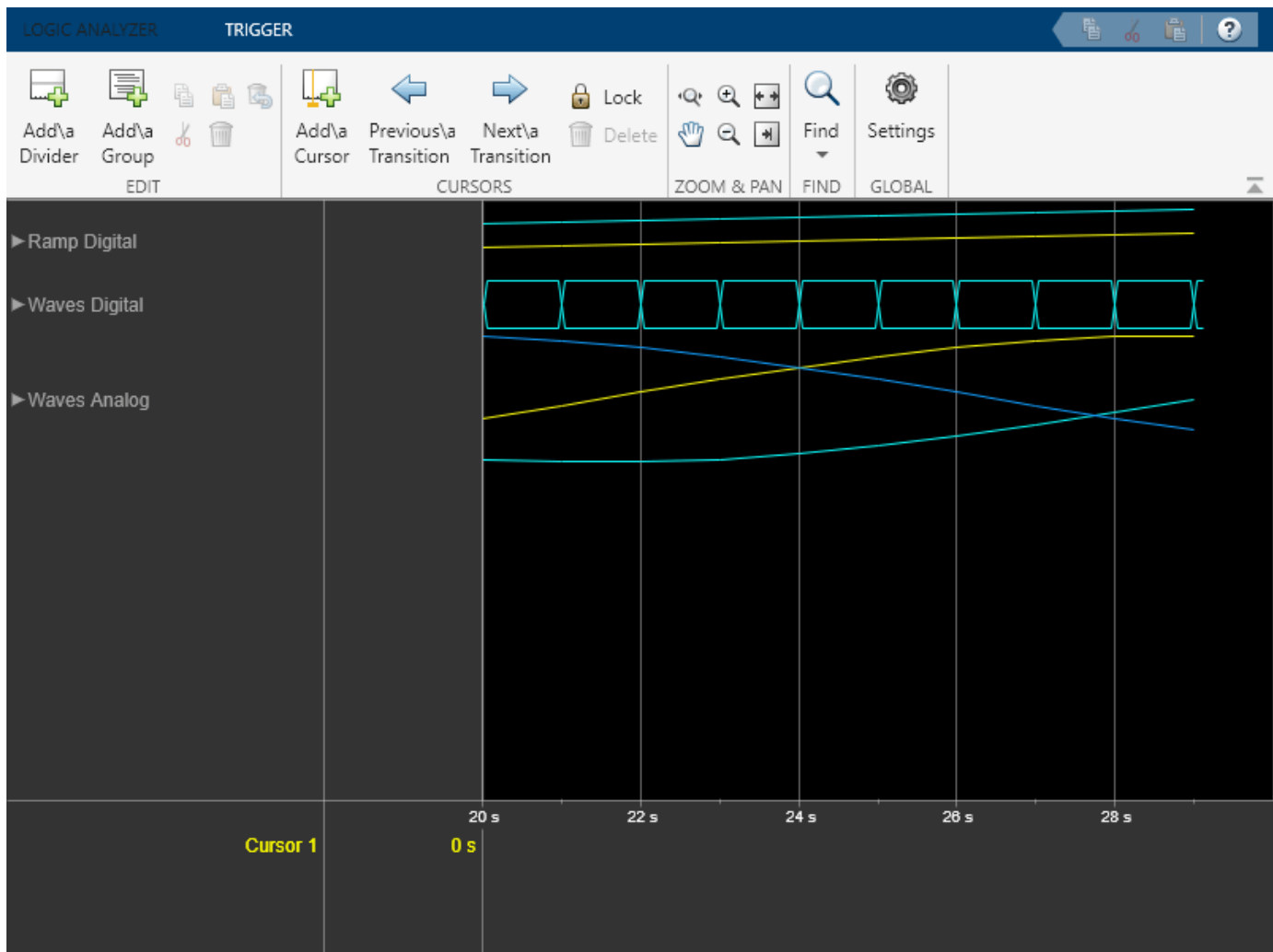
### Duplicate Wave and Check Information

```
hide(scope)
addWave(scope, 'InputChannel', 2, 'Name', 'Waves Digital', 'Format', 'Digital', ...
    'Height', 30, 'DisplayChannel', 3);
show(scope)
```



### Remove Dividers

```
hide(scope)
deleteDisplayChannel(scope,digitalDividerTag)
deleteDisplayChannel(scope,analogDividerTag)
show(scope)
```



### Clear variables

```
clear analogDividerTag cosValVec cosValVecOffset count digitalDividerTag duplicateWave scope sin
```

### Input Arguments

**scope** — The Logic Analyzer object from which you want to return all display channel tags  
 dsp.LogicAnalyzer object

The Logic Analyzer object from which you want to return all display channel tags, specified as a handle to the dsp.LogicAnalyzer object.

### Output Arguments

**displayChannelTags** — The display channel tags  
 cell array of character vectors

The display channel tags, returned as a cell array of character vectors.

Example: { 'W1' }

Example: { 'W1', 'W2', 'W3' }

Data Types: cell

## **Version History**

**Introduced in R2013a**

### **See Also**

[dsp.LogicAnalyzer](#) | [getDisplayChannelInfo](#) | [modifyDisplayChannel](#) | [moveDisplayChannel](#) | [getCursorTags](#)



# getFilterOrders

**Package:** dsp

Get orders of digital down converter or digital up converter filter cascade

## Syntax

```
S = getFilterOrders(Conv)
```

## Description

`S = getFilterOrders(Conv)` returns a structure, `S`, that contains the number of CIC filter sections and the orders of the FIR filter stages of a digital down converter or digital up converter, `Conv`. The converter usually implements the conversion using three filter stages. Sometimes, one of the stages is bypassed and the order of that filter stage is returned as an empty field.

## Examples

### Get Filter Orders of Digital Up Converter

Get orders of each decimation filter stage of the `dsp.DigitalUpConverter` System object™.

Create a `dsp.DigitalUpConverter` System object with the default settings.

```
upConv = dsp.DigitalUpConverter
upConv =
    dsp.DigitalUpConverter with properties:
        InterpolationFactor: 100
        MinimumOrderDesign: true
        Bandwidth: 200000
        StopbandFrequencySource: 'Auto'
        PassbandRipple: 0.1000
        StopbandAttenuation: 60
        Oscillator: 'Sine wave'
        CenterFrequency: 14000000
        SampleRate: 300000
```

Show all properties

Using the `getFilterOrders` function, obtain the number of CIC decimator sections, order of the CIC compensation filter stage, and order of the third filter stage.

```
S = getFilterOrders(upConv)
S = struct with fields:
    FirstFilterOrder: 24
    SecondFilterOrder: 12
    NumCICSections: 4
```

The first filter order field is empty when the object bypasses the first filter stage.

### Get Filter Orders of Digital Up Converter

Get orders of each decimation filter stage of the `dsp.DigitalUpConverter` System object™.

Create a `dsp.DigitalUpConverter` System object with the default settings.

```
upConv = dsp.DigitalUpConverter

upConv =
  dsp.DigitalUpConverter with properties:

    InterpolationFactor: 100
    MinimumOrderDesign: true
    Bandwidth: 200000
    StopbandFrequencySource: 'Auto'
    PassbandRipple: 0.1000
    StopbandAttenuation: 60
    Oscillator: 'Sine wave'
    CenterFrequency: 14000000
    SampleRate: 300000
```

Show all properties

Using the `getFilterOrders` function, obtain the number of CIC decimator sections, order of the CIC compensation filter stage, and order of the third filter stage.

```
S = getFilterOrders(upConv)

S = struct with fields:
    FirstFilterOrder: 24
    SecondFilterOrder: 12
    NumCICSections: 4
```

The first filter order field is empty when the object bypasses the first filter stage.

## Input Arguments

### Conv — Digital down converter or digital up converter

`dsp.DigitalDownConverter` | `dsp.DigitalUpConverter`

Digital down converter or digital up converter, specified as a `dsp.DigitalDownConverter` or `dsp.DigitalUpConverter` System object.

## Output Arguments

### S — Filter order information

structure

Filter order information, returned as a structure containing the number of CIC sections, and the orders of the two FIR filter stages. For a digital down converter object, the structure contains these fields:

- `NumCICSections` -- Number of sections of the CIC decimator. The default is 4.
- `SecondFilterOrder` -- Order of the CIC compensation filter stage. The default is 12.
- `ThirdFilterOrder` -- Order of the third filter stage. The default is 24. The `ThirdFilterOrder` structure field is empty if the third filter stage has been bypassed.

For a digital up converter object, the structure contains these fields:

- `FirstFilterOrder` -- Order of the first filter stage. The default is 24. The `FirstFilterOrder` structure field is empty if the first filter stage has been bypassed.
- `SecondFilterOrder` -- Order of the CIC compensation filter stage. The default is 12.
- `NumCICSections` -- Number of sections of the CIC interpolator. The default is 4.

## Version History

Introduced in R2012a

### See Also

#### Functions

`getInterpolationFactors` | `getDecimationFactors` | `getFilters` | `fvtool` | `groupDelay` | `visualizeFilterStages`

#### Objects

`dsp.DigitalDownConverter` | `dsp.DigitalUpConverter`

## getFilters

**Package:** dsp

Get handles to digital down converter or digital up converter filter cascade objects

### Syntax

```
S = getFilters(Conv)
S = getFilters(Conv, 'Arithmetic', arithType)
```

### Description

`S = getFilters(Conv)` returns a structure, `S`, that contains copies of the filter System objects and the CIC normalization factor that form a digital down converter or digital up converter, `Conv`. The CIC normalization factor equals the inverse of the CIC filter gain. This gain can include a correction factor to ensure that the cascade response meets the ripple specifications.

The converter usually implements the conversion using three filter stages. Sometimes, one of the stages is bypassed and that filter stage is returned as an empty field.

`S = getFilters(Conv, 'Arithmetic', arithType)` specifies the arithmetic type of the filter stages. Set `arithType` to `'double'`, `'single'`, or `'Fixed-point'`. When the `Conv` object is in an unlocked state, you must specify the arithmetic input. When the `Conv` object is in a locked state, it ignores the arithmetic input argument.

### Examples

#### Get Filters of Digital Down Converter

Get handles to decimation filter objects of the `dsp.DigitalDownConverter` System object™.

Create a `dsp.DigitalDownConverter` System object with the default settings.

```
dwnConv = dsp.DigitalDownConverter

dwnConv =
  dsp.DigitalDownConverter with properties:

    DecimationFactor: 100
  MinimumOrderDesign: true
    Bandwidth: 200000
  StopbandFrequencySource: 'Auto'
    PassbandRipple: 0.1000
  StopbandAttenuation: 60
    Oscillator: 'Sine wave'
    CenterFrequency: 14000000
    SampleRate: 30000000
```

Show all properties

Use the `getFilters` function to obtain the filter System objects and the CIC normalization factor that form the decimation filter cascade.

To use the `getFilters` function on an unlocked System object, you must specify the filter arithmetic through the 'Arithmetic' input of the `getFilters` function.

```
S = getFilters(dwnConv, 'Arithmetic', 'Fixed-point')

S = struct with fields:
    CICDecimator: [1x1 dsp.CICDecimator]
    CICNormalizationFactor: 2.5600e-06
    SecondFilterStage: [1x1 dsp.FIRDecimator]
    ThirdFilterStage: [1x1 dsp.FIRDecimator]
```

Alternatively, you can lock the System object by passing a valid input to the object algorithm. In this case, you can use the `getFilters` function without specifying the filter arithmetic.

### Get Filters of Digital Up Converter

Get handles to decimation filter objects of the `dsp.DigitalUpConverter` System object™.

Create a `dsp.DigitalUpConverter` System object with the default settings.

```
upConv = dsp.DigitalUpConverter

upConv =
    dsp.DigitalUpConverter with properties:

        InterpolationFactor: 100
        MinimumOrderDesign: true
        Bandwidth: 200000
    StopbandFrequencySource: 'Auto'
        PassbandRipple: 0.1000
    StopbandAttenuation: 60
        Oscillator: 'Sine wave'
        CenterFrequency: 14000000
        SampleRate: 300000
```

Show all properties

Use the `getFilters` function to obtain the filter System objects and the CIC normalization factor that form the decimation filter cascade.

To use the `getFilters` function on an unlocked System object, you must specify the filter arithmetic through the 'Arithmetic' input of the `getFilters` function.

```
S = getFilters(upConv, 'Arithmetic', 'Fixed-point')

S = struct with fields:
    FirstFilterStage: [1x1 dsp.FIRInterpolator]
    CICNormalizationFactor: 6.4000e-05
    SecondFilterStage: [1x1 dsp.FIRInterpolator]
    CICInterpolator: [1x1 dsp.CICInterpolator]
```

The first filter field is empty when the object bypasses the first filter stage.

Alternatively, you can lock the System object by passing a valid input to the object algorithm. In this case, you can use the `getFilters` function without specifying the filter arithmetic.

## Input Arguments

### Conv — Digital down converter or digital up converter

`dsp.DigitalDownConverter` | `dsp.DigitalUpConverter`

Digital down converter or digital up converter, specified as a `dsp.DigitalDownConverter` or `dsp.DigitalUpConverter` System object.

### arithType — Arithmetic type

'double' (default) | 'single' | 'fixed-point'

When the Conv object is in an unlocked state, you must specify the arithmetic type. When the Conv object is in a locked state, it ignores the arithmetic input argument.

When Conv is in an unlocked state, and you specify the arithmetic type as 'fixed-point', the `getFilters` function returns filter System objects. The custom coefficient data type properties of these System objects are set to the values that the `dsp.DigitalDownConverter` or `dsp.DigitalUpConverter` objects use to process data when you call the object. All other fixed-point properties are set to their default values.

When Conv is in a locked state, and the input to the object algorithm is of a fixed-point data type, the `getFilters` function returns filter System objects. All fixed-point properties of these System objects are set to the exact values that the `dsp.DigitalDownConverter` or `dsp.DigitalUpConverter` objects use to process the data.

## Output Arguments

### S — Handles to decimation or interpolation filter objects

structure

The output is a structure, S, containing three filter object handles and the `CICNormalizationFactor`. For a digital down converter object, the structure contains these fields:

- `CICDecimator` -- Handle to the `dsp.CICDecimator` object.
- `CICNormalizationFactor` -- Inverse of the CIC filter gain.
- `SecondFilterStage` -- Handle to the CIC compensation filter stage.
- `ThirdFilterStage` -- Handle to the third filter stage. This field is empty if the third filter stage has been bypassed.

For a digital up converter object, the structure contains these fields:

- `FirstFilterStage` -- Handle to the first filter stage. This field is empty if the first filter stage has been bypassed.
- `CICNormalizationFactor` -- Inverse of the CIC filter gain.
- `SecondFilterStage` -- Handle to the CIC compensation filter stage.

- `CICInterpolator` -- Handle to the `dsp.CICInterpolator` object.

## Version History

Introduced in R2012a

### See Also

#### Functions

`getInterpolationFactors` | `getDecimationFactors` | `getFilterOrders` | `fvtool` | `groupDelay` | `visualizeFilterStages`

#### Objects

`dsp.DigitalDownConverter` | `dsp.DigitalUpConverter`

## groupDelay

**Package:** dsp

Group delay of digital down converter or digital up converter filter cascade

### Syntax

```
D = groupDelay(Conv,N)
[D,F] = groupDelay(Conv,N)
```

### Description

`D = groupDelay(Conv,N)` returns a vector of group delays, `D`, of a digital down converter or digital up converter, `Conv`, evaluated at `N` frequency points. The frequency points are equally spaced around the upper half of the unit circle.

`[D,F] = groupDelay(Conv,N)` returns a vector of frequencies, `F`, at which the group delay has been computed.

### Examples

#### Group Delays of Digital Down Converter

Compute the group delays of the digital down converter using the `groupDelay` function.

Create a `dsp.DigitalDownConverter` System object with the default settings.

```
dwnConv = dsp.DigitalDownConverter

dwnConv =
    dsp.DigitalDownConverter with properties:

        DecimationFactor: 100
        MinimumOrderDesign: true
            Bandwidth: 200000
    StopbandFrequencySource: 'Auto'
        PassbandRipple: 0.1000
    StopbandAttenuation: 60
        Oscillator: 'Sine wave'
        CenterFrequency: 14000000
        SampleRate: 30000000

    Show all properties
```

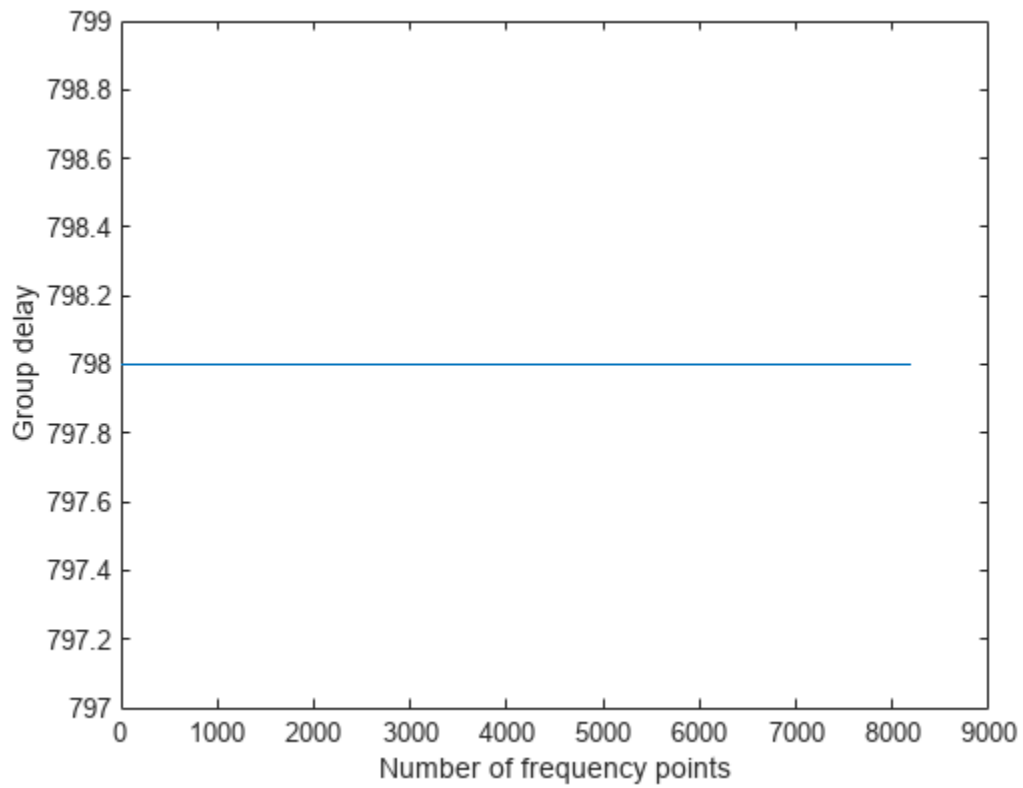
Use the `groupDelay` function to compute the vector of group delays. By default, the function evaluates the group delays at 8192 frequency points equally spaced around the upper half of the unit circle.

```
D = groupDelay(dwnConv);
```



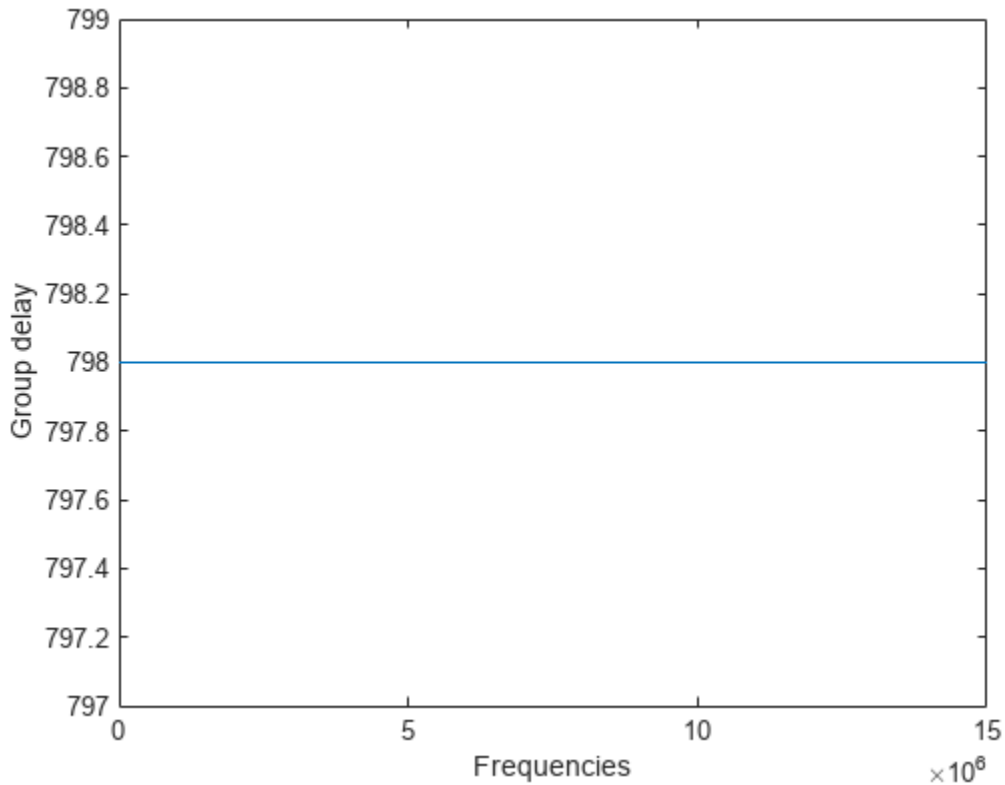
Plot the vector of group delays.

```
plot(D)
xlabel('Number of frequency points')
ylabel('Group delay')
```



Determine the vector of frequencies over which the group delays are computed and plot them.

```
[D,F] = groupDelay(dwnConv);
plot(F,D)
xlabel('Frequencies')
ylabel('Group delay')
```



## Input Arguments

### Conv — Digital down converter or digital up converter

`dsp.DigitalDownConverter` | `dsp.DigitalUpConverter`

Digital down converter or digital up converter, specified as a `dsp.DigitalDownConverter` or `dsp.DigitalUpConverter` System object.

### N — Number of frequency points

8192 (default) | positive scalar

Number of frequency points over which the group delays are evaluated, specified as a positive scalar. These points are equally spaced around the upper half of the unit circle.

Data Types: `double` | `single`

## Output Arguments

### D — Vector of group delays

column vector

Vector of group delays of the digital down converter or digital up converter, evaluated at `N` frequency points equally spaced around the upper half of the unit circle.

Data Types: `double`

**F — Vector of frequencies**

column vector

Frequencies at which the group delays are evaluated, returned as a column vector.

Data Types: double

**Version History**

Introduced in R2012a

**See Also****Functions**[getDecimationFactors](#) | [getInterpolationFactors](#) | [getFilterOrders](#) | [getFilters](#) | [fvtool](#) | [visualizeFilterStages](#)**Objects**[dsp.DigitalDownConverter](#) | [dsp.DigitalUpConverter](#)

## getFixedPointInfo

**Package:** dsp

Get fixed-point word and fraction lengths

### Syntax

```
[WLs, FLs] = getFixedPointInfo(cicObj,nt)
```

### Description

[WLs, FLs] = getFixedPointInfo(cicObj,nt) returns all the word lengths and fraction lengths of the fixed-point sections and the output of the CIC filter System objects, dsp.CICDecimator and dsp.CICInterpolator, based on the input numeric type, nt. For locked objects or when the FixedPointDataType property of the unlocked CIC filter object is set to 'Specify word and fraction lengths', the input numeric type argument, nt, is optional.

### Examples

#### Determine the Section and Output Word Lengths and Fraction Lengths

Using the getFixedPointInfo function, you can determine the word lengths and fraction lengths of the fixed-point sections and the output of the dsp.CICDecimator and dsp.CICInterpolator System objects. The data types of the filter sections and the output depend on the FixedPointDataType property of the filter System object™.

#### Full precision

Create a dsp.CICDecimator object. The default value of the NumSections property is 2. This value indicates that there are two integrator and comb sections. The WLs and FLs vectors returned by the getFixedPointInfo function contain five elements each. The first two elements represent the two integrator sections. The third and fourth elements represent the two comb sections. The last element represents the filter output.

```
cicD = dsp.CICDecimator

cicD =
  dsp.CICDecimator with properties:
    DecimationFactor: 2
    DifferentialDelay: 1
    NumSections: 2
    FixedPointDataType: 'Full precision'
```

By default, the FixedPointDataType property of the object is set to 'Full precision'. Calling the getFixedPointInfo function on this object with the input numeric type, nt, yields the following word length and fraction length vectors.

```
nt = numerictype(1,16,15)
```

```

nt =

    DataTypeMode: Fixed-point: binary point scaling
    Signedness: Signed
    WordLength: 16
    FractionLength: 15

```

```
[WLs,FLs] = getFixedPointInfo(cicD,nt) %#ok
```

```

WLs = 1x5

    18    18    18    18    18

```

```

FLs = 1x5

    15    15    15    15    15

```

For details on how the word lengths and fraction lengths are computed, see the description for *Output Arguments*.

If you lock the `cicD` object by passing an input to its algorithm, you do not need to pass the `nt` argument to the `getFixedPointInfo` function.

```

input = int64(randn(8,1))

input = 8x1 int64 column vector

     1
     2
    -2
     1
     0
    -1
     0
     0

```

```
output = cicD(input)
```

```

output =

     0
     1
     3
     0

    DataTypeMode: Fixed-point: binary point scaling
    Signedness: Signed
    WordLength: 66
    FractionLength: 0

```

```
[WLs,FLs] = getFixedPointInfo(cicD) %#ok
```

```

WLs = 1x5

    66    66    66    66    66

```

```
FLs = 1x5
      0      0      0      0      0
```

The output and section word lengths are the sum of input word length, 64 in this case, and the number of sections, 2. The output and section fraction lengths are 0 since the input is a built-in integer.

### Minimum section word lengths

Release the object and change the `FixedPointDataType` property to 'Minimum section word lengths'. Determine the section and output fixed-point information when the input is fixed-point data, `fi(randn(8,2),1,24,15)`.

```
release(cicD);
cicD.FixedPointDataType = 'Minimum section word lengths'
```

```
cicD =
  dsp.CICDecimator with properties:

    DecimationFactor: 2
    DifferentialDelay: 1
    NumSections: 2
    FixedPointDataType: 'Minimum section word lengths'
    OutputWordLength: 32
```

```
inputF = fi(randn(8,2),1,24,15)
```

```
inputF =
  3.5784   -0.1241
  2.7694    1.4897
 -1.3499    1.4090
  3.0349    1.4172
  0.7254    0.6715
 -0.0630   -1.2075
  0.7148    0.7172
 -0.2050    1.6302

    DataTypeMode: Fixed-point: binary point scaling
    Signedness: Signed
    WordLength: 24
    FractionLength: 15
```

```
[WLs, FLs] = getFixedPointInfo(cicD,numericType(inputF)) %#ok
```

```
WLs = 1x5
      26      26      26      26      32
```

```
FLs = 1x5
      15      15      15      15      21
```

### Specify word and fraction lengths

Change the `FixedPointDataType` property to 'Specify word and fraction lengths'. Determine the fixed-point information using the `getFixedPointInfo` function.

```
cicD.FixedPointDataType = 'Specify word and fraction lengths'

cicD =
    dsp.CICDecimator with properties:

        DecimationFactor: 2
        DifferentialDelay: 1
        NumSections: 2
        FixedPointDataType: 'Specify word and fraction lengths'
        SectionWordLengths: [16 16 16 16]
        SectionFractionLengths: 0
        OutputWordLength: 32
        OutputFractionLength: 0
```

```
[WLs, FLs] = getFixedPointInfo(cicD,numericType(inputF)) %#ok
```

```
WLs = 1x5
```

```
    16    16    16    16    32
```

```
FLs = 1x5
```

```
     0     0     0     0     0
```

The section and output word lengths and fraction lengths are assigned as per the respective fixed-point properties of the `cicD` object. These values are not determined by the input numeric type. To confirm, call the `getFixedPointInfo` function without passing the `numericType` input argument.

```
[WLs, FLs] = getFixedPointInfo(cicD) %#ok
```

```
WLs = 1x5
```

```
    16    16    16    16    32
```

```
FLs = 1x5
```

```
     0     0     0     0     0
```

### Specify word lengths

To specify the word lengths of the filter section and output, set the `FixedPointDataType` property to 'Specify word lengths'.

```
cicD.FixedPointDataType = 'Specify word lengths'
```

```
cicD =
    dsp.CICDecimator with properties:
```

```
        DecimationFactor: 2
```

```

    DifferentialDelay: 1
      NumSections: 2
    FixedPointDataType: 'Specify word lengths'
    SectionWordLengths: [16 16 16 16]
    OutputWordLength: 32

```

The `getFixedPointInfo` function requires the input numeric type because that information is used to compute the section and word fraction lengths.

```
[WLs, FLs] = getFixedPointInfo(cicD, numerictype(inputF))
```

```
WLs = 1x5
```

```
    16    16    16    16    32
```

```
FLs = 1x5
```

```
     5     5     5     5    21
```

For more details on how the function computes the word and fraction lengths, see the description for *Output Arguments*.

## Input Arguments

### **cicObj** — CIC filter System object

`dsp.CICDecimator` | `dsp.CICInterpolator`

CIC filter System object, specified as either a `dsp.CICDecimator` or `dsp.CICInterpolator` System object.

### **nt** — Input numeric type

`numerictype`

Input data numeric type, specified as a `numerictype` object. Specify this input when the System object is unlocked and the `FixedPointDataType` property of the CIC filter is set to 'Full precision', 'Minimum section word lengths', or 'Specify word lengths'. If the `FixedPointDataType` property is set to 'Specify word and fraction lengths', the word and fraction lengths of the filter sections and the output are specified through the object parameters. In this case, the `nt` input is optional. Alternatively, if the object is locked, the fixed-point data input to the object specifies the input word length and fraction length. The `nt` argument in this case is also optional.

Example: `numerictype(1,16,15)`

Example: `input = fi(randn(16,1),1,32,30); numerictype(input)`

## Output Arguments

### **WLs** — Section and output word lengths

row vector

Section and output word lengths, returned as a row vector. The first  $2 \times \text{NumSections}$  elements in the row vector correspond to the word lengths of the integrator and comb sections of the CIC filter.



The value of the `NumSections` property specifies the number of sections in either the integrator part or the comb part of the filter. The last element in the vector corresponds to the word length of the object output.

The word length of the CIC filter sections and the object output depend on the `FixedPointDataType` property of the CIC filter object.

### Full precision

When the `FixedPointDataType` property of the CIC filter object is set to 'Full precision', the section and output word lengths are computed using the following equation:

$$WL_{\text{section}} = WL_{\text{input}} + NumSect$$

$$WL_{\text{output}} = WL_{\text{input}} + NumSect$$

where,

- $WL_{\text{section}}$  -- Word length of the CIC filter section.
- $WL_{\text{output}}$  -- Word length of the output data.
- $WL_{\text{input}}$  -- Word length of the input data.
- $NumSect$  -- Number of CIC filter sections specified through the `NumSections` property.

For locked objects,  $WL_{\text{input}}$  is inherited from the data input you pass to the object algorithm. For unlocked objects, the  $WL_{\text{input}}$  is inherited from the `nt` argument.

### Minimum section word lengths

When the `FixedPointDataType` property is set to 'Minimum section word lengths', the section word length is given by the following equation:

$$WL_{\text{section}} = WL_{\text{input}} + NumSect$$

The output word length is the value you specify in `OutputWordLength` property of the CIC filter object.

### Specify word and fraction lengths

When the `FixedPointDataType` property is set to 'Specify word and fraction lengths', the section word lengths and output word length are the values you specify in the `SectionWordLengths` and `OutputWordLength` properties of the CIC filter object.

### Specify word lengths

When the `FixedPointDataType` property is set to 'Specify word lengths', the section word lengths and the output word length are the values you specify in the `SectionWordLengths` and `OutputWordLength` properties of the CIC filter object.

Example: [20 20 20 20 20]

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### FLs — Section and output fraction lengths

row vector

Section and output fraction lengths, returned as a row vector. The first  $2 \times NumSections$  elements in the row vector correspond to the fraction lengths of the integrator and comb sections of the CIC filter.

The value of `NumSections` property specifies the number of sections in either the integrator part or the comb part of the filter. The last element in the vector corresponds to the fraction length of the object output.

The fraction length of the CIC filter sections and the object output depend on the `FixedPointDataType` property of the CIC filter object.

### Full precision

When the `FixedPointDataType` property of the CIC filter object is set to 'Full precision', the section and output fraction lengths are computed using the following equation:

$$FL_{\text{section}} = WL_{\text{section}} - (WL_{\text{input}} - FL_{\text{input}} + NumSect)$$

$$FL_{\text{output}} = WL_{\text{output}} - (WL_{\text{input}} - FL_{\text{input}} + NumSect)$$

For inputs of built-in integer data types, the section and output fraction lengths are 0.

### Minimum section word lengths

When the `FixedPointDataType` property of the CIC filter object is set to 'Minimum section word lengths', the section and output fraction lengths,  $FL_{\text{section}}$  and  $FL_{\text{output}}$  are given by the following equation:

$$FL_{\text{section}} = WL_{\text{section}} - (WL_{\text{input}} - FL_{\text{input}} + NumSect)$$

$$FL_{\text{output}} = WL_{\text{output}} - (WL_{\text{input}} - FL_{\text{input}} + NumSect)$$

For inputs of built-in integer data types, the section and output fraction lengths are 0.

### Specify word and fraction lengths

When the `FixedPointDataType` property of the CIC filter object is set to 'Specify word and fraction lengths', the section and output fraction lengths are the values you specify in the `SectionFractionLengths` and `OutputFractionLength` properties.

### Specify word lengths

When the `FixedPointDataType` property of the CIC filter object is set to 'Specify word lengths', the section and output fraction lengths,  $FL_{\text{section}}$  and  $FL_{\text{output}}$  are given by the following equation:

$$FL_{\text{section}} = WL_{\text{section}} - (WL_{\text{input}} - FL_{\text{input}} + NumSect)$$

$$FL_{\text{output}} = WL_{\text{output}} - (WL_{\text{input}} - FL_{\text{input}} + NumSect)$$

Example: [12 12 12 12 12]

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

## Version History

Introduced in R2018a

## References

- [1] Hogenauer, E.B. "An Economical Class of Digital Filters for Decimation and Interpolation." *IEEE Transactions on Acoustics, Speech and Signal Processing*. Volume 29, Number 2, 1981, 155-162.
- [2] Meyer-Baese, U. *Digital Signal Processing with Field Programmable Gate Arrays*. New York: Springer, 2001.
- [3] Harris, Fredric J. *Multirate Signal Processing for Communication Systems*. Indianapolis, IN: Prentice Hall PTR, 2004.

## See Also

### Functions

generatehdl | impz | freqz | phasez | fvtool

### Objects

dsp.CICDecimator | dsp.CICInterpolator

## getFilters

**Package:** dsp

Return matrix of channelizer FIR filters

### Syntax

```
B = getFilters(obj)
B = getFilters(obj,ind)
```

### Description

`B = getFilters(obj)` returns a matrix `B` of filter coefficients corresponding to each filter in the `dsp.Channelizer` System object filter bank. Each row contains the coefficients for the corresponding bandpass filter. The channelizer does not actually use all these filters in the implementation. It only uses the prototype lowpass filter (the first row of matrix `B`) and an FFT to implement the filter bank. The combination of polyphase implementation of the prototype lowpass and the FFT effectively implements all the filters in `B`, but does so in a very efficient manner.

`B = getFilters(obj,ind)` returns the filters with indices corresponding to the elements in the vector `ind`. `ind` is a row vector of indices between 1 and `obj.NumFrequencyBands`. By default, this vector is `[1:N]`, where `N` is the number of frequency bands specified in the `obj.NumFrequencyBands` property.

### Examples

#### Coefficients of Channelizer Filters

Using the `getFilters` function, you can access the coefficients of the lowpass prototype filter and the modulated bandpass filters of the channelizer.

```
channelizer = dsp.Channelizer;
B = getFilters(channelizer);
```

The first row corresponds to the coefficients of the prototype filter. The subsequent rows correspond to the coefficients of the respective modulated filters. Compare the first row with the coefficients returned by the `tf` function.

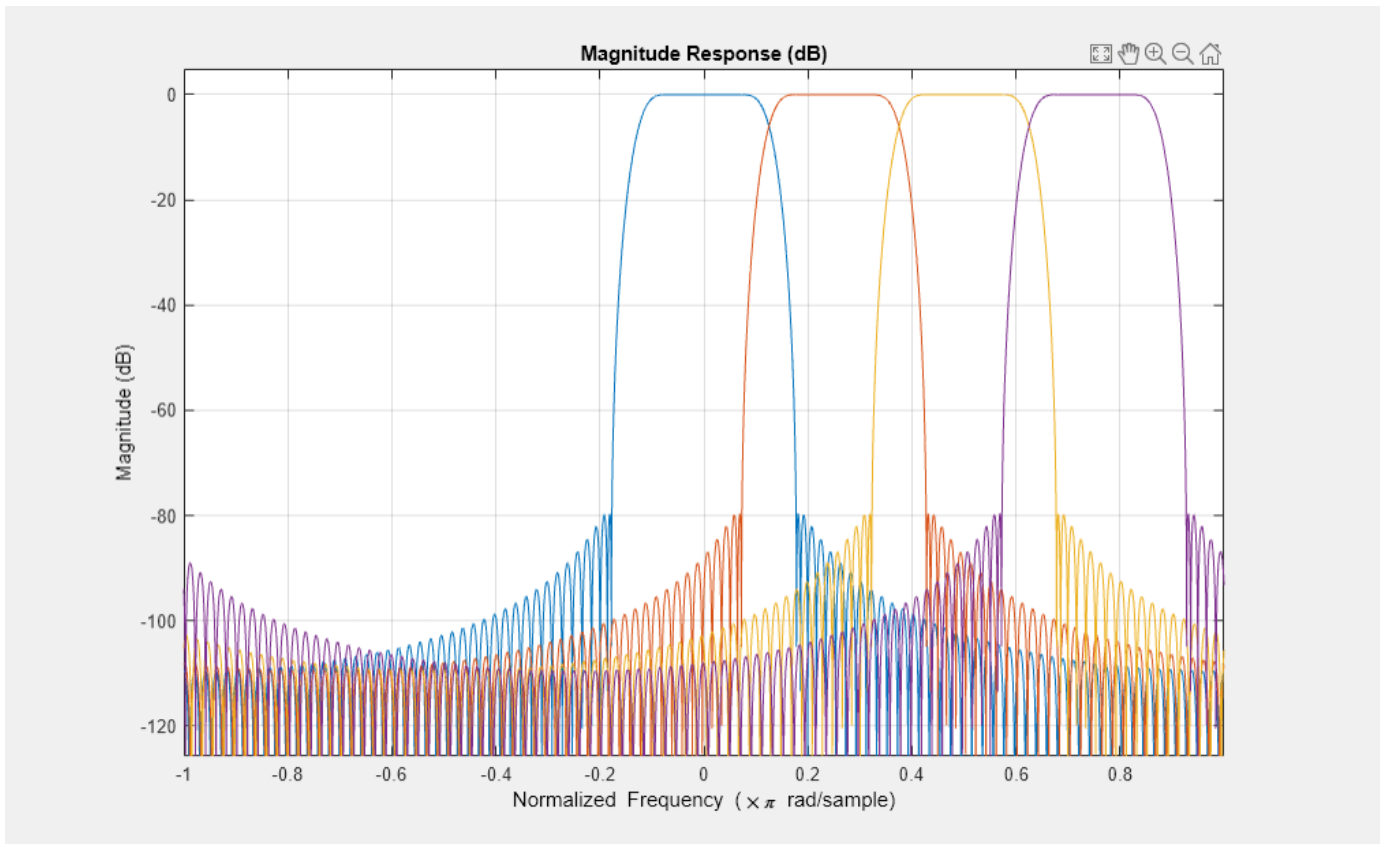
```
b = tf(channelizer);
isequal(b,B(1,:))
```

```
ans = logical
     1
```

The output of the `tf` function and the first row of the `B` matrix are equal.

Visualize the frequency response of the first 4 filters of the channelizer.

```
fvtool(B(1,:),1, B(2,:),1, B(3,:),1, B(4,:),1);
```



## Input Arguments

### **obj** — Input filter System object

`dsp.Channelizer`

Input filter, specified as `dsp.Channelizer` System object. The `getFilters` function returns a matrix of filter coefficients corresponding to each filter in the channelizer filter bank.

### **ind** — Filter indices

row vector

Filter indices, specified as a row vector in the range  $[1 \text{ obj}.\text{NumFrequencyBands}]$ . If not specified, `ind` is  $1:N$ , where  $N$  is the number of frequency bands specified through the `obj.NumFrequencyBands` property.

Example: `getFilters(channelizer,[1:4]);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

## Output Arguments

### **B** — Matrix of channelizer FIR filter coefficients

matrix

Matrix of channelizer finite impulse response (FIR) filter coefficients, returned as a matrix. Each row in the matrix corresponds to a filter in the filter bank. The first row corresponds to the prototype filter as returned by  $b = \text{tf}(\text{obj})$ . The remaining rows in  $B$  are given by:

$$b_k = b \times e^{jw_k n/N}$$

$N$  is the number of frequency bands, and  $k$  is the row index - 1.

## **Version History**

**Introduced in R2017b**

### **See Also**

#### **Functions**

`polyphase` | `tf` | `fvtool` | `freqz` | `bandedgeFrequencies` | `centerFrequencies` | `coeffs`

#### **Objects**

`dsp.Channelizer`

# generateScript

## Package:

Generate MATLAB script to create scope with current settings

## Syntax

```
generateScript(scope)
```

## Description

`generateScript(scope)` generates a MATLAB script that can recreate a scope object with the current settings in the scope.

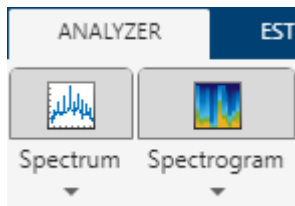
## Examples

### Generate Script from spectrumAnalyzer

- 1 Create a spectrumAnalyzer object.

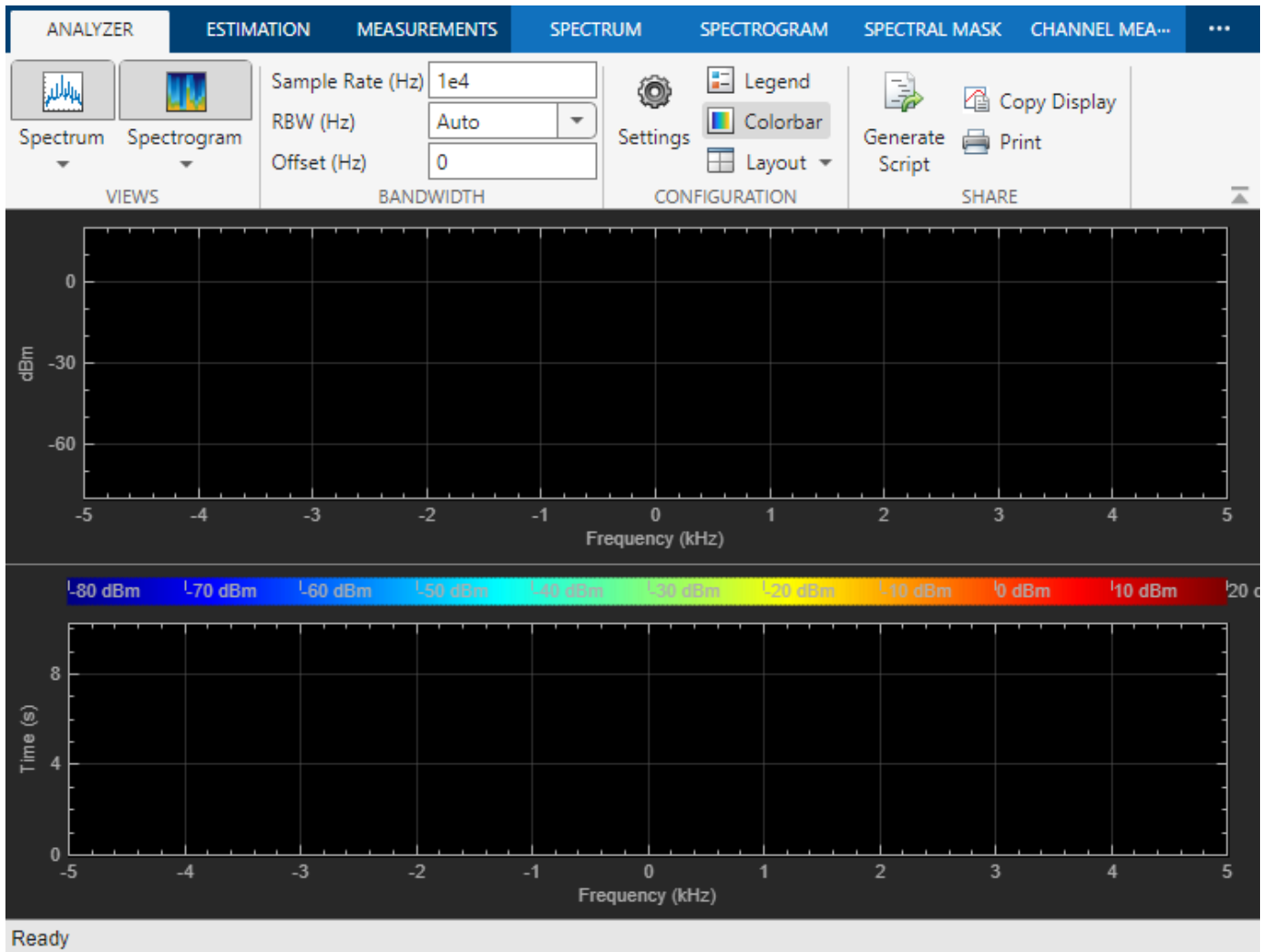
```
scope = spectrumAnalyzer();  
show(scope);
```

- 2 Set options in the Spectrum Analyzer. For this example, enable both **Spectrum** and **Spectrogram** to be shown in the **Analyzer** tab of the Spectrum Analyzer toolstrip.



In addition, enable **Max-Hold Trace** and **Min-Hold Trace** in the **Spectrum** tab.





- 3 Generate a script to recreate the `spectrumAnalyzer` with the modified settings. Either click **Generate Script** from the **Analyzer** tab, or enter this command in the MATLAB command prompt.

```
generateScript(scope);
```

A new editor window opens with code to regenerate the same scope.

```
% Generated by MATLAB(R) 9.12 (R2022a) and DSP System Toolbox 9.14 (R2022a).
% Generated on 15-Nov-2021 15:24:16 -0500.

% Construct a spectrumAnalyzer object
specScope = spectrumAnalyzer(ViewType="spectrum-and-spectrogram", ...
    PlotMaxHoldTrace=true, ...
    PlotMinHoldTrace=true, ...
    Position=[184 190 742 563]);
```

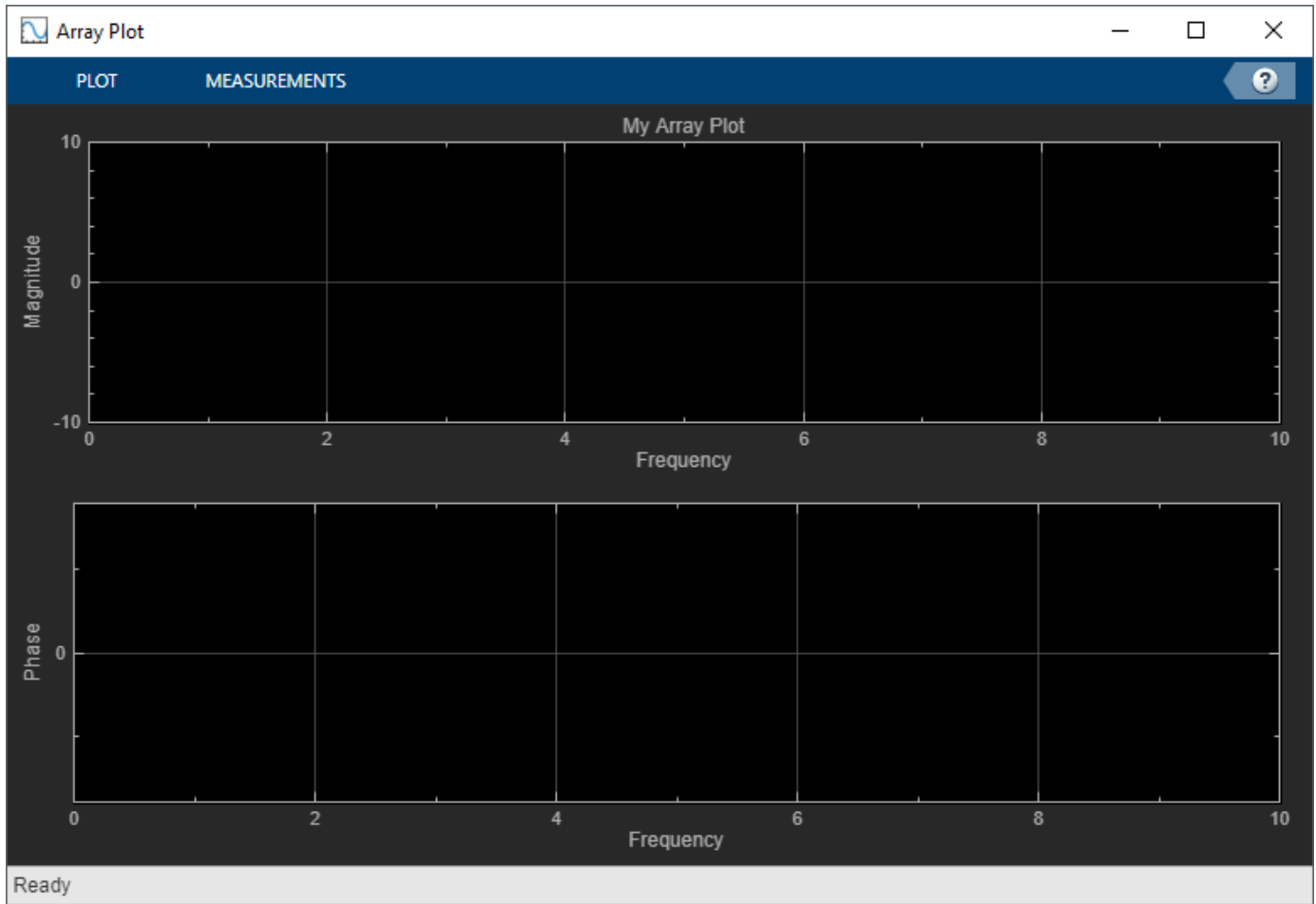
### Generate Script from `dsp.ArrayPlot`

- 1 Create a `dsp.ArrayPlot` object.

```
scope = dsp.ArrayPlot();
show(scope);
```



- Set options in the Array Plot. For this example, from the **Measurements** tab, turn on the **Data Cursors**. From the **Plot** tab, turn on **Legend** and **Magnitude and Phase**. Click **Settings** and give the plot an XLabel and Title.



- Generate a script to recreate the `dsp.ArrayPlot` with the same modified settings. Either select **Generate Script** from the **Plot** tab, or enter the following in the MATLAB command prompt:

```
generateScript(scope);
```

A new editor window opens with code to regenerate the same scope.

```
% Generated by MATLAB(R) 9.12 (R2022a) and DSP System Toolbox 9.14 (R2022a).
% Generated on 8-Dec-2021 15:37:21 -0500.

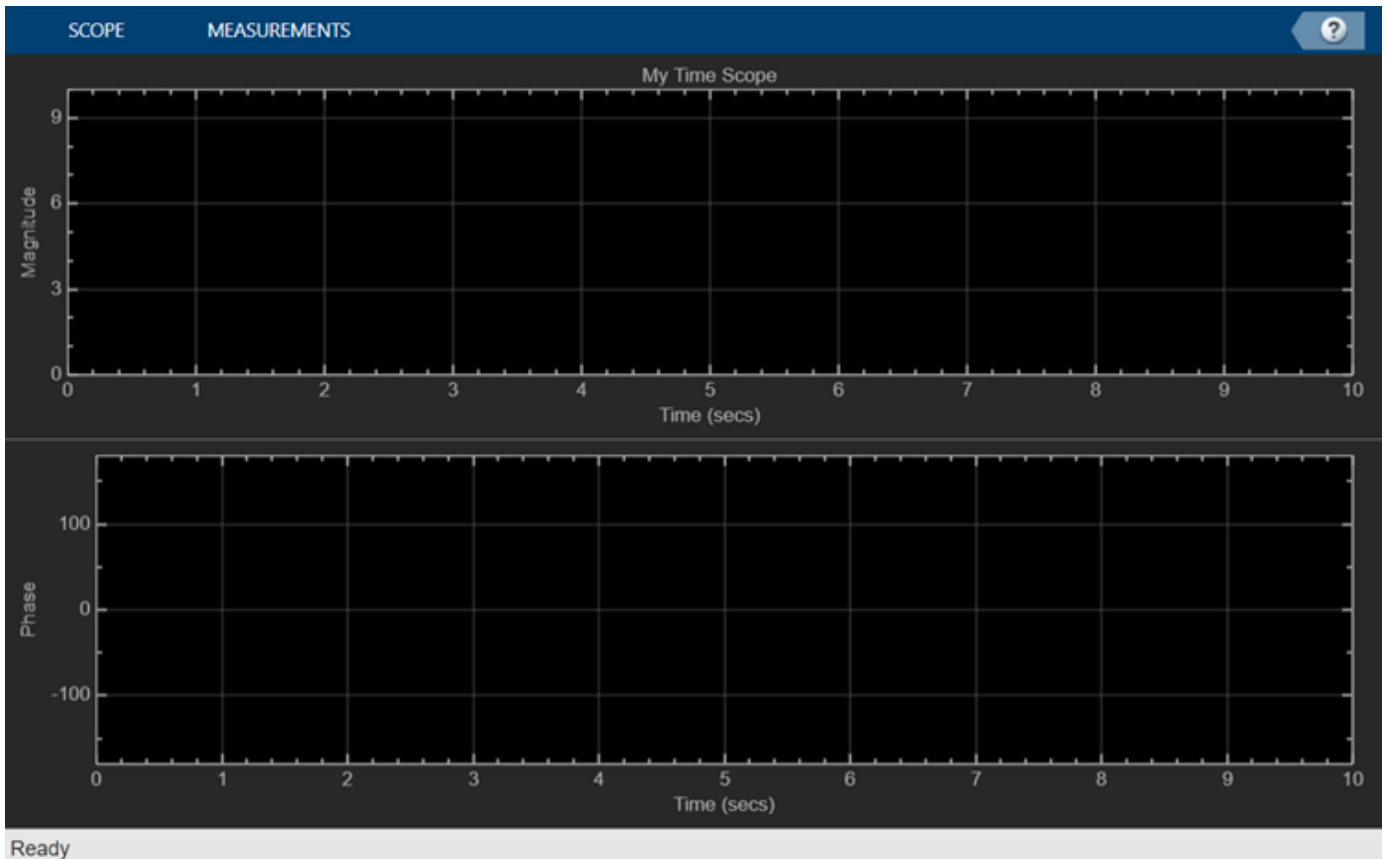
% Construct a dsp.ArrayPlot object
arrayPlot = dsp.ArrayPlot(PlotAsMagnitudePhase=true, ...
    Title="My Array Plot", ...
    XLabel="Frequency", ...
    ShowLegend=true);
% Cursor Measurements Configuration
arrayPlot.CursorMeasurements.Enabled = true;
```

## Generate Script from timescope

- Create a `timescope` object.

```
scope = timescope;
show(scope)
```

- 2 Set options in the Time Scope. For this example, on the **Scope** tab, click **Settings**. Under **Display and Labels**, select **Show Legend** and **Magnitude Phase Plot**. Set the **Title** as well.



- 3 Generate a script to recreate the `timescope` with the same modified settings. Either select **Generate Script** from the **Scope** tab, or enter the following in the MATLAB command prompt:

```
generateScript(scope);
```

A new editor window opens with code to regenerate the same scope.

```
% Generated by MATLAB(R) 9.14 (R2023a) and DSP System Toolbox 9.15 (R2023a).
% Generated on 1-Dec-2022 12:33:51 -0500.

% Construct a timescope object
timeScope = timescope(Position=[2672 404 800 500], ...
    PlotAsMagnitudePhase=true, ...
    Title="My Time Scope", ...
    ShowLegend=true);
```

## Input Arguments

### `scope` — Scope object

scope object

Scope object whose settings you want to recreate with a script, specified as one of the following:

- `spectrumAnalyzer` object

- dsp.ArrayPlot object
- timescope object
- dsp.DynamicFilterVisualizer object

## Version History

Introduced in R2019a

### R2022a: Support for dsp.SpectrumAnalyzer object will be removed

*Not recommended starting in R2022a*

The generateScript function will stop supporting the dsp.SpectrumAnalyzer object in a future release. Use the spectrumAnalyzer object instead.

### Update Code

No updates to your code are required except replacing instances of dsp.SpectrumAnalyzer with spectrumAnalyzer.

This table shows how the generateScript function typically uses the dsp.SpectrumAnalyzer System object and explains how to update existing code to use the spectrumAnalyzer object.

| Discouraged Usage   | Recommended Replacement  |
|---|--|
| <pre>scope = dsp.SpectrumAnalyzer; show(scope)</pre> <p>Make changes in the UI. Generate script.</p> <pre>generateScript(scope);</pre> <p>A new editor window opens with code to regenerate the same scope.</p> <pre>% Creation Code for 'dsp.SpectrumAnalyzer'. % Generated by Spectrum Analyzer on 16-Nov-2021 11:35:13 -0500. specScope = dsp.SpectrumAnalyzer('ViewType','Spectrum and spectrogram', ...     'PlotMaxHoldTrace',true, ...     'PlotMinHoldTrace',true);</pre> | <pre>scope = spectrumAnalyzer; show(scope)</pre> <p>Make changes in the UI. Generate script.</p> <pre>generateScript(scope);</pre> <p>A new editor window opens with code to regenerate the same scope.</p> <pre>% Generated by MATLAB(R) 9.12 (R2022a) and DSP System Toolbox 9.14 (R2022a) % Generated on 16-Nov-2021 11:36:19 -0500. % Construct a spectrumAnalyzer object specScope = spectrumAnalyzer(ViewType="spectrum-and-spectrogram", ...     PlotMaxHoldTrace=true, ...     PlotMinHoldTrace=true);</pre> |

### R2022b: New support for dsp.DynamicFilterVisualizer object

Starting in R2022b, the generateScript function supports the dsp.DynamicFilterVisualizer object.

## See Also

spectrumAnalyzer | dsp.ArrayPlot | timescope | dsp.DynamicFilterVisualizer

## Topics

“Configure Spectrum Analyzer”

## getSpectralMaskStatus

Get test results of current spectral mask

### Syntax

```
results = getSpectralMaskStatus(scope)
```

### Description

`results = getSpectralMaskStatus(scope)` returns the current status of the spectral mask on the spectrum analyzer, `scope`, in a structure, `results`.

### Examples

#### Get Spectral Mask Status

Add a spectral mask to an existing `spectrumAnalyzer` object. Use the `getSpectralMaskStatus` function to get the spectral mask status.

```
sine = dsp.SineWave(Frequency=[98 100],SampleRate=1000);
sine.SamplesPerFrame = 1024;
scope = spectrumAnalyzer(SampleRate=sine.SampleRate,...
    PlotAsTwoSidedSpectrum=false,ShowLegend=true,YLimits=[-60 40]);
hide(scope);

scope.SpectralMask.EnabledMasks = "upper-and-lower";
upperMask = [0 -10; 90 -10; 90 30; 110 30; 110 -10; 500 -10];
set(scope.SpectralMask,UpperMask=upperMask,LowerMask=-55);

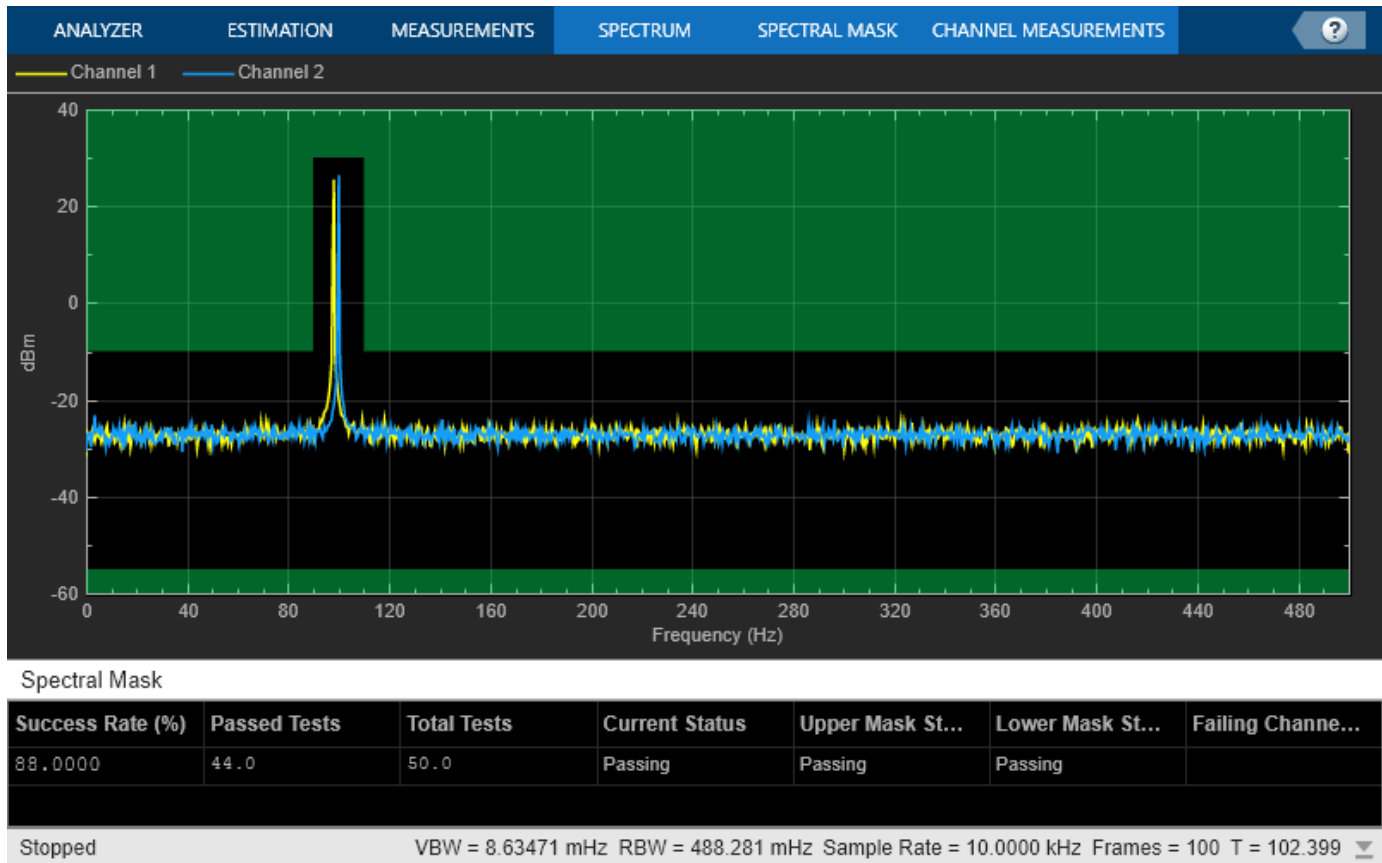
for i=1:100
    scope(sine() + 0.05*randn(1024,2));
end

res = getSpectralMaskStatus(scope)

res = struct with fields:
    IsCurrentlyPassing: 1
    NumPassedTests: 44
    NumTotalTests: 50
    SuccessRate: 88
    FailingMasks: ''
    FailingChannels: [1x0 double]
    SimulationTime: 102.3990
```

You can also view the status in the **Spectral Mask** tab on the Spectrum Analyzer toolstrip.

```
show(scope);
release(scope);
```



## Input Arguments

### scope — Spectrum Analyzer

object name | block configuration

Spectrum Analyzer with spectral masks, specified as one of the following:

- spectrumAnalyzer object
- SpectrumAnalyzerConfiguration object for a Spectrum Analyzer block

## Output Arguments

### results — Current status of spectral mask

structure

Current status of the spectral mask, returned as a structure with these fields.

| Field              | Description   |
|--------------------|---|
| IsCurrentlyPassing | Pass status of one or more masks.<br>1 — All masks have passed<br>0 — One or more masks have failed |
| NumPassedTests     | Number of mask tests that have passed   |
| NumTotalTests      | Total number of mask tests  |
| SuccessRate        | Percentage of tests that have passed  |
| FailingChannels    | Array of channel numbers that have currently failed the mask test                                   |
| FailingMasks       | Masks that have currently failed: "none", "upper", "lower", or "upper-and-lower"                    |
| SimulationTime     | Simulation time   |

## Version History

Introduced in R2017a

### **R2022a: Support for `dsp.SpectrumAnalyzer` object will be removed**

*Not recommended starting in R2022a*

The `getSpectralMaskStatus` function will stop supporting the `dsp.SpectrumAnalyzer` object in a future release. Use the `spectrumAnalyzer` object instead.

### **Update Code**

No updates to your code are required except replacing instances of `dsp.SpectrumAnalyzer` with `spectrumAnalyzer`.

This table shows how the `getSpectralMaskStatus` function typically uses the `dsp.SpectrumAnalyzer` System object and explains how to update existing code to use the `spectrumAnalyzer` object.

| Discouraged Usage  | Recommended Replacement  |
|--|--|
| <p>Add a spectral mask to an existing <code>dsp.SpectrumAnalyzer</code> System object scope and get its status using the <code>getSpectralMaskStatus</code> function.</p> <pre>sine = dsp.SineWave('Frequency',[98 100],'SampleRate',1000); sine.SamplesPerFrame = 1024; scope = dsp.SpectrumAnalyzer('SampleRate',sine.SampleRate,     'PlotAsTwoSidedSpectrum',false,'ShowLegend',true,     'YLimits',[-60 40]); hide(scope);  scope.SpectralMask.EnabledMasks = 'Upper and lower'; upperMask = [0 -10; 90 -10; 90 30; 110 30; 110 -10; 50 -10]; set(scope.SpectralMask,'UpperMask',upperMask);  for i=1:100     scope(sine() + 0.05*randn(1024,2)); end  res = getSpectralMaskStatus(scope)  res =      struct with fields:          IsCurrentlyPassing: 0         NumPassedTests: 1         NumTotalTests: 33         SuccessRate: 3.0303         FailingMasks: 'Lower'         FailingChannels: [1 2]         SimulationTime: 101.3760</pre> <p>You can see the same information in the Spectral Mask panel of the Spectrum Analyzer.</p> <pre>show(scope); release(scope);</pre> | <p>Add a spectral mask to an existing <code>spectrumAnalyzer</code> object scope and get its status using the <code>getSpectralMaskStatus</code> function.</p> <p>The <code>spectrumAnalyzer</code> object uses the filter bank method as the default spectral estimation method. Compared to the Welch method, the filter bank method has a lower noise floor, better frequency resolution, and also requires fewer samples per update.</p> <p>To retain the same default behavior as the <code>dsp.SpectrumAnalyzer</code> System object, change the <code>Method</code> property to "welch" and the <code>AveragingMethod</code> property to "exponential". If needed, adjust the value of the forgetting factor.</p> <pre>sine = dsp.SineWave(Frequency=[98 100],SampleRate=1000); sine.SamplesPerFrame = 1024; scope = spectrumAnalyzer(SampleRate=sine.SampleRate,     PlotAsTwoSidedSpectrum=false,ShowLegend=true,     Method="welch",AveragingMethod="exponential",     ForgettingFactor=0.4,YLimits=[-60 40]); hide(scope); scope.SpectralMask.EnabledMasks = "upper-and-lower"; upperMask = [0 -10; 90 -10; 90 30; 110 30; 110 -10; 50 -10]; set(scope.SpectralMask,UpperMask=upperMask,LowerMask=lowerMask);  for i=1:100     scope(sine() + 0.05*randn(1024,2)); end  res = getSpectralMaskStatus(scope)  res = struct with fields:     IsCurrentlyPassing: 0     NumPassedTests: 0     NumTotalTests: 0     SuccessRate: 0     FailingMasks: ''     FailingChannels: [1x0 double]     SimulationTime: 2.0470</pre> <p>You can see the same information in the <b>Spectral Mask</b> tab on the Spectrum Analyzer toolstrip.</p> <pre>show(scope); release(scope);</pre> |

## See Also

`spectrumAnalyzer` | `SpectrumAnalyzerConfiguration` | `getMeasurementsData`

## getSpectrumData

Save spectrum data shown in spectrum analyzer

### Syntax

```
spectrumTable = getSpectrumData(scope)
```

### Description

`spectrumTable = getSpectrumData(scope)` returns the spectrum and spectrogram displayed on the spectrum analyzer along with additional statistics about the spectrum.

### Examples

#### Save Spectrum Data

Create a Spectrum Analyzer object and generate data.

```
scope = spectrumAnalyzer;  
scope(randn(6000,3))
```





Save data from the last spectrum shown in the Spectrum Analyzer to a table.

```
specTable = getSpectrumData(scope)
```

```
specTable =
```

```
1x3 table
```

| SimulationTime | Spectrum        | FrequencyVector |
|----------------|-----------------|-----------------|
| {[0.5999]}     | {1024x3 double} | {1024x1 double} |

## Input Arguments

### scope — Spectrum analyzer

object name | block configuration

Spectrum Analyzer you want to query, specified as one of the following:

- spectrumAnalyzer object
- SpectrumAnalyzerConfiguration object for a Spectrum Analyzer block

## Output Arguments

### spectrumTable — Data about current spectrum estimation

table

A spectrum table is returned containing the following fields:

| Field           | Description                                |
|-----------------|--|
| SimulationTime  | Simulation time                            |
| Spectrum        | Power, power density, or RMS spectrum data |
| Spectrogram     | Spectrogram data                           |
| MinHoldTrace    | Minimum hold trace data                    |
| MaxHoldTrace    | Maximum hold trace data                    |
| FrequencyVector | Frequency vector                           |

## Version History

### Introduced in R2017b

### R2022a: Support for dsp.SpectrumAnalyzer object will be removed

*Not recommended starting in R2022a*

The getSpectrumData function will stop supporting the dsp.SpectrumAnalyzer object in a future release. Use the spectrumAnalyzer object instead.

### Update Code

No updates to your code are required except replacing instances of `dsp.SpectrumAnalyzer` with `spectrumAnalyzer`.

This table shows how the `getSpectrumData` function typically uses the `dsp.SpectrumAnalyzer` System object and explains how to update existing code to use the `spectrumAnalyzer` object.

| Discouraged Usage  | Recommended Replacement  |
|--|--|
| <pre>scope = dsp.SpectrumAnalyzer;<br/>scope(randn(100000,1))<br/>specTable = getSpectrumData(scope)</pre> | <pre>scope = spectrumAnalyzer;<br/>scope(randn(100000,1))<br/>specTable = getSpectrumData(scope)</pre> |

## See Also

### Functions

[isNewDataReady](#) | [getSpectralMaskStatus](#) | [getMeasurementsData](#)

### Objects

[spectrumAnalyzer](#) | [SpectrumAnalyzerConfiguration](#)

### Blocks

[Spectrum Analyzer](#)

# getMeasurementsData

Get the current measurement data displayed on the spectrum analyzer

## Syntax

```
data = getMeasurementsData(scope)
data = getMeasurementsData(scope,"all")
```

## Description

`data = getMeasurementsData(scope)` returns a data table about the current spectrum analyzer measurements in use.

`data = getMeasurementsData(scope,"all")` returns a data table about all spectrum analyzer measurements for the current time step.

## Examples

### Obtain Measurements Data Programmatically for spectrumAnalyzer object

Compute and display the power spectrum of a noisy sinusoidal input signal using the `spectrumAnalyzer` MATLAB® object. Measure the peaks, cursor placements, adjacent channel power ratio, and distortion values in the spectrum by enabling these properties:

- `PeakFinder`
- `CursorMeasurements`
- `ChannelMeasurements`
- `DistortionMeasurements`

### Initialization

The input sine wave has two frequencies: 1000 Hz and 5000 Hz. Create two `dsp.SineWave` System objects to generate these two frequencies. Create a `spectrumAnalyzer` object to compute and display the power spectrum.

```
Fs = 44100;
Sineobject1 = dsp.SineWave(SamplesPerFrame=1024,PhaseOffset=10,...
    SampleRate=Fs,Frequency=1000);
Sineobject2 = dsp.SineWave(SamplesPerFrame=1024,...
    SampleRate=Fs,Frequency=5000);
SA = spectrumAnalyzer(SampleRate=Fs,SpectrumType="power",...
    PlotAsTwoSidedSpectrum=false,ChannelNames={'Power spectrum of the input'},...
    YLimits=[-120 40],ShowLegend=true);
```

### Enable Measurements Data

To obtain the measurements, set the `Enabled` property to `true`.

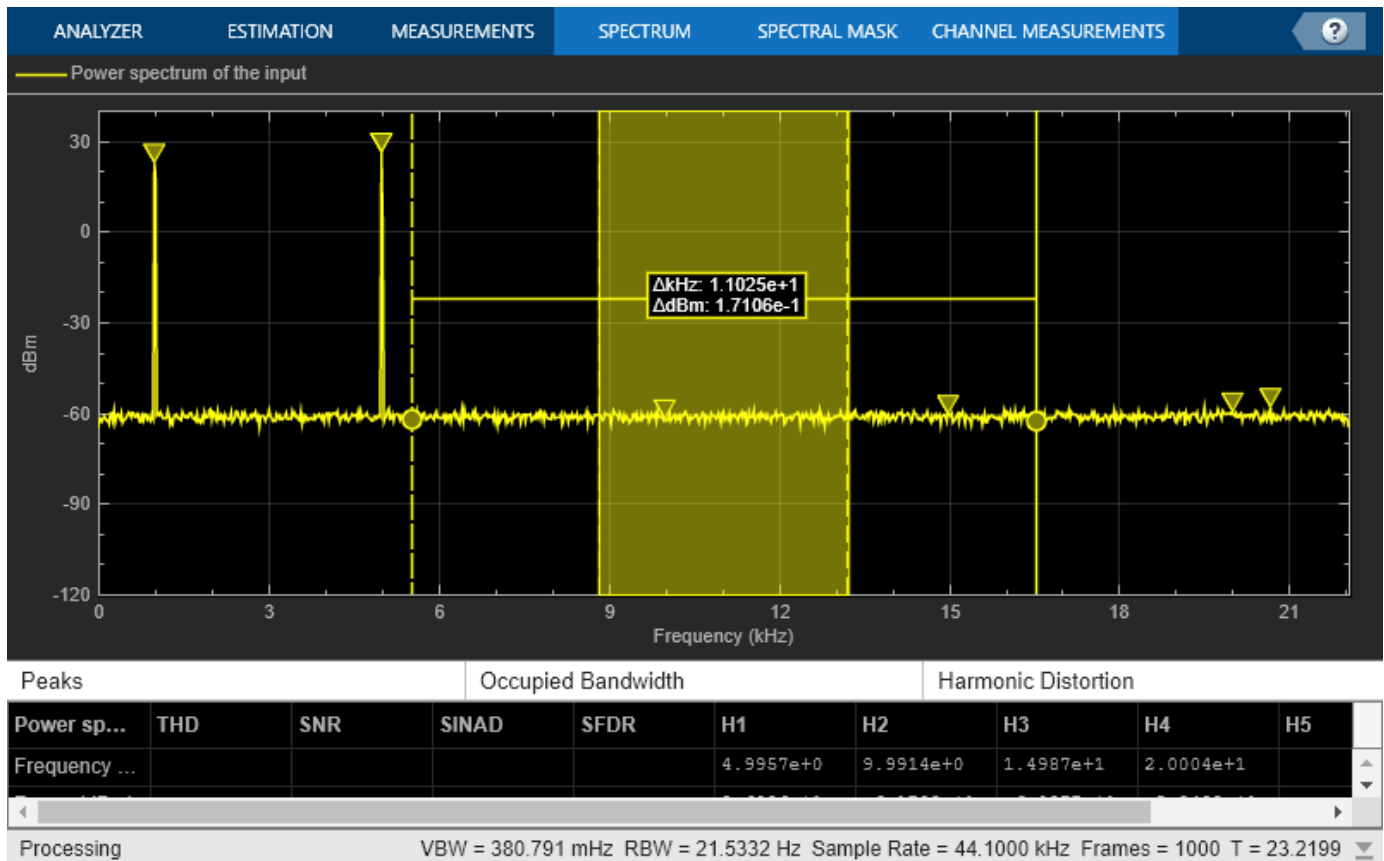
```
SA.CursorMeasurements.Enabled = true;
SA.ChannelMeasurements.Enabled = true;
```

```
SA.PeakFinder.Enabled = true;
SA.DistortionMeasurements.Enabled = true;
```

### Use getMeasurementsData

Stream in the noisy sine wave input signal and estimate the power spectrum of the signal using the spectrumAnalyzer object. Measure the characteristics of the spectrum. Use the getMeasurementsData function to obtain these measurements programmatically. The isNewDataReady function returns true when there is new spectrum data. Store the measured data in the variable data.

```
data = [];
for Iter = 1:1000
    Sinewave1 = Sineobject1();
    Sinewave2 = Sineobject2();
    Input = Sinewave1 + Sinewave2;
    NoisyInput = Input + 0.001*randn(1024,1);
    SA(NoisyInput);
    if SA.isNewDataReady
        data = [data;getMeasurementsData(SA)];
    end
end
```



The panes at the bottom of the scope window display the measurements that you have enabled. The values in these panes match the values in the last time step of the data variable. You can access the individual fields of data to obtain the various measurements programmatically.

## Compare Peak Values

Use the `PeakFinder` property to obtain peak values. Verify that the peak values in the last time step of `data` match the values in the spectrum analyzer plot.

```
peakvalues = data.PeakFinder(end).Value
```

```
peakvalues = 3×1
```

```
    26.3957  
    22.7830  
   -57.9977
```

```
frequencieskHz = data.PeakFinder(end).Frequency/1000
```

```
frequencieskHz = 3×1
```

```
    4.9957  
    0.9905  
   20.6719
```

## Obtain Measurements Data Programmatically for Spectrum Analyzer Block

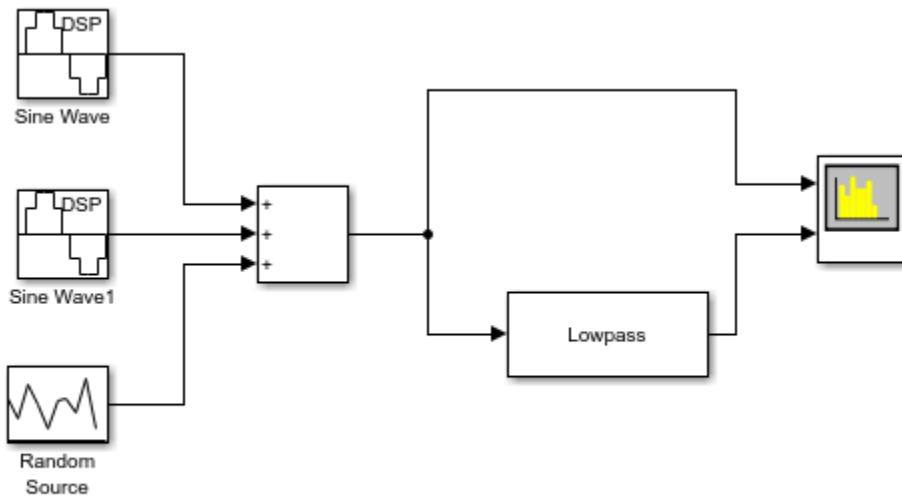
Compute and display the power spectrum of a noisy sinusoidal input signal using the Spectrum Analyzer block. Measure the cursor placements, adjacent channel power ratio, distortion, and peak values in the spectrum by enabling these block configuration properties:

- `CursorMeasurements`
- `ChannelMeasurements`
- `DistortionMeasurements`
- `PeakFinder`

## Open and Inspect the Model

Filter a streaming noisy sinusoidal input signal using a Lowpass Filter block. The input signal consists of two sinusoidal tones: 1 kHz and 15 kHz. The noise is white Gaussian noise with a mean of 0 and a variance of 0.05. The sampling frequency is 44.1 kHz. Open the model and inspect the parameter values in the blocks.

```
model = 'spectrumanalyzer_measurements.slx';  
open_system(model)
```



Copyright 2018-2022 The MathWorks, Inc.

Access the configuration properties of the Spectrum Analyzer block using the `get_param` function.

```
sablock = 'spectrumanalyzer_measurements/Spectrum Analyzer';  
cfg = get_param(sablock, 'ScopeConfiguration');
```

### Enable Measurements Data

To obtain the measurements, set the Enabled property to `true`.

```
cfg.CursorMeasurements.Enabled = true;  
cfg.ChannelMeasurements.Enabled = true;  
cfg.DistortionMeasurements.Enabled = true;  
cfg.PeakFinder.Enabled = true;
```

### Simulate the Model

Run the model. The Spectrum Analyzer block compares the original spectrum with the filtered spectrum.

```
sim(model)
```



The panes at the bottom of the spectrum analyzer window display the measurements that you have enabled.

### Use getMeasurementsData function

Use the `getMeasurementsData` function to obtain the measurements programmatically.

```
data = getMeasurementsData(cfg)
```

```
data =
```

```
1x5 table
```

```
SimulationTime
```

```
9.9962
```

```
PeakFinder
```

```
1x1 struct
```

```
CursorMeasurements
```

```
1x1 struct
```

```
ChannelMeasurements
```

```
1x1 struct
```

```
DistortionMeasurements
```

```
1x1 struct
```

The values shown in the measurement panels match the values shown in `data`. You can access the individual fields of `data` to obtain the various measurements programmatically.

### Compare Peak Values

As an example, compare the peak values. Verify that the peak values obtained by `data.PeakFinder` match with the values in the spectrum analyzer window.

```
peakvalues = data.PeakFinder.Value
frequencieskHz = data.PeakFinder.Frequency/1000
```

```
peakvalues =
```

```
    26.8653
    26.2638
    -3.4329
```

```
frequencieskHz =
```

```
    15.0015
     1.0049
     3.7755
```

## Input Arguments

### scope — Spectrum Analyzer

object name | block configuration

Spectrum Analyzer you want to query, specified as one of the following:

- spectrumAnalyzer object
- SpectrumAnalyzerConfiguration object for a Spectrum Analyzer block

## Output Arguments

### data — Measurements data

table

Measurements data, returned as a table. When you specify "all", the function returns a measurements table containing the following fields.

| Field  | Description                  |
|--|------------------------------|
| SimulationTime   | Simulation time              |
| PeakFinder   | Peak finder data             |
| CursorMeasurements   | Cursor measurements data     |
| ChannelMeasurements  | Channel measurements data    |
| DistortionMeasurements   | Distortion measurements data |
| CCDFMeasurements (applies only for SpectrumAnalyzerConfiguration object) | CCDF measurements data       |

When you do not specify "all", the data table contains only the Spectrum Analyzer measurements currently in use.

## Version History

Introduced in R2018b



**R2022a: Support for dsp.SpectrumAnalyzer object will be removed***Not recommended starting in R2022a*

The `getMeasurementsData` function will stop supporting the `dsp.SpectrumAnalyzer` object in a future release. Use the `spectrumAnalyzer` object instead.

**Update Code**

No updates to your code are required except replacing instances of `dsp.SpectrumAnalyzer` with `spectrumAnalyzer`.

This table shows how the `getMeasurementsData` function typically uses the `dsp.SpectrumAnalyzer` System object and explains how to update existing code to use the `spectrumAnalyzer` object.

| Discouraged Usage   | Recommended Replacement   |
|---|---|
| <pre>scope = dsp.SpectrumAnalyzer; scope.CursorMeasurements.Enable = true; scope.ChannelMeasurements.Enable = true; scope(randn(100000,1)) getMeasurementsData(scope)</pre> | <pre>scope = spectrumAnalyzer; scope.CursorMeasurements.Enabled = true; scope.ChannelMeasurements.Enabled = true; scope(randn(100000,1)) getMeasurementsData(scope)</pre> |

**See Also****Functions**

`isNewDataReady` | `getSpectrumData` | `getSpectralMaskStatus`

**Objects**

`spectrumAnalyzer` | `SpectrumAnalyzerConfiguration`

**Blocks**

Spectrum Analyzer

## getOctaveBandwidth

**Package:** dsp

Bandwidth in number of octaves

### Syntax

```
N = getOctaveBandwidth(npFilter)
```

### Description

`N = getOctaveBandwidth(npFilter)` returns the bandwidth of the notch peak filter, measured in number of octaves.

### Examples

#### Get Octave Bandwidth of Notch Peak Filter

Create a `dsp.NotchPeakFilter` object in the default configuration.

```
np = dsp.NotchPeakFilter
```

```
np =  
    dsp.NotchPeakFilter with properties:
```

```
    Specification: 'Bandwidth and center frequency'  
    Bandwidth: 2205  
    CenterFrequency: 11025  
    SampleRate: 44100
```

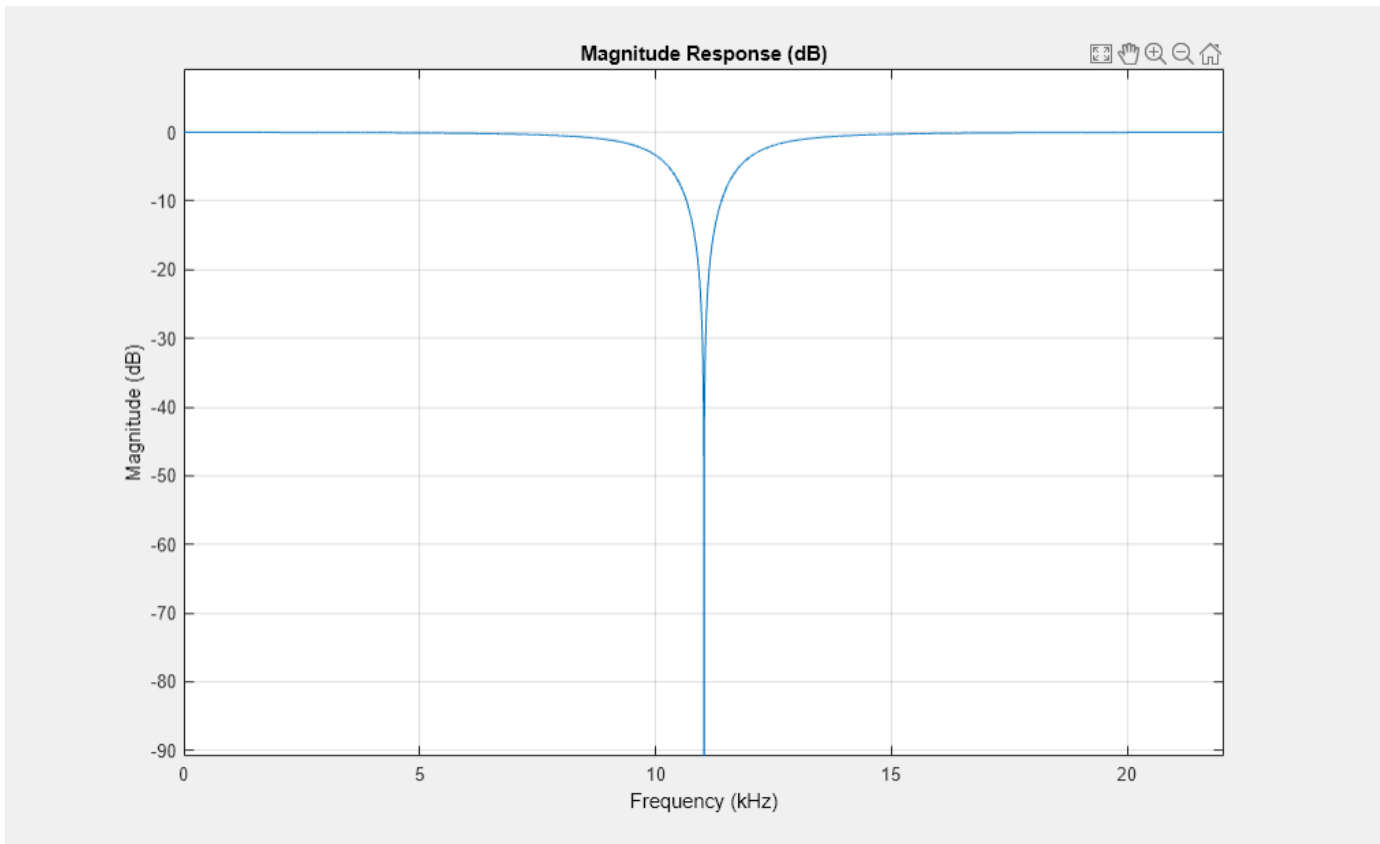
Determine the octave bandwidth of the filter using the `getOctaveBandwidth` function.

```
getOctaveBandwidth(np)
```

```
ans = 0.2881
```

Visualize the filter response using `fvtool`.

```
fvtool(np)
```



## Input Arguments

**npFilter** — Notch peak filter  
`dsp.NotchPeakFilter`

Notch peak filter whose bandwidth is measured in octaves, specified as a `dsp.NotchPeakFilter` object.

## Output Arguments

**N** — Number of octaves  
scalar

Bandwidth of the filter measured in number of octaves, returned as a scalar.

Data Types: `double`

## Version History

Introduced in R2014a

## **See Also**

### **Functions**

getBandwidth | getCenterFrequency | getQualityFactor | tf

### **Objects**

dsp.NotchPeakFilter

# getQualityFactor

**Package:** dsp

Get quality factor

## Syntax

```
Q = getQualityFactor(npFilter)
```

## Description

`Q = getQualityFactor(npFilter)` returns the quality factor (Q factor) of the notch peak filter. The Q factor is defined as the center frequency divided by the bandwidth.

## Examples

### Compute Quality Factor of Notch Peak Filter

Create a `dsp.NotchPeakFilter` object in the default configuration, where the `Specification` property is set to 'Bandwidth and center frequency'.

```
np = dsp.NotchPeakFilter
```

```
np =  
    dsp.NotchPeakFilter with properties:
```

```
    Specification: 'Bandwidth and center frequency'  
    Bandwidth: 2205  
    CenterFrequency: 11025  
    SampleRate: 44100
```

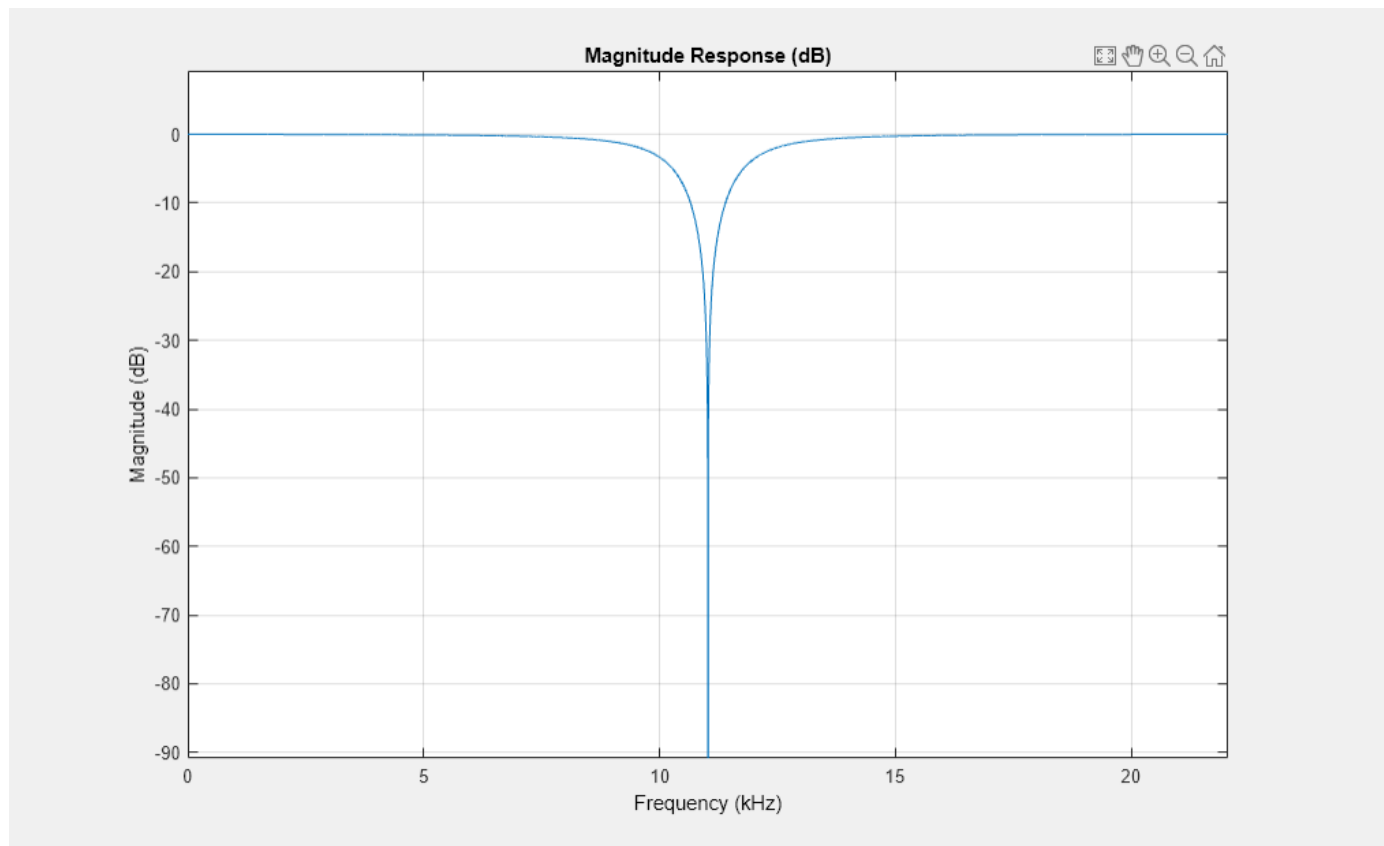
Determine the quality factor of the filter using the `getQualityFactor` function. The quality factor is given by the ratio of the center frequency to the bandwidth.

```
getQualityFactor(np)
```

```
ans = 5
```

Visualize the filter response using `fvtool`.

```
fvtool(np)
```



## Input Arguments

**npFilter** — Notch peak filter  
`dsp.NotchPeakFilter`

Notch peak filter whose quality factor is computed, specified as a `dsp.NotchPeakFilter` object.

## Output Arguments

**Q** — Quality factor  
scalar

Quality factor of the filter, returned as a scalar. The Q factor is defined as the center frequency divided by the bandwidth.

Data Types: `double`

## Version History

Introduced in R2014a

## See Also

### Functions

[getBandwidth](#) | [getCenterFrequency](#) | [getOctaveBandwidth](#) | [tf](#)

### Objects

[dsp.NotchPeakFilter](#)

## tf

**Package:** dsp

Transfer function

### Syntax

```
[B,A] = tf(npFilter)
[B,A,B2,A2] = tf(npFilter)
```

### Description

`[B,A] = tf(npFilter)` returns the vector of numerator coefficients, B, and the vector of denominators, A, for the equivalent transfer function corresponding to the notch filter.

`[B,A,B2,A2] = tf(npFilter)` also returns the vector of numerator coefficients, B2, and the vector of denominator coefficients, A2, for the equivalent transfer function corresponding to the peak filter.

### Examples

#### Determine Transfer Function of Notch Peak Filter

Create a `dsp.NotchPeakFilter` System object™. Obtain the coefficients of the transfer function corresponding to the notch and peak filters.

```
notchpeak = dsp.NotchPeakFilter;
[Bnotch,Anotch,Bpeak,Apeak] = tf(notchpeak)
```

```
Bnotch = 1×3
```

```
    0.8633    -0.0000    0.8633
```

```
Anotch = 1×3
```

```
    1.0000    -0.0000    0.7265
```

```
Bpeak = 1×3
```

```
    0.1367         0    -0.1367
```

```
Apeak = 1×3
```

```
    1.0000    -0.0000    0.7265
```



**Bnotch** and **Anotch** are the vectors of numerator and denominator coefficients for the equivalent transfer function corresponding to the notch filter. **Bpeak** and **Apeak** are the vectors of numerator and denominator coefficients for the equivalent transfer function corresponding to the peak filter.

## Input Arguments

### **npFilter** — Notch peak filter object

`dsp.NotchPeakFilter`

Notch peak filter object, specified as a `dsp.NotchPeakFilter` object.

## Output Arguments

### **B** — Numerator coefficients of notch filter

vector

Numerator coefficients for the equivalent transfer function corresponding to the notch filter, specified as a vector.

Data Types: `double`

### **A** — Denominator coefficients of notch filter

vector

Denominator coefficients for the equivalent transfer function corresponding to the notch filter, specified as a vector.

Data Types: `double`

### **B2** — Numerator coefficients of peak filter

vector

Numerator coefficients for the equivalent transfer function corresponding to the peak filter, specified as a vector.

Data Types: `double`

### **A2** — Denominator coefficients of peak filter

vector

Denominator coefficients for the equivalent transfer function corresponding to the peak filter, specified as a vector.

Data Types: `double`

## Version History

Introduced in R2014a

## See Also

### Functions

`getCenterFrequency` | `getOctaveBandwidth` | `getQualityFactor` | `getBandwidth`

**Objects**

dsp.NotchPeakFilter

# grpdelay

**Package:** dsp

Group delay response of discrete-time filter System object

## Syntax

```
[gd,w] = grpdelay(sysobj)
[gd,w] = grpdelay(sysobj,n)
[gd,w] = grpdelay(sysobj,'Arithmetic',arithType)
grpdelay(sysobj)
```

## Description

`[gd,w] = grpdelay(sysobj)` returns the group delay `gd` of the filter System object, `sysobj`, based on the current filter coefficients. The vector `w` contains the frequencies (in radians) at which the group delay is evaluated. The group delay is defined as:

$$-\frac{d}{dw}(\text{angle}(w))$$

The group delay is evaluated at 8192 points equally spaced around the upper half of the unit circle.

`[gd,w] = grpdelay(sysobj,n)` returns the group delay of the filter System object and the corresponding frequencies at `n` points equally spaced around the upper half of the unit circle.

`[gd,w] = grpdelay(sysobj,'Arithmetic',arithType)` computes the group delay of the filter System object, based on the arithmetic specified in `arithType`, using either of the previous syntaxes.

`grpdelay(sysobj)` plots the group delay of the filter System object in the `fvtool`.

For more input options, see `grpdelay` in Signal Processing Toolbox.

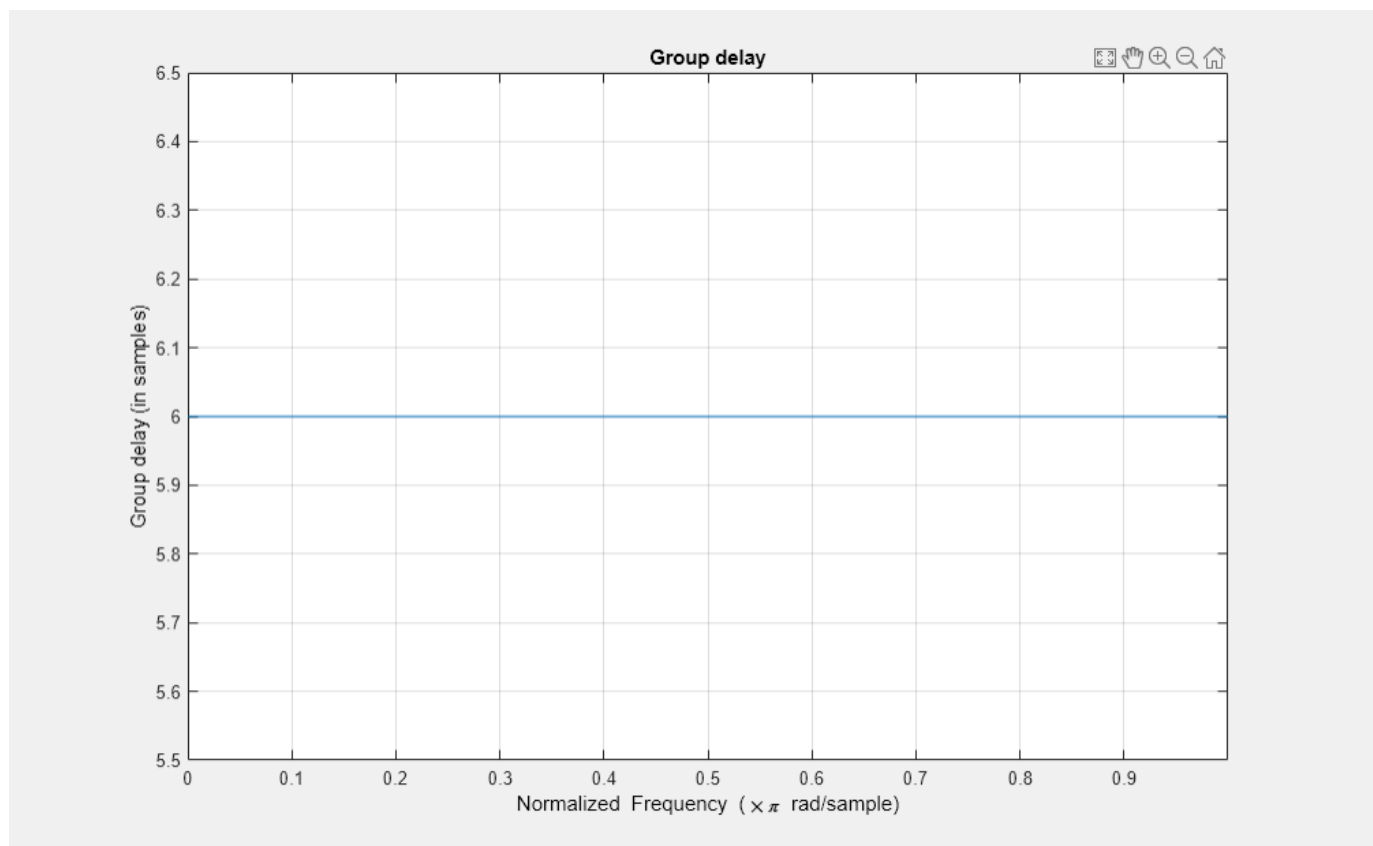
## Examples

### Group Delay of Discrete-Time Multirate Filter

```
CICComp = dsp.CICCompensationDecimator;
```

`grpdelay` computes the group delay of the filter and displays it using `fvtool`.

```
grpdelay(CICComp);
```



## Input Arguments

### **sysobj** — Input filter

filter System object

Input filter, specified as as one of the following filter System objects:

- `dsp.AllpassFilter`
- `dsp.AllpoleFilter`
- `dsp.BiquadFilter`
- `dsp.CICCompensationDecimator`
- `dsp.CICCompensationInterpolator`
- `dsp.CICDecimator`
- `dsp.CICInterpolator`
- `dsp.CoupledAllpassFilter`
- `dsp.Differentiator`
- `dsp.FarrowRateConverter`
- `dsp.FilterCascade`
- `dsp.FIRDecimator`

- `dsp.FIRFilter`
- `dsp.FIRHalfbandDecimator`
- `dsp.FIRHalfbandInterpolator`
- `dsp.FIRInterpolator`
- `dsp.FIRRateConverter`
- `dsp.FourthOrderSectionFilter`
- `dsp.HighpassFilter`
- `dsp.IIRFilter`
- `dsp.IIRHalfbandDecimator`
- `dsp.IIRHalfbandInterpolator`
- `dsp.LowpassFilter`
- `dsp.NotchPeakFilter`
- `dsp.SOSFilter`
- `dsp.VariableBandwidthFIRFilter`
- `dsp.VariableBandwidthIIRFilter`

#### **n — Number of samples**

8192 (default) | positive integer

Number of samples, specified as a positive integer. For an FIR filter where *n* is a power of two, the computation is done faster using FFTs.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

#### **arithType — Arithmetic type**

'double' (default) | 'single' | 'Fixed'

Arithmetic used in the filter analysis, specified as 'double', 'single', or 'Fixed'. When the arithmetic input is not specified and the filter System object is unlocked, the analysis tool assumes a double-precision filter. When the arithmetic input is not specified and the System object is locked, the function performs the analysis based on the data type of the locked input.

The 'Fixed' value applies to filter System objects with fixed-point properties only.

When the 'Arithmetic' input argument is specified as 'Fixed' and the filter object has the data type of the coefficients set to 'Same word length as input', the arithmetic analysis depends on whether the System object is unlocked or locked.

- **unlocked** -- The analysis object function cannot determine the coefficients data type. The function assumes that the coefficients data type is signed, has a 16-bit word length, and is auto scaled. The function performs fixed-point analysis based on this assumption.
- **locked** -- When the input data type is 'double' or 'single', the analysis object function cannot determine the coefficients data type. The function assumes that the data type of the coefficients is signed, has a 16-bit word length, and is auto scaled. The function performs fixed-point analysis based on this assumption.

To check if the System object is locked or unlocked, use the `isLocked` function.

When the arithmetic input is specified as 'Fixed' and the filter object has the data type of the coefficients set to a custom numeric type, the object function performs fixed-point analysis based on the custom numeric data type.

## Output Arguments

### **gd** — Group delay

vector

Group delay vector of length  $n$ . If  $n$  is not specified, the function uses a default value of 8192.

Data Types: double

### **w** — Frequencies

vector

Frequency vector of length  $n$ , in radians/sample.  $w$  consists of  $n$  points equally spaced around the upper half of the unit circle (from 0 to  $\pi$  radians/sample). If  $n$  is not specified, the function uses a default value of 8192.

Data Types: double

## Version History

Introduced in R2011a

## See Also

### Functions

grpdelay

### Topics

"Analysis Methods for Filter System Objects" on page 3-2

# help

Help for design method with filter specification

## Syntax

```
help(d, 'designmethod')
```

## Description

`help(d, 'designmethod')` displays help in the command window for the design algorithm `designmethod` for the current specifications of the filter specification object `d`. `designmethod` must be one of the design algorithms returned by `designmethods` for `d`, the design object.

## Examples

### Get Help for Designing Butterworth Filters

Get specific help for designing lowpass Butterworth filters.

The first lowpass filter uses the default specification 'Fp,Fst,Ap,Ast' and returns help text specific to the specification.

```
d = fdesign.lowpass;
designmethods(d, 'Systemobject', true)
```

Design Methods that support System objects for class `fdesign.lowpass` (Fp,Fst,Ap,Ast):

```
butter
cheby1
cheby2
ellip
equiripple
ifir
kaiserwin
multistage
```

```
help(d, 'butter')
```

DESIGN Design a Butterworth IIR filter.

`HD = DESIGN(D, 'butter')` designs a Butterworth filter specified by the `FDESIGN` object `D`, and returns the `DFILT/MFILT` object `HD`.

`HD = DESIGN(D, ..., 'SystemObject', true)` implements the filter, `HD`, using a System object instead of a `DFILT/MFILT` object.

`HD = DESIGN(..., 'FilterStructure', STRUCTURE)` returns a filter with the structure `STRUCTURE`. `STRUCTURE` is 'df2sos' by default and can be any of the following:

```
'df1sos'
```

```
'df2sos'
'dfltsos'
'df2tsos'
'cascadeallpass'
'cascadewdfallpass'
```

Some of the listed structures may not be supported by System object filters. Type `validstructures(D, 'butter', 'SystemObject', true)` to get a list of structures supported by System objects.

`HD = DESIGN(..., 'MatchExactly', MATCH)` designs a Butterworth filter and matches the frequency and magnitude specification for the band MATCH exactly. The other band will exceed the specification. MATCH can be 'stopband' or 'passband' and is 'stopband' by default.

`HD = DESIGN(..., 'SOSScaleNorm', NORM)` designs an SOS filter and scales the coefficients using the P-Norm NORM. NORM can be either a discrete-time-domain norm or a frequency-domain norm. Valid time-domain norms are 'l1', 'l2', and 'linf'. Valid frequency-domain norms are 'L1', 'L2', and 'Linf'. Note that L2-norm is equal to l2-norm (Parseval's theorem) but the same is not true for other norms.

The different norms can be ordered in terms of how stringent they are as follows: 'l1' >= 'Linf' >= 'L2' = 'l2' >= 'L1' >= 'linf'. Using the most stringent scaling, 'l1', the filter is the least prone to overflow, but also has the worst signal-to-noise ratio. Linf-scaling is the most commonly used scaling in practice.

Scaling is turned off by default, which is equivalent to setting `SOSScaleNorm = ''`.

`HD = DESIGN(..., 'SOSScaleOpts', OPTS)` designs an SOS filter and scales the coefficients using an `FDOPTS.SOSSCALING` object OPTS. Scaling options are:

| Property               | Default    | Description/Valid values   |
|------------------------|------------|--|
| 'sosReorder'           | 'auto'     | Reorder section prior to scaling.<br>{'auto', 'none', 'up', 'down', 'lowpass', 'highpass', 'bandpass', 'bandstop'} |
| 'MaxNumerator'         | 2          | Maximum value for numerator coefficients   |
| 'NumeratorConstraint'  | 'none'     | {'none', 'unit', 'normalize', 'po2'}   |
| 'OverflowMode'         | 'wrap'     | {'wrap', 'saturate'}   |
| 'ScaleValueConstraint' | 'unit'     | {'unit', 'none', 'po2'}  |
| 'MaxScaleValue'        | 'Not used' | Maximum value for scale values   |

When `sosReorder` is set to 'auto', the sections will be automatically reordered depending on the response type of the design (lowpass, highpass, etc.).

Note that 'MaxScaleValue' will only be used when 'ScaleValueConstraint' is set to something other than 'unit'. If 'MaxScaleValue' is set to a number, the 'ScaleValueConstraint' will be changed to 'none'. Further, if `SOSScaleNorm` is off (as it is by default), then all the `SOSScaleOpts` will be ignored.

For more information about P-Norm and scaling options see help for `DFILT\SCALE`.

```
% Example #1 - Compare passband and stopband MatchExactly.
```



```

h      = fdesign.lowpass('Fp,Fst,Ap,Ast', .1, .3, 1, 60);
Hd     = design(h, 'butter', 'MatchExactly', 'passband');
Hd(2) = design(h, 'butter', 'MatchExactly', 'stopband');

% Compare the passband edges in FVTool.
fvtool(Hd);
axis([.09 .11 -2 0]);

```

Note the discussion of the `MatchExactly` input option. When you use a design object that uses a different specification, such as `'N,F3dB'`, the help content for the `butter` design method changes.

In this case, the `MatchExactly` option does not appear in the help because it is not an available input argument for the specification `'N,F3dB'`.

```

d = fdesign.lowpass('N,F3dB')

d =
  lowpass with properties:

      Response: 'Lowpass'
  Specification: 'N,F3dB'
  Description: {2x1 cell}
  NormalizedFrequency: 1
      FilterOrder: 10
          F3dB: 0.5000

```

```
designmethods(d, 'SystemObject', true)
```

Design Methods that support System objects for class `fdesign.lowpass (N,F3dB)`:

```
butter
maxflat
```

```
help(d, 'butter')
```

DESIGN Design a Butterworth IIR filter.

`HD = DESIGN(D, 'butter')` designs a Butterworth filter specified by the `FDESIGN` object `D`, and returns the `DFILT/MFILT` object `HD`.

`HD = DESIGN(D, ..., 'SystemObject', true)` implements the filter, `HD`, using a System object instead of a `DFILT/MFILT` object.

`HD = DESIGN(..., 'FilterStructure', STRUCTURE)` returns a filter with the structure `STRUCTURE`. `STRUCTURE` is `'df2sos'` by default and can be any of the following:

```

'df1sos'
'df2sos'
'df1tsos'
'df2tsos'
'cascadeallpass'
'cascadewdfallpass'

```

Some of the listed structures may not be supported by System object filters. Type `validstructures(D, 'butter', 'SystemObject', true)` to get a list of structures supported by System objects.

`HD = DESIGN(..., 'SOSScaleNorm', NORM)` designs an SOS filter and scales the coefficients using the P-Norm `NORM`. `NORM` can be either a discrete-time-domain norm or a frequency-domain norm. Valid time-domain norms are 'l1', 'l2', and 'linf'. Valid frequency-domain norms are 'L1', 'L2', and 'Linf'. Note that L2-norm is equal to l2-norm (Parseval's theorem) but the same is not true for other norms.

The different norms can be ordered in terms of how stringent they are as follows: 'l1' >= 'Linf' >= 'L2' = 'l2' >= 'L1' >= 'linf'. Using the most stringent scaling, 'l1', the filter is the least prone to overflow, but also has the worst signal-to-noise ratio. Linf-scaling is the most commonly used scaling in practice.

Scaling is turned off by default, which is equivalent to setting `SOSScaleNorm = ''`.

`HD = DESIGN(..., 'SOSScaleOpts', OPTS)` designs an SOS filter and scales the coefficients using an `FDOPTS.SOSSCALING` object `OPTS`. Scaling options are:

| Property               | Default    | Description/Valid values   |
|------------------------|------------|--|
| 'sosReorder'           | 'auto'     | Reorder section prior to scaling.<br>{'auto', 'none', 'up', 'down', 'lowpass', 'highpass', 'bandpass', 'bandstop'} |
| 'MaxNumerator'         | 2          | Maximum value for numerator coefficients   |
| 'NumeratorConstraint'  | 'none'     | {'none', 'unit', 'normalize', 'po2'}   |
| 'OverflowMode'         | 'wrap'     | {'wrap', 'saturate'}   |
| 'ScaleValueConstraint' | 'unit'     | {'unit', 'none', 'po2'}  |
| 'MaxScaleValue'        | 'Not used' | Maximum value for scale values   |

When `sosReorder` is set to 'auto', the sections will be automatically reordered depending on the response type of the design (lowpass, highpass, etc.).

Note that 'MaxScaleValue' will only be used when 'ScaleValueConstraint' is set to something other than 'unit'. If 'MaxScaleValue' is set to a number, the 'ScaleValueConstraint' will be changed to 'none'. Further, if `SOSScaleNorm` is off (as it is by default), then all the `SOSScaleOpts` will be ignored.

For more information about P-Norm and scaling options see help for `DFILT\SCALE`.

```
% Example #1 - Design a lowpass Butterworth filter in the DF2TSOS structure.
h = fdesign.lowpass('N,F3dB');
Hd = design(h, 'butter', 'FilterStructure', 'df2tsos');
```

## Version History

Introduced in R2011a

### See Also

`fdesign` | `design` | `designmethods` | `designopts`

# hide

## Package:

Hide scope window

## Syntax

```
hide(scope)
```

## Description

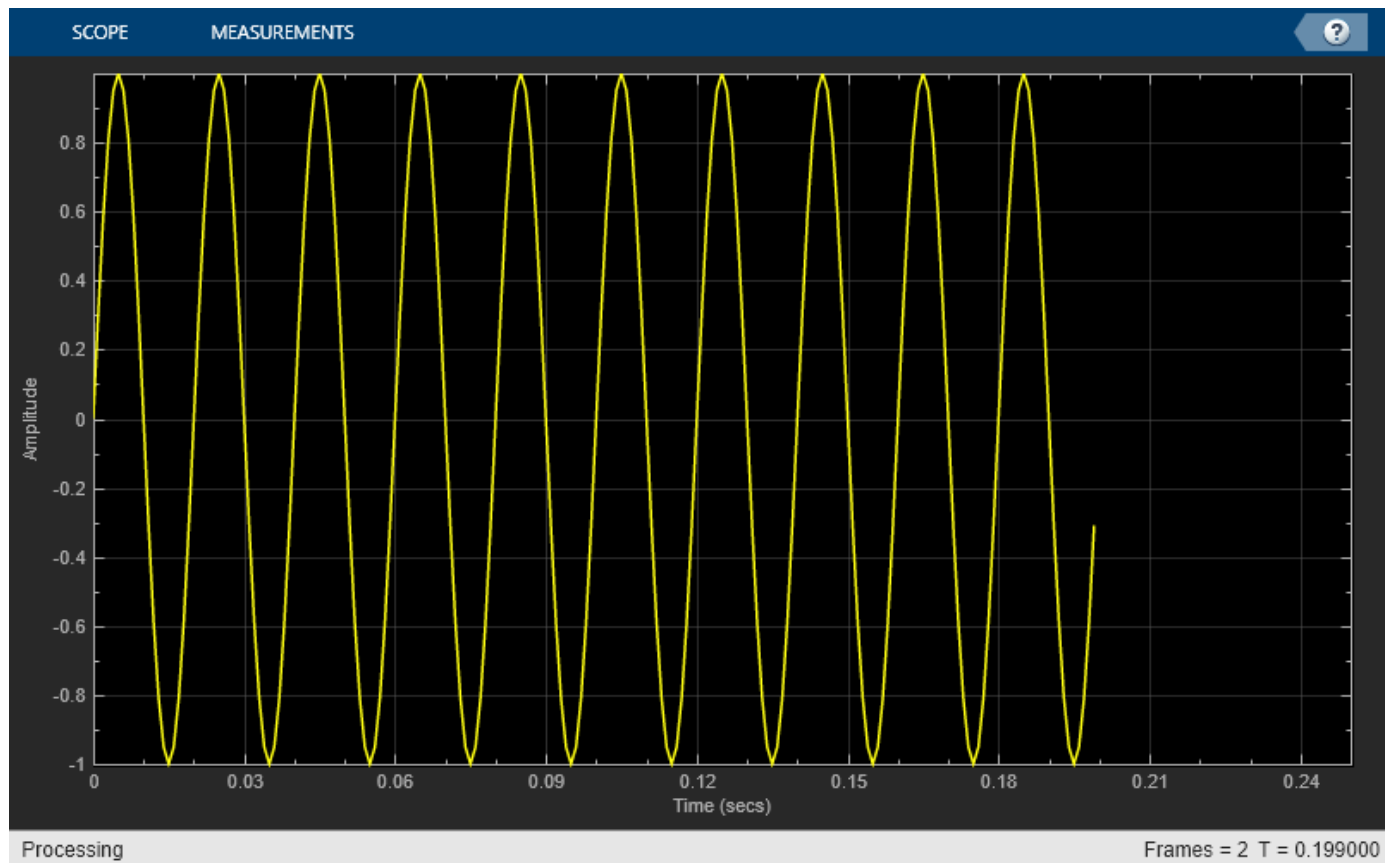
hide(scope) hides the window of the scope.

## Examples

### Hide and Show Time Scope

Create a sine wave signal and view it in the scope.

```
Fs = 1000; % Sample rate
signal = dsp.SineWave(Frequency=50, SampleRate=Fs, ...
    SamplesPerFrame=100);
scope = timescope(SampleRate=Fs, TimeSpanSource="property", ...
    TimeSpan=0.25, YLimits=[-1 1]);
for ii = 1:2
    xsine = signal();
    scope(xsine)
end
```

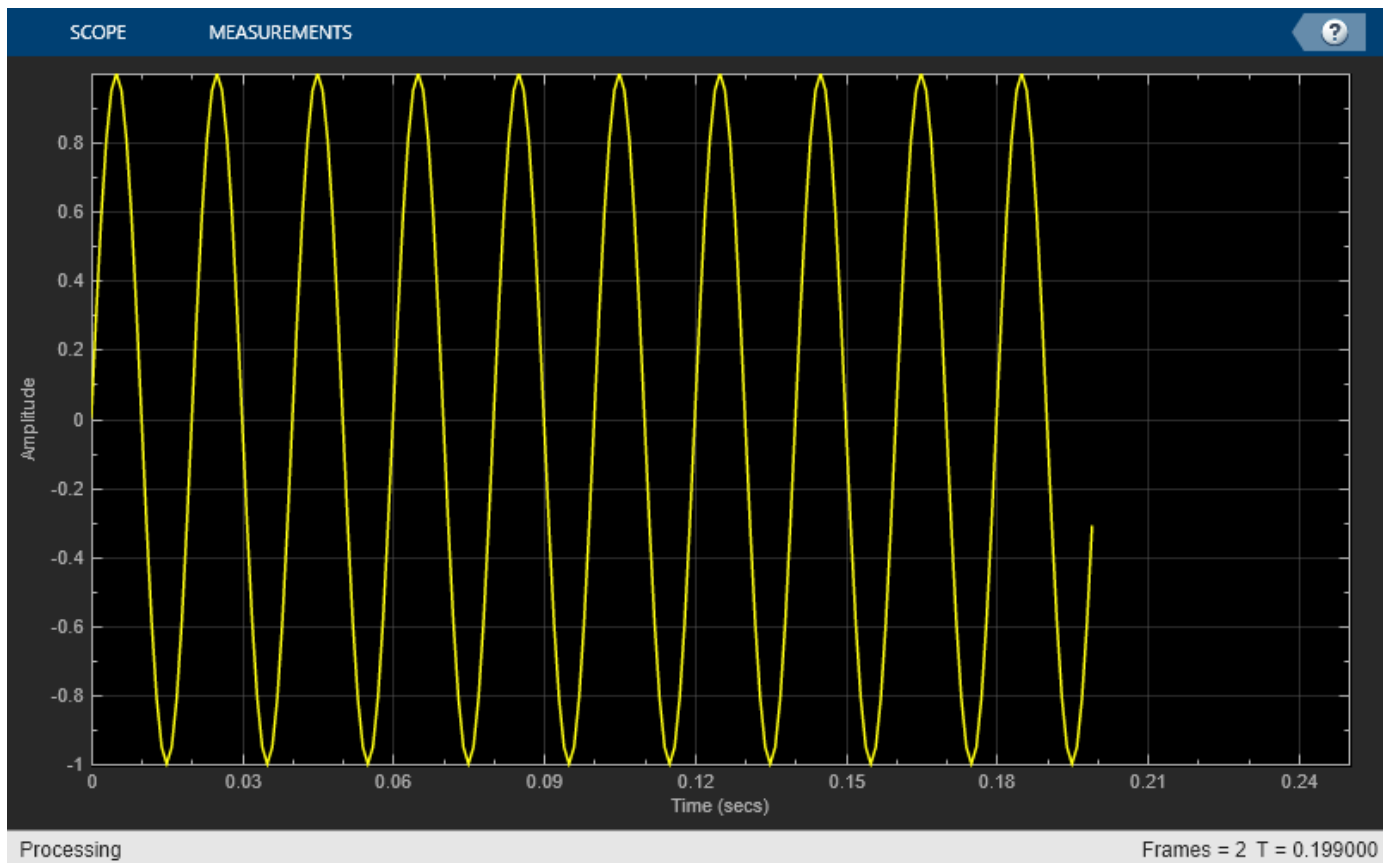


Hide the scope window.

```
if(isVisible(scope))  
    hide(scope)  
end
```

Show the scope window.

```
if(~isVisible(scope))  
    show(scope)  
end
```



Clean up workspace variables.

```
clear scope Fs sine ii xsine
```

## Input Arguments

### **scope** — Scope object

scope object

Scope object, specified as one of the following:

- spectrumAnalyzer object
- dsp.ArrayPlot object
- dsp.LogicAnalyzer System object
- timescope object
- dsp.DynamicFilterVisualizer object

Example: `myScope = timescope; hide(myScope)`

## Version History

Introduced in R2011a

**R2022a: Support for `dsp.SpectrumAnalyzer` object will be removed**

*Not recommended starting in R2022a*

The `hide` function will stop supporting the `dsp.SpectrumAnalyzer` object in a future release. Use the `spectrumAnalyzer` object instead.

**Update Code**

No updates to your code are required except replacing instances of `dsp.SpectrumAnalyzer` with `spectrumAnalyzer`.

This table shows how the `hide` function typically uses the `dsp.SpectrumAnalyzer System` object and explains how to update existing code to use the `spectrumAnalyzer` object.

| Discouraged Usage   | Recommended Replacement   |
|---|---|
| <pre>scope = dsp.SpectrumAnalyzer;<br/>scope(randn(100000,1))<br/>hide(scope)</pre> | <pre>scope = spectrumAnalyzer;<br/>scope(randn(100000,1))<br/>hide(scope)</pre> |

**See Also****Functions**

`show` | `isVisible` | `step`

**Objects**

`dsp.DynamicFilterVisualizer` | `timescope` | `dsp.ArrayPlot` | `dsp.LogicAnalyzer` | `spectrumAnalyzer`

# ifir

Interpolated FIR filter design

## Syntax

```
[h,g] = ifir(l,type,f,dev)
[h,g,d] = ifir(l,type,f,dev)
[...] = ifir(...,str)
```

## Description

`[h,g] = ifir(l,type,f,dev)` designs a periodic filter  $h(z^l)$ , where  $l$  is the interpolation factor. It also finds an image-suppressor filter  $g(z)$ , such that the cascade of the two filters represents the optimal minimax FIR approximation of the desired response. This response is specified by `type`, with band edge frequencies contained in vector `f`. This is done while not exceeding the maximum deviations or ripples (linear) specified in vector `dev`.

When `type` is set to 'low', the filter design is a lowpass design. When `type` is set to 'high', the filter design is a highpass design. `f` is a two-element vector with passband and stopband edge frequency values. For narrowband lowpass filters and wideband highpass filters,  $l \times f(2)$  is less than 1. For wideband lowpass filters and narrowband highpass filters, specify `f` so that  $l \times (1 - f(1))$  is less than 1.

`dev` is a two-element vector that contains the peak ripple or deviation (in linear units) allowed for both the passband and the stopband.

The `ifir` design algorithm achieves an efficient design in the sense that it reduces the total number of multipliers required. To do this, the design problem is broken into two stages. In the first stage, the filter is upsampled to achieve the stringent specifications without using many multipliers. In the second stage, the filter removes the images created when upsampling the previous filter.

`[h,g,d] = ifir(l,type,f,dev)` returns a delay `d` that is connected in parallel with the cascade of  $h(z^l)$  and  $g(z)$  for both wideband lowpass and highpass filters. This is necessary to obtain the desired response.

`[...] = ifir(...,str)` uses `str` to choose the algorithm level of optimization used. Possible values for `str` are 'simple', 'intermediate' (default) or 'advanced'. `str` provides for a tradeoff between design speed and filter order optimization. The 'advanced' option can result in substantial filter order reduction, especially for  $g(z)$ .

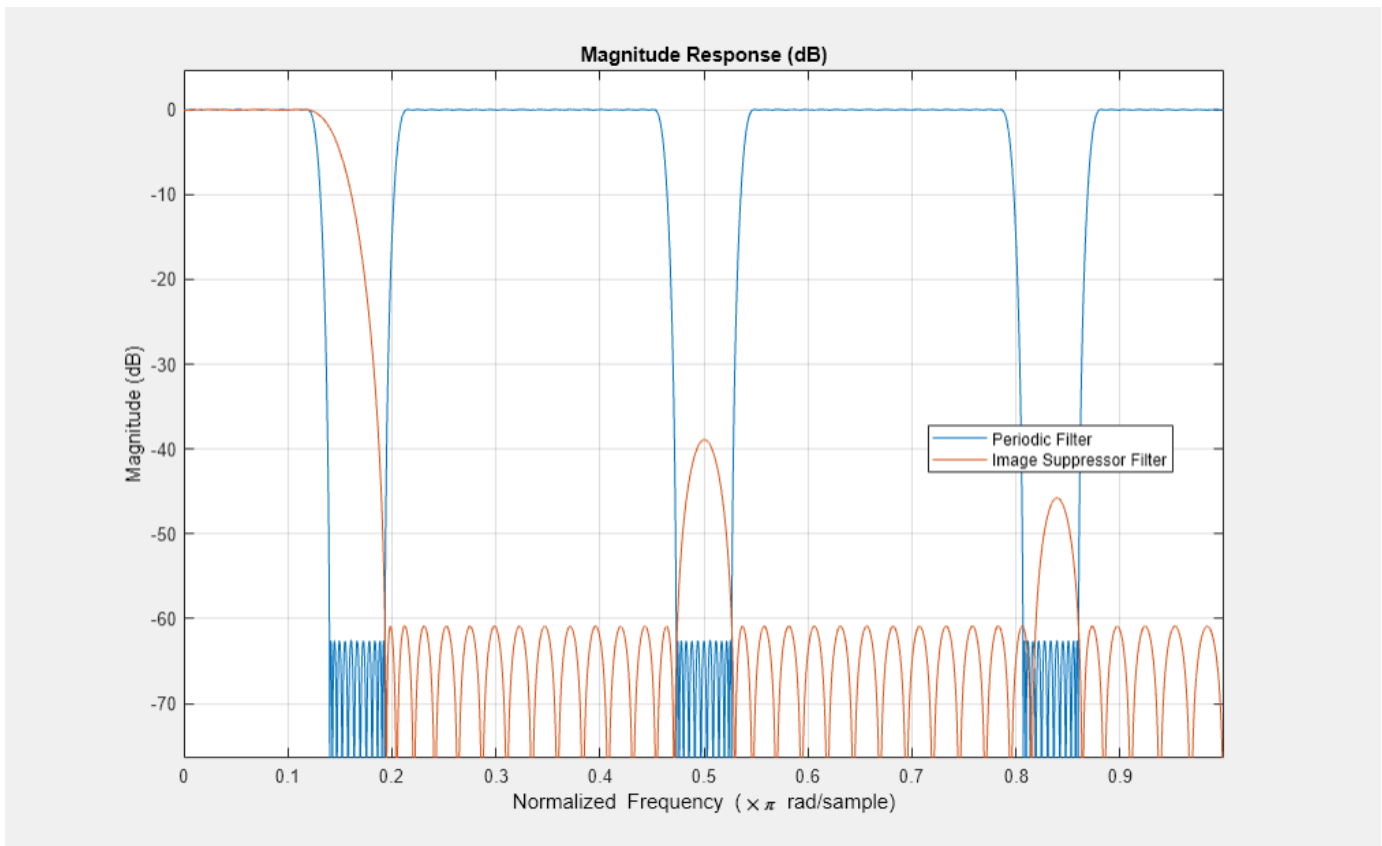
## Examples

### Narrowband lowpass design using an interpolation factor of 6

This example shows how to use the function `ifir` to design a narrowband lowpass filter.

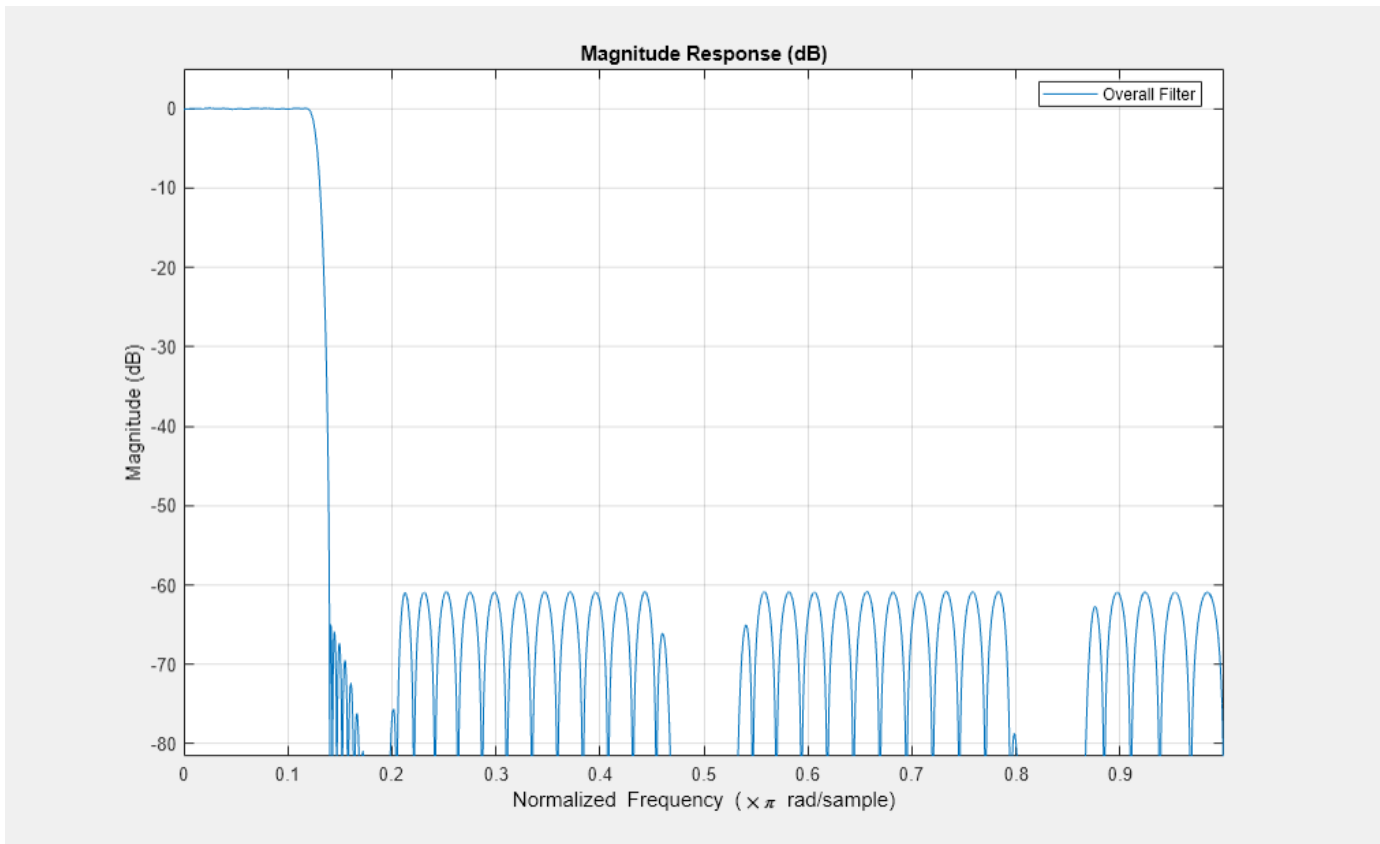
```
[h,g] = ifir(6,'low',[.12 .14],[.01 .001]);
H = dsp.FIRFilter('Numerator',h);
G = dsp.FIRFilter('Numerator',g);
```

```
hfv = fvtool(H,G);  
legend(hfv, 'Periodic Filter', 'Image Suppressor Filter')
```



```
Hcas = cascade(H,G);  
hfv2 = fvtool(Hcas);  
legend(hfv2, 'Overall Filter')
```

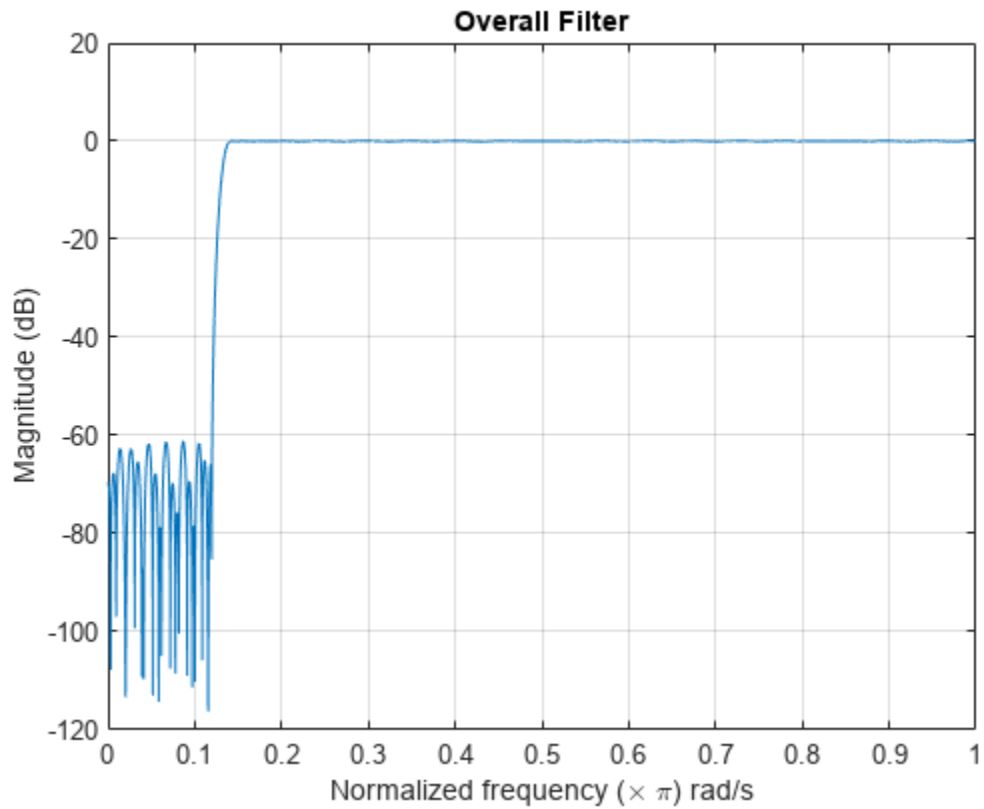




### Wideband highpass design using an interpolation factor of 6

This example shows how to use `ifir` to design a wideband highpass filter.

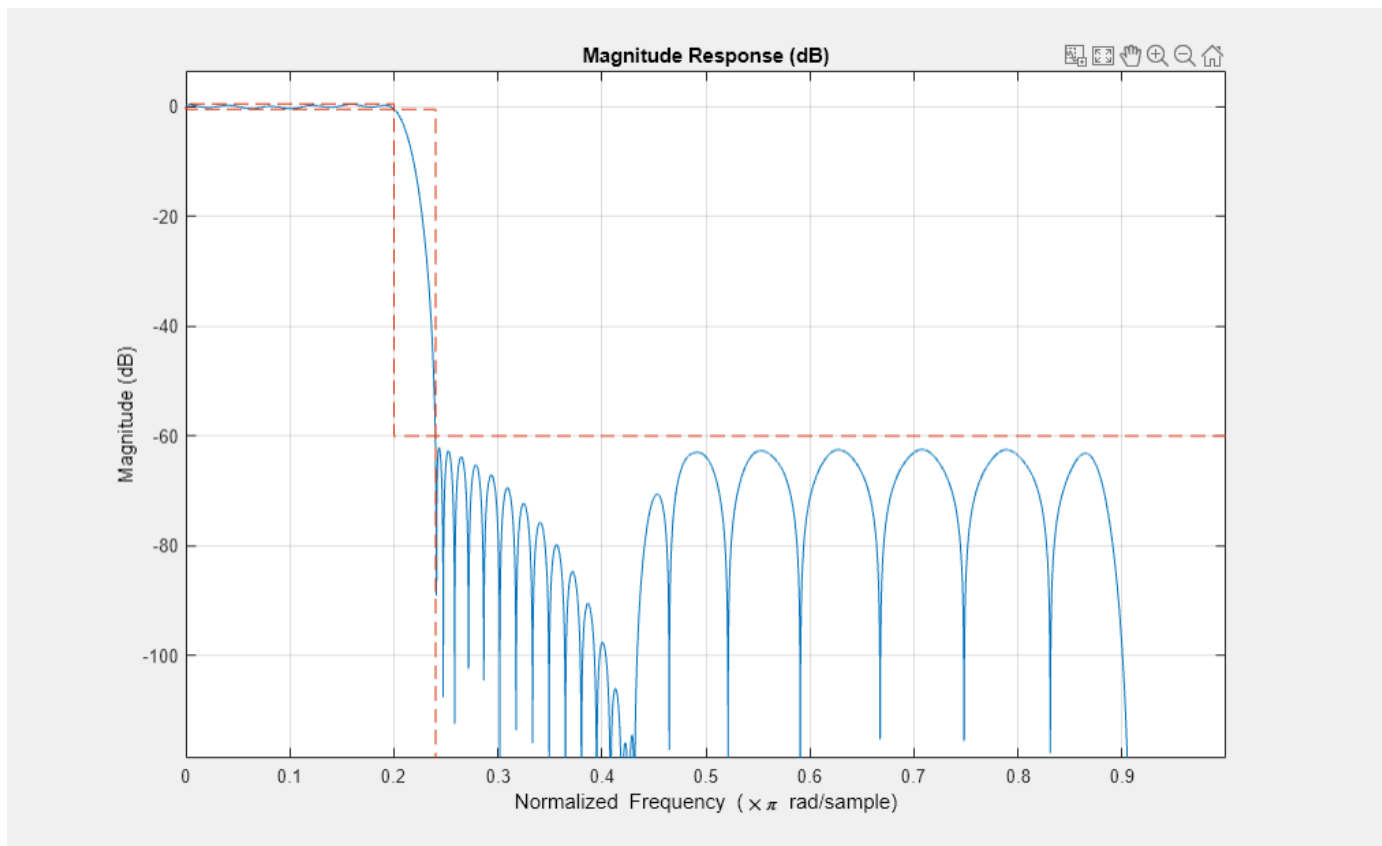
```
[h,g,d]=ifir(6,'high',[.12 .14],[.001 .01]);
H = dsp.FIRFilter('Numerator',h); G = dsp.FIRFilter('Numerator',g);
b1 = cascade(H,G); % Branch 1
b2 = dsp.FIRFilter('Numerator',d); % Branch 2
Hoverall = freqz(b1) + freqz(b2); % Overall wideband highpass
plot(linspace(0,1,length(Hoverall)),20*log10(abs(Hoverall)))
xlabel('Normalized frequency (\times \pi) rad/s')
ylabel('Magnitude (dB)')
title('Overall Filter');
grid on
```



### Design a cascade of lowpass filters

This example shows how to use `fdesign.lowpass` to design a cascade of lowpass filters. After designing the filter, use `fvtool` to plot the response curve.

```
fpass = 0.2;  
fstop = 0.24;  
d1 = fdesign.lowpass(fpass, fstop);  
lowpassCascade = design(d1, 'ifir', 'Systemobject', true);  
fvtool(lowpassCascade)
```



## Version History

Introduced in R2011a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

All inputs must be constant. Expressions or variables are allowed if their values do not change.

## See Also

### Functions

fdesign | firgr | fir1 | fir1s | firpm

## iirbpc2bpc

Transform IIR complex bandpass filter to IIR complex bandpass filter with different characteristics

### Syntax

```
[Num,Den,AllpassNum,AllpassDen] = iirbpc2bpc(B,A,Wo,Wt)
```

### Description

`[Num,Den,AllpassNum,AllpassDen] = iirbpc2bpc(B,A,Wo,Wt)` transform IIR complex bandpass filter to IIR complex bandpass filter with different characteristics.

The `iirbpc2bpc` function returns the numerator and denominator vectors, `Num` and `Den`, respectively of the target filter transformed from the complex bandpass prototype by applying a first-order complex bandpass to complex bandpass frequency transformation. For more details, see IIR Complex Bandpass Frequency Transformation on page 5-910.

The function also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is specified with the numerator `B` and a denominator `A`.

Frequencies must be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

### Examples

#### Shift Complex Bandpass

Design a prototype real IIR lowpass elliptic filter with a gain of about -3 dB at  $0.5\pi$  rad/sample.

```
[b,a] = ellip(3,0.1,30,0.409);
```

Create a complex passband from  $0.25\pi$  to  $0.75\pi$ .

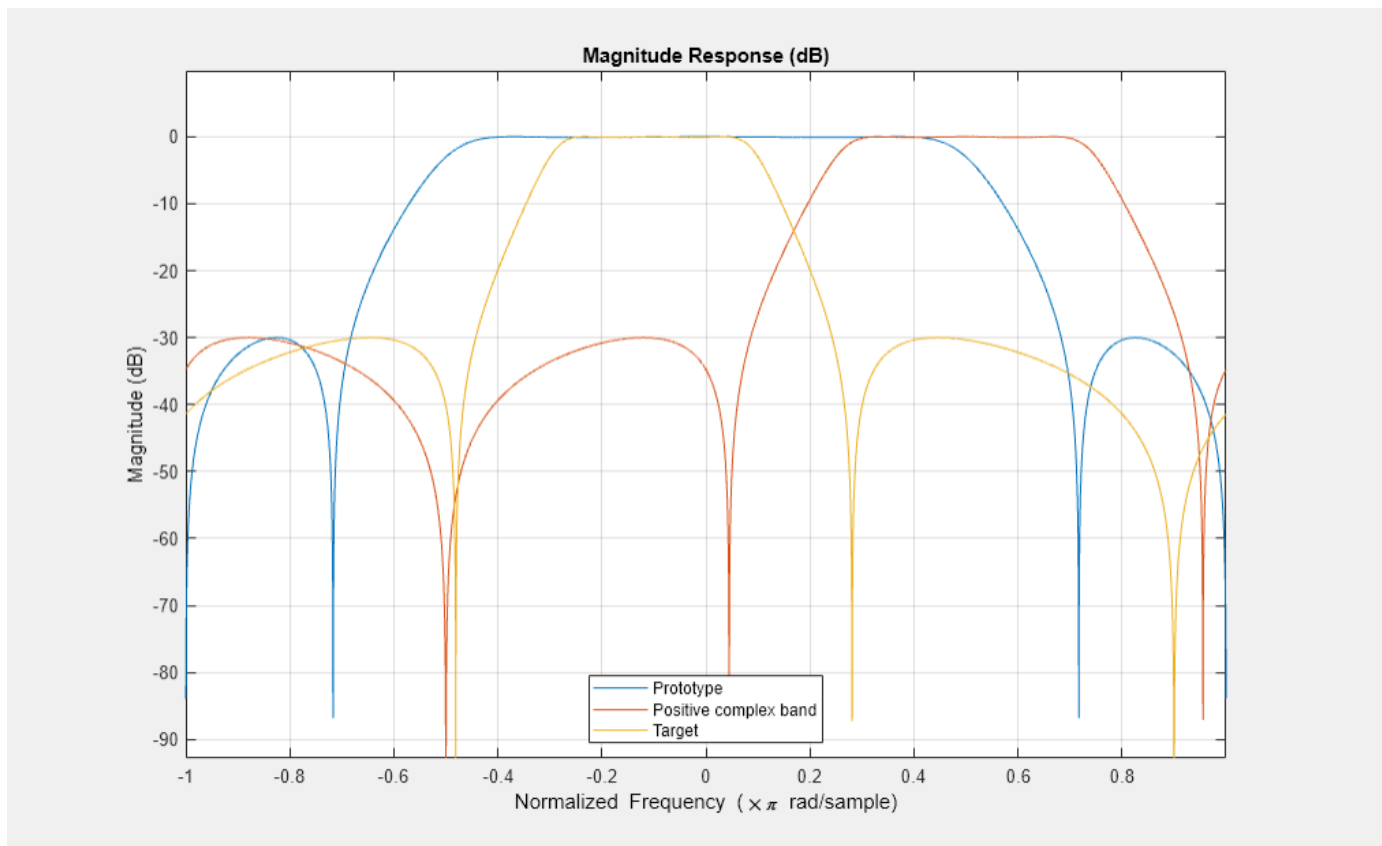
```
[bc,ac] = iirlp2bpc(b,a,0.5,[0.25 0.75]);
```

Move the bandpass to between  $-0.3\pi$  and  $0.1\pi$ .

```
[num,den] = iirbpc2bpc(bc,ac,[0.25 0.75],[-0.3 0.1]);
```

Compare the three filters in FVTool.

```
hvft = fvtool(b,a,bc,ac,num,den);
legend(hvft,'Prototype','Positive complex band','Target')
```



## Input Arguments

### **B** — Numerator of prototype lowpass filter

row vector

Numerator of the prototype lowpass filter, specified as a row vector.

Data Types: `single` | `double`

Complex Number Support: Yes

### **A** — Denominator of prototype lowpass filter

row vector

Denominator of the prototype lowpass filter, specified as a row vector.

Data Types: `single` | `double`

Complex Number Support: Yes

### **Wo** — Frequency values to be transformed from prototype filter

two-element vector

Frequency values to be transformed from the prototype filter, specified as a two-element vector. Frequencies in **Wo** must be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

Data Types: `single` | `double`

**Wt — Desired frequency locations in transformed target filter**

two-element vector

Desired frequency locations in the transformed target filter, specified as a two-element vector. Frequencies in **Wt** must be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

Data Types: single | double

**Output Arguments****Num — Numerator of target filter**

row vector

Numerator of the target filter, returned as a row vector.

Data Types: single | double

Complex Number Support: Yes

**Den — Denominator of target filter**

row vector

Denominator of the target filter, returned as a row vector.

Data Types: single | double

Complex Number Support: Yes

**AllpassNum — Numerator of mapping filter**

row vector

Numerator of the mapping filter, returned as a row vector.

Data Types: single | double

Complex Number Support: Yes

**AllpassDen — Denominator of mapping filter**

row vector

Denominator of the mapping filter, returned as a row vector.

Data Types: single | double

**More About****IIR Complex Bandpass Frequency Transformation**

IIR Complex Bandpass Frequency transformation effectively places two features of the original filter, located at frequencies  $W_{o1}$  and  $W_{o2}$ , at the required target frequency locations,  $W_{t1}$ , and  $W_{t2}$  respectively. It is assumed that  $W_{t2}$  is greater than  $W_{t1}$ . In most of the cases the features selected for the transformation are the band edges of the filter passbands. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Relative positions of other features of the original filter do not change in the target filter. This means that it is possible to select two features of an original filter,  $F_1$  and  $F_2$ , with  $F_1$  preceding  $F_2$ . Feature  $F_1$  will still precede  $F_2$  after the transformation. However, the distance between  $F_1$  and  $F_2$  will not be the same before and after the transformation.

IIR complex bandpass frequency transformation can also be used for transforming other types of filters; e.g., complex notch filters or resonators can be repositioned at two distinct desired frequencies at any place around the unit circle for example in the adaptive system.

## **Version History**

**Introduced in R2011a**

### **See Also**

[iirfttransf](#) | [allpassbpc2bpc](#) | [zpkbpc2bpc](#)

## iircomb

IIR comb notch or peak filter

### Syntax

```
[num,den] = iircomb(n,bw)
[num,den] = iircomb(n,bw,ab)
[num,den] = iircomb( ____,type)
```

### Description

`[num,den] = iircomb(n,bw)` returns a digital notching filter with order  $n$  and with the width of the filter notch at -3 dB set to  $bw$ , the filter bandwidth.

For the notching filter, the transfer function takes the form:

$$H(z) = b \frac{1 - z^{-n}}{1 - \alpha z^{-n}}$$

where  $\alpha$  and  $b$  are positive scalars and  $n$  is the filter order or the number of notches in the filter minus 1.

`[num,den] = iircomb(n,bw,ab)` returns a digital notching filter whose bandwidth,  $bw$ , is specified at a level of  $-ab$  decibels.

`[num,den] = iircomb( ____,type)` returns either a digital notch filter or a digital peaking filter as specified by `type`.

The transfer function for peaking filters is:

$$H(z) = b \frac{1 - z^{-n}}{1 + az^{-n}}$$

### Examples

#### Design IIR Notch Filter

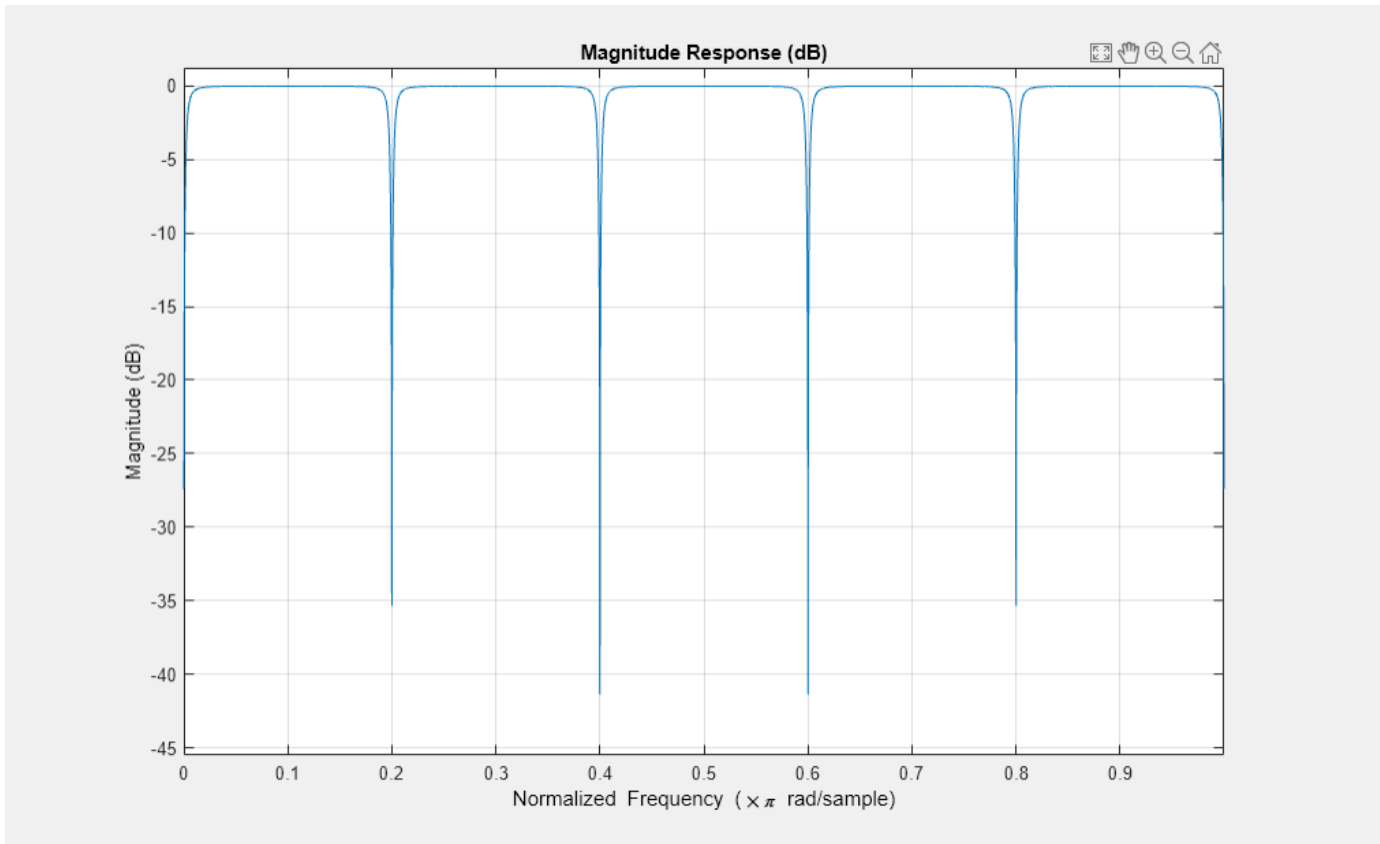
Design and plot an IIR notch filter with 11 notches (equal to filter order plus 1) that removes a 60 Hz tone ( $f_0$ ) from a signal at 600 Hz( $f_s$ ). For this example, set the Q factor for the filter to 35 and use it to specify the filter bandwidth.

```
fs = 600;
fo = 60;
q = 35;
bw = (fo/(fs/2))/q;
[b,a] = iircomb(fs/fo,bw,'notch'); % Note type flag 'notch'
```

Use the Filter Visualization Tool (`fvtool`) to generate the plot showing the filter notches.



```
fvtool(b,a);
```



Note that the notches are evenly spaced and one falls at exactly 60 Hz.

## Input Arguments

### **n** — Order of digital filter

positive integer

Order of the digital filter, specified as a positive integer.  $n$  also defines the number of notches (or peaks) in the filter across the frequency range 0 to  $2\pi$ . The number of notches (or peaks) equals  $n+1$ .

Data Types: `single` | `double`

### **bw** — Bandwidth of filter

positive scalar in the range  $(0, 1)$

Bandwidth of the filter at a level of  $-ab$  decibels, specified as a positive scalar in the range  $(0, 1)$ .

The quality factor (Q factor)  $q$  for the filter is related to the filter bandwidth by  $q = \omega_0/bw$  where  $\omega_0$  is the frequency to remove from the signal.

Data Types: `single` | `double`

### **ab** — Magnitude response at filter bandwidth

-3 (default) | scalar

Magnitude response at the filter bandwidth `bw`, specified as a scalar in dB. You can include the optional input argument `ab` to specify the magnitude response at a level other than the default -3 dB, for example, -6 dB or 0 dB. If you do not include the argument `ab`, the function sets the magnitude response at the default -3 dB level.

Data Types: `single` | `double`

### **type** — Type of digital filter

'notch' (default) | 'peak'

Type of the digital filter, specified as either:

- 'notch' — Design an IIR notch filter. Notch filters attenuate the response at the specified frequencies. This is the default type. When you omit the `type` input argument, `iircomb` function returns a notch filter.
- 'peak' — Design an IIR peaking filter. Peaking filters boost the signal at the specified frequencies.

## **Output Arguments**

### **num** — Numerator coefficients

real-valued row vector

Numerator coefficients of the IIR filter transfer function, returned as a real-valued row vector. For an  $n$ th-order filter, the number of elements in the row vector is  $n+1$ .

Data Types: `single` | `double`

### **den** — Denominator coefficients

real-valued row vector

Denominator coefficients of the IIR filter transfer function, returned as a real-valued row vector. For an  $n$ th-order filter, the number of elements in the row vector is  $n+1$ .

Data Types: `single` | `double`

## **Version History**

Introduced in R2011a

## **References**

[1] Orfanidis, S.J., *Introduction to Signal Processing*, Englewood Cliffs, New Jersey: Prentice-Hall, 1996.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

## **See Also**

`firgr` | `iirnotch` | `iirpeak`

# iirftansf

IIR frequency transformation of digital filter

## Syntax

```
[num,den] = iirftansf(b,a,allpassNum,allpassDen)
```

## Description

`[num,den] = iirftansf(b,a,allpassNum,allpassDen)` returns the numerator and the denominator coefficients of the transformed filter.

The `iirftansf` function transforms a prototype filter, specified by the numerator `b` and denominator `a`, by using an allpass mapping filter, specified by the numerator `allpassNum` and the denominator `allpassDen`. If you do not specify an allpass mapping filter, then the function returns an original filter.

## Examples

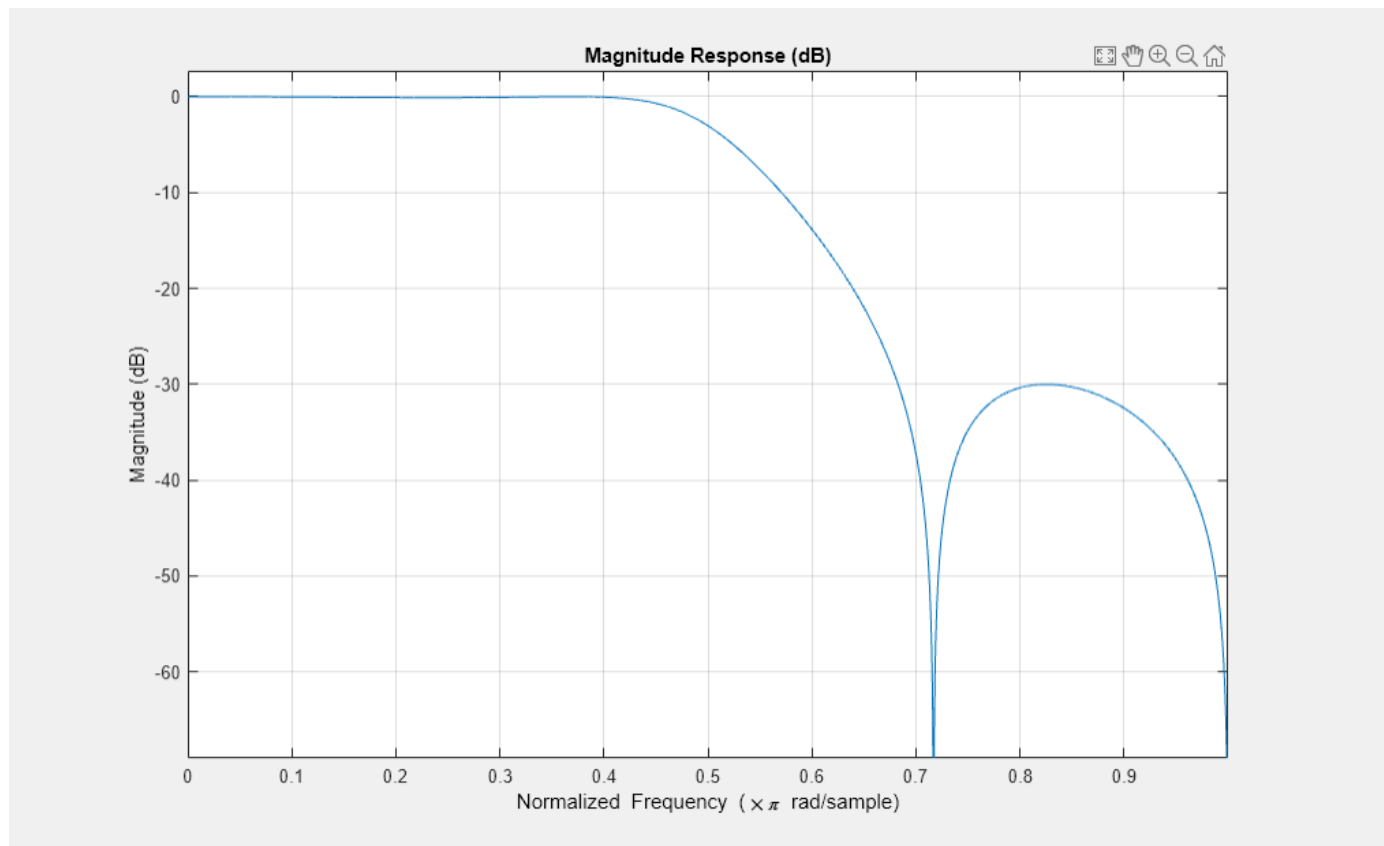
### Transform Lowpass Filter Using Allpass Mapping Filter

Using the `iirftansf` function, extend the passband of a lowpass IIR filter by using an allpass mapping filter.

#### Input Lowpass IIR Filter

Design a prototype real IIR lowpass elliptic filter with a gain of about -3 dB at  $0.5\pi$  rad/sample.

```
[b,a] = ellip(3,0.1,30,0.409);  
fvtool(b,a)
```



### Transform Filter Using `iirftransf`

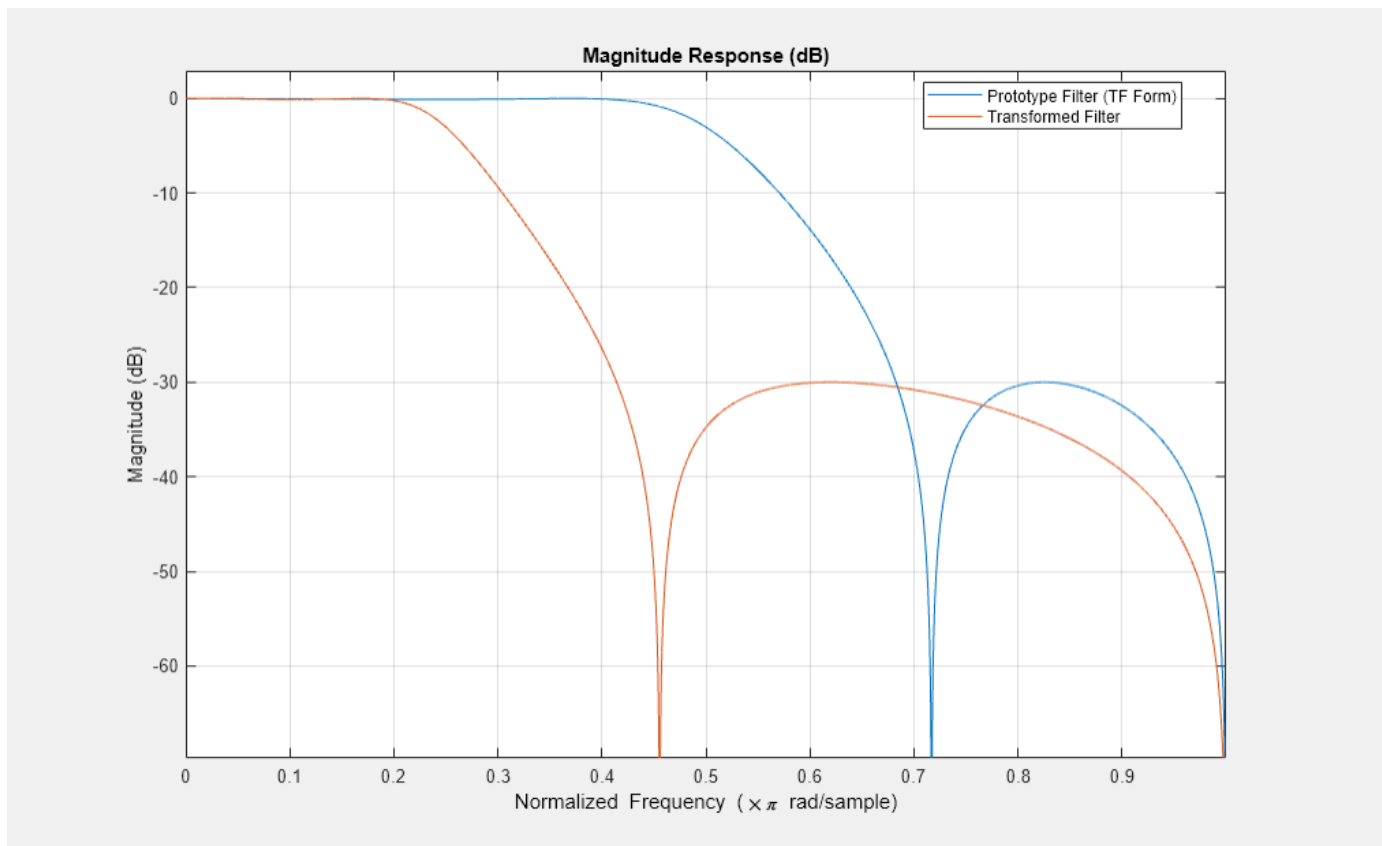
Extend the passband of the real prototype filter by specifying the allpass mapping filter as a vector of numerator and denominator coefficients, `alpnum` and `alpden` respectively. Use the `allpasslp2lp` function to generate the allpass mapping filter coefficients.

Specify the prototype filter as a vector of numerator and denominator coefficients, `b` and `a` respectively.

```
[b,a] = ellip(3,0.1,30,0.409);
[alpnum,alpden] = allpasslp2lp(0.5,0.25);
[num,den] = iirftransf(b,a,alpnum,alpden);
```

Compare the magnitude response of the filters using FVTool.

```
fvt = fvtool(b,a,num,den);
legend(fvt,"Prototype Filter (TF Form)",...
       "Transformed Filter")
```

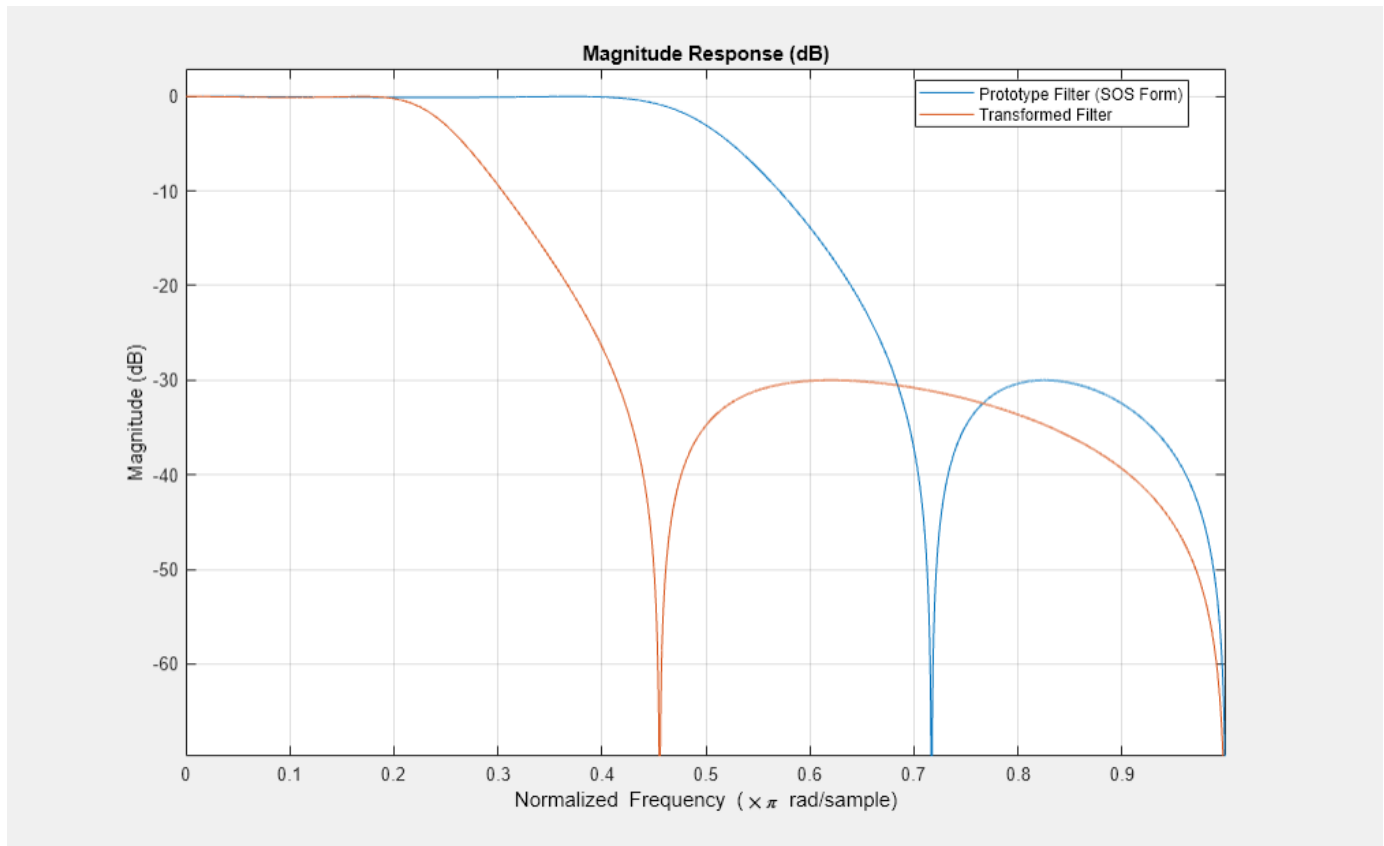


You can also specify the input lowpass IIR filter as a matrix of coefficients. Pass the second-order section matrices as inputs. The numerator and the denominator coefficients of the transformed filter are given by `num2` and `den2`, respectively.

```
ss = tf2sos(b,a);
[num2,den2] = iirftansf(ss(:,1:3),ss(:,4:6),...
    alpnum,alpden);
```

Compare the magnitude response of the filters using `FVTool`.

```
hvft = fvtool(ss,[num2 den2]);
legend(hvft,"Prototype Filter (SOS Form)",...
    "Transformed Filter")
```



## Input Arguments

### **b** — Numerator coefficients of prototype lowpass IIR filter

row vector | matrix

Numerator coefficients of the prototype lowpass IIR filter, specified as either:

- Row vector -- Specifies the values of  $[b_0, b_1, \dots, b_n]$ , given this transfer function form:

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_0 + b_1z^{-1} + \dots + b_nz^{-n}}{a_0 + a_1z^{-1} + \dots + a_nz^{-n}}$$

where  $n$  is the order of the filter.

- Matrix -- Specifies the numerator coefficients in the form of an  $P$ -by- $(Q+1)$  matrix, where  $P$  is the number of filter sections and  $Q$  is the order of each filter section. If  $Q = 2$ , the filter is a second-order section filter. For higher-order sections, make  $Q > 2$ .

$$\mathbf{b} = \begin{bmatrix} b_{01} & b_{11} & b_{21} & \dots & b_{Q1} \\ b_{02} & b_{12} & b_{22} & \dots & b_{Q2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ b_{0P} & b_{1P} & b_{2P} & \dots & b_{QP} \end{bmatrix}$$

In the transfer function form, the numerator coefficient matrix  $b_{ik}$  of the IIR filter can be represented using the following equation:

$$H(z) = \prod_{k=1}^P H_k(z) = \prod_{k=1}^P \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2} + \dots + b_{Qk}z^{-Q}}{a_{0k} + a_{1k}z^{-1} + a_{2k}z^{-2} + \dots + a_{Qk}z^{-Q}},$$

where,

- **a** -- Denominator coefficients matrix. For more information on how to specify this matrix, see **a**.
- *k* -- Row index.
- *i* -- Column index.

When specified in the matrix form, **b** and **a** matrices must have the same number of rows (filter sections) *Q*.

Data Types: `single` | `double`

### **a** — Denominator coefficients of prototype lowpass IIR filter

row vector | matrix

Denominator coefficients for a prototype lowpass IIR filter, specified as one of these options:

- Row vector -- Specifies the values of  $[a_0, a_1, \dots, a_n]$ , given this transfer function form:

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_0 + b_1z^{-1} + \dots + b_nz^{-n}}{a_0 + a_1z^{-1} + \dots + a_nz^{-n}},$$

where *n* is the order of the filter.

- Matrix -- Specifies the denominator coefficients in the form of an *P*-by- $(Q+1)$  matrix, where *P* is the number of filter sections and *Q* is the order of each filter section. If *Q* = 2, the filter is a second-order section filter. For higher-order sections, make *Q* > 2.

$$\mathbf{a} = \begin{bmatrix} a_{01} & a_{11} & a_{21} & \dots & a_{Q1} \\ a_{02} & a_{12} & a_{22} & \dots & a_{Q2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{0P} & a_{1P} & a_{2P} & \dots & a_{QP} \end{bmatrix}$$

In the transfer function form, the denominator coefficient matrix  $a_{ik}$  of the IIR filter can be represented using the following equation:

$$H(z) = \prod_{k=1}^P H_k(z) = \prod_{k=1}^P \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2} + \dots + b_{Qk}z^{-Q}}{a_{0k} + a_{1k}z^{-1} + a_{2k}z^{-2} + \dots + a_{Qk}z^{-Q}},$$

where,

- **b** -- Numerator coefficients matrix. For more information on how to specify this matrix, see **b**.
- *k* -- Row index.
- *i* -- Column index.

When specified in the matrix form, **a** and **b** matrices must have the same number of rows (filter sections) *P*.

Data Types: `single` | `double`

**allpassNum — Numerator coefficients of mapping filter**

row vector

Numerator coefficients of the mapping filter, specified as a row vector.

Data Types: `single` | `double`

Complex Number Support: Yes

**allpassDen — Denominator coefficients of mapping filter**

row vector

Denominator coefficients of the mapping filter, specified as a row vector.

Data Types: `single` | `double`

Complex Number Support: Yes

## Output Arguments

**num — Numerator coefficients of transformed filter**

row vector | matrix

Numerator coefficients of the transformed filter, returned as one of the following:

- Row vector of length  $n+1$ , where  $n$  is the order of the input filter. The `num` output is a row vector when the input coefficients `b` and `a` are row vectors.
- $P$ -by- $(Q+1)$  matrix, where  $P$  is the number of filter sections and  $Q$  is the order of each section of the transformed filter. The `num` output is a matrix when the input coefficients `b` and `a` are matrices.

Data Types: `single` | `double`**den — Denominator coefficients of transformed filter**

row vector | matrix

Denominator coefficients of the transformed filter, returned as one of the following:

- Row vector of length  $n+1$ , where  $n$  is the order of the input filter. The `den` output is a row vector when the input coefficients `b` and `a` are row vectors.
- $P$ -by- $(Q+1)$  matrix, where  $P$  is the number of filter sections and  $Q$  is the order of each section of the transformed filter. The `den` output is a matrix when the input coefficients `b` and `a` are matrices.

Data Types: `single` | `double`

## Version History

Introduced in R2011a

## Extended Capabilities

**C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.



## **See Also**

### **Functions**

zpkfttransf

## iirgrpdelay

Optimal IIR filter with prescribed group-delay

### Syntax

```
[num,den] = iirgrpdelay(n,f,edges,a)
[num,den] = iirgrpdelay(n,f,edges,a,w)
[num,den] = iirgrpdelay(n,f,edges,a,w,radius)
[num,den] = iirgrpdelay(n,f,edges,a,w,radius,p)
[num,den] = iirgrpdelay(n,f,edges,a,w,radius,p,dens)
[num,den] = iirgrpdelay(n,f,edges,a,w,radius,p,dens,initden)
[num,den] = iirgrpdelay(n,f,edges,a,w,radius,p,dens,initden,tau)
[num,den,tau] = iirgrpdelay(n,f,edges,a,w)
```

### Description

`[num,den] = iirgrpdelay(n,f,edges,a)` returns an allpass IIR filter of order  $n$  ( $n$  must be even) which is the best approximation to the relative group-delay response described by  $f$  and  $a$  in the least- $p$ th sense.  $f$  is a vector of frequencies between 0 and 1 and  $a$  is specified in samples. The vector  $edges$  specifies the band-edge frequencies for multi-band designs. `iirgrpdelay` uses a constrained Newton-type algorithm. Always check your resulting filter using `grpdelay` or `freqz`.

`[num,den] = iirgrpdelay(n,f,edges,a,w)` uses the weights in  $w$  to weight the error.  $w$  has one entry per frequency point and must be the same length as  $f$  and  $a$ ). Entries in  $w$  tell `iirgrpdelay` how much emphasis to put on minimizing the error in the vicinity of each specified frequency point relative to the other points.

$f$  and  $a$  must have the same number of elements.  $f$  and  $a$  can contain more elements than the vector  $edges$  contains. This lets you use  $f$  and  $a$  to specify a filter that has any group-delay contour within each band.

`[num,den] = iirgrpdelay(n,f,edges,a,w,radius)` returns a filter having a maximum pole radius equal to `radius`, where  $0 < \text{radius} < 1$ . `radius` defaults to 0.999999. Filters whose pole radius you constrain to be less than 1.0 can better retain transfer function accuracy after quantization.

`[num,den] = iirgrpdelay(n,f,edges,a,w,radius,p)`, where  $p$  is a two-element vector  $[p_{\min} \ p_{\max}]$ , lets you determine the minimum and maximum values of  $p$  used in the least- $p$ th algorithm.  $p$  defaults to  $[2 \ 128]$  which yields filters very similar to the L-infinity, or Chebyshev, norm.  $p_{\min}$  and  $p_{\max}$  should be even. If  $p$  is 'inspect', no optimization occurs. You might use this feature to inspect the initial pole/zero placement.

`[num,den] = iirgrpdelay(n,f,edges,a,w,radius,p,dens)` specifies the grid density `dens` used in the optimization process. The number of grid points is  $(\text{dens} * (n+1))$ . The default is 20. `dens` can be specified as a single-element cell array. The grid is not equally spaced.

`[num,den] = iirgrpdelay(n,f,edges,a,w,radius,p,dens,initden)` allows you to specify the initial estimate of the denominator coefficients in vector `initden`. This can be useful for difficult optimization problems. The pole-zero editor in Signal Processing Toolbox software can be used for generating `initden`.

`[num,den] = iirgrpdelay(n,f,edges,a,w,radius,p,dens,initden,tau)` allows the initial estimate of the group delay offset to be specified by the value of `tau`, in samples.

`[num,den,tau] = iirgrpdelay(n,f,edges,a,w)` returns the resulting group delay offset. In all cases, the resulting filter has a group delay that approximates `[a + tau]`. Allpass filters can have only positive group delay and a non-zero value of `tau` accounts for any additional group delay that is needed to meet the shape of the contour specified by `(f,a)`. The default for `tau` is `max(a)`.

Hint: If the zeros or poles cluster together, your filter order may be too low or the pole radius may be too small (overly constrained). Try increasing `n` or `radius`.

For group-delay equalization of an IIR filter, compute `a` by subtracting the filter's group delay from its maximum group delay. For example,

```
[be,ae] = ellip(4,1,40,0.2);
f = 0:0.001:0.2;
g = grpdelay(be,ae,f,2); % Equalize only the passband.
a = max(g)-g;
[num,den]=iirgrpdelay(8, f, [0 0.2], a);
```

## Version History

Introduced in R2011a

## References

Antoniou, A., *Digital Filters: Analysis, Design, and Applications*, Second Edition, McGraw-Hill, Inc. 1993.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- All inputs must be constant. Expressions or variables are allowed if their values do not change.
- Does not support syntaxes that have cell array input.

## See Also

`freqz` | `filter` | `grpdelay` | `iirlpnorm` | `iirlpnormc` | `zplane`

## iirlinphase

Quasi-linear phase IIR filter from halfband filter specification

### Syntax

```
iirlinFilter= design(d,'iirlinphase','SystemObject',true)
hd = design(...,'filterstructure',structure,'SystemObject',true)
```

### Description

`iirlinFilter= design(d,'iirlinphase','SystemObject',true)` designs a quasi-linear phase filter `iirlinFilter` specified by the filter specification object `d`.

`hd = design(...,'filterstructure',structure,'SystemObject',true)` returns a filter with the structure specified by `structure`. By default, the filter structure is a cascade structure. The following table lists all the structures `design` supports for the IIR linear phase response.

| Structure                      | Filter Structure                               |
|--------------------------------|--|
| <code>cascadeallpass</code>    | Cascade of allpass filters                     |
| <code>cascadewdfallpass</code> | Cascade of allpass wave digital filters        |
| <code>iirdecim</code>          | IIR polyphase decimator                        |
| <code>iirwdfdecim</code>       | IIR wave digital filter polyphase decimator    |
| <code>iirinterp</code>         | IIR polyphase interpolator                     |
| <code>iirwdfinterp</code>      | IIR wave digital filter polyphase interpolator |

To get a list of all the structures the specific `fdesign` method supports, type the following in the MATLAB command prompt.

```
d = fdesign.halfband;
strucs = validstructures(d,'SystemObject',true);
```

To get a list of structures `iirlinphase` supports, type the following in the MATLAB command prompt.

```
iirlinphaseStrucs = strucs.iirlinphase;
```

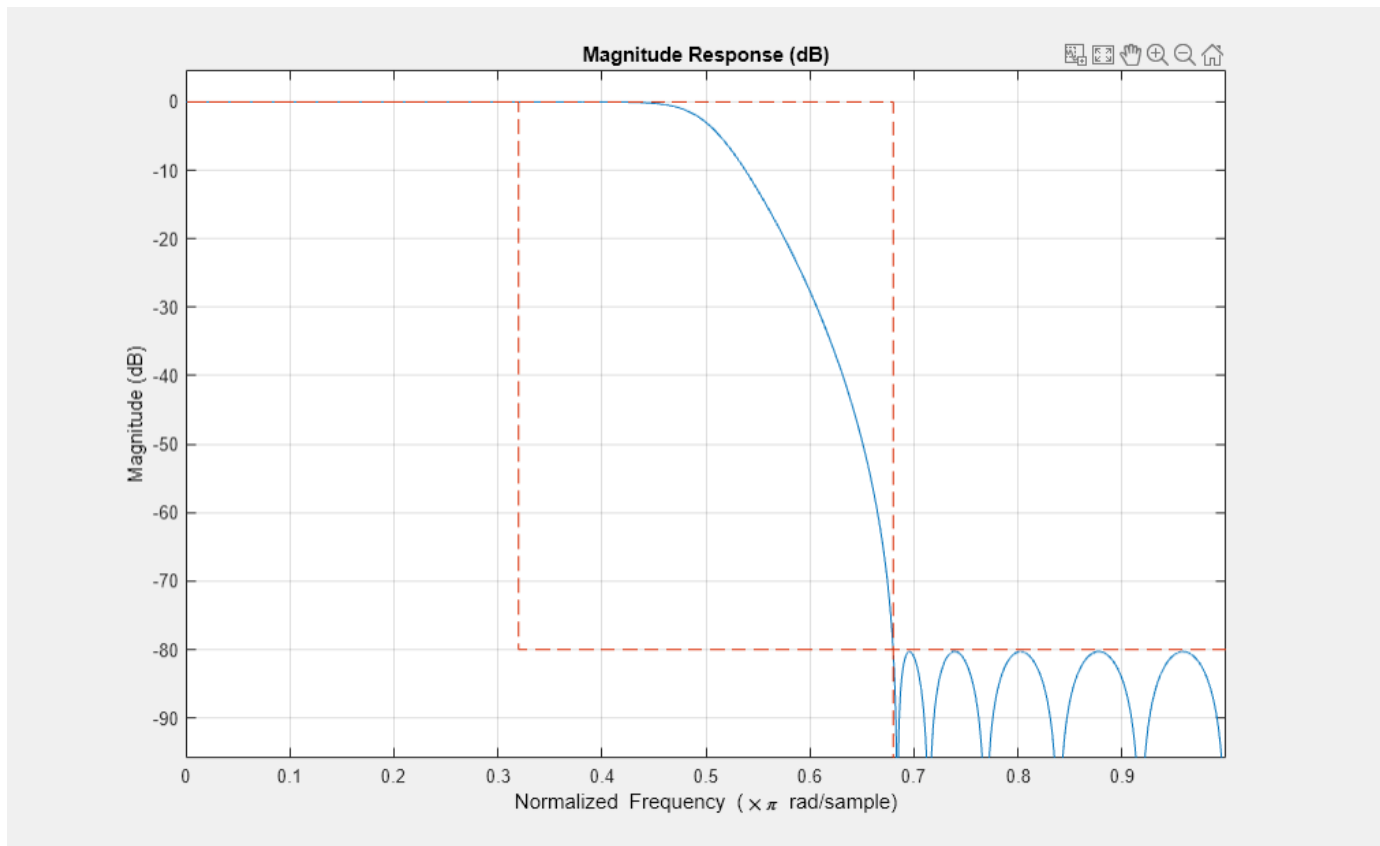
### Examples

#### Design a Minimum-order Halfband IIR Filter

Design a quasi-linear phase, minimum-order halfband IIR filter with transition width of 0.36 and stopband attenuation of at least 80 dB.

```
tw = 0.36;
ast = 80;
d = fdesign.halfband('tw,ast',tw,ast); % Transition width,
                                     % stopband attenuation.
```

```
halfbandIIR = design(d,'iirlinphase','SystemObject',true);
fvtool(halfbandIIR)
```



Notice the characteristic halfband nature of the ripple in the stopband. If you measure the resulting filter, you see it meets the specifications.

```
measure(halfbandIIR)
```

```
ans =
Sample Rate      : N/A (normalized frequency)
Passband Edge    : 0.32
3-dB Point       : 0.5
6-dB Point       : 0.51911
Stopband Edge    : 0.68
Passband Ripple  : 4.0866e-08 dB
Stopband Atten.  : 80.2642 dB
Transition Width  : 0.36
```

## Version History

Introduced in R2011a

**See Also**

`fdesign.halfband`

## iirlp2bp

Transform IIR lowpass filter to IIR bandpass filter

### Syntax

```
[num,den,allpassNum,allpassDen] = iirlp2bp(b,a,wo,wt)
```

### Description

`[num,den,allpassNum,allpassDen] = iirlp2bp(b,a,wo,wt)` transforms an IIR lowpass filter to an IIR bandpass filter.

The `iirlp2bp` function returns the numerator and denominator coefficients of the transformed bandpass filter. The function transforms the real lowpass prototype filter, specified by the numerator coefficients `b` and denominator coefficients `a`, by applying a second-order real lowpass to real bandpass frequency mapping. For more information, see “IIR Lowpass to IIR Bandpass Transformation” on page 5-933.

The function also returns the numerator and the denominator coefficients of the allpass mapping filter, `allpassNum` and `allpassDen` respectively.

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

### Examples

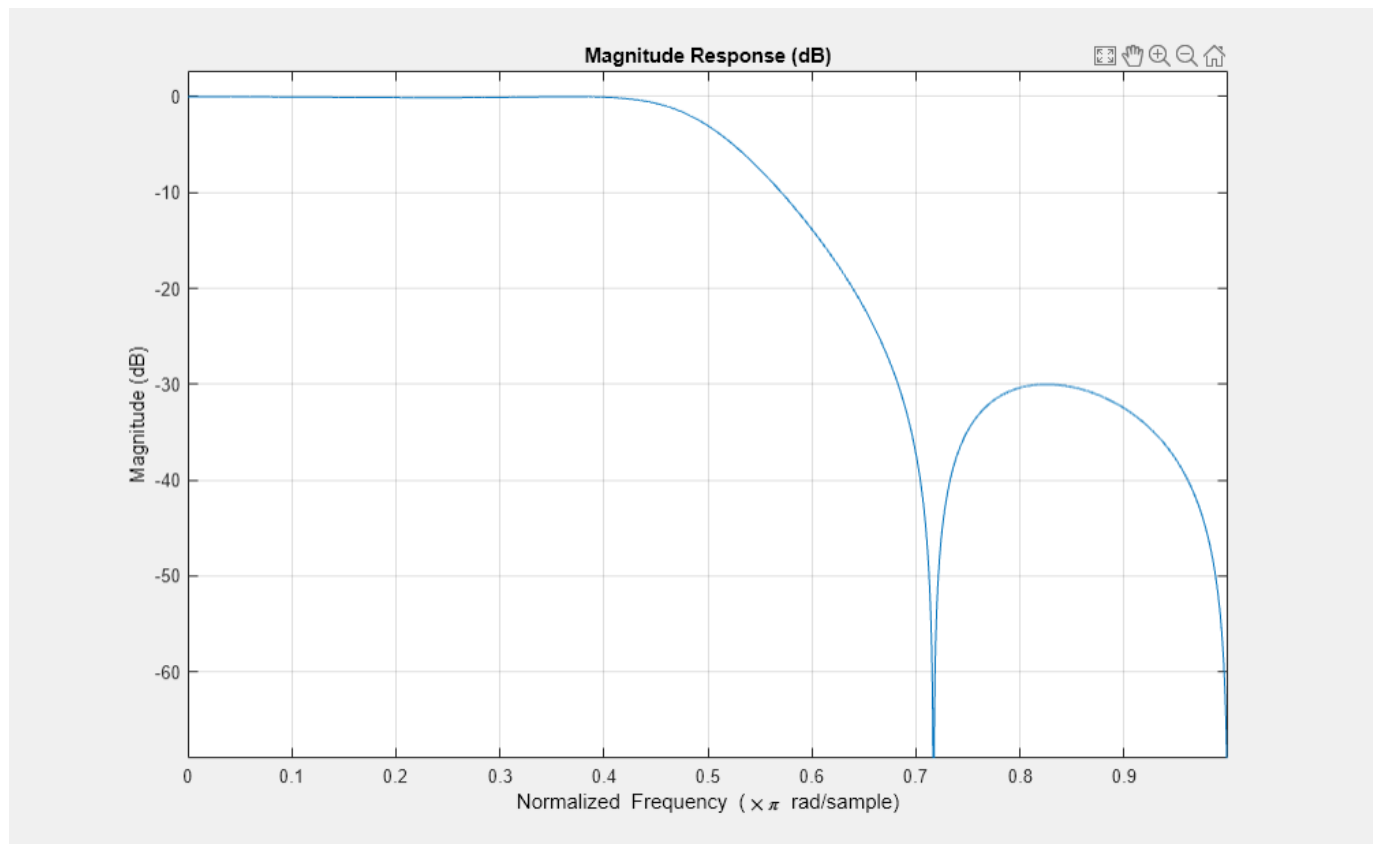
#### Transform Lowpass Filter to Bandpass Filter

Transform a lowpass IIR filter to a bandpass filter using the `iirlp2bp` function.

#### Input Lowpass IIR Filter

Design a prototype real IIR lowpass elliptic filter using the `ellip` function. The filter has a gain of about -3 dB at  $0.5\pi$  rad/sample.

```
[b,a] = ellip(3,0.1,30,0.409);  
fvtool(b,a)
```



### Transform Filter Using `iirlp2bp`

Transform the prototype lowpass filter into a bandpass filter by placing the cutoff frequencies of the prototype filter at  $0.25\pi$  and  $0.75\pi$ .

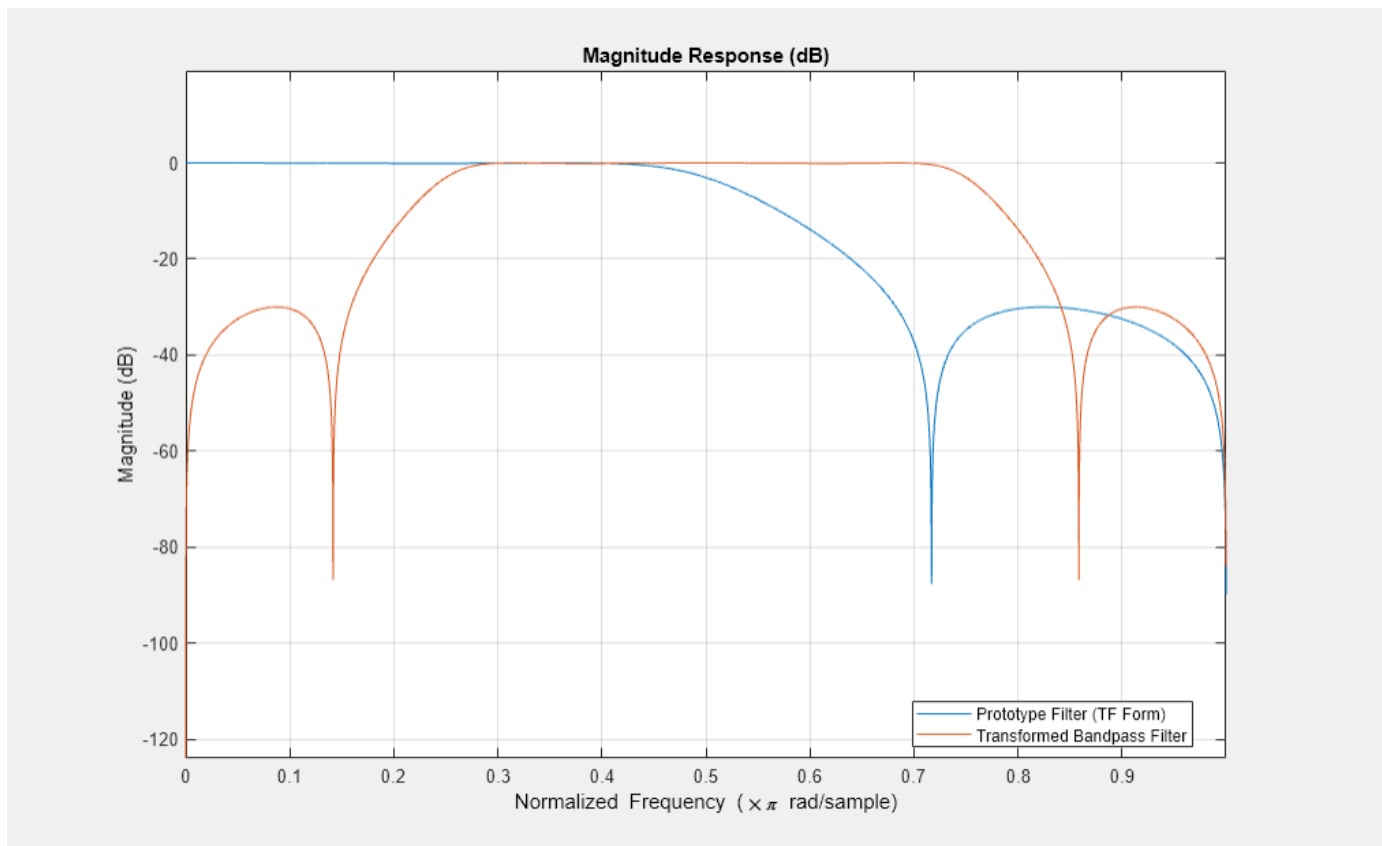
Specify the prototype filter as a vector of numerator and denominator coefficients, `b` and `a` respectively.

```
[num,den] = iirlp2bp(b,a,0.5,[0.25 0.75]);
```

Compare the magnitude response of the filters using `FVTool`.

```
hvft = fvtool(b,a,num,den);
legend(hvft,"Prototype Filter (TF Form)","Transformed Bandpass Filter")
```





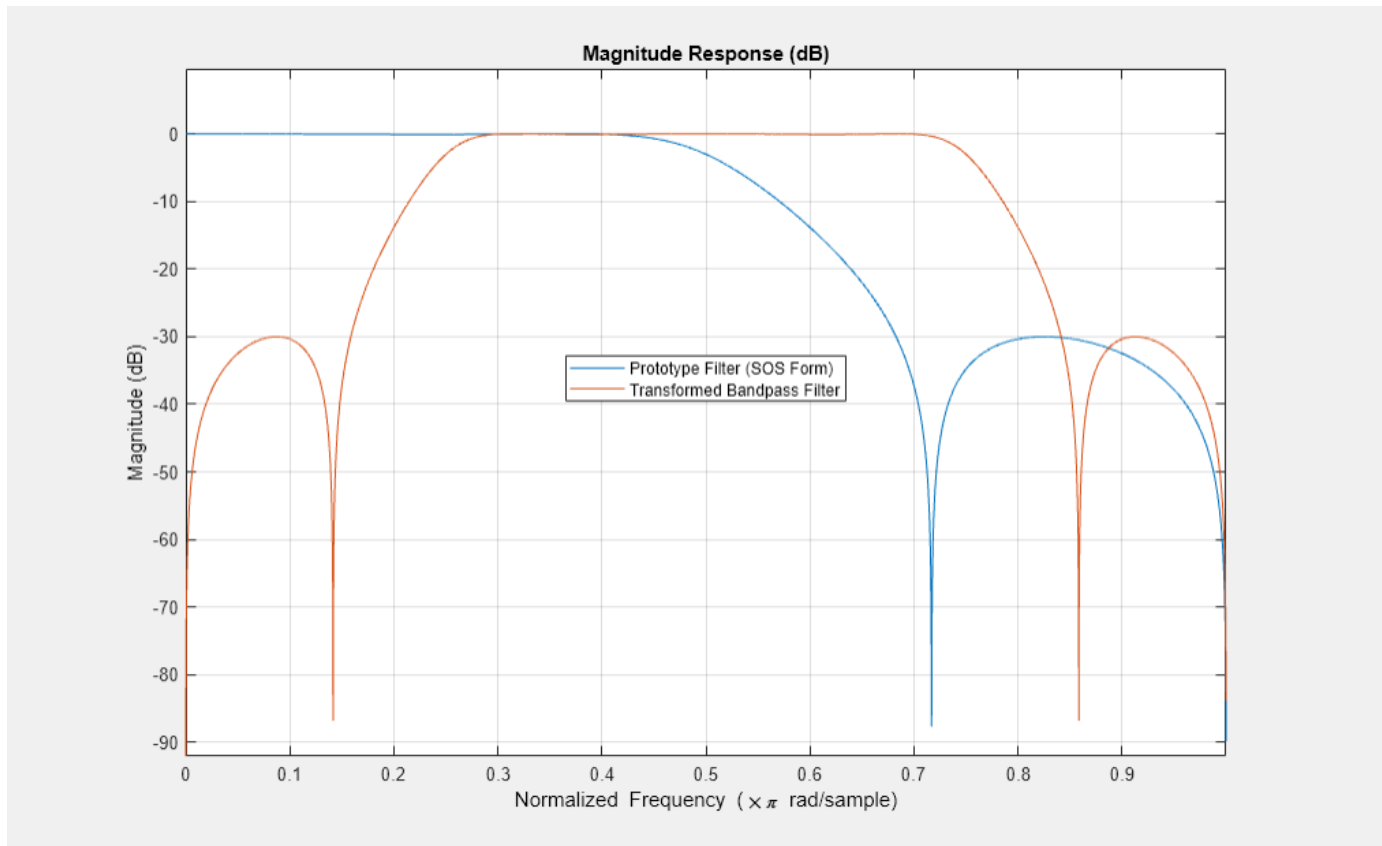
You can also specify the input lowpass IIR filter as a matrix of coefficients. Pass the second-order section matrices as inputs. The transformed filter is a fourth-order section filter given by the numerator and the denominator coefficients `num2` and `den2`.

```
ss = tf2sos(b,a);
[num2,den2] = iirlp2bp(ss(:,1:3),ss(:,4:6),0.5,[0.25 0.75]);
```

Compare the magnitude response of the filters using `FVTool`.

To visualize the magnitude response of the fourth-order section filter, first pass the filter coefficients to the `dsp.FourthOrderSectionFilter` object. Then, use this object as an input to the `FVTool`.

```
fos = dsp.FourthOrderSectionFilter(Numerator=num2,...
    Denominator=den2);
hvft = fvtool(ss, fos);
legend(hvft, "Prototype Filter (SOS Form)", ...
    "Transformed Bandpass Filter")
```



Copyright 2012-2021 The MathWorks, Inc.

## Input Arguments

### **b** — Numerator coefficients of prototype lowpass IIR filter

row vector | matrix

Numerator coefficients of the prototype lowpass IIR filter, specified as either:

- Row vector -- Specifies the values of  $[b_0, b_1, \dots, b_n]$ , given this transfer function form:

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_0 + b_1z^{-1} + \dots + b_nz^{-n}}{a_0 + a_1z^{-1} + \dots + a_nz^{-n}}$$

where  $n$  is the order of the filter.

- Matrix -- Specifies the numerator coefficients in the form of an  $P$ -by- $(Q+1)$  matrix, where  $P$  is the number of filter sections and  $Q$  is the order of each filter section. If  $Q = 2$ , the filter is a second-order section filter. For higher-order sections, make  $Q > 2$ .

$$\mathbf{b} = \begin{bmatrix} b_{01} & b_{11} & b_{21} & \dots & b_{Q1} \\ b_{02} & b_{12} & b_{22} & \dots & b_{Q2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ b_{0P} & b_{1P} & b_{2P} & \dots & b_{QP} \end{bmatrix}$$

In the transfer function form, the numerator coefficient matrix  $b_{ik}$  of the IIR filter can be represented using the following equation:

$$H(z) = \prod_{k=1}^P H_k(z) = \prod_{k=1}^P \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2} + \dots + b_{Qk}z^{-Q}}{a_{0k} + a_{1k}z^{-1} + a_{2k}z^{-2} + \dots + a_{Qk}z^{-Q}},$$

where,

- **a** -- Denominator coefficients matrix. For more information on how to specify this matrix, see [a](#).
- $k$  -- Row index.
- $i$  -- Column index.

When specified in the matrix form, **b** and **a** matrices must have the same number of rows (filter sections)  $Q$ .

Data Types: `single` | `double`

### **a** — Denominator coefficients of prototype lowpass IIR filter

row vector | matrix

Denominator coefficients for a prototype lowpass IIR filter, specified as one of these options:

- Row vector -- Specifies the values of  $[a_0, a_1, \dots, a_n]$ , given this transfer function form:

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_0 + b_1z^{-1} + \dots + b_nz^{-n}}{a_0 + a_1z^{-1} + \dots + a_nz^{-n}},$$

where  $n$  is the order of the filter.

- Matrix -- Specifies the denominator coefficients in the form of an  $P$ -by- $(Q+1)$  matrix, where  $P$  is the number of filter sections and  $Q$  is the order of each filter section. If  $Q = 2$ , the filter is a second-order section filter. For higher-order sections, make  $Q > 2$ .

$$\mathbf{a} = \begin{bmatrix} a_{01} & a_{11} & a_{21} & \dots & a_{Q1} \\ a_{02} & a_{12} & a_{22} & \dots & a_{Q2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{0P} & a_{1P} & a_{2P} & \dots & a_{QP} \end{bmatrix}$$

In the transfer function form, the denominator coefficient matrix  $a_{ik}$  of the IIR filter can be represented using the following equation:

$$H(z) = \prod_{k=1}^P H_k(z) = \prod_{k=1}^P \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2} + \dots + b_{Qk}z^{-Q}}{a_{0k} + a_{1k}z^{-1} + a_{2k}z^{-2} + \dots + a_{Qk}z^{-Q}},$$

where,

- **b** -- Numerator coefficients matrix. For more information on how to specify this matrix, see [b](#).
- $k$  -- Row index.
- $i$  -- Column index.

When specified in the matrix form, **a** and **b** matrices must have the same number of rows (filter sections)  $P$ .

Data Types: `single` | `double`

**wo** — **Frequency value to be transformed from prototype filter**

scalar

Frequency value to transform from the prototype filter, specified as a scalar. Frequency **wo** must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

Data Types: `single` | `double`

**wt** — **Desired frequency locations in transformed target filter**

two-element vector

Desired frequency locations in the transformed target filter, specified as a two-element vector. Frequencies in **wt** must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

Data Types: `single` | `double`

## Output Arguments

**num** — **Numerator coefficients of transformed bandpass filter**

row vector | matrix

Numerator coefficients of the transformed bandpass filter, returned as one of the following:

- Row vector of length  $2n+1$ , where  $n$  is the order of the input filter. The **num** output is a row vector when the input coefficients **b** and **a** are row vectors.
- $P$ -by- $(2Q+1)$  matrix, where  $P$  is the number of filter sections and  $Q$  is the order of each section of the transformed filter. The **num** output is a matrix when the input coefficients **b** and **a** are matrices.

Data Types: `single` | `double`

**den** — **Denominator coefficients of transformed bandpass filter**

row vector | matrix

Denominator coefficients of the transformed bandpass filter, returned as one of the following:

- Row vector of length  $2n+1$ , where  $n$  is the order of the input filter. The **den** output is a row vector when the input coefficients **b** and **a** are row vectors.
- $P$ -by- $(2Q+1)$  matrix, where  $P$  is the number of filter sections and  $Q$  is the order of each section of the transformed filter. The **den** output is a matrix when the input coefficients **b** and **a** are matrices.

Data Types: `single` | `double`

**allpassNum** — **Numerator coefficients of mapping filter**

row vector

Numerator coefficients of the mapping filter, returned as a row vector.

Data Types: `single` | `double`

**allpassDen — Denominator coefficients of mapping filter**

row vector

Denominator coefficients of the mapping filter, returned as a row vector.

Data Types: `single` | `double`

**More About****IIR Lowpass to IIR Bandpass Transformation**

IIR lowpass to IIR bandpass transformation effectively places one feature of the original filter, located at frequency  $-w_0$ , at the required target frequency location  $w_{t1}$ , and the second feature, originally at  $+w_0$ , at the new location  $w_{t2}$ . It is assumed that  $w_{t2}$  is greater than  $w_{t1}$ . This transformation implements the “*DC Mobility*,” meaning that the Nyquist feature stays at Nyquist, but the DC feature moves to a location dependent on the selection of  $w_{ts}$ .

Relative positions of other features of the original filter do not change in the target filter. This means that it is possible to select two features of an original filter,  $F_1$  and  $F_2$ , with  $F_1$  preceding  $F_2$ . Feature  $F_1$  will still precede  $F_2$  after the transformation. However, the distance between  $F_1$  and  $F_2$  will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to bandpass transformation is not restricted only to the cutoff frequency of an original lowpass filter. You can choose to transform any feature of the original filter like stopband edge, DC, deep minimum in the stopband, or others.

Real lowpass to bandpass transformation can also be used to transform other types of filters, for example, real notch filters or resonators can be doubled and positioned at two distinct desired frequencies.

**Version History**

Introduced in R2011a

**References**

- [1] Nowrouzian, B., and A.G. Constantinides. “Prototype Reference Transfer Function Parameters in the Discrete-Time Frequency Transformations.” In *Proceedings of the 33rd Midwest Symposium on Circuits and Systems*, 1078-82. Calgary, Alta., Canada: IEEE, 1991. <https://doi.org/10.1109/MWSCAS.1990.140912>.
- [2] Nowrouzian, B., and L.T. Bruton. “Closed-Form Solutions for Discrete-Time Elliptic Transfer Functions.” In [1992] *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, 784-87. Washington, DC, USA: IEEE, 1992. <https://doi.org/10.1109/MWSCAS.1992.271206>.
- [3] Constantinides, A.G. “Design of Bandpass Digital Filters.” *Proceedings of the IEEE* 57, no. 6 (1969): 1229-31. <https://doi.org/10.1109/PROC.1969.7216>.
- [4] Constantinides, A.G. “Spectral transformations for digital filters.” *Proceedings of the IEEE*, vol. 117, no. 8: 1585-1590. August 1970.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

## **See Also**

### **Functions**

`iirftransf` | `allpasslp2bp` | `zpklp2bp`

## iirlp2bpc

Transform IIR lowpass filter to complex bandpass filter

### Syntax

```
[num,den,allpassNum,allpassDen] = iirlp2bpc(b,a,wo,wt)
```

### Description

`[num,den,allpassNum,allpassDen] = iirlp2bpc(b,a,wo,wt)` transforms an IIR lowpass filter to a complex bandpass filter.

The function transforms a real lowpass prototype filter, specified as the numerator and denominator coefficients `b` and `a` respectively, to a complex bandpass filter by applying a first-order real lowpass to complex bandpass frequency transformation.

The function returns the numerator and denominator coefficients of the transformed complex bandpass filter. The function also returns the numerator and denominator coefficients of the allpass mapping filter, `allpassNum` and `allpassDen` respectively.

For more details on the transformation, see “IIR Lowpass to Complex Bandpass Transformation” on page 5-941.

### Examples

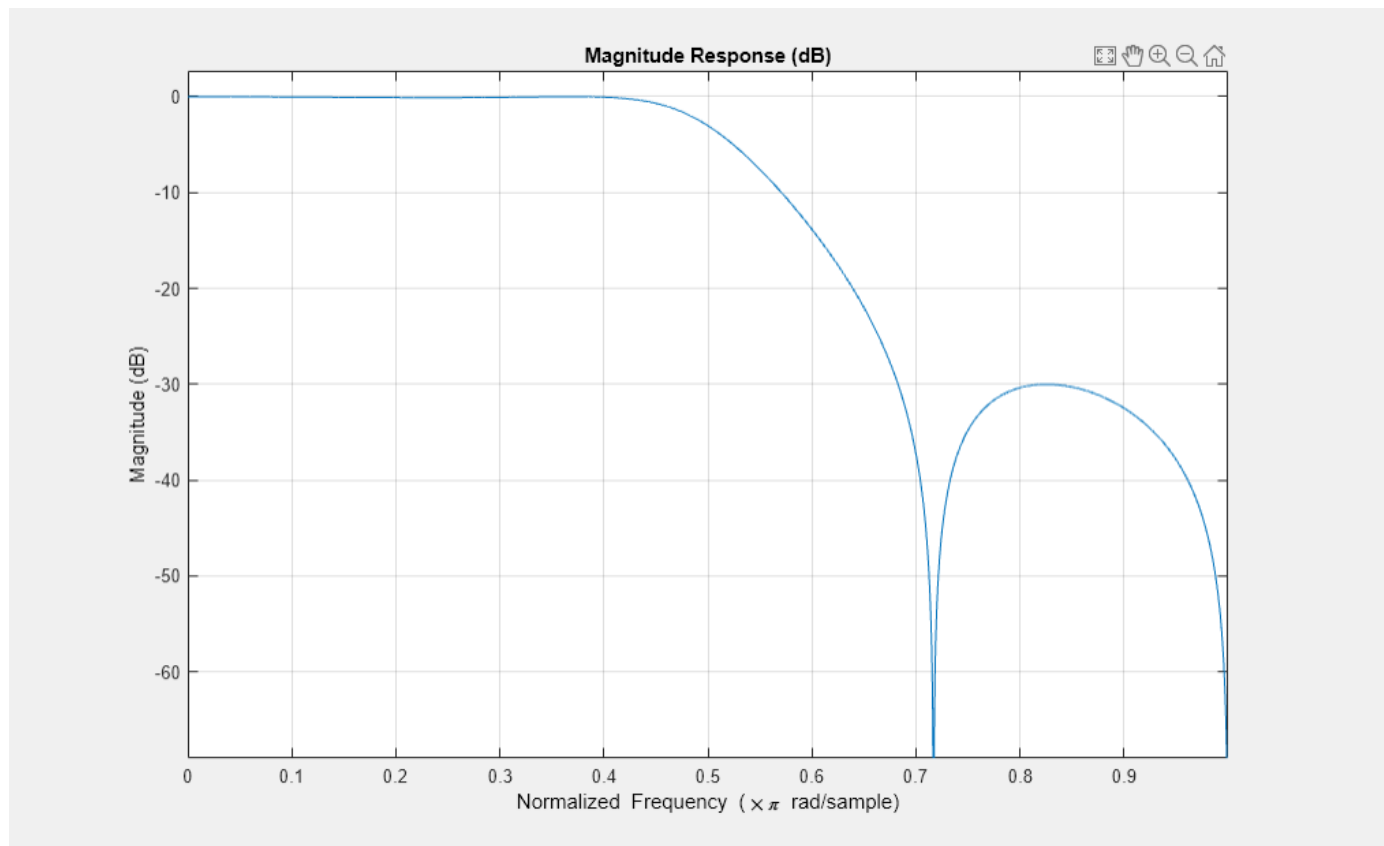
#### Transform Lowpass Filter to Complex Bandpass Filter

Transform a lowpass IIR filter to a complex bandpass filter using the `iirlp2bpc` function.

#### Input Lowpass IIR Filter

Design a prototype real IIR lowpass elliptic filter with a gain of about -3 dB at  $0.5\pi$  rad/sample.

```
[b,a] = ellip(3,0.1,30,0.409);  
fvtool(b,a)
```



### Transform Filter Using `iirlp2bpc`

Transform the prototype lowpass filter into a complex bandpass filter by placing the cutoff frequencies of the prototype filter at  $0.25\pi$  and  $0.75\pi$ .

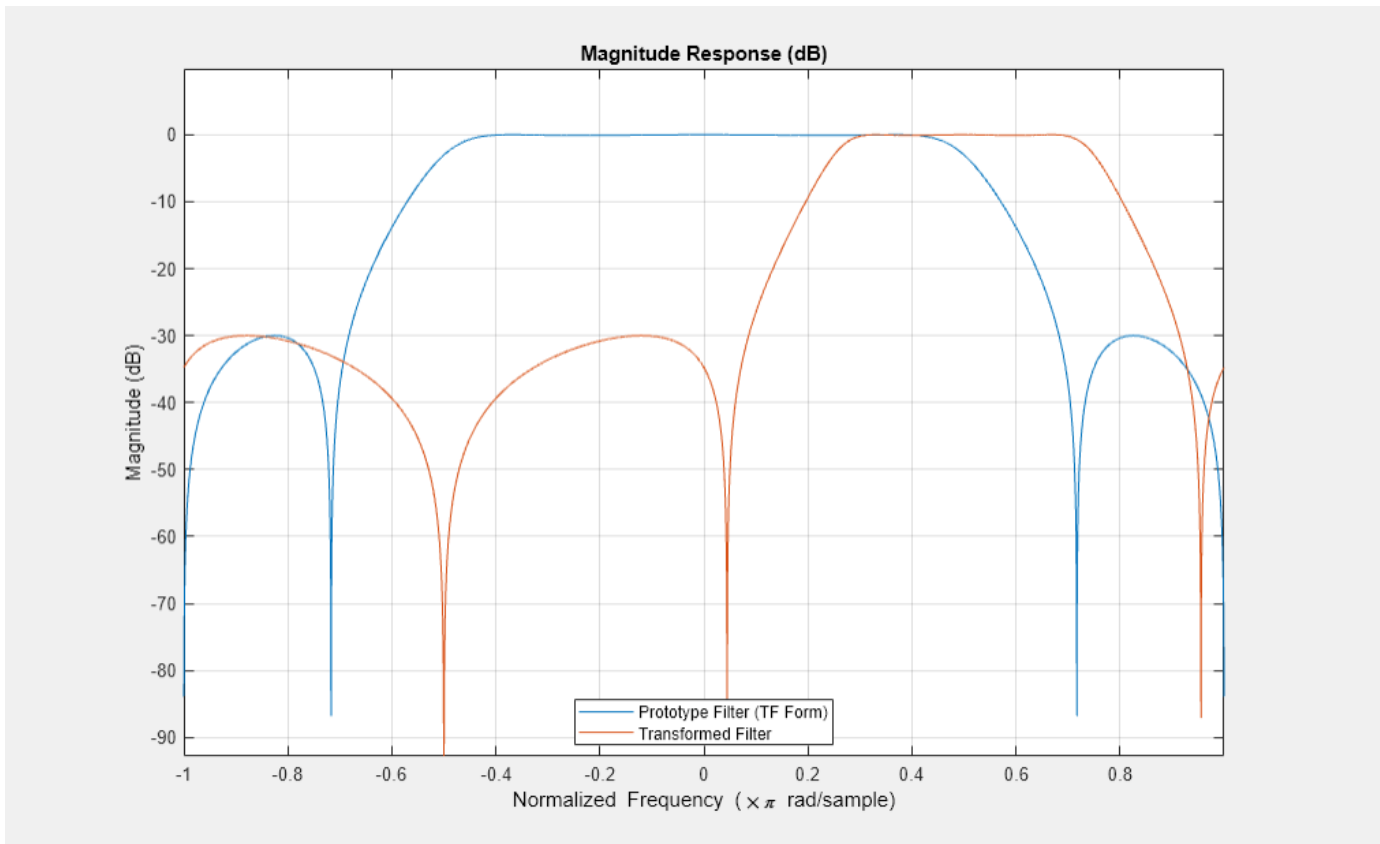
Specify the prototype filter as a vector of numerator and denominator coefficients, `b` and `a` respectively.

```
[num,den] = iirlp2bpc(b,a,0.5,[0.25 0.75]);
```

Compare the magnitude response of the filters using `FVTool`.

```
hvft = fvtool(b,a,num,den);
legend(hvft,"Prototype Filter (TF Form)",...
       "Transformed Filter")
```



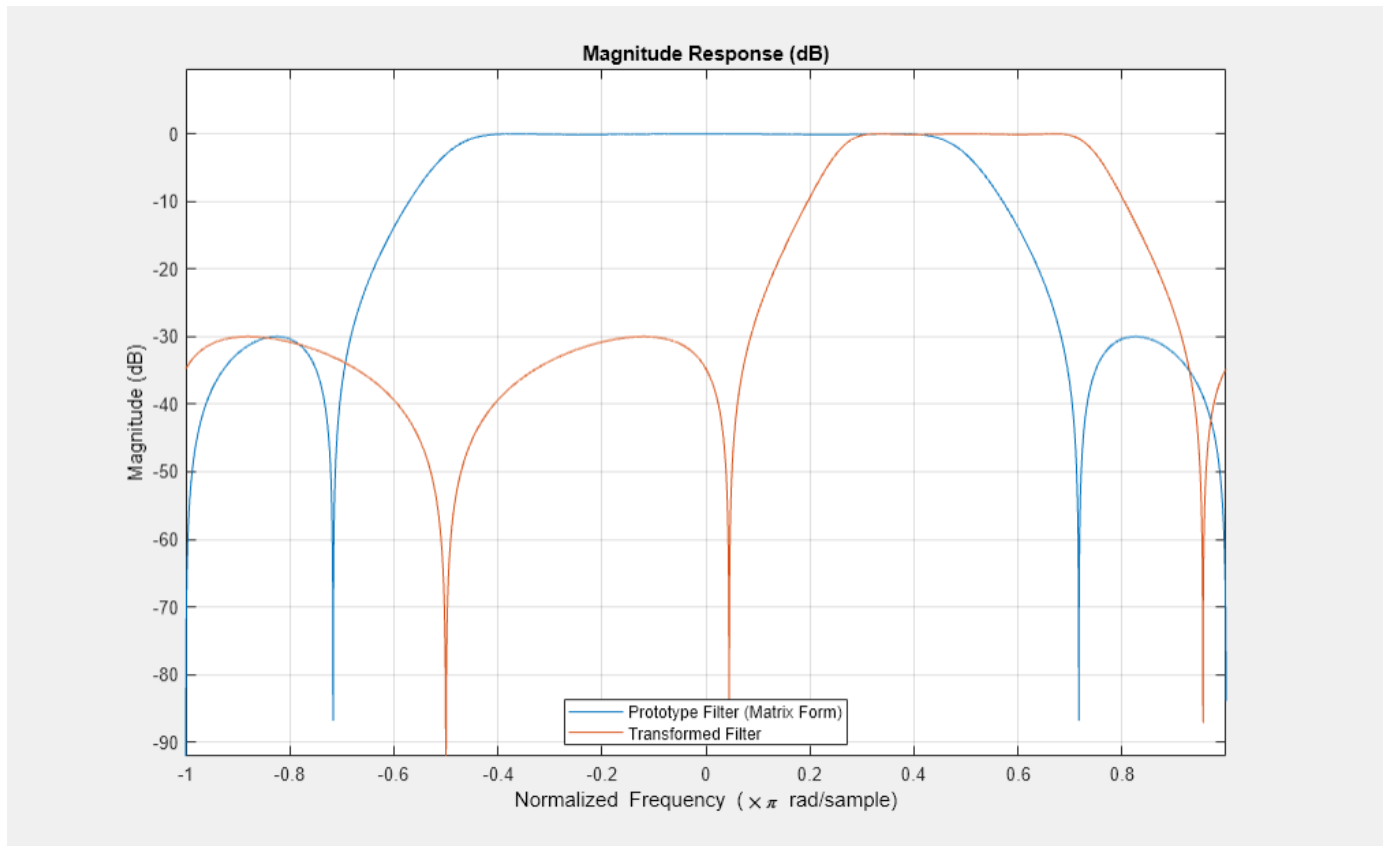


Alternatively, you can also specify the input lowpass IIR filter as a matrix of coefficients. Pass the second order section coefficient matrices as inputs.

```
ss = tf2sos(b,a);
[num2,den2] = iirlp2bpc(ss(:,1:3),ss(:,4:6),0.5,[0.25 0.75]);
```

Compare the magnitude response of the filters using FVTool.

```
hvft = fvtool(ss,[num2 den2]);
legend(hvft,"Prototype Filter (Matrix Form)", ...
       "Transformed Filter")
```



Copyright 2012-2021 The MathWorks, Inc.

## Input Arguments

### **b** — Numerator coefficients of prototype lowpass IIR filter

row vector | matrix

Numerator coefficients of the prototype lowpass IIR filter, specified as either:

- Row vector -- Specifies the values of  $[b_0, b_1, \dots, b_n]$ , given this transfer function form:

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_0 + b_1z^{-1} + \dots + b_nz^{-n}}{a_0 + a_1z^{-1} + \dots + a_nz^{-n}}$$

where  $n$  is the order of the filter.

- Matrix -- Specifies the numerator coefficients in the form of an  $P$ -by- $(Q+1)$  matrix, where  $P$  is the number of filter sections and  $Q$  is the order of each filter section. If  $Q = 2$ , the filter is a second-order section filter. For higher-order sections, make  $Q > 2$ .

$$\mathbf{b} = \begin{bmatrix} b_{01} & b_{11} & b_{21} & \dots & b_{Q1} \\ b_{02} & b_{12} & b_{22} & \dots & b_{Q2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ b_{0P} & b_{1P} & b_{2P} & \dots & b_{QP} \end{bmatrix}$$

In the transfer function form, the numerator coefficient matrix  $b_{ik}$  of the IIR filter can be represented using the following equation:

$$H(z) = \prod_{k=1}^P H_k(z) = \prod_{k=1}^P \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2} + \dots + b_{Qk}z^{-Q}}{a_{0k} + a_{1k}z^{-1} + a_{2k}z^{-2} + \dots + a_{Qk}z^{-Q}},$$

where,

- **a** -- Denominator coefficients matrix. For more information on how to specify this matrix, see [a](#).
- $k$  -- Row index.
- $i$  -- Column index.

When specified in the matrix form, **b** and **a** matrices must have the same number of rows (filter sections)  $Q$ .

Data Types: `single` | `double`

### **a** — Denominator coefficients of prototype lowpass IIR filter

row vector | matrix

Denominator coefficients for a prototype lowpass IIR filter, specified as one of these options:

- Row vector -- Specifies the values of  $[a_0, a_1, \dots, a_n]$ , given this transfer function form:

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_0 + b_1z^{-1} + \dots + b_nz^{-n}}{a_0 + a_1z^{-1} + \dots + a_nz^{-n}},$$

where  $n$  is the order of the filter.

- Matrix -- Specifies the denominator coefficients in the form of an  $P$ -by- $(Q+1)$  matrix, where  $P$  is the number of filter sections and  $Q$  is the order of each filter section. If  $Q = 2$ , the filter is a second-order section filter. For higher-order sections, make  $Q > 2$ .

$$\mathbf{a} = \begin{bmatrix} a_{01} & a_{11} & a_{21} & \dots & a_{Q1} \\ a_{02} & a_{12} & a_{22} & \dots & a_{Q2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{0P} & a_{1P} & a_{2P} & \dots & a_{QP} \end{bmatrix}$$

In the transfer function form, the denominator coefficient matrix  $a_{ik}$  of the IIR filter can be represented using the following equation:

$$H(z) = \prod_{k=1}^P H_k(z) = \prod_{k=1}^P \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2} + \dots + b_{Qk}z^{-Q}}{a_{0k} + a_{1k}z^{-1} + a_{2k}z^{-2} + \dots + a_{Qk}z^{-Q}},$$

where,

- **b** -- Numerator coefficients matrix. For more information on how to specify this matrix, see [b](#).
- $k$  -- Row index.
- $i$  -- Column index.

When specified in the matrix form, **a** and **b** matrices must have the same number of rows (filter sections)  $P$ .

Data Types: `single` | `double`

**wo — Frequency value to transform from prototype filter**

scalar

Frequency value to transform from the prototype filter, specified as a real scalar. Frequency  $w_0$  should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

Data Types: `single` | `double`

**wt — Desired frequency locations in transformed target filter**

two-element vector

Desired frequency locations in the transformed target filter, specified as a two-element vector. Frequencies in  $w_t$  should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

Data Types: `single` | `double`

## Output Arguments

**num — Numerator coefficients of transformed complex bandpass filter**

row vector | matrix

Numerator coefficients of the transformed complex bandpass filter, returned as one of the following:

- Row vector of length  $n+1$ , where  $n$  is the order of the input filter. The `num` output is a row vector when the input coefficients `b` and `a` are row vectors.
- $P$ -by- $(Q+1)$  matrix, where  $P$  is the number of filter sections and  $Q$  is the order of each section of the transformed filter. The `num` output is a matrix when the input coefficients `b` and `a` are matrices.

Data Types: `single` | `double`

**den — Denominator coefficients of transformed complex bandpass filter**

row vector | matrix

Denominator coefficients of the transformed complex bandpass filter, returned as one of the following:

- Row vector of length  $n+1$ , where  $n$  is the order of the input filter. The `den` output is a row vector when the input coefficients `b` and `a` are row vectors.
- $P$ -by- $(Q+1)$  matrix, where  $P$  is the number of filter sections and  $Q$  is the order of each section of the transformed filter. The `den` output is a matrix when the input coefficients `b` and `a` are matrices.

Data Types: `single` | `double`

**allpassNum — Numerator coefficients of mapping filter**

row vector

Numerator of the mapping filter, returned as a row vector.

Data Types: single | double

### **allpassDen — Denominator coefficients of mapping filter**

row vector

Denominator of the mapping filter, returned as a row vector.

Data Types: single | double

Complex Number Support: Yes

## **More About**

### **IIR Lowpass to Complex Bandpass Transformation**

IIR lowpass to complex bandpass transformation effectively places one feature of the original filter, located at frequency  $-w_o$ , at the required target frequency location,  $w_{t1}$ , and the second feature, originally at  $w_o$ , at the new location,  $w_{t2}$ . It is assumed that  $w_{t2}$  is greater than  $w_{t1}$ .

Relative positions of other features of the original filter do not change in the target filter. This means that it is possible to select two features of an original filter,  $F_1$  and  $F_2$ , with  $F_1$  preceding  $F_2$ . Feature  $F_1$  will still precede  $F_2$  after the transformation. However, the distance between  $F_1$  and  $F_2$  will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to bandpass transformation is not restricted only to the cutoff frequency of an original lowpass filter. You can choose to transform any feature of the original filter like stopband edge, DC, deep minimum in the stopband, or others.

Lowpass to bandpass transformation can also be used to transform other types of filters, for example real notch filters or resonators can be doubled and positioned at two distinct desired frequencies at any place around the unit circle, forming a pair of complex notches or resonators. You can use this transformation to design bandpass filters for radio receivers from the high-quality prototype lowpass filter.

## **Version History**

Introduced in R2011a

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

## **See Also**

### **Functions**

iirfftransf | allpasslp2bpc | zpklp2bpc

## iirlp2bs

Transform IIR lowpass filter to IIR bandstop filter

### Syntax

```
[num,den,allpassNum,allpassDen] = iirlp2bs(b,a,wo,wt)
```

### Description

`[num,den,allpassNum,allpassDen] = iirlp2bs(b,a,wo,wt)` transforms an IIR lowpass filter to an IIR bandstop filter.

The lowpass filter is specified using the numerator and denominator coefficients `b` and `a` respectively. The function returns the numerator and denominator coefficients of the transformed bandstop digital filter. The function also returns the numerator and denominator coefficients of the allpass mapping filter, `allpassNum` and `allpassDen` respectively.

For more details on the transformation, see “IIR Lowpass Filter to IIR Bandstop Filter Transformation” on page 5-948.

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

### Examples

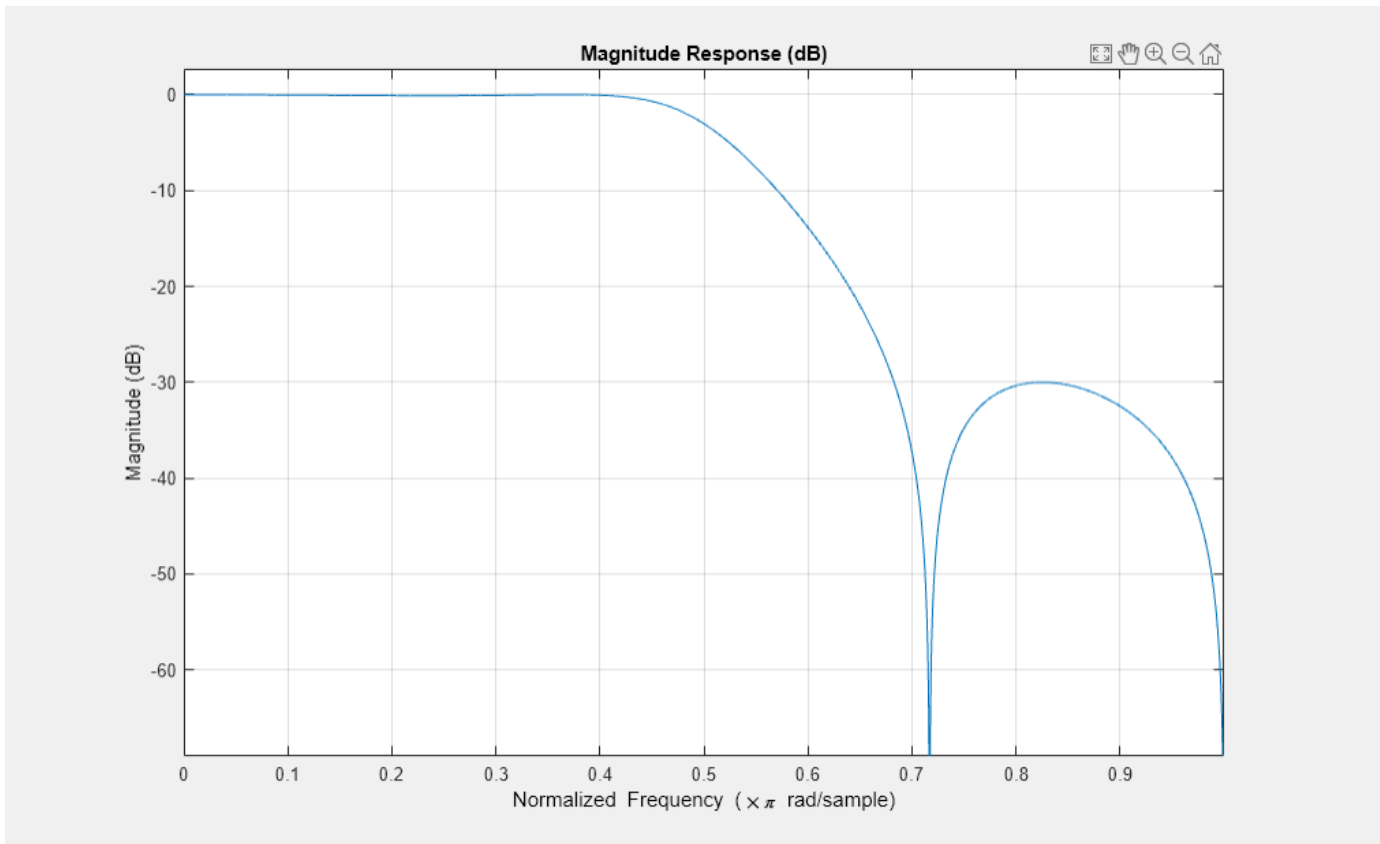
#### Transform Lowpass Filter to Bandstop Filter

Transform a lowpass IIR filter to a bandstop filter using the `iirlp2bs` function.

#### Input Lowpass IIR Filter

Design a prototype real IIR lowpass elliptic filter with a gain of about -3 dB at  $0.5\pi$  rad/sample.

```
[b,a] = ellip(3,0.1,30,0.409);  
fvtool(b,a)
```



### Transform Filter Using `iirlp2bs`

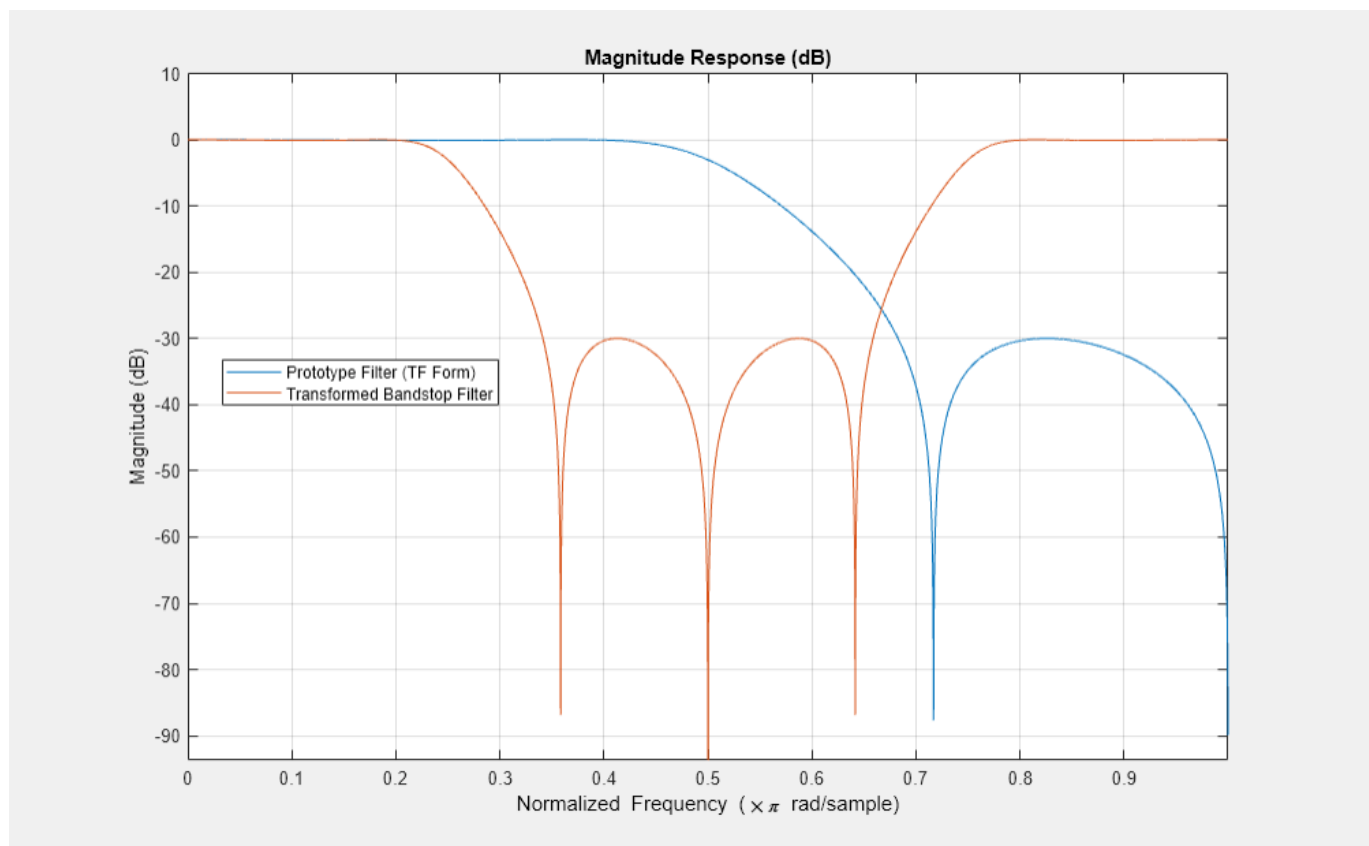
Transform the prototype lowpass filter into a bandstop filter by placing the cutoff frequencies of the prototype filter at  $0.25\pi$  and  $0.75\pi$ .

Specify the prototype filter as a vector of numerator and denominator coefficients, `b` and `a` respectively.

```
[num,den] = iirlp2bs(b,a,0.5,[0.25 0.75]);
```

Compare the magnitude response of the filters using `FVTool`.

```
hfvt = fvtool(b,a,num,den);
legend(hfvt,"Prototype Filter (TF Form)",...
       "Transformed Bandstop Filter")
```



Alternatively, you can also specify the input lowpass IIR filter as a matrix of coefficients. Pass the second-order section matrices as inputs. The transformed filter is a fourth-order section filter given by the numerator and the denominator coefficients `num2` and `den2`.

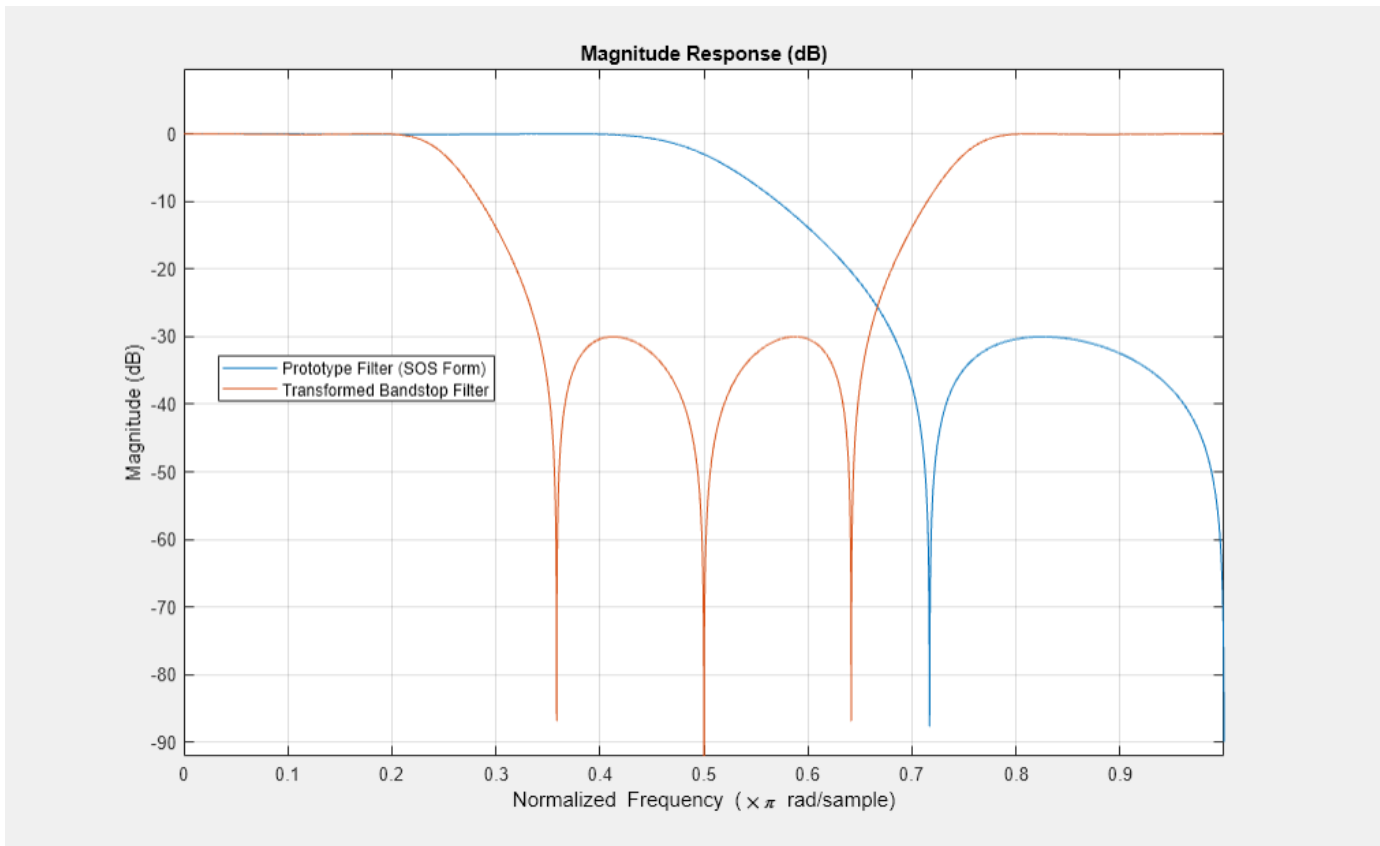
```
ss = tf2sos(b,a);
[num2,den2] = iirlp2bs(ss(:,1:3),ss(:,4:6),0.5,[0.25 0.75]);
```

Compare the magnitude response of the filters using `FVTool`.

To visualize the magnitude response of the fourth-order section filter, first pass the filter coefficients to the `dsp.FourthOrderSectionFilter` object. Then, use this object as an input to the `FVTool`.

```
fos = dsp.FourthOrderSectionFilter(Numerator=num2,...
    Denominator=den2);
hvft = fvtool(ss,fos);
legend(hvft,"Prototype Filter (SOS Form)",...
    "Transformed Bandstop Filter")
```





Copyright 2012-2021 The MathWorks, Inc.

## Input Arguments

### **b** — Numerator coefficients of prototype lowpass IIR filter

row vector | matrix

Numerator coefficients of the prototype lowpass IIR filter, specified as either:

- Row vector -- Specifies the values of  $[b_0, b_1, \dots, b_n]$ , given this transfer function form:

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_0 + b_1z^{-1} + \dots + b_nz^{-n}}{a_0 + a_1z^{-1} + \dots + a_nz^{-n}}$$

where  $n$  is the order of the filter.

- Matrix -- Specifies the numerator coefficients in the form of an  $P$ -by- $(Q+1)$  matrix, where  $P$  is the number of filter sections and  $Q$  is the order of each filter section. If  $Q = 2$ , the filter is a second-order section filter. For higher-order sections, make  $Q > 2$ .

$$\mathbf{b} = \begin{bmatrix} b_{01} & b_{11} & b_{21} & \dots & b_{Q1} \\ b_{02} & b_{12} & b_{22} & \dots & b_{Q2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ b_{0P} & b_{1P} & b_{2P} & \dots & b_{QP} \end{bmatrix}$$

In the transfer function form, the numerator coefficient matrix  $b_{ik}$  of the IIR filter can be represented using the following equation:

$$H(z) = \prod_{k=1}^P H_k(z) = \prod_{k=1}^P \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2} + \dots + b_{Qk}z^{-Q}}{a_{0k} + a_{1k}z^{-1} + a_{2k}z^{-2} + \dots + a_{Qk}z^{-Q}},$$

where,

- **a** -- Denominator coefficients matrix. For more information on how to specify this matrix, see [a](#).
- $k$  -- Row index.
- $i$  -- Column index.

When specified in the matrix form, **b** and **a** matrices must have the same number of rows (filter sections)  $Q$ .

Data Types: `single` | `double`

### **a** — Denominator coefficients of prototype lowpass IIR filter

row vector | matrix

Denominator coefficients for a prototype lowpass IIR filter, specified as one of these options:

- Row vector -- Specifies the values of  $[a_0, a_1, \dots, a_n]$ , given this transfer function form:

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_0 + b_1z^{-1} + \dots + b_nz^{-n}}{a_0 + a_1z^{-1} + \dots + a_nz^{-n}},$$

where  $n$  is the order of the filter.

- Matrix -- Specifies the denominator coefficients in the form of an  $P$ -by- $(Q+1)$  matrix, where  $P$  is the number of filter sections and  $Q$  is the order of each filter section. If  $Q = 2$ , the filter is a second-order section filter. For higher-order sections, make  $Q > 2$ .

$$\mathbf{a} = \begin{bmatrix} a_{01} & a_{11} & a_{21} & \dots & a_{Q1} \\ a_{02} & a_{12} & a_{22} & \dots & a_{Q2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{0P} & a_{1P} & a_{2P} & \dots & a_{QP} \end{bmatrix}$$

In the transfer function form, the denominator coefficient matrix  $a_{ik}$  of the IIR filter can be represented using the following equation:

$$H(z) = \prod_{k=1}^P H_k(z) = \prod_{k=1}^P \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2} + \dots + b_{Qk}z^{-Q}}{a_{0k} + a_{1k}z^{-1} + a_{2k}z^{-2} + \dots + a_{Qk}z^{-Q}},$$

where,

- **b** -- Numerator coefficients matrix. For more information on how to specify this matrix, see [b](#).
- $k$  -- Row index.
- $i$  -- Column index.

When specified in the matrix form, **a** and **b** matrices must have the same number of rows (filter sections)  $P$ .

Data Types: `single` | `double`

**wo** — **Frequency value to transform from prototype filter**

scalar

Frequency value to transform from the prototype filter, specified as a real scalar. Frequency  $\omega_0$  must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

Data Types: `single` | `double`

**wt** — **Desired frequency locations in transformed target filter**

two-element vector

Desired frequency locations in the transformed target filter, specified as a two-element vector. Frequencies in  $\omega_t$  must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

Data Types: `single` | `double`

## Output Arguments

**num** — **Numerator coefficients of transformed bandstop filter**

row vector | matrix

Numerator coefficients of the transformed bandstop filter, returned as one of the following:

- Row vector of length  $2n+1$ , where  $n$  is the order of the input filter. The `num` output is a row vector when the input coefficients `b` and `a` are row vectors.
- $P$ -by- $(2Q+1)$  matrix, where  $P$  is the number of filter sections and  $Q$  is the order of each section of the transformed filter. The `num` output is a matrix when the input coefficients `b` and `a` are matrices.

Data Types: `single` | `double`

**den** — **Denominator coefficients of transformed bandstop filter**

row vector | matrix

Denominator coefficients of the transformed bandstop filter, returned as one of the following:

- Row vector of length  $2n+1$ , where  $n$  is the order of the input filter. The `den` output is a row vector when the input coefficients `b` and `a` are row vectors.
- $P$ -by- $(2Q+1)$  matrix, where  $P$  is the number of filter sections and  $Q$  is the order of each section of the transformed filter. The `den` output is a matrix when the input coefficients `b` and `a` are matrices.

Data Types: `single` | `double`

**allpassNum** — **Numerator coefficients of mapping filter**

row vector

Numerator of the mapping filter, returned as a row vector.

Data Types: `single` | `double`

**allpassDen — Denominator coefficients of mapping filter**

row vector

Denominator of the mapping filter, returned as a row vector.

Data Types: `single` | `double`**More About****IIR Lowpass Filter to IIR Bandstop Filter Transformation**

IIR lowpass filter to IIR bandstop filter transformation effectively places one feature of the original filter, located at frequency  $-w_o$ , at the required target frequency location,  $w_{t1}$ , and the second feature, originally at  $w_o$ , at the new location,  $w_{t2}$ . Choice of the feature subject to the lowpass to bandstop transformation is not restricted only to the cutoff frequency of an original lowpass filter. You can choose to transform any feature of the original filter like stopband edge, DC, deep minimum in the stopband, or others. It is assumed that  $w_{t2}$  is greater than  $w_{t1}$ . Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

This transformation implements the "Nyquist Mobility," which means that the DC feature stays at DC, but the Nyquist feature moves to a location dependent on the selection of  $w_o$  and  $w_{ts}$ .

Relative positions of other features of the original filter change in the target filter. This means that it is possible to select two features of the original filter,  $F_1$  and  $F_2$ , with  $F_1$  preceding  $F_2$ . After the transformation feature  $F_2$  will precede  $F_1$  in the target filter. In addition, the distance between  $F_1$  and  $F_2$  will not be the same before and after the transformation.

For more details on the lowpass to bandstop frequency transformation, see "Digital Frequency Transformations".

**Version History**

Introduced in R2011a

**References**

- [1] Nowrouzian, B., and A.G. Constantinides. "Prototype Reference Transfer Function Parameters in the Discrete-Time Frequency Transformations." In *Proceedings of the 33rd Midwest Symposium on Circuits and Systems*, 1078-82. Calgary, Alta., Canada: IEEE, 1991. <https://doi.org/10.1109/MWSCAS.1990.140912>.
- [2] Nowrouzian, B., and L.T. Bruton. "Closed-Form Solutions for Discrete-Time Elliptic Transfer Functions." In [1992] *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, 784-87. Washington, DC, USA: IEEE, 1992. <https://doi.org/10.1109/MWSCAS.1992.271206>.
- [3] Constantinides, A.G. "Design of Bandpass Digital Filters." *Proceedings of the IEEE* 57, no. 6 (1969): 1229-31. <https://doi.org/10.1109/PROC.1969.7216>.
- [4] Constantinides, A.G. "Spectral transformations for digital filters." *Proceedings of the IEEE*, vol. 117, no. 8: 1585-1590. August 1970.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

### Functions

[iirftransf](#) | [allpasslp2bs](#) | [zpklp2bs](#)

### Topics

“Digital Frequency Transformations”

## iirlp2bsc

Transform IIR lowpass filter to complex bandstop filter

### Syntax

```
[num,den,allpassNum,allpassDen] = iirlp2bsc(b,a,wo,wt)
```

### Description

`[num,den,allpassNum,allpassDen] = iirlp2bsc(b,a,wo,wt)` transform an IIR lowpass filter to a complex bandstop filter.

The prototype lowpass filter is specified with the numerator `b` and the denominator `a`. The function returns the numerator and denominator coefficients of the transformed complex bandstop filter. The function also returns the numerator, `allpassNum`, and the denominator, `allpassDen`, of the allpass mapping filter.

The function transforms a real lowpass prototype filter to a complex bandstop filter by applying a first-order real lowpass to complex bandstop frequency transformation. For more details, see “IIR Lowpass Filter to IIR Complex Bandstop Filter Transformation” on page 5-956.

### Examples

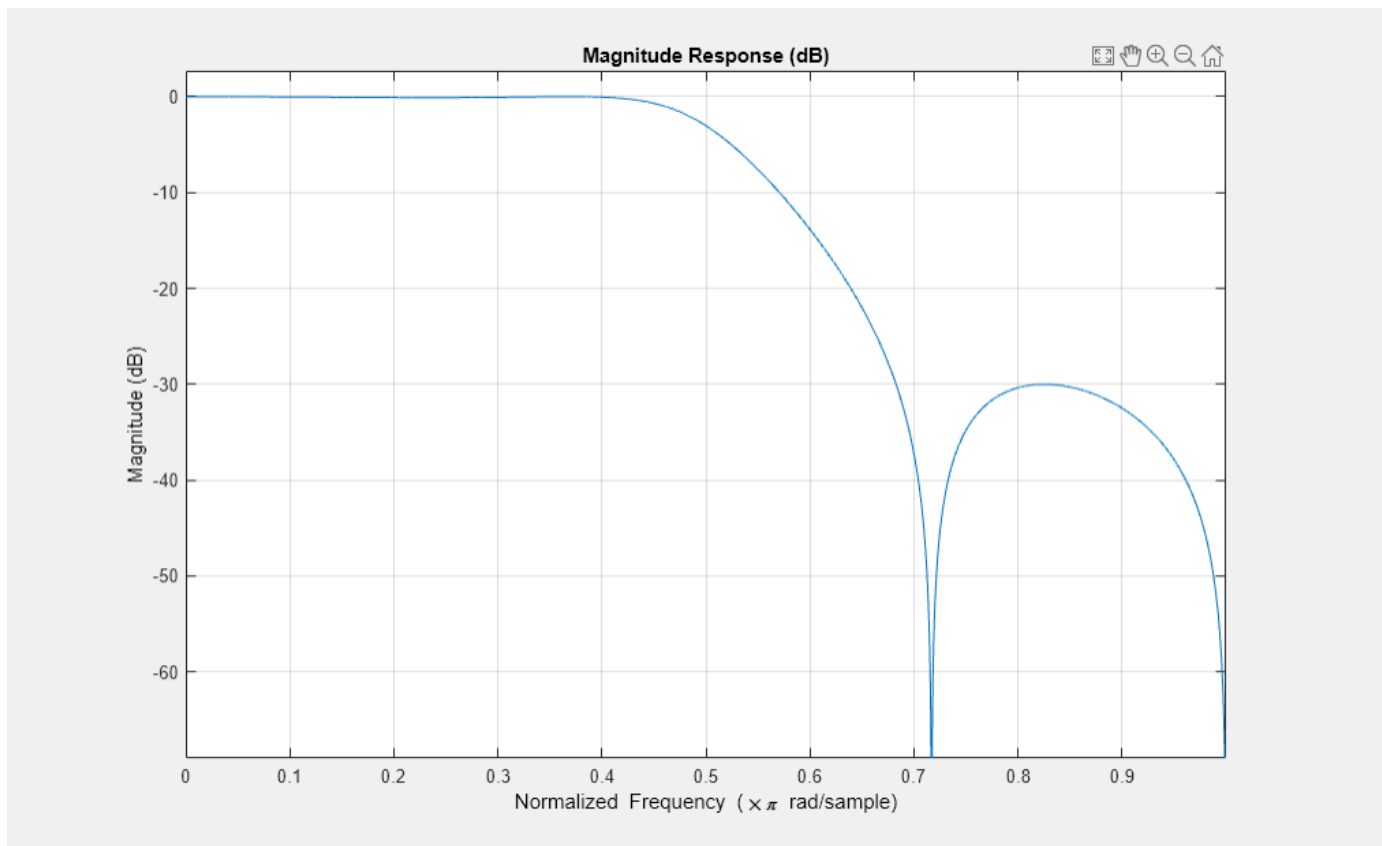
#### Transform Lowpass Filter to Complex Bandstop Filter

Transform a lowpass IIR filter to a complex bandstop filter using the `iirlp2bsc` function.

#### Input Lowpass IIR Filter

Design a prototype real IIR lowpass elliptic filter with a gain of about -3 dB at  $0.5\pi$  rad/sample.

```
[b,a] = ellip(3,0.1,30,0.409);  
fvtool(b,a)
```



### Transform Filter Using `iirlp2bsc`

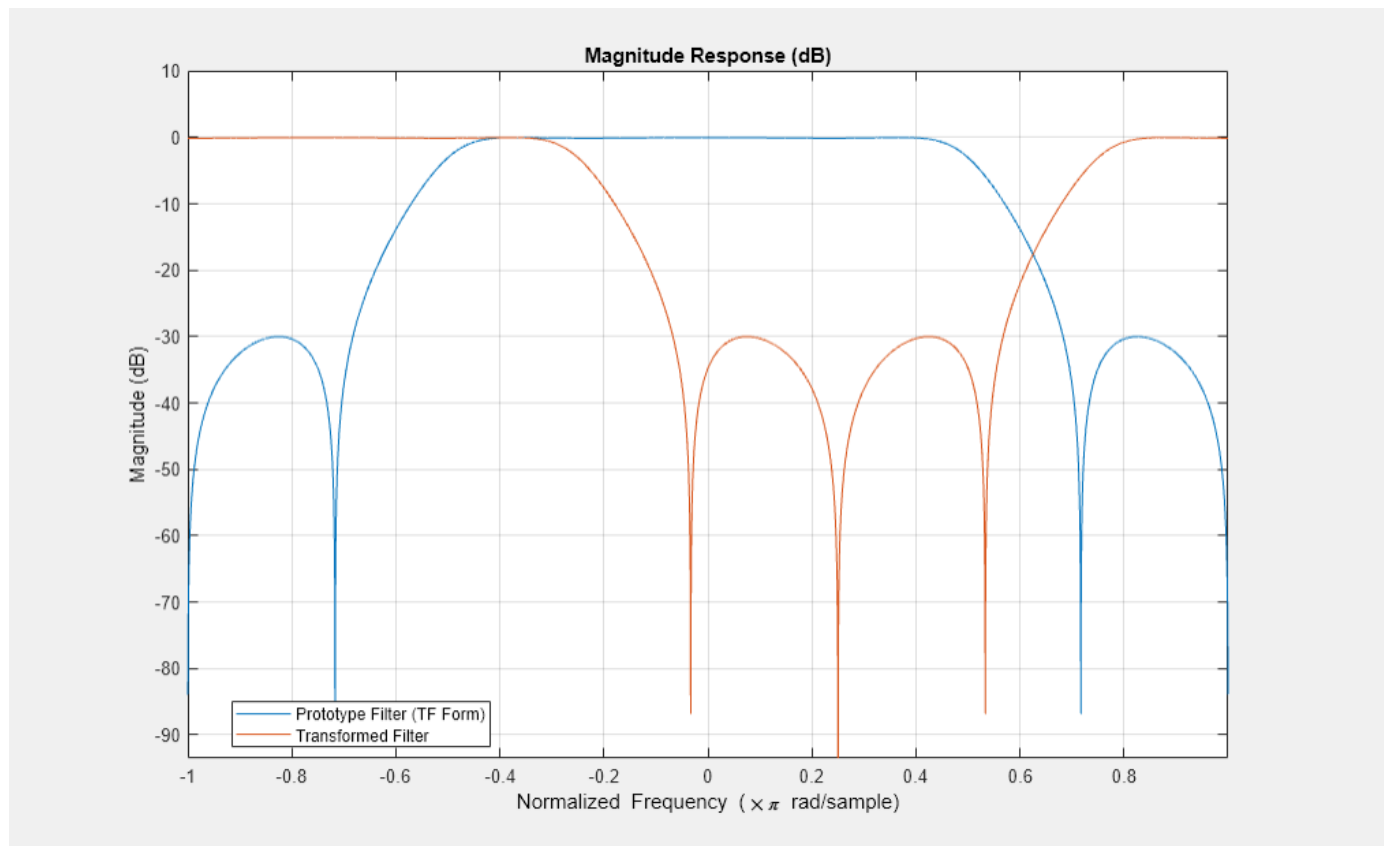
Transform the prototype lowpass filter into a complex bandstop filter by placing the cutoff frequencies of the prototype filter at  $-0.25\pi$  and  $0.75\pi$ .

Specify the prototype filter as a vector of numerator and denominator coefficients, `b` and `a` respectively.

```
[num,den] = iirlp2bsc(b,a,0.5,[-0.25 0.75]);
```

Compare the magnitude response of the filters using `FVTool`.

```
fvt = fvtool(b,a,num,den);
legend(fvt,"Prototype Filter (TF Form)",...
       "Transformed Filter")
```



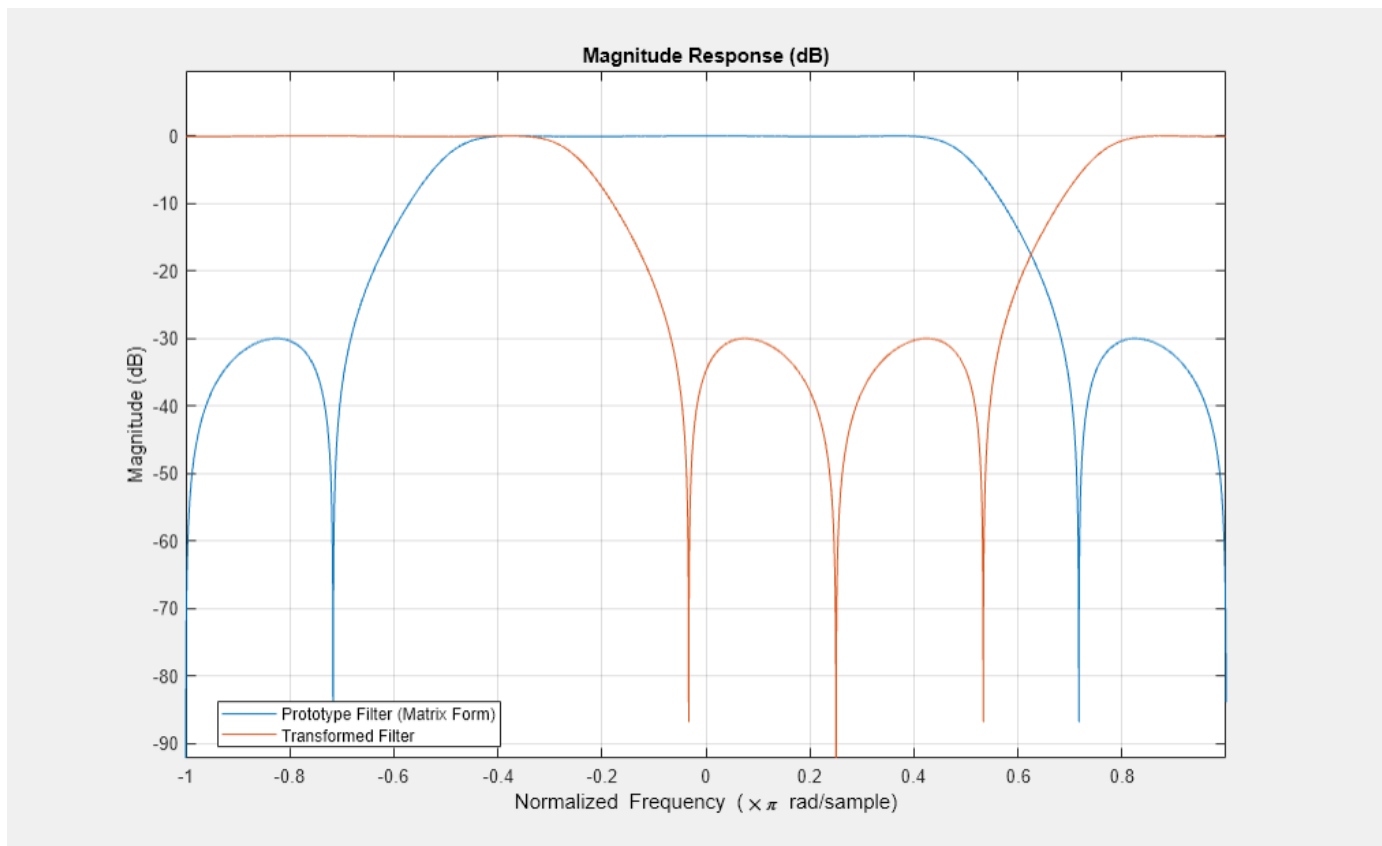
Alternatively, you can also specify the input lowpass IIR filter as a matrix of coefficients. Pass the second order section coefficient matrices as inputs.

```
ss = tf2sos(b,a);
[num2,den2] = iirlp2bsc(ss(:,1:3),ss(:,4:6),0.5,[-0.25 0.75]);
```

Compare the magnitude response of the filters using FVTool.

```
hvft = fvtool(ss,[num2 den2]);
legend(hvft,"Prototype Filter (Matrix Form)", ...
       "Transformed Filter")
```





Copyright 2012-2021 The MathWorks, Inc.

## Input Arguments

### **b** — Numerator coefficients of prototype lowpass IIR filter

row vector | matrix

Numerator coefficients of the prototype lowpass IIR filter, specified as either:

- Row vector -- Specifies the values of  $[b_0, b_1, \dots, b_n]$ , given this transfer function form:

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_0 + b_1z^{-1} + \dots + b_nz^{-n}}{a_0 + a_1z^{-1} + \dots + a_nz^{-n}},$$

where  $n$  is the order of the filter.

- Matrix -- Specifies the numerator coefficients in the form of an  $P$ -by- $(Q+1)$  matrix, where  $P$  is the number of filter sections and  $Q$  is the order of each filter section. If  $Q = 2$ , the filter is a second-order section filter. For higher-order sections, make  $Q > 2$ .

$$\mathbf{b} = \begin{bmatrix} b_{01} & b_{11} & b_{21} & \dots & b_{Q1} \\ b_{02} & b_{12} & b_{22} & \dots & b_{Q2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ b_{0P} & b_{1P} & b_{2P} & \dots & b_{QP} \end{bmatrix}$$

In the transfer function form, the numerator coefficient matrix  $b_{ik}$  of the IIR filter can be represented using the following equation:

$$H(z) = \prod_{k=1}^P H_k(z) = \prod_{k=1}^P \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2} + \dots + b_{Qk}z^{-Q}}{a_{0k} + a_{1k}z^{-1} + a_{2k}z^{-2} + \dots + a_{Qk}z^{-Q}},$$

where,

- **a** -- Denominator coefficients matrix. For more information on how to specify this matrix, see [a](#).
- $k$  -- Row index.
- $i$  -- Column index.

When specified in the matrix form, **b** and **a** matrices must have the same number of rows (filter sections)  $Q$ .

Data Types: `single` | `double`

### **a** — Denominator coefficients of prototype lowpass IIR filter

row vector | matrix

Denominator coefficients for a prototype lowpass IIR filter, specified as one of these options:

- Row vector -- Specifies the values of  $[a_0, a_1, \dots, a_n]$ , given this transfer function form:

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_0 + b_1z^{-1} + \dots + b_nz^{-n}}{a_0 + a_1z^{-1} + \dots + a_nz^{-n}},$$

where  $n$  is the order of the filter.

- Matrix -- Specifies the denominator coefficients in the form of an  $P$ -by- $(Q+1)$  matrix, where  $P$  is the number of filter sections and  $Q$  is the order of each filter section. If  $Q = 2$ , the filter is a second-order section filter. For higher-order sections, make  $Q > 2$ .

$$\mathbf{a} = \begin{bmatrix} a_{01} & a_{11} & a_{21} & \dots & a_{Q1} \\ a_{02} & a_{12} & a_{22} & \dots & a_{Q2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{0P} & a_{1P} & a_{2P} & \dots & a_{QP} \end{bmatrix}$$

In the transfer function form, the denominator coefficient matrix  $a_{ik}$  of the IIR filter can be represented using the following equation:

$$H(z) = \prod_{k=1}^P H_k(z) = \prod_{k=1}^P \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2} + \dots + b_{Qk}z^{-Q}}{a_{0k} + a_{1k}z^{-1} + a_{2k}z^{-2} + \dots + a_{Qk}z^{-Q}},$$

where,

- **b** -- Numerator coefficients matrix. For more information on how to specify this matrix, see [b](#).
- $k$  -- Row index.
- $i$  -- Column index.

When specified in the matrix form, **a** and **b** matrices must have the same number of rows (filter sections)  $P$ .

Data Types: `single` | `double`

#### **wo** — Frequency value to transform from prototype filter

real scalar

Frequency value to transform from the prototype filter, specified as a scalar. Frequency  $\omega_0$  should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

Data Types: `single` | `double`

#### **wt** — Desired frequency locations in transformed target filter

two-element vector

Desired frequency locations in the transformed target filter, specified as a two-element vector. Frequencies in  $\omega_t$  should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

Data Types: `single` | `double`

## Output Arguments

#### **num** — Numerator coefficients of transformed complex bandstop filter

row vector | matrix

Numerator coefficients of the transformed complex bandstop filter, returned as one of the following:

- Row vector of length  $n+1$ , where  $n$  is the order of the input filter. The `num` output is a row vector when the input coefficients `b` and `a` are row vectors.
- $P$ -by- $(Q+1)$  matrix, where  $P$  is the number of filter sections and  $Q$  is the order of each section of the transformed filter. The `num` output is a matrix when the input coefficients `b` and `a` are matrices.

Data Types: `single` | `double`

#### **den** — Denominator coefficients of transformed complex bandstop filter

row vector | matrix

Denominator coefficients of the transformed complex bandstop filter, returned as one of the following:

- Row vector of length  $n+1$ , where  $n$  is the order of the input filter. The `den` output is a row vector when the input coefficients `b` and `a` are row vectors.
- $P$ -by- $(Q+1)$  matrix, where  $P$  is the number of filter sections and  $Q$  is the order of each section of the transformed filter. The `den` output is a matrix when the input coefficients `b` and `a` are matrices.

Data Types: `single` | `double`

#### **allpassNum** — Numerator coefficients of mapping filter

row vector

Numerator of the mapping filter, returned as a row vector.

Data Types: `single` | `double`

### **allpassDen — Denominator coefficients of mapping filter**

row vector

Denominator of the mapping filter, returned as a row vector.

Data Types: `single` | `double`

## **More About**

### **IIR Lowpass Filter to IIR Complex Bandstop Filter Transformation**

IIR Lowpass Filter to IIR Complex Bandstop Filter transformation effectively places one feature of the original filter, located at frequency  $-w_o$ , at the required target frequency location,  $w_{t1}$ , and the second feature, originally at  $w_o$ , at the new location,  $w_{t2}$ . It is assumed that  $w_{t2}$  is greater than  $w_{t1}$ . Additionally, the transformation swaps passbands with stopbands in the target filter.

Relative positions of other features of the original filter do not change in the target filter. This means that it is possible to select two features of an original filter,  $F_1$  and  $F_2$ , with  $F_1$  preceding  $F_2$ . Feature  $F_1$  will still precede  $F_2$  after the transformation. However, the distance between  $F_1$  and  $F_2$  will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to bandstop transformation is not restricted only to the cutoff frequency of an original lowpass filter. You can choose to transform any feature of the original filter like stopband edge, DC, deep minimum in the stopband, or others.

Lowpass to bandpass transformation can also be used to transform other types of filters, for example. real notch filters or resonators can be doubled and positioned at two distinct desired frequencies at any place around the unit circle forming a pair of complex notches/resonators. This transformation can be used for designing bandstop filters for band attenuation or frequency equalizers, from the high-quality prototype lowpass filter.

## **Version History**

**Introduced in R2011a**

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

## **See Also**

### **Functions**

`iirftransf` | `allpasslp2bsc` | `zpklp2bsc`

## iirlp2hp

Transform lowpass IIR filter to highpass filter

### Syntax

```
[num,den] = iirlp2hp(b,a,wo,wt)
[num,den,allpassNum,allpassDen] = iirlp2hp(b,a,wo,wt)
```

### Description

`[num,den] = iirlp2hp(b,a,wo,wt)` transforms a lowpass IIR filter to a highpass IIR filter.

The `iirlp2hp` function returns the numerator and denominator coefficients of the transformed highpass filter. The function accepts `wo`, frequency value to be transformed from the prototype filter, and `wt`, desired frequency in the transformed highpass filter, and applies the lowpass to highpass frequency transformation on the input prototype filter. The input prototype lowpass filter is specified with the numerator and denominator coefficients, `b` and `a` respectively. For more details on the transformation, see “IIR Lowpass to Highpass Frequency Transformation” on page 5-963.

`[num,den,allpassNum,allpassDen] = iirlp2hp(b,a,wo,wt)` additionally returns the numerator and the denominator coefficients of the mapping filter.

### Examples

#### Transform Lowpass Filter to Highpass Filter

Transform an IIR filter from lowpass to highpass by moving the magnitude response at one frequency in the source filter to a new location in the transformed filter.

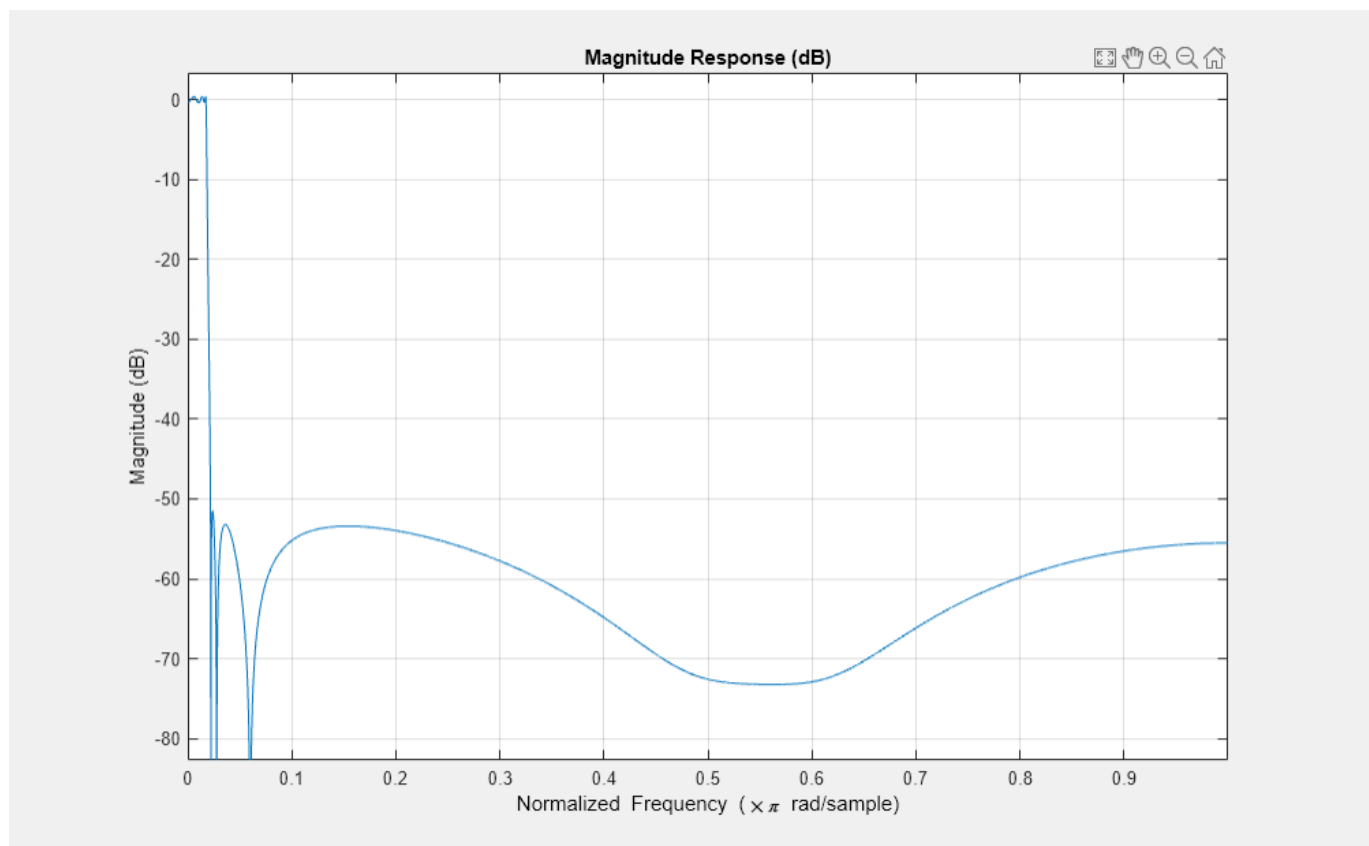
#### Input Lowpass IIR Filter

Generate a least P-norm optimal IIR lowpass filter with varying attenuation levels in the stopband. Specify a numerator order of 10 and a denominator order of 6. The function returns the coefficients both in the vector form and in the second-order sections (SOS) form. The output argument `g` specifies the overall gain of the filter when expressed in the second-order sections form.

```
[b,a,~,sos,g] = iirlpnorm(10,6,[0 0.0175 0.02 0.0215 0.025 1], ...
    [0 0.0175 0.02 0.0215 0.025 1],[1 1 0 0 0 0], ...
    [1 1 1 1 20 20]);
```

Visualize the magnitude response of the filter.

```
fvtool(b,a)
```



### Transform Filter Using `iirlp2hp`

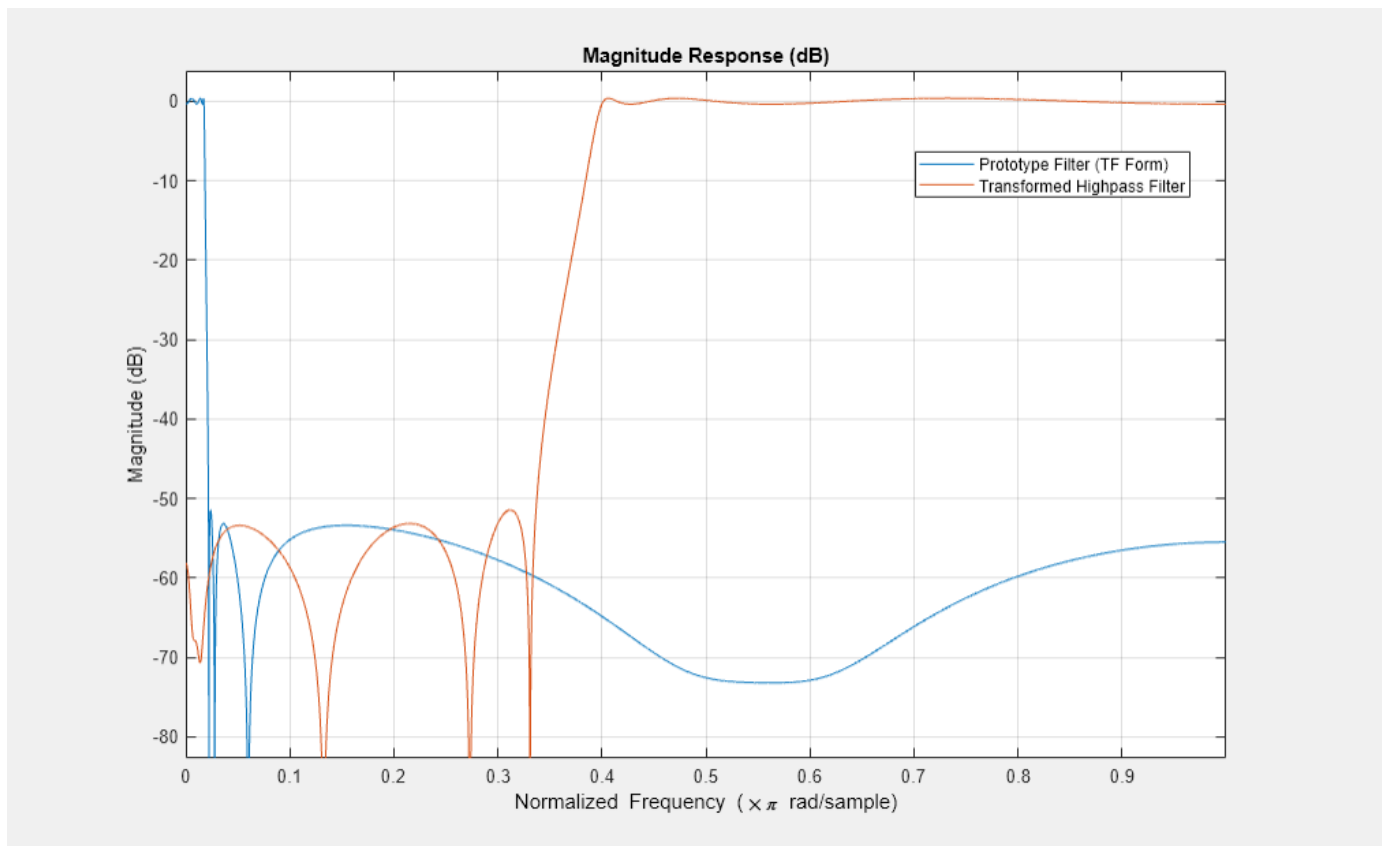
Transform the lowpass IIR filter using the `iirlp2hp` function. Specify the filter as a vector of numerator and denominator coefficients.

To generate a highpass filter whose passband flattens out at  $0.4\pi$  rad/sample, select the frequency in the lowpass filter at  $0.0175\pi$ , the frequency where the passband starts to roll off, and move it to the new location.

```
wc = 0.0175;
wd = 0.4;
[num,den] = iirlp2hp(b,a,wc,wd);
```

Compare the magnitude responses of the filters using FVTool. The transition band for the highpass filter is essentially the mirror image of the transition for the lowpass filter from  $0.0175\pi$  to  $0.025\pi$ , stretched out over a wider frequency range. In the passbands, the filter share common ripple characteristics and magnitude.

```
hvft = fvtool(b,a,num,den);
legend(hvft,"Prototype Filter (TF Form)",...
       "Transformed Highpass Filter")
```

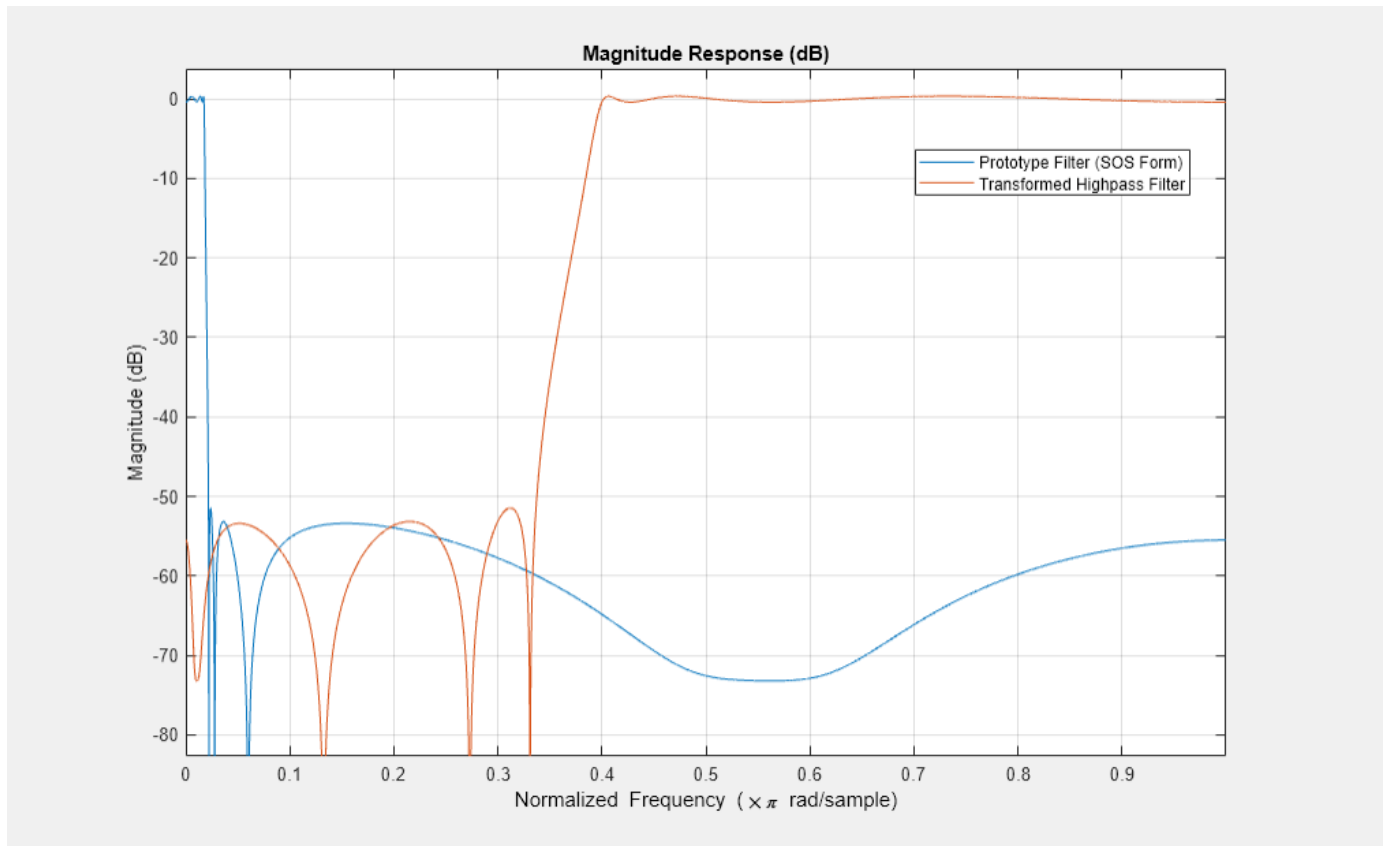


Alternatively, you can also specify the input lowpass IIR filter as a matrix of coefficients. Pass the scaled second order section coefficient matrices as inputs. Apply the scaling factor  $g$  to the first section of the filter.

```
sosg = sos;
sosg(1,1:3) = g*sosg(1,1:3);
[num2,den2] = iirlp2hp(sosg(:,1:3),sosg(:,4:6),wc,wd);
```

Compare the magnitude response of the filters using FVTool.

```
hvft = fvtool(sosg,[num2 den2]);
legend(hvft,"Prototype Filter (SOS Form)",...
       "Transformed Highpass Filter")
```



Copyright 2012-2021 The MathWorks, Inc.

## Input Arguments

### **b** — Numerator coefficients of prototype lowpass IIR filter

row vector | matrix

Numerator coefficients of the prototype lowpass IIR filter, specified as either:

- Row vector -- Specifies the values of  $[b_0, b_1, \dots, b_n]$ , given this transfer function form:

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_0 + b_1z^{-1} + \dots + b_nz^{-n}}{a_0 + a_1z^{-1} + \dots + a_nz^{-n}}$$

where  $n$  is the order of the filter.

- Matrix -- Specifies the numerator coefficients in the form of an  $P$ -by- $(Q+1)$  matrix, where  $P$  is the number of filter sections and  $Q$  is the order of each filter section. If  $Q = 2$ , the filter is a second-order section filter. For higher-order sections, make  $Q > 2$ .

$$\mathbf{b} = \begin{bmatrix} b_{01} & b_{11} & b_{21} & \dots & b_{Q1} \\ b_{02} & b_{12} & b_{22} & \dots & b_{Q2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ b_{0P} & b_{1P} & b_{2P} & \dots & b_{QP} \end{bmatrix}$$



In the transfer function form, the numerator coefficient matrix  $b_{ik}$  of the IIR filter can be represented using the following equation:

$$H(z) = \prod_{k=1}^P H_k(z) = \prod_{k=1}^P \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2} + \dots + b_{Qk}z^{-Q}}{a_{0k} + a_{1k}z^{-1} + a_{2k}z^{-2} + \dots + a_{Qk}z^{-Q}},$$

where,

- **a** -- Denominator coefficients matrix. For more information on how to specify this matrix, see [a](#).
- $k$  -- Row index.
- $i$  -- Column index.

When specified in the matrix form, **b** and **a** matrices must have the same number of rows (filter sections)  $Q$ .

Data Types: `single` | `double`

### **a** — Denominator coefficients of prototype lowpass IIR filter

row vector | matrix

Denominator coefficients for a prototype lowpass IIR filter, specified as one of these options:

- Row vector -- Specifies the values of  $[a_0, a_1, \dots, a_n]$ , given this transfer function form:

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_0 + b_1z^{-1} + \dots + b_nz^{-n}}{a_0 + a_1z^{-1} + \dots + a_nz^{-n}},$$

where  $n$  is the order of the filter.

- Matrix -- Specifies the denominator coefficients in the form of an  $P$ -by- $(Q+1)$  matrix, where  $P$  is the number of filter sections and  $Q$  is the order of each filter section. If  $Q = 2$ , the filter is a second-order section filter. For higher-order sections, make  $Q > 2$ .

$$\mathbf{a} = \begin{bmatrix} a_{01} & a_{11} & a_{21} & \dots & a_{Q1} \\ a_{02} & a_{12} & a_{22} & \dots & a_{Q2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{0P} & a_{1P} & a_{2P} & \dots & a_{QP} \end{bmatrix}$$

In the transfer function form, the denominator coefficient matrix  $a_{ik}$  of the IIR filter can be represented using the following equation:

$$H(z) = \prod_{k=1}^P H_k(z) = \prod_{k=1}^P \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2} + \dots + b_{Qk}z^{-Q}}{a_{0k} + a_{1k}z^{-1} + a_{2k}z^{-2} + \dots + a_{Qk}z^{-Q}},$$

where,

- **b** -- Numerator coefficients matrix. For more information on how to specify this matrix, see [b](#).
- $k$  -- Row index.
- $i$  -- Column index.

When specified in the matrix form, **a** and **b** matrices must have the same number of rows (filter sections)  $P$ .

Data Types: `single` | `double`

**wo** — Frequency value to be transformed from prototype filter

positive scalar in  $(0, 1)$

Frequency value to transform from the prototype filter, specified as a real positive scalar in the range  $(0, 1)$ .

Data Types: `single` | `double`

**wt** — Desired frequency location in transformed highpass filter

positive scalar in  $(0, 1)$

Desired frequency location in the transformed highpass filter, specified as a real positive scalar in the range  $(0, 1)$ .

Data Types: `single` | `double`

## Output Arguments

**num** — Numerator coefficients of transformed highpass filter

row vector | matrix

Numerator coefficients of the transformed highpass filter, returned as one of the following:

- Row vector of length  $n+1$ , where  $n$  is the order of the input filter. The **num** output is a row vector when the input coefficients **b** and **a** are row vectors.
- $P$ -by- $(Q+1)$  matrix, where  $P$  is the number of filter sections and  $Q$  is the order of each section of the transformed filter. The **num** output is a matrix when the input coefficients **b** and **a** are matrices.

Data Types: `single` | `double`

**den** — Denominator coefficients of transformed highpass filter

row vector | matrix

Denominator coefficients of the transformed highpass filter, returned as one of the following:

- Row vector of length  $n+1$ , where  $n$  is the order of the input filter. The **den** output is a row vector when the input coefficients **b** and **a** are row vectors.
- $P$ -by- $(Q+1)$  matrix, where  $P$  is the number of filter sections and  $Q$  is the order of each section of the transformed filter. The **den** output is a matrix when the input coefficients **b** and **a** are matrices.

Data Types: `single` | `double`

**allpassNum** — Numerator coefficients of mapping filter

row vector

Numerator coefficients of the mapping filter, returned as a row vector.

Data Types: `single` | `double`

## **aIIPassDen — Denominator coefficients of mapping filter**

row vector

Denominator coefficients of the mapping filter, returned as a row vector.

Data Types: `single` | `double`

## **More About**

### **IIR Lowpass to Highpass Frequency Transformation**

IIR lowpass to highpass frequency transformation takes a selected frequency from the lowpass filter,  $\omega_0$ , and maps the corresponding magnitude response value onto the desired frequency location in the highpass filter,  $\omega_t$ . Note that all frequencies are normalized between 0 and 1 and that the filter order does not change when you transform to a highpass filter.

When you select  $\omega_0$  and designate  $\omega_t$ , the transformation algorithm sets the magnitude response at the  $\omega_t$  values of your bandstop filter to be the same as the magnitude response of your lowpass filter at  $\omega_0$ . Filter performance between the values in  $\omega_t$  is not specified, except that the stopband retains the ripple nature of your original lowpass filter and the magnitude response in the stopband is equal to the peak response of your lowpass filter. To accurately specify the filter magnitude response across the stopband of your bandpass filter, use a frequency value from within the stopband of your lowpass filter as  $\omega_0$ . Then your bandstop filter response is the same magnitude and ripple as your lowpass filter stopband magnitude and ripple.

The transformation retains the shape of the original filter is what makes this function useful. If you have a lowpass filter whose characteristics, such as rolloff or passband ripple, particularly meet your needs, the transformation function lets you create a new filter with the same characteristic performance features, but in a highpass version without designing the highpass filter from the beginning.

In some cases transforming your filter may cause numerical problems, resulting in incorrect conversion to the highpass filter. Use `fvtool` to verify the response of your converted filter.

## **Version History**

**Introduced in R2011a**

## **References**

- [1] Nowrouzian, B., and A.G. Constantinides. "Prototype Reference Transfer Function Parameters in the Discrete-Time Frequency Transformations." In *Proceedings of the 33rd Midwest Symposium on Circuits and Systems*, 1078-82. Calgary, Alta., Canada: IEEE, 1991. <https://doi.org/10.1109/MWSCAS.1990.140912>.
- [2] Nowrouzian, B., and L.T. Bruton. "Closed-Form Solutions for Discrete-Time Elliptic Transfer Functions." In [1992] *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, 784-87. Washington, DC, USA: IEEE, 1992. <https://doi.org/10.1109/MWSCAS.1992.271206>.
- [3] Constantinides, A.G. "Spectral transformations for digital filters." *Proceedings of the IEEE*, vol. 117, no. 8: 1585-1590. August 1970.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

## **See Also**

### **Functions**

`iirlp2bp` | `iirlp2bs` | `iirlp2lp` | `firlp2lp` | `firlp2hp`

## iirlp2lp

Transform lowpass IIR filter to different lowpass filter

### Syntax

```
[num,den] = iirlp2lp(b,a,wo,wt)
[num,den,allpassNum,allpassDen] = iirlp2lp(b,a,wo,wt)
```

### Description

`[num,den] = iirlp2lp(b,a,wo,wt)` transforms lowpass IIR filter to different lowpass filter.

The prototype lowpass filter is specified with the numerator and denominator coefficients, `b` and `a` respectively. The function returns the numerator and denominator coefficients of the transformed lowpass digital filter.

The function transforms the magnitude response from lowpass to a different lowpass. For more details, see “Lowpass IIR Filter to Different Lowpass Filter Transformation” on page 5-971.

`[num,den,allpassNum,allpassDen] = iirlp2lp(b,a,wo,wt)` in addition returns the numerator and the denominator coefficients of the allpass mapping filter.

### Examples

#### Extend Passband of Lowpass Filter

Transform the passband of a lowpass IIR filter by moving the magnitude response from one frequency in the source filter to a new location in the transformed filter.

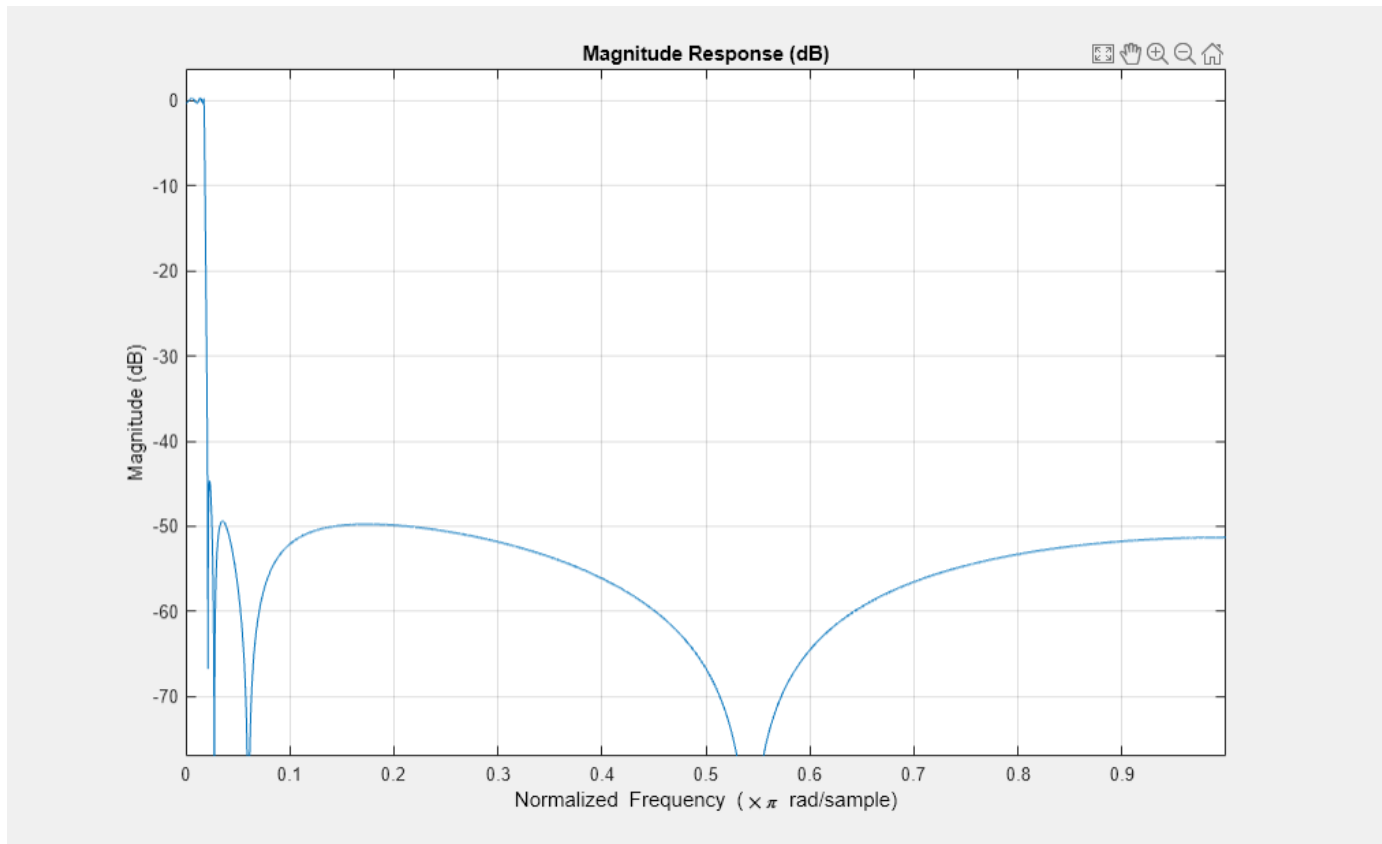
#### Input Lowpass IIR Filter

Generate a least P-norm optimal IIR lowpass filter using the `iirlpnorm` function. Specify a numerator order of 10 and a denominator order of 6. The function returns the coefficients both in the vector form and in the second-order sections (SOS) form. The output argument `g` specifies the overall gain of the filter when expressed in the second-order sections form.

```
[b,a,~,sos,g] = iirlpnorm(10,6, ...
    [0 0.0175 0.02 0.0215 0.025 1], ...
    [0 0.0175 0.02 0.0215 0.025 1],[1 1 0 0 0 0], ...
    [1 1 1 1 10 10]);
```

Visualize the magnitude response of the filter.

```
fvtool(b,a)
```



### Transform Filter Using `iirlp2lp`

Transform the passband of the lowpass IIR filter using the `iirlp2lp` function. Specify the filter as a vector of numerator and denominator coefficients.

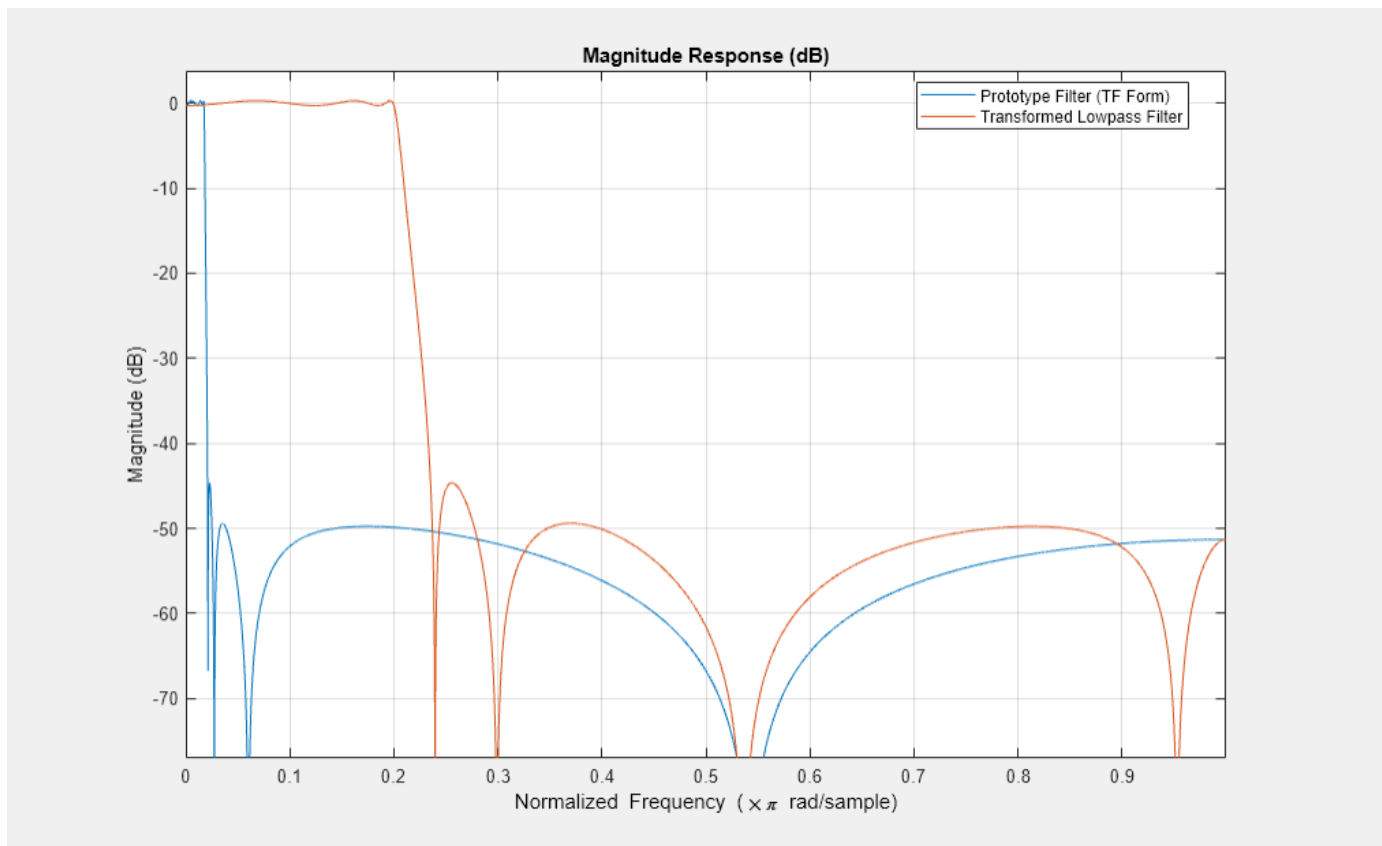
To generate a lowpass filter whose passband extends out to  $0.2\pi$  rad/sample, select the frequency in the lowpass filter at  $0.0175\pi$ , the frequency where the passband starts to roll off, and move it to the new location.

Moving the edge of the passband from  $0.0175\pi$  to  $0.2\pi$  results in a new lowpass filter whose peak response in-band is the same as in the original filter, with the same ripple and the same absolute magnitude. The rolloff is slightly less steep and the stopband profiles are the same for both filters. The new filter stopband is a "stretched" version of the original, as is the passband of the new filter.

```
wc = 0.0175;
wd = 0.2;
[num,den] = iirlp2lp(b,a,wc,wd);
```

Compare the magnitude response of the filters using FVTool.

```
hvft = fvtool(b,a,num,den);
legend(hvft,"Prototype Filter (TF Form)", ...
      "Transformed Lowpass Filter")
```

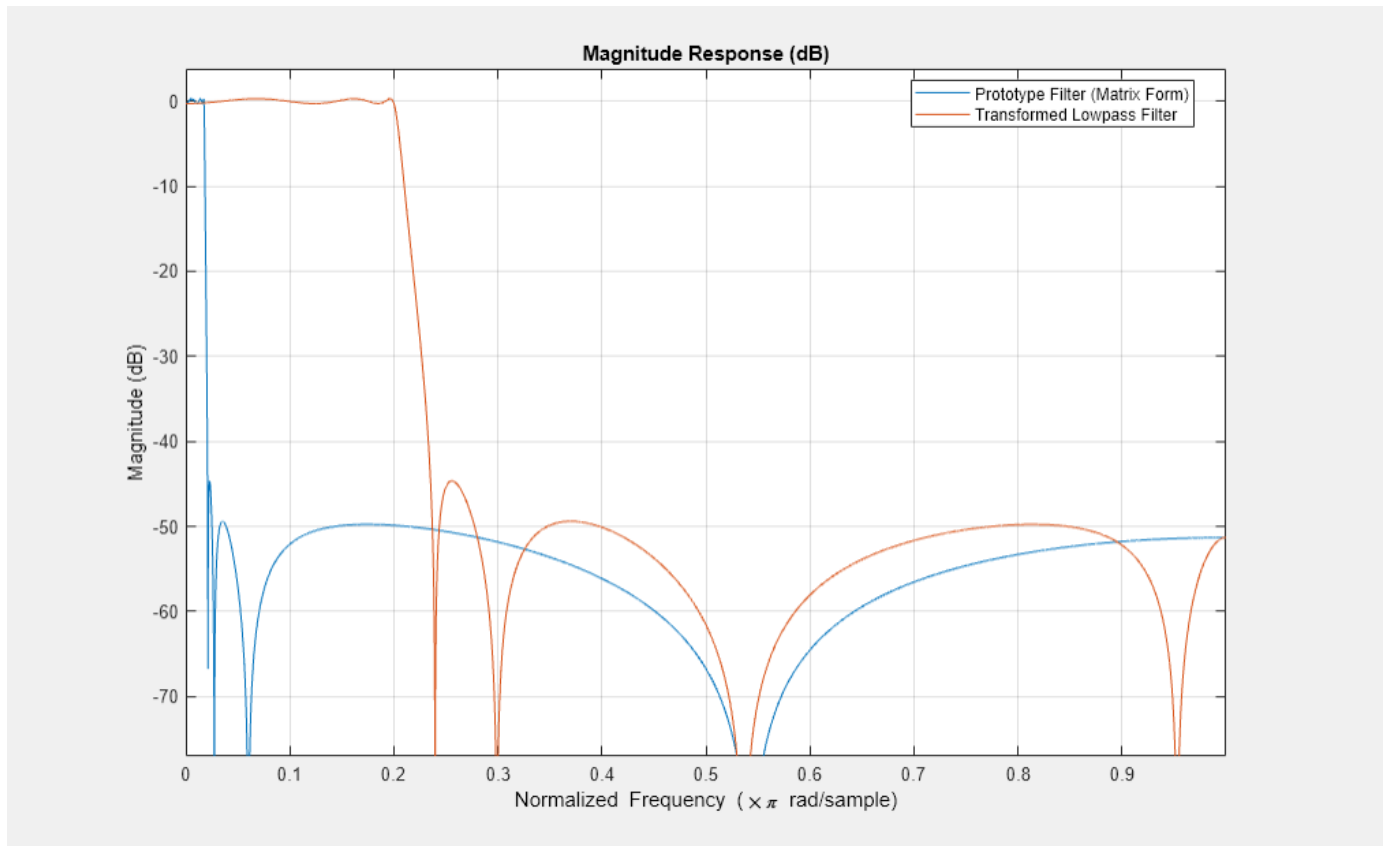


Alternatively, you can also specify the input lowpass IIR filter as a matrix of coefficients. Pass the scaled second order section coefficient matrices as inputs. Apply the scaling factor  $g$  to the first section of the filter.

```
sosg = sos;
sosg(1,1:3) = g*sosg(1,1:3);
[num2,den2] = iirlp2lp(sosg(:,1:3),sosg(:,4:6),wc,wd);
```

Compare the magnitude response of the filters using FVTool.

```
hvft = fvtool(sosg,[num2 den2]);
legend(hvft,"Prototype Filter (Matrix Form)",...
        "Transformed Lowpass Filter")
```



Copyright 2012-2021 The MathWorks, Inc.

## Input Arguments

### **b** — Numerator coefficients of prototype lowpass IIR filter

row vector | matrix

Numerator coefficients of the prototype lowpass IIR filter, specified as either:

- Row vector -- Specifies the values of  $[b_0, b_1, \dots, b_n]$ , given this transfer function form:

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_0 + b_1z^{-1} + \dots + b_nz^{-n}}{a_0 + a_1z^{-1} + \dots + a_nz^{-n}}$$

where  $n$  is the order of the filter.

- Matrix -- Specifies the numerator coefficients in the form of an  $P$ -by- $(Q+1)$  matrix, where  $P$  is the number of filter sections and  $Q$  is the order of each filter section. If  $Q = 2$ , the filter is a second-order section filter. For higher-order sections, make  $Q > 2$ .

$$\mathbf{b} = \begin{bmatrix} b_{01} & b_{11} & b_{21} & \dots & b_{Q1} \\ b_{02} & b_{12} & b_{22} & \dots & b_{Q2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ b_{0P} & b_{1P} & b_{2P} & \dots & b_{QP} \end{bmatrix}$$



In the transfer function form, the numerator coefficient matrix  $b_{ik}$  of the IIR filter can be represented using the following equation:

$$H(z) = \prod_{k=1}^P H_k(z) = \prod_{k=1}^P \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2} + \dots + b_{Qk}z^{-Q}}{a_{0k} + a_{1k}z^{-1} + a_{2k}z^{-2} + \dots + a_{Qk}z^{-Q}},$$

where,

- **a** -- Denominator coefficients matrix. For more information on how to specify this matrix, see [a](#).
- $k$  -- Row index.
- $i$  -- Column index.

When specified in the matrix form, **b** and **a** matrices must have the same number of rows (filter sections)  $Q$ .

Data Types: `single` | `double`

### **a** — Denominator coefficients of prototype lowpass IIR filter

row vector | matrix

Denominator coefficients for a prototype lowpass IIR filter, specified as one of these options:

- Row vector -- Specifies the values of  $[a_0, a_1, \dots, a_n]$ , given this transfer function form:

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_0 + b_1z^{-1} + \dots + b_nz^{-n}}{a_0 + a_1z^{-1} + \dots + a_nz^{-n}},$$

where  $n$  is the order of the filter.

- Matrix -- Specifies the denominator coefficients in the form of an  $P$ -by- $(Q+1)$  matrix, where  $P$  is the number of filter sections and  $Q$  is the order of each filter section. If  $Q = 2$ , the filter is a second-order section filter. For higher-order sections, make  $Q > 2$ .

$$\mathbf{a} = \begin{bmatrix} a_{01} & a_{11} & a_{21} & \dots & a_{Q1} \\ a_{02} & a_{12} & a_{22} & \dots & a_{Q2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{0P} & a_{1P} & a_{2P} & \dots & a_{QP} \end{bmatrix}$$

In the transfer function form, the denominator coefficient matrix  $a_{ik}$  of the IIR filter can be represented using the following equation:

$$H(z) = \prod_{k=1}^P H_k(z) = \prod_{k=1}^P \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2} + \dots + b_{Qk}z^{-Q}}{a_{0k} + a_{1k}z^{-1} + a_{2k}z^{-2} + \dots + a_{Qk}z^{-Q}},$$

where,

- **b** -- Numerator coefficients matrix. For more information on how to specify this matrix, see [b](#).
- $k$  -- Row index.
- $i$  -- Column index.

When specified in the matrix form, **a** and **b** matrices must have the same number of rows (filter sections)  $P$ .

Data Types: `single` | `double`

**wo** — **Frequency value to transform from prototype filter**

positive scalar

Frequency value to transform from the prototype filter, specified as a real positive scalar. Frequency **wo** must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

Data Types: `single` | `double`

**wt** — **Desired frequency location in transformed target filter**

positive scalar

Desired frequency location in the transformed target filter, specified as a real positive scalar. Frequency **wt** must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

Data Types: `single` | `double`

## Output Arguments

**num** — **Numerator coefficients of transformed lowpass filter**

row vector | matrix

Numerator coefficients of the transformed lowpass filter, returned as one of the following:

- Row vector of length  $n+1$ , where  $n$  is the order of the input filter. The **num** output is a row vector when the input coefficients **b** and **a** are row vectors.
- $P$ -by- $(Q+1)$  matrix, where  $P$  is the number of filter sections and  $Q$  is the order of each section of the transformed filter. The **num** output is a matrix when the input coefficients **b** and **a** are matrices.

Data Types: `single` | `double`

**den** — **Denominator coefficients of transformed lowpass filter**

row vector | matrix

Denominator coefficients of the transformed lowpass filter, returned as one of the following:

- Row vector of length  $n+1$ , where  $n$  is the order of the input filter. The **den** output is a row vector when the input coefficients **b** and **a** are row vectors.
- $P$ -by- $(Q+1)$  matrix, where  $P$  is the number of filter sections and  $Q$  is the order of each section of the transformed filter. The **den** output is a matrix when the input coefficients **b** and **a** are matrices.

Data Types: `single` | `double`

**allpassNum** — **Numerator coefficients of mapping filter**

row vector

Numerator coefficients of the mapping filter, returned as a row vector.

Data Types: `single` | `double`

## **aIIRpassDen — Denominator coefficients of mapping filter**

row vector

Denominator coefficients of the mapping filter, returned as a row vector.

Data Types: `single` | `double`

## **More About**

### **Lowpass IIR Filter to Different Lowpass Filter Transformation**

Lowpass IIR filter to different lowpass filter transformation takes a selected frequency from your lowpass filter,  $\omega_0$ , and maps the corresponding magnitude response value onto the desired frequency location in the transformed lowpass filter,  $\omega_t$ . Note that all frequencies are normalized between zero and one and that the filter order does not change when you transform to the target lowpass filter.

When you select  $\omega_0$  and designate  $\omega_t$ , the transformation algorithm sets the magnitude response at the  $\omega_t$  values of your bandstop filter to be the same as the magnitude response of your lowpass filter at  $\omega_0$ . Filter performance between the values in  $\omega_t$  is not specified, except that the stopband retains the ripple nature of your original lowpass filter and the magnitude response in the stopband is equal to the peak response of your lowpass filter. To accurately specify the filter magnitude response across the stopband of your bandpass filter, use a frequency value from within the stopband of your lowpass filter as  $\omega_c$ . Then your bandstop filter response is the same magnitude and ripple as your lowpass filter stopband magnitude and ripple.

The fact that the transformation retains the shape of the original filter is what makes this function useful. If you have a lowpass filter whose characteristics, such as rolloff or passband ripple, particularly meet your needs, the transformation function lets you create a new filter with the same characteristic performance features.

In some cases transforming your filter may cause numerical problems, resulting in incorrect conversion to the target filter. Use `fvtool` to verify the response of your converted filter.

## **Version History**

**Introduced in R2011a**

## **References**

- [1] Nowrouzian, B., and A.G. Constantinides. "Prototype Reference Transfer Function Parameters in the Discrete-Time Frequency Transformations." In *Proceedings of the 33rd Midwest Symposium on Circuits and Systems*, 1078-82. Calgary, Alta., Canada: IEEE, 1991. <https://doi.org/10.1109/MWSCAS.1990.140912>.
- [2] Nowrouzian, B., and L.T. Bruton. "Closed-Form Solutions for Discrete-Time Elliptic Transfer Functions." In [1992] *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, 784-87. Washington, DC, USA: IEEE, 1992. <https://doi.org/10.1109/MWSCAS.1992.271206>.
- [3] Constantinides, A.G. "Spectral transformations for digital filters." *Proceedings of the IEEE*, vol. 117, no. 8: 1585-1590. August 1970.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

## **See Also**

### **Functions**

`iirlp2bp` | `iirlp2bs` | `iirlp2hp` | `firlp2lp` | `firlp2hp`

# iirlp2mb

Transform IIR lowpass filter to IIR multiband filter

## Syntax

```
[num,den,allpassNum,allpassDen] = iirlp2mb(b,a,wo,wt)
[num,den,allpassNum,allpassDen] = iirlp2mb(b,a,wo,wt,pass)
```

## Description

`[num,den,allpassNum,allpassDen] = iirlp2mb(b,a,wo,wt)` transforms an IIR lowpass filter to an IIR multiband filter.

The `iirlp2mb` function returns the numerator and denominator coefficients of the transformed IIR multiband filter. The function also returns the numerator, `allpassNum`, and the denominator, `allpassDen`, of the  $M^{\text{th}}$  order allpass mapping filter. The prototype lowpass filter is specified with the numerator `b` and the denominator `a`.

The function transforms a real lowpass prototype filter to a multiband filter by applying an  $M^{\text{th}}$ -order real lowpass to real multiple bandpass frequency mapping. Parameter  $M$  is the number of times an original feature is replicated in the transformed filter. By default the DC feature is kept at its original location. For more details, see “IIR Lowpass Filter to IIR Multiband Filter Transformation” on page 5-979.

`[num,den,allpassNum,allpassDen] = iirlp2mb(b,a,wo,wt,pass)` allows you to specify an additional parameter, `pass` as 'pass' or 'stop', which chooses between using the “*Nyquist Mobility*” and the “*DC Mobility*”, respectively. In the case of “*Nyquist Mobility*”, the DC feature is kept at an original frequency and the Nyquist feature is free to move. In the case of “*DC Mobility*”, the Nyquist feature stays at its original location and the DC feature is free to move.

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

## Examples

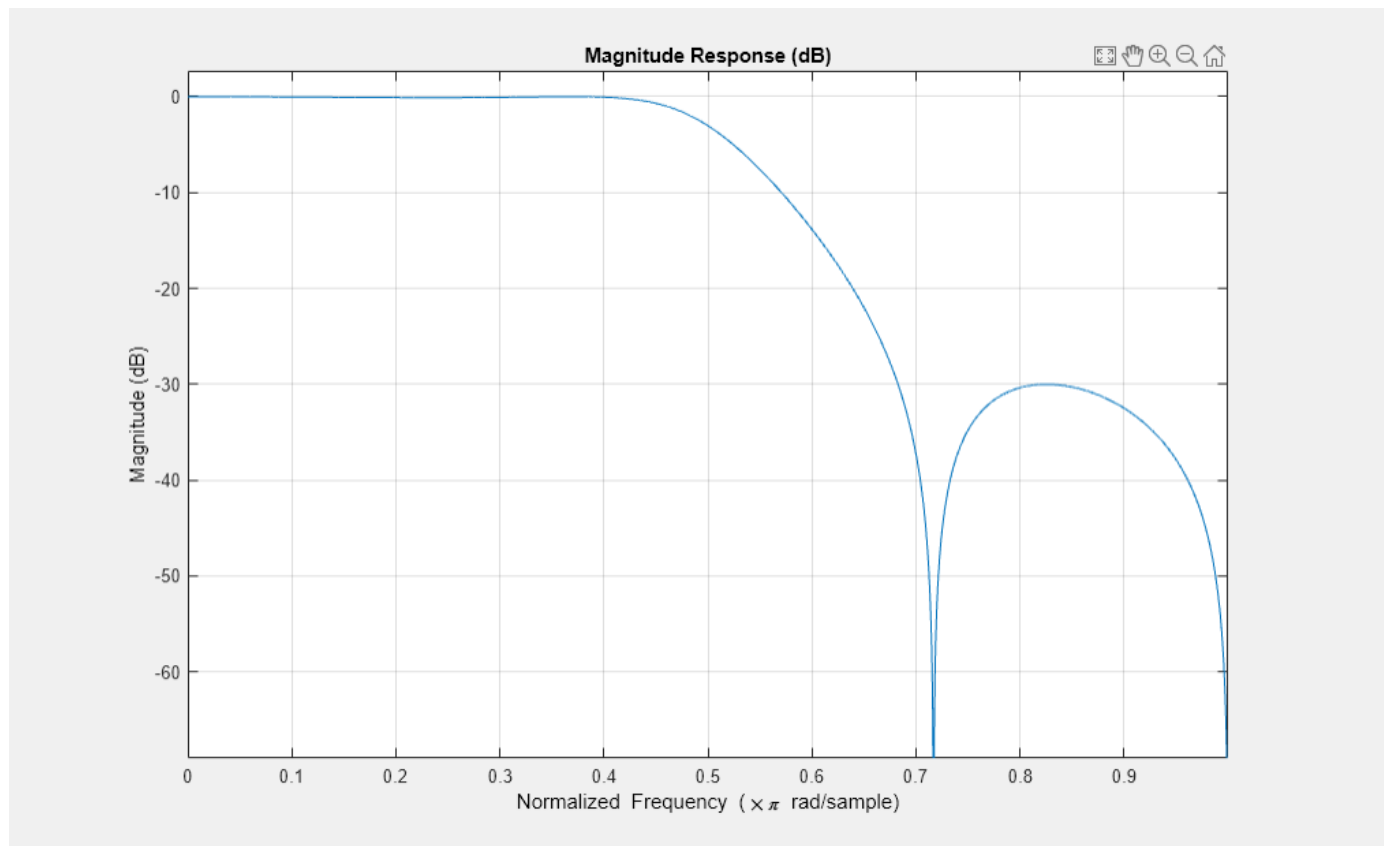
### Transform Lowpass Filter to Multiband Filter

Transform a lowpass IIR filter to a multiband IIR filter using the `iirlp2mb` function.

#### Input Lowpass IIR Filter

Design a prototype real IIR lowpass elliptic filter with a gain of about -3 dB at  $0.5\pi$  rad/sample.

```
[b,a] = ellip(3,0.1,30,0.409);
fvtool(b,a)
```



### Transform Filter Using `iirlp2mb`

Transform the real prototype lowpass filter into a real multiband filter with two passbands.

Specify the prototype filter as a vector of numerator and denominator coefficients, `b` and `a` respectively.

```
[num1,den1] = iirlp2mb(b,a,0.5,[2 4 6 8]/10);
```

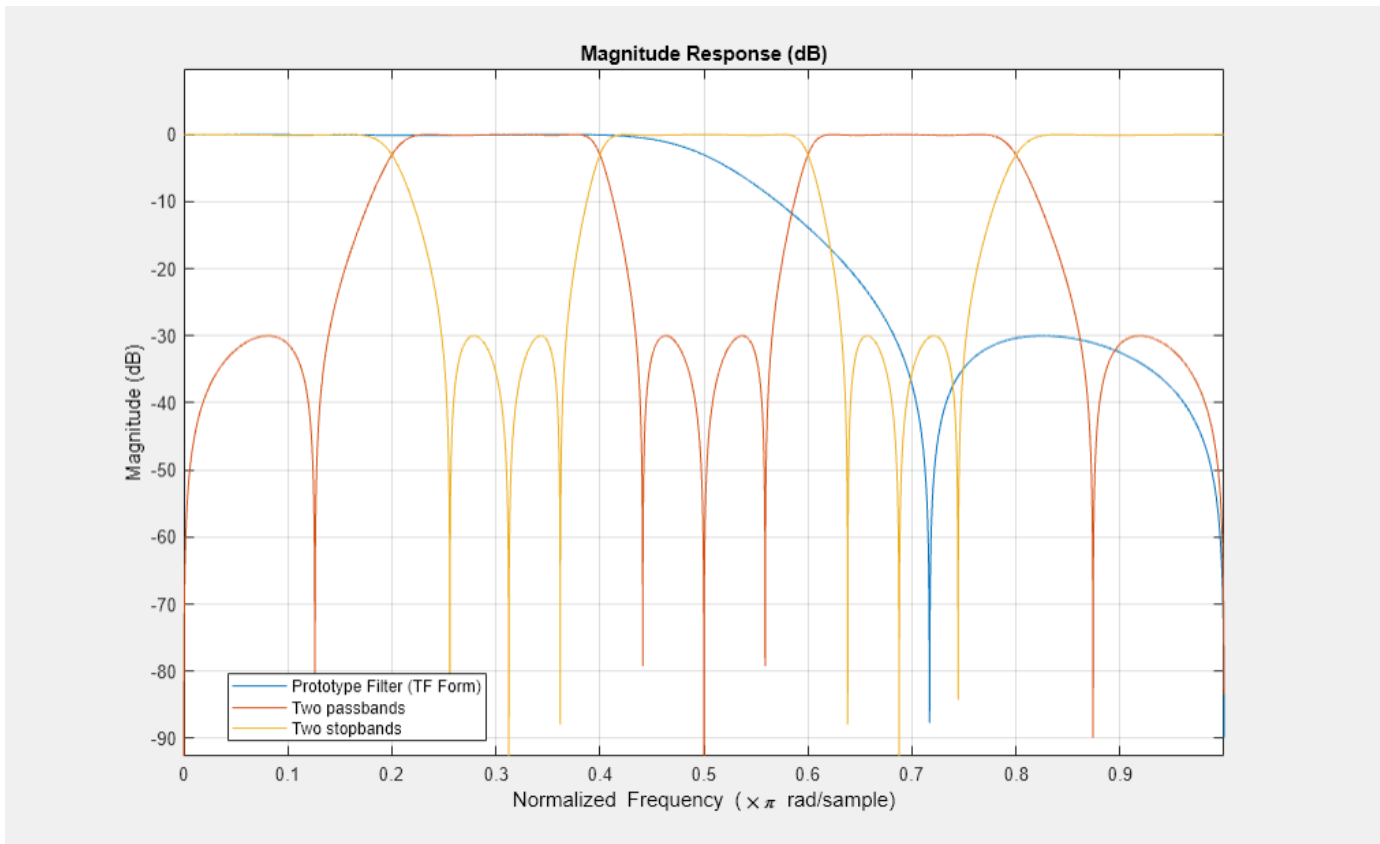
Transform the prototype filter into a real multiband filter with two stopbands.

Specify the prototype filter as a vector of numerator and denominator coefficients, `b` and `a` respectively.

```
[num2,den2] = iirlp2mb(b,a,0.5,[2 4 6 8]/10,"stop");
```

Compare the magnitude response of the filters using `FVTool`.

```
hvft = fvtool(b,a,num1,den1,num2,den2);
legend(hvft,"Prototype Filter (TF Form)",...
       "Two passbands","Two stopbands")
```



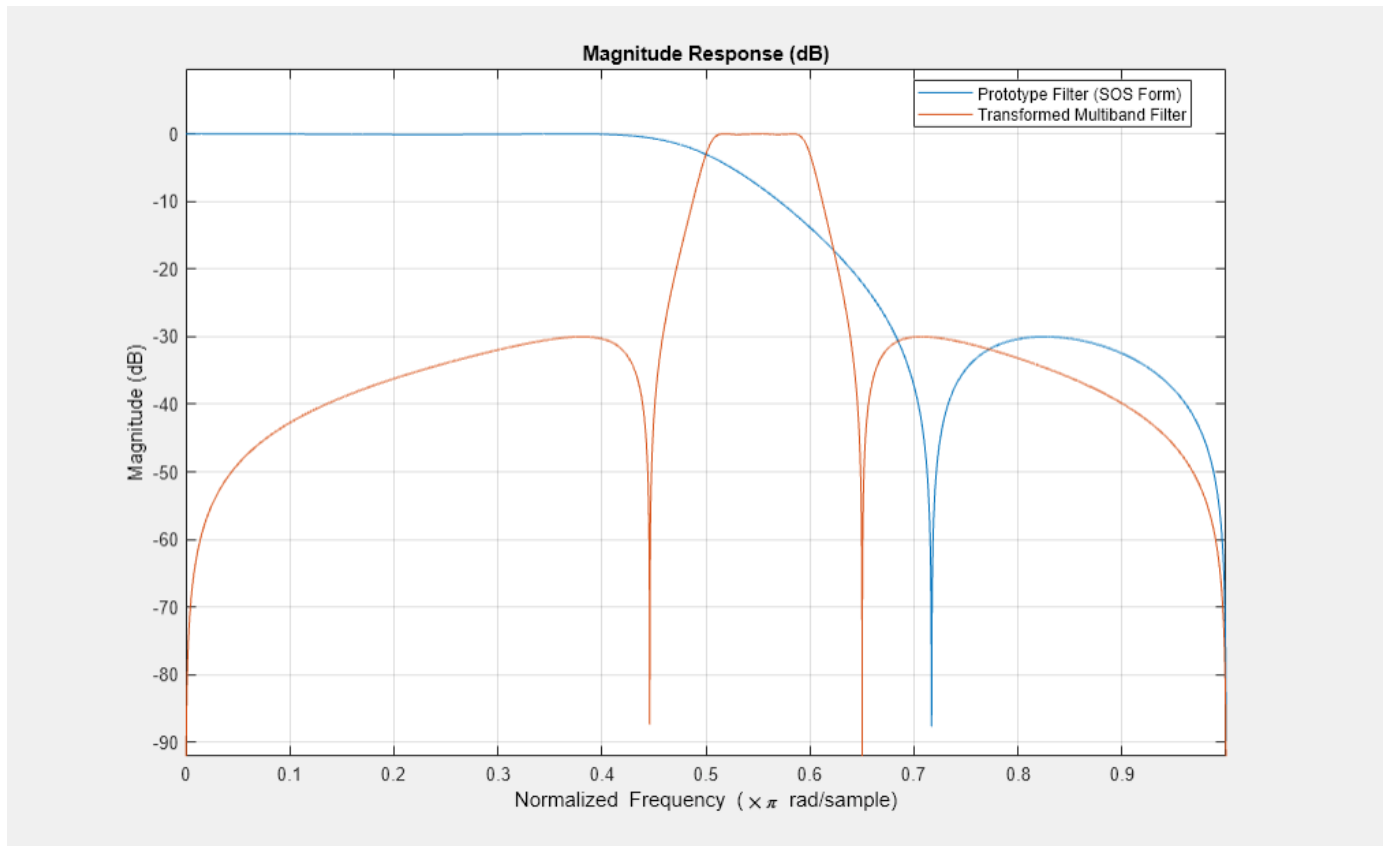
You can also specify the input lowpass IIR filter as a matrix of coefficients. Pass the second-order section matrices as inputs. Change the desired frequency locations to be [0.5 0.6] in normalized frequency units. The transformed 2-band filter is a fourth-order section filter given by the numerator and the denominator coefficients num3 and den3.

```
ss = tf2sos(b,a);
[num3, den3] = iirlp2mb(ss(:,1:3), ss(:,4:6), ...
    0.5, [5 6]/10);
```

Compare the magnitude response of the filters using FVTool.

To visualize the magnitude response of the fourth-order section filter, first pass the filter coefficients to the `dsp.FourthOrderSectionFilter` object. Then, use this object as an input to the FVTool.

```
fos = dsp.FourthOrderSectionFilter(Numerator=num3, ...
    Denominator=den3);
hvft = fvtool(ss, fos);
legend(hvft, "Prototype Filter (SOS Form)", ...
    "Transformed Multiband Filter")
```



Copyright 2012-2021 The MathWorks, Inc.

## Input Arguments

### **b** — Numerator coefficients of prototype lowpass IIR filter

row vector | matrix

Numerator coefficients of the prototype lowpass IIR filter, specified as either:

- Row vector -- Specifies the values of  $[b_0, b_1, \dots, b_n]$ , given this transfer function form:

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_0 + b_1z^{-1} + \dots + b_nz^{-n}}{a_0 + a_1z^{-1} + \dots + a_nz^{-n}}$$

where  $n$  is the order of the filter.

- Matrix -- Specifies the numerator coefficients in the form of an  $P$ -by- $(Q+1)$  matrix, where  $P$  is the number of filter sections and  $Q$  is the order of each filter section. If  $Q = 2$ , the filter is a second-order section filter. For higher-order sections, make  $Q > 2$ .

$$\mathbf{b} = \begin{bmatrix} b_{01} & b_{11} & b_{21} & \dots & b_{Q1} \\ b_{02} & b_{12} & b_{22} & \dots & b_{Q2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ b_{0P} & b_{1P} & b_{2P} & \dots & b_{QP} \end{bmatrix}$$



In the transfer function form, the numerator coefficient matrix  $b_{ik}$  of the IIR filter can be represented using the following equation:

$$H(z) = \prod_{k=1}^P H_k(z) = \prod_{k=1}^P \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2} + \dots + b_{Qk}z^{-Q}}{a_{0k} + a_{1k}z^{-1} + a_{2k}z^{-2} + \dots + a_{Qk}z^{-Q}},$$

where,

- **a** -- Denominator coefficients matrix. For more information on how to specify this matrix, see [a](#).
- $k$  -- Row index.
- $i$  -- Column index.

When specified in the matrix form, **b** and **a** matrices must have the same number of rows (filter sections)  $Q$ .

Data Types: `single` | `double`

### **a** — Denominator coefficients of prototype lowpass IIR filter

row vector | matrix

Denominator coefficients for a prototype lowpass IIR filter, specified as one of these options:

- Row vector -- Specifies the values of  $[a_0, a_1, \dots, a_n]$ , given this transfer function form:

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_0 + b_1z^{-1} + \dots + b_nz^{-n}}{a_0 + a_1z^{-1} + \dots + a_nz^{-n}},$$

where  $n$  is the order of the filter.

- Matrix -- Specifies the denominator coefficients in the form of an  $P$ -by- $(Q+1)$  matrix, where  $P$  is the number of filter sections and  $Q$  is the order of each filter section. If  $Q = 2$ , the filter is a second-order section filter. For higher-order sections, make  $Q > 2$ .

$$\mathbf{a} = \begin{bmatrix} a_{01} & a_{11} & a_{21} & \dots & a_{Q1} \\ a_{02} & a_{12} & a_{22} & \dots & a_{Q2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{0P} & a_{1P} & a_{2P} & \dots & a_{QP} \end{bmatrix}$$

In the transfer function form, the denominator coefficient matrix  $a_{ik}$  of the IIR filter can be represented using the following equation:

$$H(z) = \prod_{k=1}^P H_k(z) = \prod_{k=1}^P \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2} + \dots + b_{Qk}z^{-Q}}{a_{0k} + a_{1k}z^{-1} + a_{2k}z^{-2} + \dots + a_{Qk}z^{-Q}},$$

where,

- **b** -- Numerator coefficients matrix. For more information on how to specify this matrix, see [b](#).
- $k$  -- Row index.
- $i$  -- Column index.

When specified in the matrix form, **a** and **b** matrices must have the same number of rows (filter sections)  $P$ .

Data Types: `single` | `double`

**wo — Frequency value to transform from prototype filter**

positive scalar

Frequency value to transform from the prototype filter, specified as a real scalar. Frequency  $\omega_0$  must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

Data Types: `single` | `double`

**wt — Desired frequency locations in transformed target filter**

row vector

Desired frequency locations in the transformed target filter, specified as a row vector. Frequencies in **wt** must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate. The value of  $M$  which is equal to the number of times an original feature is replicated in the transformed filter equals the length of the **wt** vector.

Data Types: `single` | `double`

**pass — Choice of passband or stopband at DC**

"pass" (default) | "stop"

Choice of passband or stopband at DC, specified as either:

- "pass" -- "Nyquist Mobility". The DC feature is kept at an original frequency and the Nyquist feature is free to move.
- "stop" -- "DC Mobility". The Nyquist feature stays at its original location and the DC feature is free to move.

## Output Arguments

**num — Numerator coefficients of transformed multiband filter**

row vector | matrix

Numerator coefficients of the transformed multiband filter, returned as one of the following:

- Row vector of length  $Mn+1$ , where  $M$  is the number of times an original feature is replicated in the transformed filter and  $n$  is the order of the input filter. The value of  $M$  equals the length of the **wt** vector.

The **num** output is a row vector when the input coefficients **b** and **a** are row vectors.

- $P$ -by- $(QM+1)$  matrix, where  $P$  is the number of filter sections,  $Q$  is the order of each section of the transformed filter, and  $M$  is the number of times an original feature is replicated in the transformed filter.

The **num** output is a matrix when the input coefficients **b** and **a** are matrices.

Data Types: `single` | `double`

**den — Denominator coefficients of transformed multiband filter**

row vector | matrix

Denominator coefficients of the transformed multiband filter, returned as one of the following:

- Row vector of length  $Mn+1$ , where  $M$  is the number of times an original feature is replicated in the transformed filter and  $n$  is the order of the input filter. The value of  $M$  equals the length of the `wt` vector.

The `den` output is a row vector when the input coefficients `b` and `a` are row vectors.

- $P$ -by- $(QM+1)$  matrix, where  $P$  is the number of filter sections,  $Q$  is the order of each section of the transformed filter, and  $M$  is the number of times an original feature is replicated in the transformed filter.

The `den` output is a matrix when the input coefficients `b` and `a` are matrices.

Data Types: `single` | `double`

### **allpassNum — Numerator coefficients of mapping filter**

row vector

Numerator coefficients of the mapping filter, returned as a row vector.

Data Types: `single` | `double`

### **allpassDen — Denominator coefficients of mapping filter**

row vector

Denominator coefficients of the mapping filter, returned as a row vector.

Data Types: `single` | `double`

## **More About**

### **IIR Lowpass Filter to IIR Multiband Filter Transformation**

IIR lowpass filter to IIR multiband filter transformation effectively places one feature of the original filter, located at frequency  $w_o$ , at the required target frequency locations,  $w_{t1}, \dots, w_{tM}$ .

Relative positions of other features of the original filter do not change in the target filter. It is possible to select two features of an original filter,  $F_1$  and  $F_2$ , with  $F_1$  preceding  $F_2$ . Feature  $F_1$  will still precede  $F_2$  after the transformation. However, the distance between  $F_1$  and  $F_2$  will not be the same before and after the transformation.

Choice of the feature subject to this transformation is not restricted to the cutoff frequency of an original lowpass filter. You can choose to transform any feature of the original filter like stopband edge, DC, deep minimum in the stopband, or others.

The IIR lowpass filter to IIR multiband filter transformation can also be used to transform other types of filters, for example, notch filters or resonators can be easily replicated at a number of required frequency locations without redesigning them. A good application would be an adaptive tone cancellation circuit reacting to the changing number and location of tones.

## **Version History**

**Introduced in R2011a**

## References

- [1] Franchitti, Jean-Claude. "All-pass filter interpolation and frequency transformation problems." *MSc Thesis*, Dept. of Electrical and Computer Engineering, University of Colorado, 1985.
- [2] Feyh, G., J.C. Franchitti and C.T. Mullis. "All-pass filter interpolation and frequency transformation problem." *Proceedings 20th Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, California, pp. 164-168, November 1986.
- [3] Mullis, C.T. and R. A. Roberts. *Digital Signal Processing*, section 6.7, Reading, Mass., Addison-Wesley, 1987.
- [4] Feyh, G., W.B. Jones and C.T. Mullis. "An extension of the Schur Algorithm for frequency transformations." *Linear Circuits, Systems and Signal Processing: Theory and Application*. C. J. Byrnes et al Eds, Amsterdam: Elsevier, 1988.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

### Functions

`iirlp2mbc` | `iirftransf` | `allpasslp2mb` | `zpklp2mb`

## iirlp2mbc

Transform IIR lowpass filter to IIR complex multiband filter

### Syntax

```
[num,den,allpassNum,allpassDen] = iirlp2mbc(b,a,wo,wt)
```

### Description

`[num,den,allpassNum,allpassDen] = iirlp2mbc(b,a,wo,wt)` transform an IIR lowpass filter to an IIR complex multiband filter.

The `iirlp2mbc` returns the numerator and denominator coefficients of the transformed complex IIR multiband filter. The function also returns the numerator, `allpassNum`, and the denominator, `allpassDen`, of the  $M^{\text{th}}$  order allpass mapping filter. The prototype lowpass filter is specified with the numerator `b` and the denominator `a`.

The function transforms a real lowpass prototype filter to a multiband filter by applying an  $M^{\text{th}}$ -order real lowpass to complex multiple bandpass frequency mapping. Parameter  $M$  is the number of times an original feature is replicated in the transformed filter. For more details, see “IIR Lowpass Filter to IIR Complex Multiband Filter Transformation” on page 5-987.

### Examples

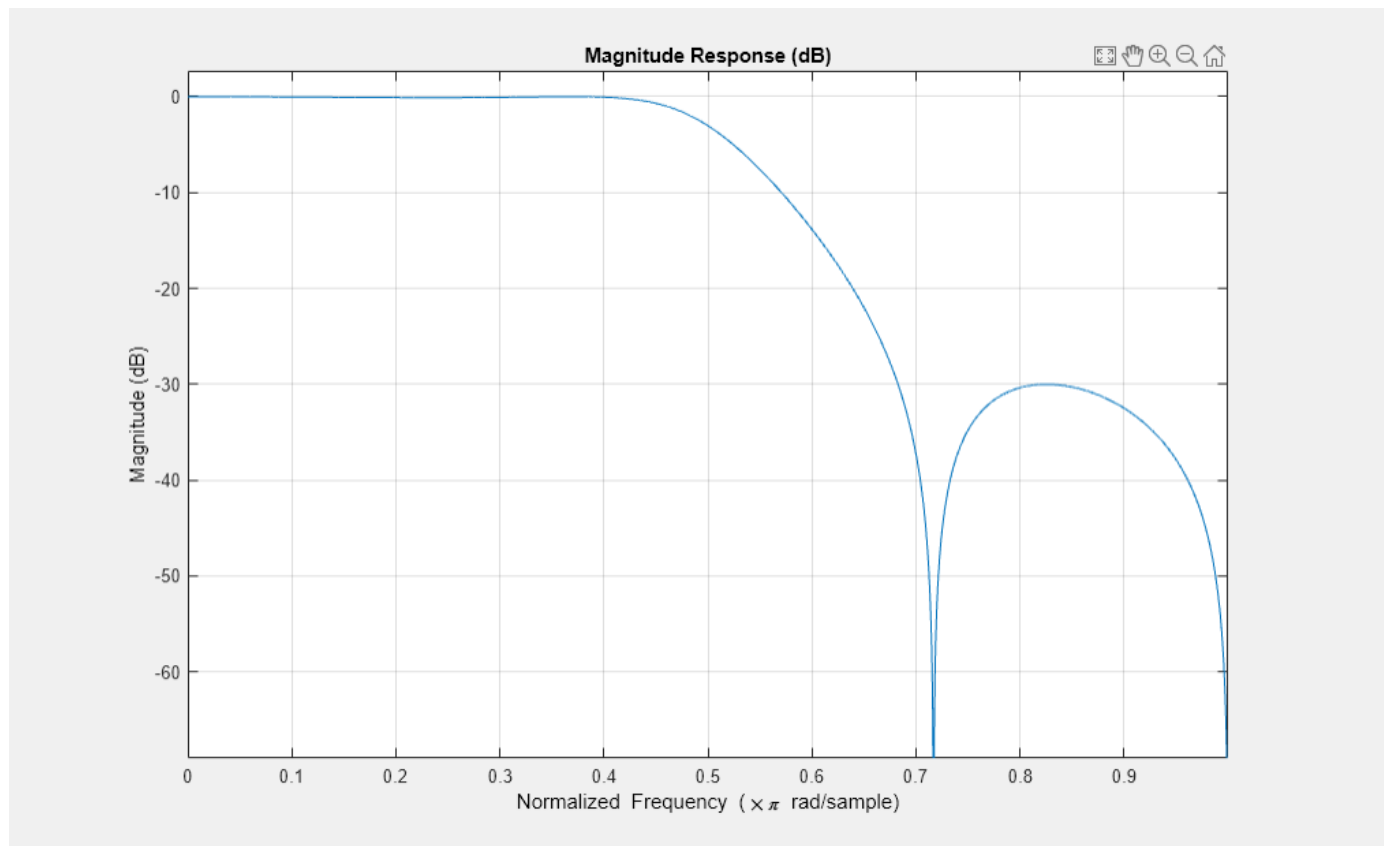
#### Transform Lowpass Filter to Complex Multiband Filter

Transform a real lowpass IIR filter to a complex multiband IIR filter using the `iirlp2mbc` function.

#### Input Lowpass IIR Filter

Design a prototype real IIR lowpass elliptic filter with a gain of about -3 dB at  $0.5\pi$  rad/sample.

```
[b,a] = ellip(3,0.1,30,0.409);  
fvtool(b,a)
```



### Transform Filter Using `iirlp2mbc`

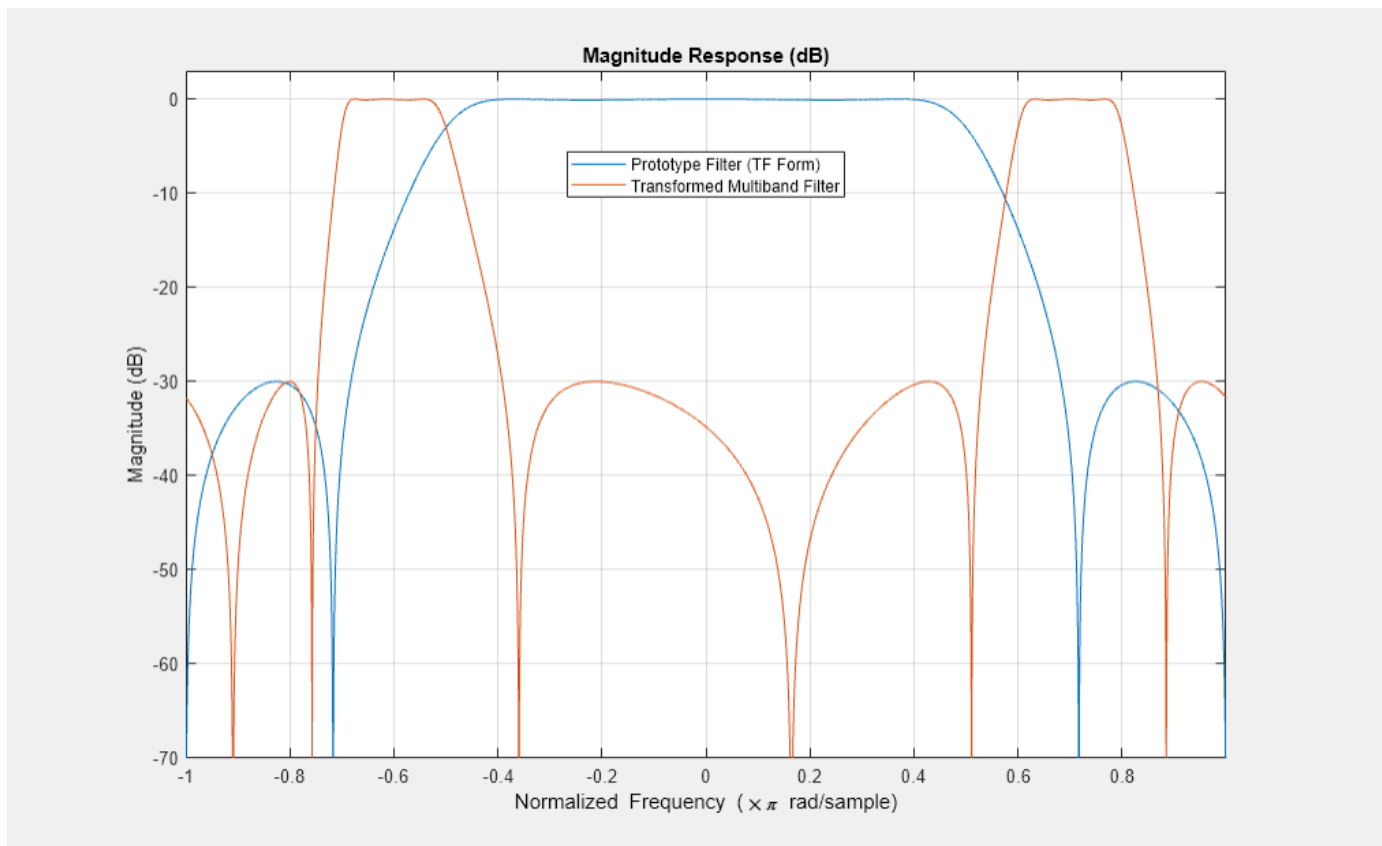
Transform the real prototype lowpass filter into a complex multiband filter with two passbands.

Specify the prototype filter as a vector of numerator and denominator coefficients, `b` and `a` respectively.

```
[num,den] = iirlp2mbc(b,a,0.5,[-7 -5 6 8]/10);
```

Compare the magnitude response of the filters using `FVTool`. `iirlp2mbc` replicates the desired feature at 0.5 in the lowpass filter at four locations in the multiband filter.

```
hvft = fvtool(b,a,num,den);
legend(hvft,"Prototype Filter (TF Form)",...
       "Transformed Multiband Filter")
```



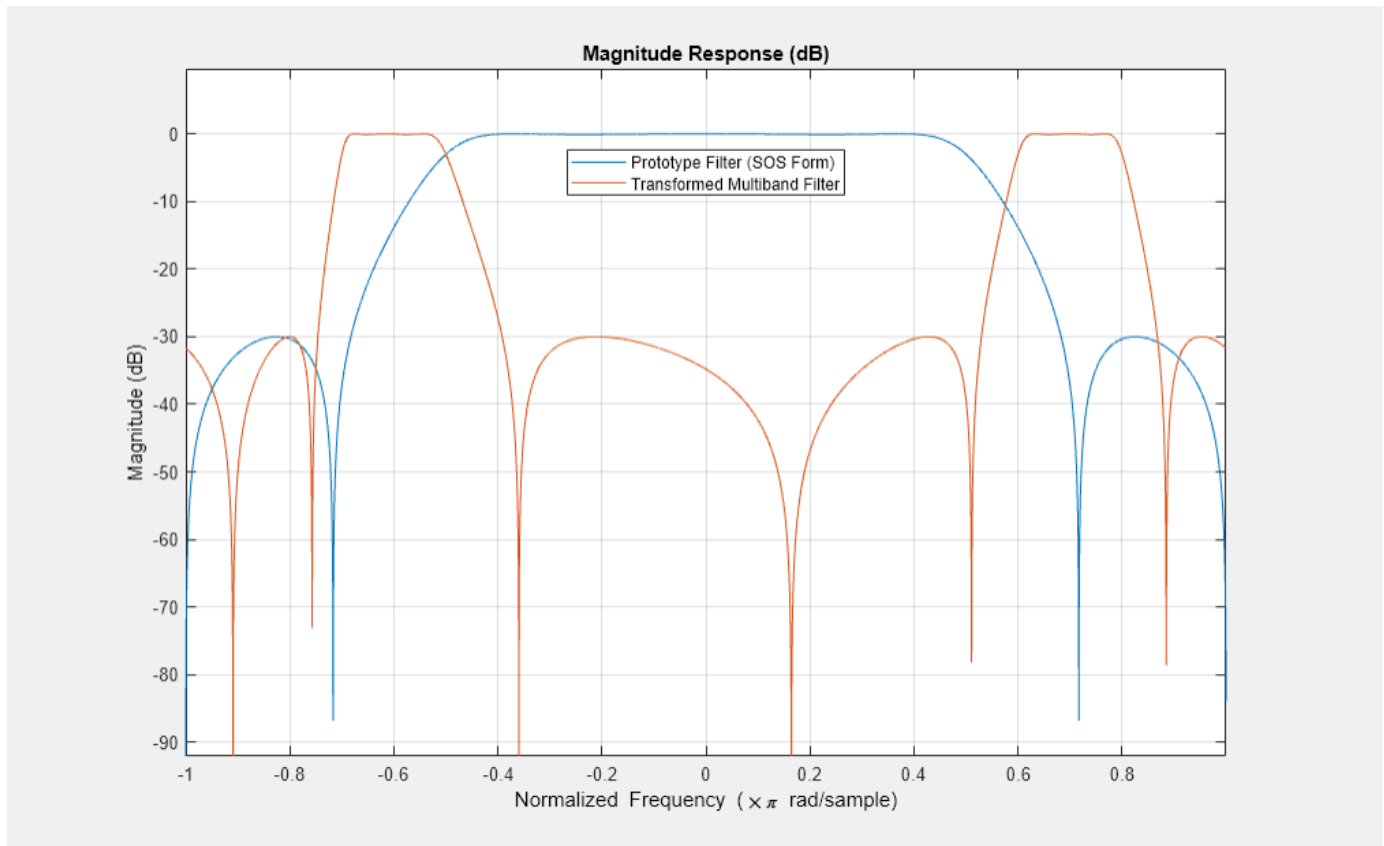
Alternatively, you can also specify the input lowpass IIR filter as a matrix of coefficients. Pass the second-order section matrices as inputs. The transformed filter is a fourth-order section filter given by the numerator and the denominator coefficients `num2` and `den2`.

```
ss = tf2sos(b,a);
[num2,den2] = iirlp2mbc(ss(:,1:3),ss(:,4:6),0.5,[-7 -5 6 8]/10);
```

Compare the magnitude response of the filters using FVTool.

To visualize the magnitude response of the fourth-order section filter, first pass the filter coefficients to the `dsp.FourthOrderSectionFilter` object. Then, use this object as an input to the FVTool.

```
fos = dsp.FourthOrderSectionFilter(Numerator=num2,...
    Denominator=den2);
hvft = fvtool(ss,fos);
legend(hvft,"Prototype Filter (SOS Form)",...
    "Transformed Multiband Filter")
```



Copyright 2012-2021 The MathWorks, Inc.

## Input Arguments

### **b** — Numerator coefficients of prototype lowpass IIR filter

row vector | matrix

Numerator coefficients of the prototype lowpass IIR filter, specified as either:

- Row vector -- Specifies the values of  $[b_0, b_1, \dots, b_n]$ , given this transfer function form:

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_0 + b_1z^{-1} + \dots + b_nz^{-n}}{a_0 + a_1z^{-1} + \dots + a_nz^{-n}}$$

where  $n$  is the order of the filter.

- Matrix -- Specifies the numerator coefficients in the form of an  $P$ -by- $(Q+1)$  matrix, where  $P$  is the number of filter sections and  $Q$  is the order of each filter section. If  $Q = 2$ , the filter is a second-order section filter. For higher-order sections, make  $Q > 2$ .

$$\mathbf{b} = \begin{bmatrix} b_{01} & b_{11} & b_{21} & \dots & b_{Q1} \\ b_{02} & b_{12} & b_{22} & \dots & b_{Q2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ b_{0P} & b_{1P} & b_{2P} & \dots & b_{QP} \end{bmatrix}$$



In the transfer function form, the numerator coefficient matrix  $b_{ik}$  of the IIR filter can be represented using the following equation:

$$H(z) = \prod_{k=1}^P H_k(z) = \prod_{k=1}^P \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2} + \dots + b_{Qk}z^{-Q}}{a_{0k} + a_{1k}z^{-1} + a_{2k}z^{-2} + \dots + a_{Qk}z^{-Q}},$$

where,

- **a** -- Denominator coefficients matrix. For more information on how to specify this matrix, see [a](#).
- $k$  -- Row index.
- $i$  -- Column index.

When specified in the matrix form, **b** and **a** matrices must have the same number of rows (filter sections)  $Q$ .

Data Types: `single` | `double`

### **a** — Denominator coefficients of prototype lowpass IIR filter

row vector | matrix

Denominator coefficients for a prototype lowpass IIR filter, specified as one of these options:

- Row vector -- Specifies the values of  $[a_0, a_1, \dots, a_n]$ , given this transfer function form:

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_0 + b_1z^{-1} + \dots + b_nz^{-n}}{a_0 + a_1z^{-1} + \dots + a_nz^{-n}},$$

where  $n$  is the order of the filter.

- Matrix -- Specifies the denominator coefficients in the form of an  $P$ -by- $(Q+1)$  matrix, where  $P$  is the number of filter sections and  $Q$  is the order of each filter section. If  $Q = 2$ , the filter is a second-order section filter. For higher-order sections, make  $Q > 2$ .

$$\mathbf{a} = \begin{bmatrix} a_{01} & a_{11} & a_{21} & \dots & a_{Q1} \\ a_{02} & a_{12} & a_{22} & \dots & a_{Q2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{0P} & a_{1P} & a_{2P} & \dots & a_{QP} \end{bmatrix}$$

In the transfer function form, the denominator coefficient matrix  $a_{ik}$  of the IIR filter can be represented using the following equation:

$$H(z) = \prod_{k=1}^P H_k(z) = \prod_{k=1}^P \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2} + \dots + b_{Qk}z^{-Q}}{a_{0k} + a_{1k}z^{-1} + a_{2k}z^{-2} + \dots + a_{Qk}z^{-Q}},$$

where,

- **b** -- Numerator coefficients matrix. For more information on how to specify this matrix, see [b](#).
- $k$  -- Row index.
- $i$  -- Column index.

When specified in the matrix form, **a** and **b** matrices must have the same number of rows (filter sections)  $P$ .

Data Types: `single` | `double`

### **wo — Frequency value to transform from prototype filter**

positive scalar

Frequency value to transform from the prototype filter, specified as a positive scalar. Frequency  $w_0$  should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

Data Types: `single` | `double`

### **wt — Desired frequency locations in transformed target filter**

row vector

Desired frequency locations in the transformed target filter, specified as a row vector. Frequencies in **wt** should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate. The value of  $M$  which is equal to the number of times an original feature is replicated in the transformed filter equals the length of the **wt** vector.

Data Types: `single` | `double`

## **Output Arguments**

### **num — Numerator coefficients of transformed multiband filter**

row vector | matrix

Numerator coefficients of the transformed multiband filter, returned as one of the following:

- Row vector of length  $Mn/2+1$ , where  $M$  is the number of times an original feature is replicated in the transformed filter and  $n$  is the order of the input filter. The value of  $M$  equals the length of the **wt** vector.

The **num** output is a row vector when the input coefficients **b** and **a** are row vectors.

- $P$ -by- $(QM/2+1)$  matrix, where  $P$  is the number of filter sections,  $Q$  is the order of each section of the transformed filter, and  $M$  is the number of times an original feature is replicated in the transformed filter.

The **num** output is a matrix when the input coefficients **b** and **a** are matrices.

Data Types: `single` | `double`

### **den — Denominator coefficients of transformed multiband filter**

row vector | matrix

Denominator coefficients of the transformed multiband filter, returned as one of the following:

- Row vector of length  $Mn/2+1$ , where  $M$  is the number of times an original feature is replicated in the transformed filter and  $n$  is the order of the input filter. The value of  $M$  equals the length of the **wt** vector.

The **den** output is a row vector when the input coefficients **b** and **a** are row vectors.

- $P$ -by- $(QM/2+1)$  matrix, where  $P$  is the number of filter sections,  $Q$  is the order of each section of the transformed filter, and  $M$  is the number of times an original feature is replicated in the transformed filter.

The den output is a matrix when the input coefficients  $b$  and  $a$  are matrices.

Data Types: `single` | `double`

### **allpassNum — Numerator coefficients of mapping filter**

row vector

Numerator coefficients of the mapping filter, returned as a row vector.

Data Types: `single` | `double`

Complex Number Support: Yes

### **allpassDen — Denominator coefficients of mapping filter**

row vector

Denominator coefficients of the mapping filter, returned as a row vector.

Data Types: `single` | `double`

Complex Number Support: Yes

## **More About**

### **IIR Lowpass Filter to IIR Complex Multiband Filter Transformation**

IIR lowpass filter to IIR complex multiband filter transformation effectively places one feature of the original filter, located at frequency  $w_o$ , at the required target frequency locations,  $w_{t1}, \dots, w_{tM}$ .

Relative positions of other features of the original filter do not change in the target filter. This means that it is possible to select two features of an original filter,  $F_1$  and  $F_2$ , with  $F_1$  preceding  $F_2$ . Feature  $F_1$  will still precede  $F_2$  after the transformation. However, the distance between  $F_1$  and  $F_2$  will not be the same before and after the transformation.

Choice of the feature subject to this transformation is not restricted to the cutoff frequency of an original lowpass filter. You can choose to transform any feature of the original filter like stopband edge, DC, deep minimum in the stopband, or others.

The IIR lowpass filter to IIR complex multiband filter transformation can also be used for transforming other types of filters, for example, notch filters or resonators can be easily replicated at a number of required frequency locations. A good application would be an adaptive tone cancellation circuit reacting to the changing number and location of tones.

## **Version History**

**Introduced in R2011a**

## **References**

- [1] Krukowski, A., and I. Kale. "High-Order Complex Frequency Transformations," *Internal report No. 27/2001*. Applied DSP and VLSI Research Group, University of Westminster.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

## **See Also**

### **Functions**

`iirlp2mb` | `iirftransf` | `allpasslp2mbc` | `zpklp2mbc`

## iirlp2xc

Transform IIR lowpass filter to IIR complex  $N$ -point filter

### Syntax

```
[num,den,allpassNum,allpassDen] = iirlp2xc(b,a,wo,wt)
```

### Description

`[num,den,allpassNum,allpassDen] = iirlp2xc(b,a,wo,wt)` transforms an IIR lowpass filter to an IIR complex  $N$ -point filter.

The prototype lowpass filter is specified with the numerator `b` and the denominator `a`. The function returns the numerator and the denominator coefficients of the transformed IIR complex  $N$ -point filter. The function also returns the numerator, `allpassNum`, and the denominator, `allpassDen`, of the allpass mapping filter.

The function transforms a real lowpass prototype filter to an  $N$ -point complex filter by applying an  $N^{\text{th}}$ -order real lowpass to complex multipoint frequency transformation, where  $N$  is the number of features being mapped and is given by the length of the vector `wo`. For more details, see “IIR Lowpass Filter to IIR Complex  $N$ -Point Filter Transformation” on page 5-995.

### Examples

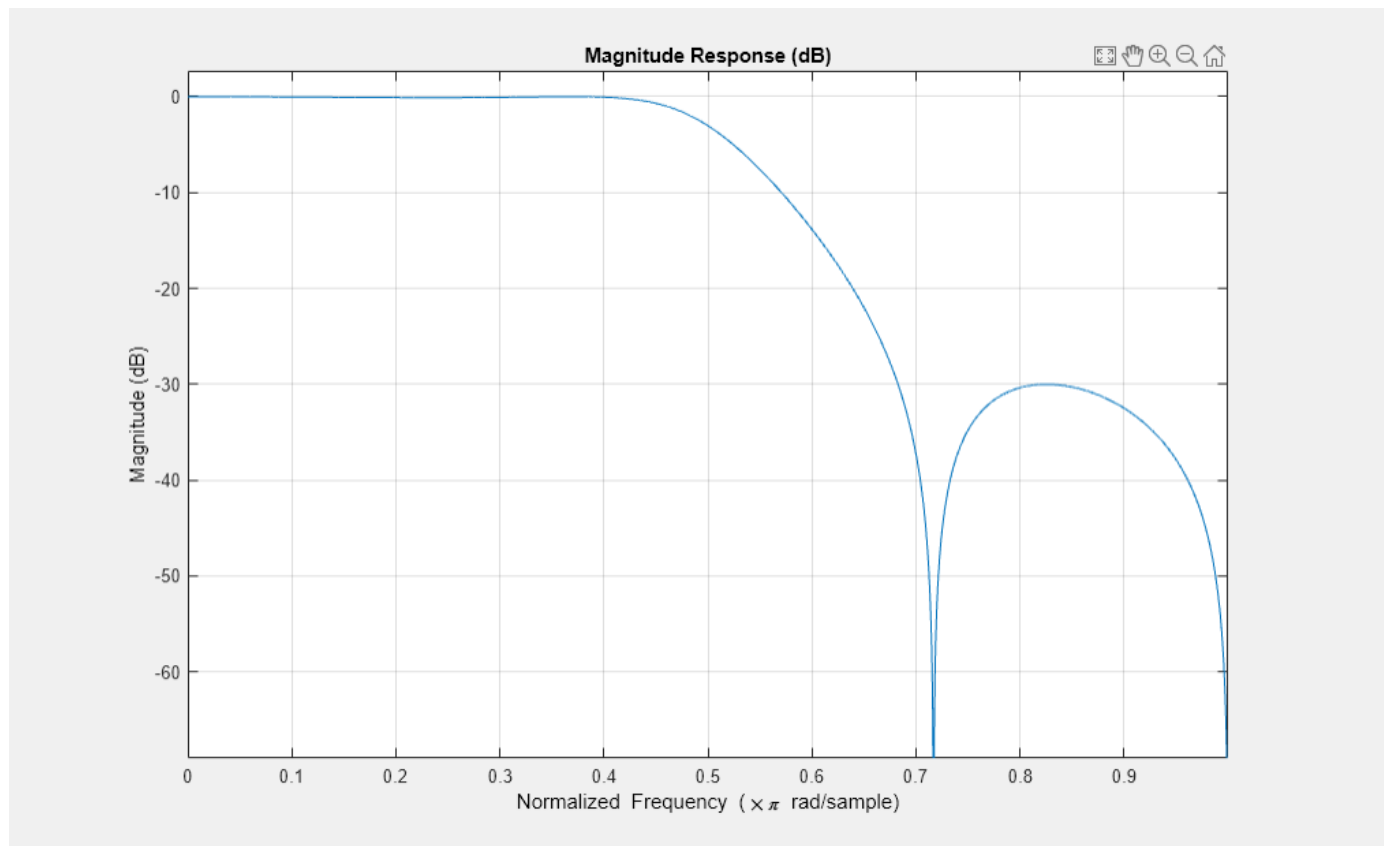
#### Transform Lowpass Filter to IIR Complex $N$ -Point Filter

Transform a lowpass IIR filter to an  $N$ -point IIR complex filter using the `iirlp2xc` function.

#### Input Lowpass IIR Filter

Design a prototype real IIR lowpass elliptic filter with a gain of about -3 dB at  $0.5\pi$  rad/sample.

```
[b,a] = ellip(3,0.1,30,0.409);  
fvtool(b,a)
```



### Transform Filter Using `iirlp2xc`

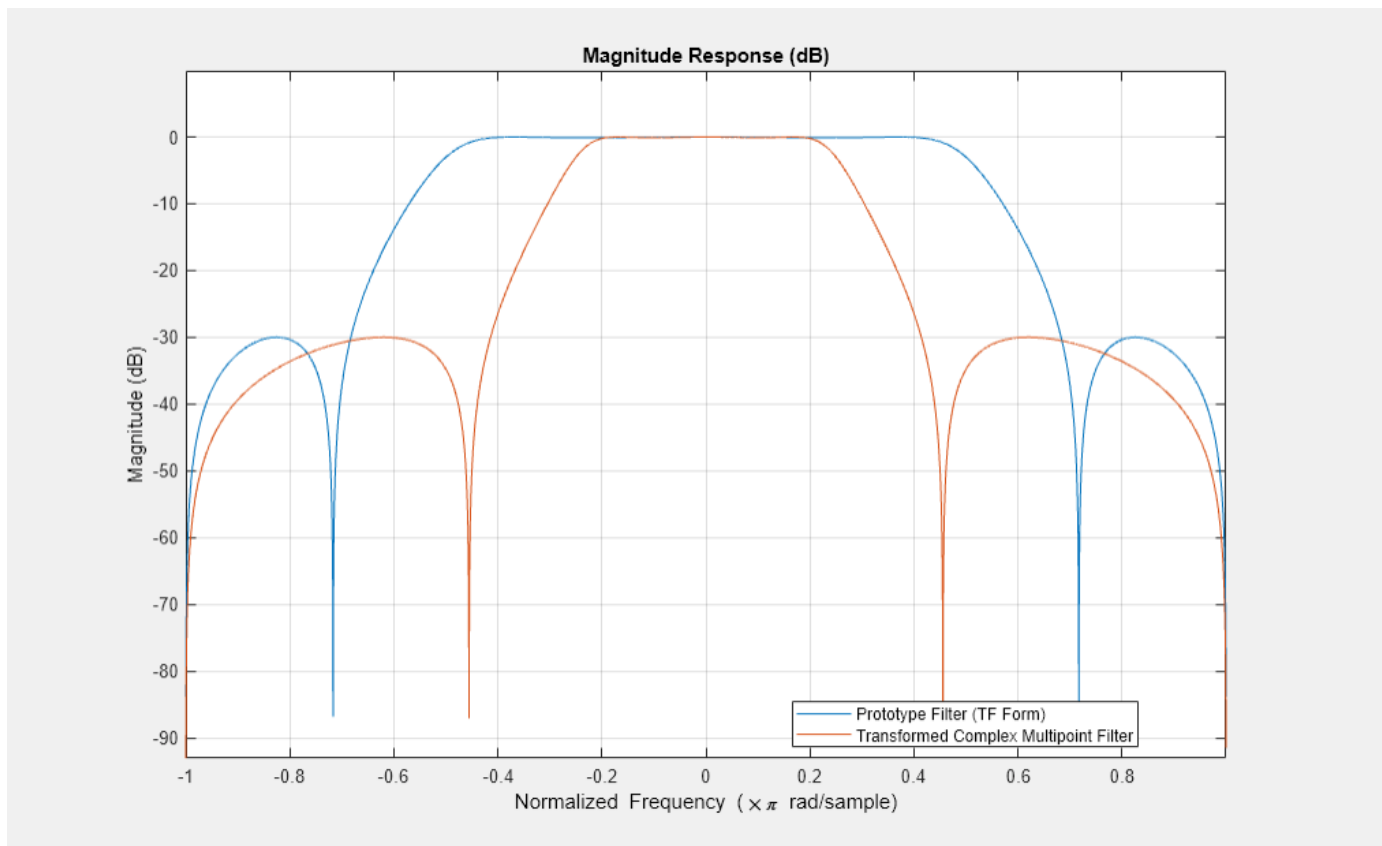
Transform the real prototype lowpass filter to an IIR complex  $N$ -point filter.

Specify the prototype filter as a vector of numerator and denominator coefficients, `b` and `a` respectively.

```
[num,den] = iirlp2xc(b,a,[-0.5 0.5],[-0.25 0.25]);
```

Compare the magnitude response of the filters using `FVTool`. The transformed filter has complex coefficients and is indeed a bandpass filter.

```
fvt = fvtool(b,a,num,den);
legend(fvt,"Prototype Filter (TF Form)",...
      "Transformed Complex Multipoint Filter")
```

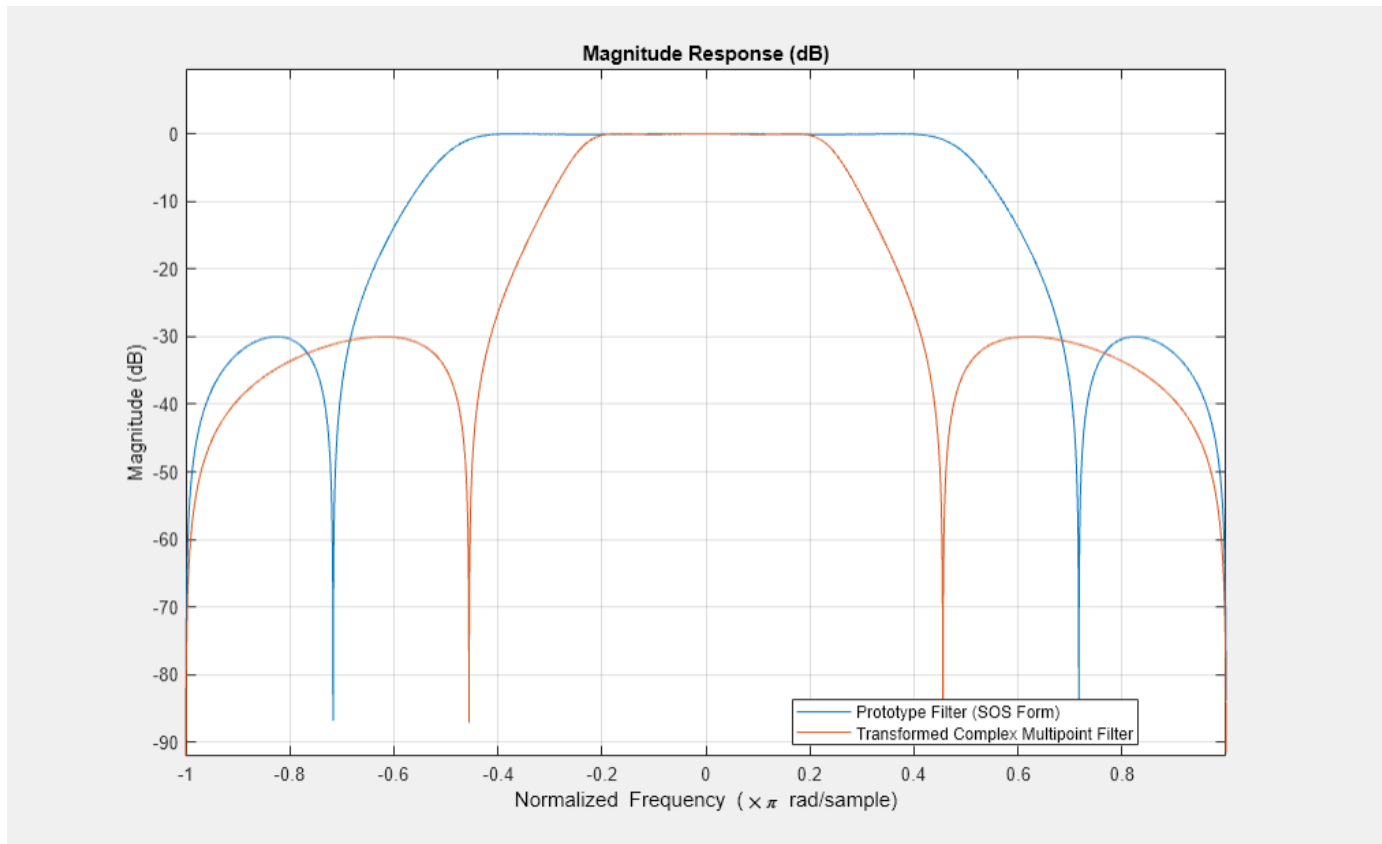


Alternatively, you can also specify the input lowpass IIR filter as a matrix of coefficients. Pass the second-order section matrices as inputs. The numerator and the denominator coefficients of the transformed filter are given by `num2` and `den2`, respectively.

```
ss = tf2sos(b,a);
[num2,den2] = iirlp2xc(ss(:,1:3),ss(:,4:6),...
    [-0.5 0.5],[-0.25 0.25]);
```

Compare the magnitude response of the filters using `FVTool`.

```
hvft = fvtool(ss,[num2 den2]);
legend(hvft,"Prototype Filter (SOS Form)",...
    "Transformed Complex Multipoint Filter")
```



Copyright 2012-2021 The MathWorks, Inc.

## Input Arguments

### **b** — Numerator coefficients of prototype lowpass IIR filter

row vector | matrix

Numerator coefficients of the prototype lowpass IIR filter, specified as either:

- Row vector -- Specifies the values of  $[b_0, b_1, \dots, b_n]$ , given this transfer function form:

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_0 + b_1z^{-1} + \dots + b_nz^{-n}}{a_0 + a_1z^{-1} + \dots + a_nz^{-n}},$$

where  $n$  is the order of the filter.

- Matrix -- Specifies the numerator coefficients in the form of an  $P$ -by- $(Q+1)$  matrix, where  $P$  is the number of filter sections and  $Q$  is the order of each filter section. If  $Q = 2$ , the filter is a second-order section filter. For higher-order sections, make  $Q > 2$ .

$$\mathbf{b} = \begin{bmatrix} b_{01} & b_{11} & b_{21} & \dots & b_{Q1} \\ b_{02} & b_{12} & b_{22} & \dots & b_{Q2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ b_{0P} & b_{1P} & b_{2P} & \dots & b_{QP} \end{bmatrix}$$



In the transfer function form, the numerator coefficient matrix  $b_{ik}$  of the IIR filter can be represented using the following equation:

$$H(z) = \prod_{k=1}^P H_k(z) = \prod_{k=1}^P \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2} + \dots + b_{Qk}z^{-Q}}{a_{0k} + a_{1k}z^{-1} + a_{2k}z^{-2} + \dots + a_{Qk}z^{-Q}},$$

where,

- **a** -- Denominator coefficients matrix. For more information on how to specify this matrix, see [a](#).
- $k$  -- Row index.
- $i$  -- Column index.

When specified in the matrix form, **b** and **a** matrices must have the same number of rows (filter sections)  $Q$ .

Data Types: `single` | `double`

### **a** — Denominator coefficients of prototype lowpass IIR filter

row vector | matrix

Denominator coefficients for a prototype lowpass IIR filter, specified as one of these options:

- Row vector -- Specifies the values of  $[a_0, a_1, \dots, a_n]$ , given this transfer function form:

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_0 + b_1z^{-1} + \dots + b_nz^{-n}}{a_0 + a_1z^{-1} + \dots + a_nz^{-n}},$$

where  $n$  is the order of the filter.

- Matrix -- Specifies the denominator coefficients in the form of an  $P$ -by- $(Q+1)$  matrix, where  $P$  is the number of filter sections and  $Q$  is the order of each filter section. If  $Q = 2$ , the filter is a second-order section filter. For higher-order sections, make  $Q > 2$ .

$$\mathbf{a} = \begin{bmatrix} a_{01} & a_{11} & a_{21} & \dots & a_{Q1} \\ a_{02} & a_{12} & a_{22} & \dots & a_{Q2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{0P} & a_{1P} & a_{2P} & \dots & a_{QP} \end{bmatrix}$$

In the transfer function form, the denominator coefficient matrix  $a_{ik}$  of the IIR filter can be represented using the following equation:

$$H(z) = \prod_{k=1}^P H_k(z) = \prod_{k=1}^P \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2} + \dots + b_{Qk}z^{-Q}}{a_{0k} + a_{1k}z^{-1} + a_{2k}z^{-2} + \dots + a_{Qk}z^{-Q}},$$

where,

- **b** -- Numerator coefficients matrix. For more information on how to specify this matrix, see [b](#).
- $k$  -- Row index.
- $i$  -- Column index.

When specified in the matrix form, **a** and **b** matrices must have the same number of rows (filter sections)  $P$ .

Data Types: `single` | `double`

**wo** — Frequency values to transform from prototype filter

row vector

Frequency values to transform from the prototype filter, specified as a row vector with even number of elements. Frequencies in **wo** should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate. The value of  $N$  equals the length of the **wo** vector. Length of vectors **wo** and **wt** must be the same.

Data Types: `single` | `double`

**wt** — Desired frequency locations in transformed target filter

row vector

Desired frequency locations in the transformed target filter, specified as a row vector with even number of elements. Frequencies in **wt** should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

---

**Note** Length of **wo** and **wt** vectors must be the same.

---

Data Types: `single` | `double`

## Output Arguments

**num** — Numerator coefficients of transformed multipoint filter

row vector | matrix

Numerator coefficients of the transformed multipoint filter, returned as one of the following:

- Row vector of length  $Nn/2+1$ , where  $n$  is the order of the input filter and  $N$  is the number of features being mapped. The value of  $N$  equals the length of the **wo** vector.

The **num** output is a row vector when the input coefficients **b** and **a** are row vectors.

- $P$ -by- $(QN/2+1)$  matrix, where  $P$  is the number of filter sections,  $Q$  is the order of each section of the transformed filter, and  $N$  is the number of features being mapped and is given by the length of the vector **wo**.

The **num** output is a matrix when the input coefficients **b** and **a** are matrices.

Data Types: `single` | `double`

**den** — Denominator coefficients of transformed multipoint filter

row vector | matrix

Denominator coefficients of the transformed multipoint filter, returned as one of the following:

- Row vector of length  $Nn/2+1$ , where  $n$  is the order of the input filter and  $N$  is the number of features being mapped. The value of  $N$  equals the length of the **wo** vector.

The **den** output is a row vector when the input coefficients **b** and **a** are row vectors.

- $P$ -by- $(QN/2+1)$  matrix, where  $P$  is the number of filter sections,  $Q$  is the order of each section of the transformed filter, and  $N$  is the number of features being mapped and is given by the length of the vector  $w_0$ .

The den output is a matrix when the input coefficients  $b$  and  $a$  are matrices.

Data Types: `single` | `double`

### **allpassNum — Numerator coefficients of mapping filter**

row vector

Numerator coefficients of the mapping filter, returned as a row vector.

Data Types: `single` | `double`

Complex Number Support: Yes

### **allpassDen — Denominator coefficients of mapping filter**

row vector

Denominator coefficients of the mapping filter, returned as a row vector.

Data Types: `single` | `double`

Complex Number Support: Yes

## **More About**

### **IIR Lowpass Filter to IIR Complex $N$ -Point Filter Transformation**

IIR lowpass filter to IIR complex  $N$ -point filter transformation effectively places  $N$  features of the original filter, located at frequencies  $w_{o1}, \dots, w_{oN}$ , at the required target frequency locations,  $w_{t1}, \dots, w_{tM}$ . The function `iirlp2xc` requires that  $N$  and  $M$  are equal.

Relative positions of other features of the original filter are the same in the target filter for the Nyquist mobility and are reversed for the DC mobility. For the Nyquist mobility this means that it is possible to select two features of an original filter,  $F_1$  and  $F_2$ , with  $F_1$  preceding  $F_2$ . Feature  $F_1$  will still precede  $F_2$  after the transformation. However, the distance between  $F_1$  and  $F_2$  will not be the same before and after the transformation. For DC mobility feature  $F_2$  will precede  $F_1$  after the transformation.

Choice of the feature subject to this transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., a stopband edge, DC, the deep minimum in the stopband, or other ones. Select features such that there is no band overlap when creating  $N$  bands around the unit circle.

IIR lowpass filter to IIR complex  $N$ -point filter transformation can also be used to transform other types of filters, for example, notch filters or resonators can be easily replicated at a number of required frequency locations. A good application would be an adaptive tone cancellation circuit reacting to the changing number and location of tones.

## **Version History**

**Introduced in R2011a**

## References

- [1] Krukowski, A., and I. Kale, "High-order complex frequency transformations," *Internal report No. 27/2001*, Applied DSP and VLSI Research Group, University of Westminster.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

### Functions

`iirfttransf` | `allpasslp2xc` | `zpklp2xc`

## iirlp2xn

Transform IIR lowpass filter to IIR real  $N$ -point filter

### Syntax

```
[num,den,allpassNum,allpassDen] = iirlp2xn(b,a,wo,wt)
[num,den,allpassNum,allpassDen] = iirlp2xn(b,a,wo,wt,pass)
```

### Description

`[num,den,allpassNum,allpassDen] = iirlp2xn(b,a,wo,wt)` transforms an IIR lowpass filter to an IIR real  $N$ -point filter.

The `iirlp2xn` function returns the numerator and denominator coefficients of the transformed IIR  $N$ -point filter. The function also returns the numerator, `allpassNum`, and the denominator, `allpassDen`, of the  $N^{\text{th}}$  order allpass mapping filter. The prototype lowpass filter is specified with the numerator `b` and the denominator `a`.

The function transforms a real lowpass prototype filter to an  $N$ -point filter by applying an  $N^{\text{th}}$ -order real lowpass to real multipoint frequency transformation, where  $N$  is the number of features being mapped and is given by the length of the vector `wo`. By default, the DC feature is kept at its original location. For more details, see “IIR Lowpass to IIR Real  $N$ -Point Filter Transformation” on page 5-1003.

`[num,den,allpassNum,allpassDen] = iirlp2xn(b,a,wo,wt,pass)` allows you to specify an additional parameter, `pass` as 'pass' or 'stop', which chooses between using the “*Nyquist Mobility*” and the “*DC Mobility*”, respectively. In the case of “*Nyquist Mobility*”, the DC feature is kept at an original frequency and the Nyquist feature is free to move. In the case of “*DC Mobility*”, the Nyquist feature stays at its original location and the DC feature is free to move.

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

### Examples

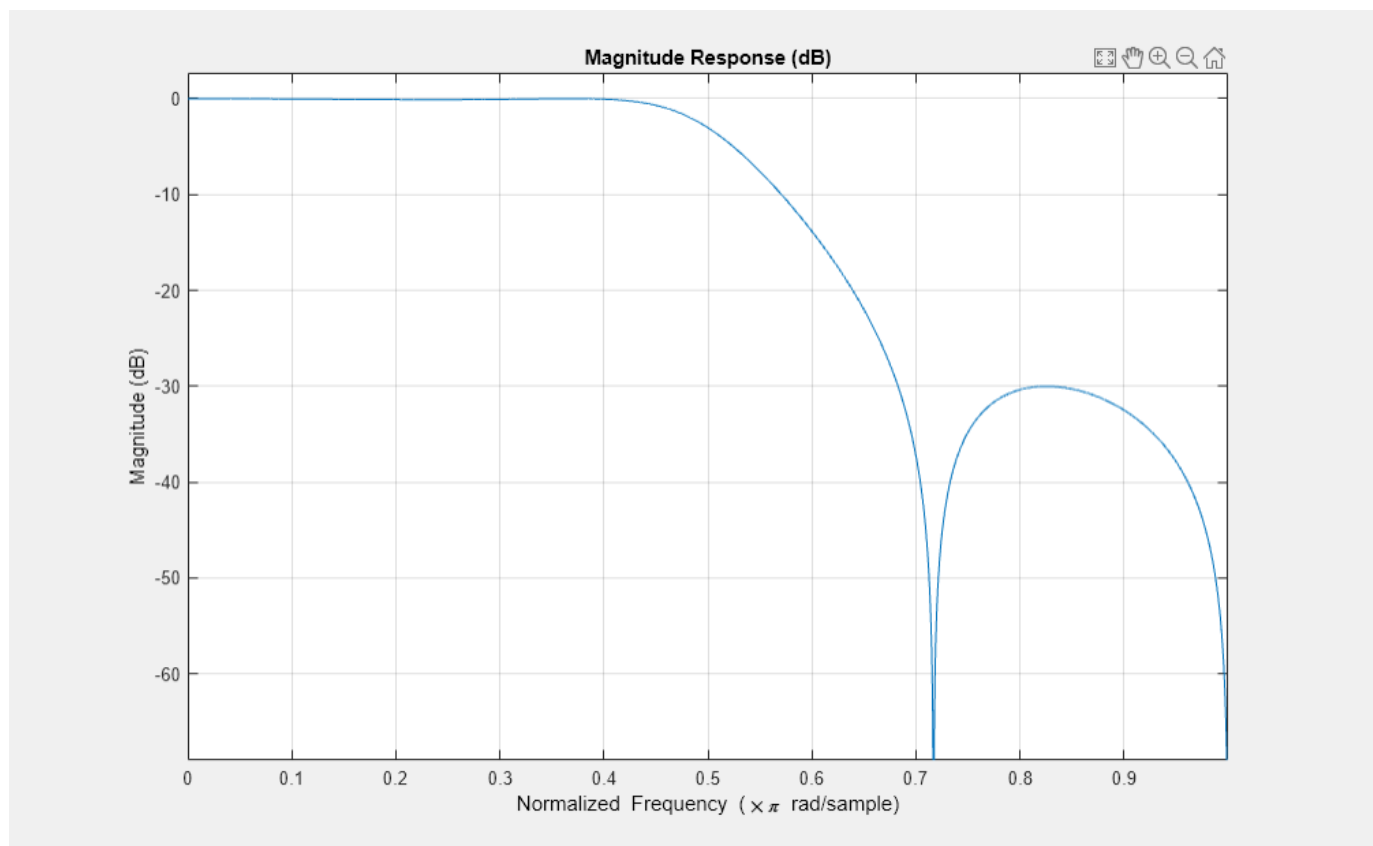
#### Transform Lowpass Filter to IIR Real $N$ -Point Filter

Transform a lowpass IIR filter to an  $N$ -point IIR filter using the `iirlp2xn` function.

#### Input Lowpass IIR Filter

Design a prototype real IIR lowpass elliptic filter with a gain of about -3 dB at  $0.5\pi$  rad/sample.

```
[b,a] = ellip(3,0.1,30,0.409);
fvtool(b,a)
```



### Transform Filter Using `iirlp2xn`

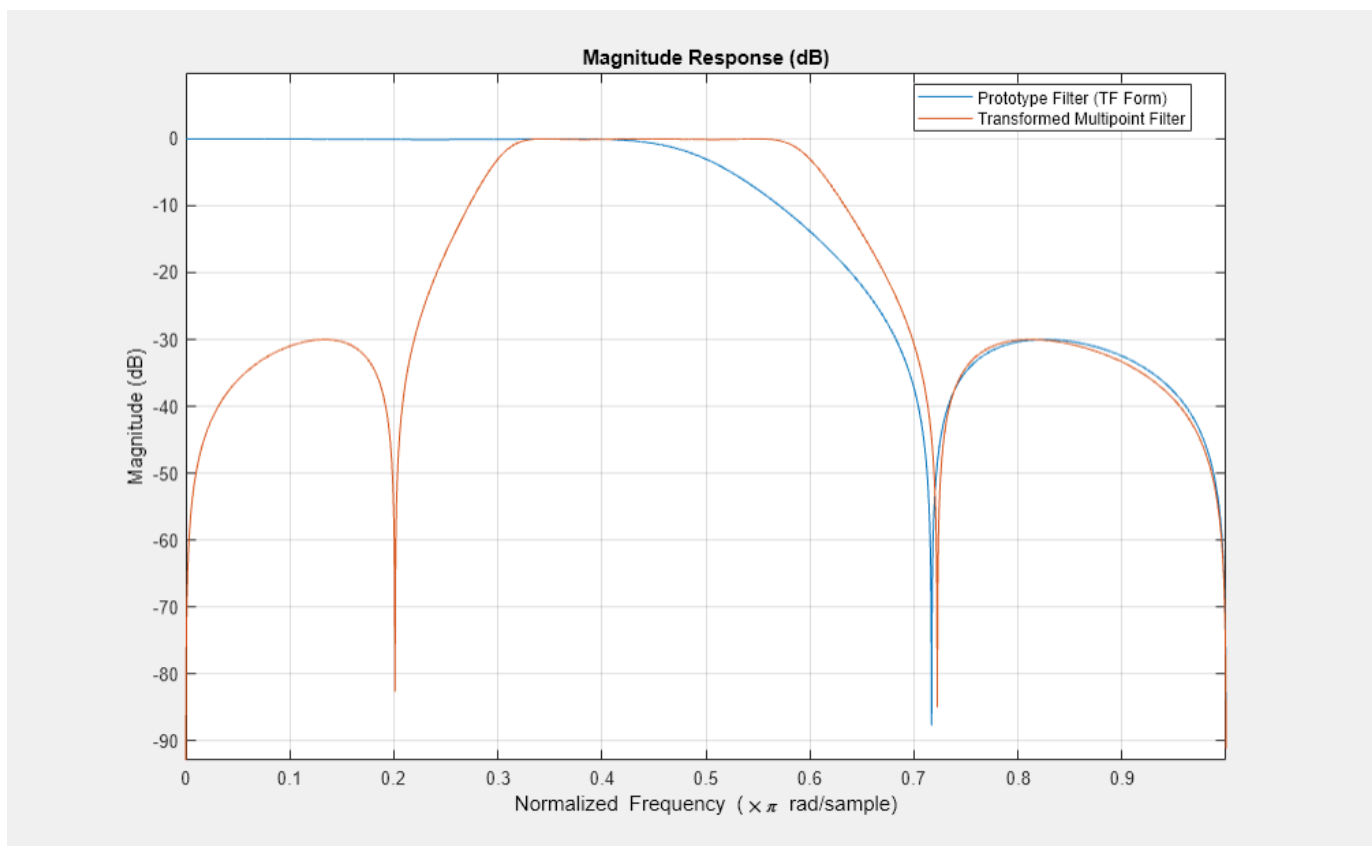
Transform the real prototype lowpass filter to an IIR real  $N$ -point filter with one passband.

Specify the prototype filter as a vector of numerator and denominator coefficients, `b` and `a` respectively.

```
[num,den] = iirlp2xn(b,a,[-0.5 0.5],[0.3 0.6],"pass");
```

Compare the magnitude response of the filters using `FVTool`.

```
hvft = fvtool(b,a,num,den);
legend(hvft,"Prototype Filter (TF Form)",...
       "Transformed Multipoint Filter");
```



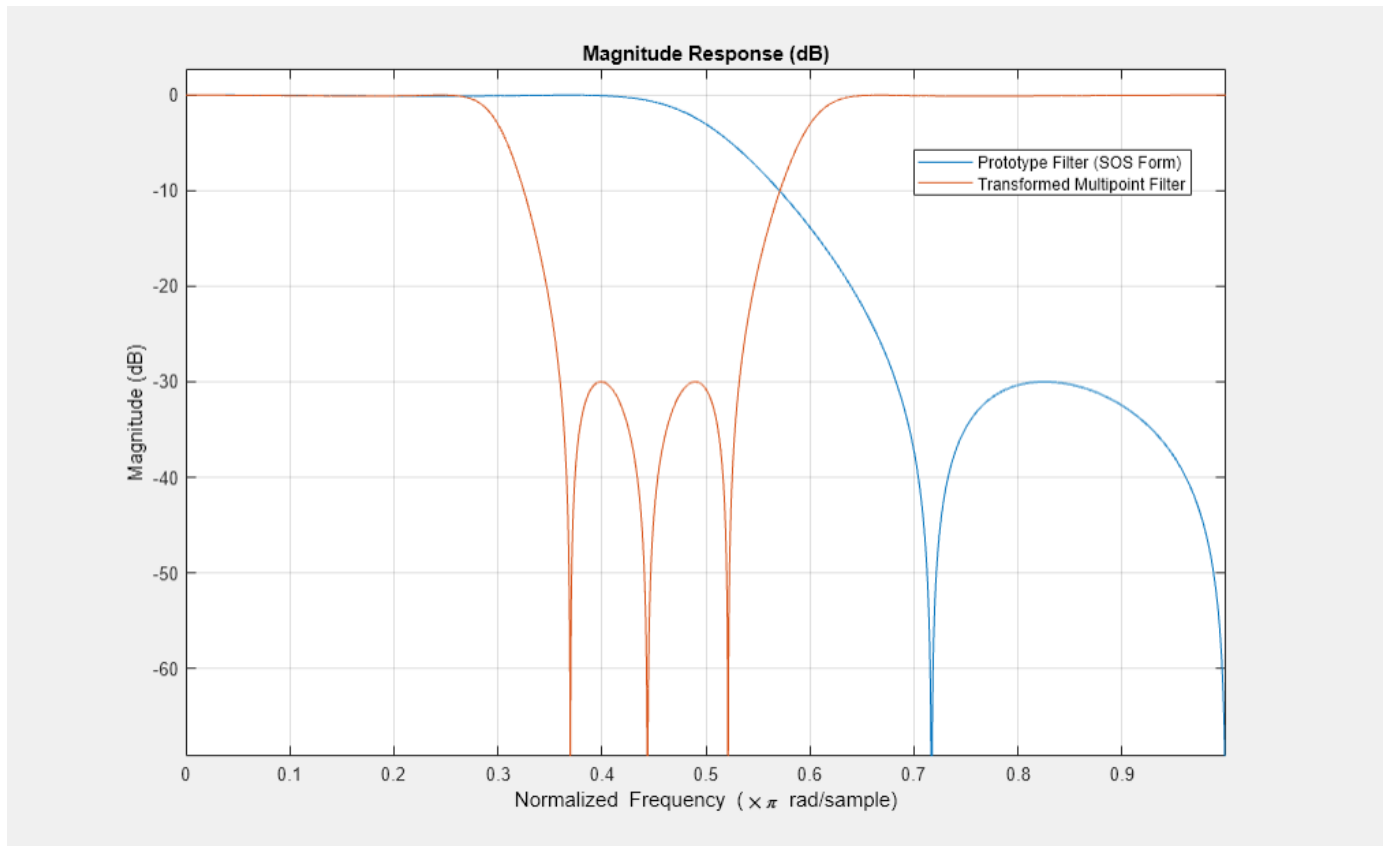
You can also specify the input lowpass IIR filter as a matrix of coefficients. Pass the second-order section matrices as inputs. The transformed filter is a fourth-order section filter with one stopband. The numerator and the denominator coefficients of the transformed filter are given by `num2` and `den2`, respectively.

```
ss = tf2sos(b,a);
[num2, den2] = iirlp2xn(ss(:,1:3),ss(:,4:6),[-0.5 0.5],...
    [0.3 0.6],"stop");
```

Compare the magnitude response of the filters using FVTool.

To visualize the magnitude response of the fourth-order section filter, first pass the filter coefficients to the `dsp.FourthOrderSectionFilter` object. Then, use this object as an input to the FVTool.

```
fos = dsp.FourthOrderSectionFilter(Numerator=num2,...
    Denominator=den2);
hvft = fvtool(ss,fos);
legend(hvft,"Prototype Filter (SOS Form)",...
    "Transformed Multipoint Filter")
```



Copyright 2012-2021 The MathWorks, Inc.

## Input Arguments

### **b** — Numerator coefficients of prototype lowpass IIR filter

row vector | matrix

Numerator coefficients of the prototype lowpass IIR filter, specified as either:

- Row vector -- Specifies the values of  $[b_0, b_1, \dots, b_n]$ , given this transfer function form:

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_0 + b_1z^{-1} + \dots + b_nz^{-n}}{a_0 + a_1z^{-1} + \dots + a_nz^{-n}}$$

where  $n$  is the order of the filter.

- Matrix -- Specifies the numerator coefficients in the form of an  $P$ -by- $(Q+1)$  matrix, where  $P$  is the number of filter sections and  $Q$  is the order of each filter section. If  $Q = 2$ , the filter is a second-order section filter. For higher-order sections, make  $Q > 2$ .

$$\mathbf{b} = \begin{bmatrix} b_{01} & b_{11} & b_{21} & \dots & b_{Q1} \\ b_{02} & b_{12} & b_{22} & \dots & b_{Q2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ b_{0P} & b_{1P} & b_{2P} & \dots & b_{QP} \end{bmatrix}$$



In the transfer function form, the numerator coefficient matrix  $b_{ik}$  of the IIR filter can be represented using the following equation:

$$H(z) = \prod_{k=1}^P H_k(z) = \prod_{k=1}^P \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2} + \dots + b_{Qk}z^{-Q}}{a_{0k} + a_{1k}z^{-1} + a_{2k}z^{-2} + \dots + a_{Qk}z^{-Q}},$$

where,

- **a** -- Denominator coefficients matrix. For more information on how to specify this matrix, see [a](#).
- $k$  -- Row index.
- $i$  -- Column index.

When specified in the matrix form, **b** and **a** matrices must have the same number of rows (filter sections)  $Q$ .

Data Types: `single` | `double`

### **a** — Denominator coefficients of prototype lowpass IIR filter

row vector | matrix

Denominator coefficients for a prototype lowpass IIR filter, specified as one of these options:

- Row vector -- Specifies the values of  $[a_0, a_1, \dots, a_n]$ , given this transfer function form:

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_0 + b_1z^{-1} + \dots + b_nz^{-n}}{a_0 + a_1z^{-1} + \dots + a_nz^{-n}},$$

where  $n$  is the order of the filter.

- Matrix -- Specifies the denominator coefficients in the form of an  $P$ -by- $(Q+1)$  matrix, where  $P$  is the number of filter sections and  $Q$  is the order of each filter section. If  $Q = 2$ , the filter is a second-order section filter. For higher-order sections, make  $Q > 2$ .

$$\mathbf{a} = \begin{bmatrix} a_{01} & a_{11} & a_{21} & \dots & a_{Q1} \\ a_{02} & a_{12} & a_{22} & \dots & a_{Q2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{0P} & a_{1P} & a_{2P} & \dots & a_{QP} \end{bmatrix}$$

In the transfer function form, the denominator coefficient matrix  $a_{ik}$  of the IIR filter can be represented using the following equation:

$$H(z) = \prod_{k=1}^P H_k(z) = \prod_{k=1}^P \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2} + \dots + b_{Qk}z^{-Q}}{a_{0k} + a_{1k}z^{-1} + a_{2k}z^{-2} + \dots + a_{Qk}z^{-Q}},$$

where,

- **b** -- Numerator coefficients matrix. For more information on how to specify this matrix, see [b](#).
- $k$  -- Row index.
- $i$  -- Column index.

When specified in the matrix form, **a** and **b** matrices must have the same number of rows (filter sections)  $P$ .

Data Types: `single` | `double`

#### **wo** — Frequency values to transform from prototype filter

row vector

Frequency values to transform from the prototype filter, specified as a row vector. Frequencies in **wo** should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

The value of  $N$  equals the length of the **wo** vector. Length of vectors **wo** and **wt** must be the same.

Data Types: `single` | `double`

#### **wt** — Desired frequency locations in transformed target filter

row vector

Desired frequency locations in the transformed target filter, specified as a row vector. Frequencies in **wt** should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

Length of vectors **wo** and **wt** must be the same.

Data Types: `single` | `double`

#### **pass** — Choice of passband or stopband at DC

`"pass"` (default) | `"stop"`

Choice of passband or stopband at DC, specified as either:

- `"pass"` -- *"Nyquist Mobility"*. The DC feature is kept at an original frequency and the Nyquist feature is free to move.
- `"stop"` -- *"DC Mobility"*. The Nyquist feature stays at its original location and the DC feature is free to move.

## Output Arguments

#### **num** — Numerator coefficients of transformed multipoint filter

row vector | matrix

Numerator coefficients of the transformed multipoint filter, returned as one of the following:

- Row vector of length  $Nn+1$ , where  $n$  is the order of the input filter and  $N$  is the number of features being mapped. The value of  $N$  equals the length of the **wo** vector.

The **num** output is a row vector when the input coefficients **b** and **a** are row vectors.

- $P$ -by- $(QN+1)$  matrix, where  $P$  is the number of filter sections,  $Q$  is the order of each section of the transformed filter, and  $N$  is the number of features being mapped and is given by the length of the vector **wo**.

The **num** output is a matrix when the input coefficients **b** and **a** are matrices.

Data Types: `single` | `double`

#### **den** — Denominator coefficients of transformed multipoint filter

row vector | matrix

Denominator coefficients of the transformed multipoint filter, returned as one of the following:

- Row vector of length  $Nn+1$ , where  $n$  is the order of the input filter and  $N$  is the number of features being mapped. The value of  $N$  equals the length of the `wo` vector.

The `den` output is a row vector when the input coefficients `b` and `a` are row vectors.

- $P$ -by- $(QN+1)$  matrix, where  $P$  is the number of filter sections,  $Q$  is the order of each section of the transformed filter, and  $N$  is the number of features being mapped and is given by the length of the vector `wo`.

The `den` output is a matrix when the input coefficients `b` and `a` are matrices.

Data Types: `single` | `double`

### **allpassNum — Numerator coefficients of mapping filter**

row vector

Numerator coefficients of the mapping filter, returned as a row vector.

Data Types: `single` | `double`

### **allpassDen — Denominator coefficients of mapping filter**

row vector

Denominator coefficients of the mapping filter, returned as a row vector.

Data Types: `single` | `double`

## **More About**

### **IIR Lowpass to IIR Real $N$ -Point Filter Transformation**

IIR Lowpass to IIR Real  $N$ -Point Filter transformation effectively places  $N$  features of the original filter, located at frequencies  $w_{o1}, \dots, w_{oN}$ , at the required target frequency locations,  $w_{t1}, \dots, w_{tM}$ . The function `iirlp2xn` requires that  $N$  and  $M$  are equal.

Relative positions of other features of the original filter are the same in the target filter for the Nyquist mobility and are reversed in DC mobility. For the Nyquist mobility this means that it is possible to select two features of an original filter,  $F_1$  and  $F_2$ , with  $F_1$  preceding  $F_2$ . Feature  $F_1$  will still precede  $F_2$  after the transformation. However, the distance between  $F_1$  and  $F_2$  will not be the same before and after the transformation. For DC mobility feature  $F_2$  will precede  $F_1$  after the transformation.

Choice of the feature subject to this transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones. Select features such that there is no band overlap when creating  $N$  bands around the unit circle.

IIR lowpass to IIR real  $N$ -point filter transformation can also be used to transforming other types of filters, for example, notch filters or resonators can be easily replicated at a number of required frequency locations. A good application would be an adaptive tone cancellation circuit reacting to the changing number and location of tones.

## Version History

Introduced in R2011a

## References

- [1] Krukowski, A., G.D. Cain, and I. Kale. "Custom Designed High-Order Frequency Transformations for IIR Filters." In *38th Midwest Symposium on Circuits and Systems. Proceedings*, 1:588-91. Rio de Janeiro, Brazil: IEEE, 1996.
- [2] Cain, G.D., A. Krukowski and I. Kale, "High Order Transformations for Flexible IIR Filter Design," *VII European Signal Processing Conference (EUSIPCO'94)*, vol. 3, pp. 1582-1585, Edinburgh, United Kingdom, September 1994.

## See Also

### Functions

`iirlp2xc` | `iirftransf` | `allpasslp2xn` | `zpklp2xn`

# iirlpnorm

Least P-norm optimal IIR filter

## Syntax

```
[num,den] = iirlpnorm(n,d,f,edges,a)
[num,den] = iirlpnorm(n,d,f,edges,a,w)
[num,den] = iirlpnorm(n,d,f,edges,a,w,p)
[num,den] = iirlpnorm(n,d,f,edges,a,w,p,dens)
[num,den] = iirlpnorm(n,d,f,edges,a,w,p,dens,initnum,initden)
[num,den,err] = iirlpnorm(...)
[num,den,err,sos,g] = iirlpnorm(...)
```

## Description

`[num,den] = iirlpnorm(n,d,f,edges,a)` returns a filter having a numerator order  $n$  and denominator order  $d$  which is the best approximation to the desired frequency response described by  $f$  and  $a$  in the least- $p$ th sense. The vector `edges` specifies the band-edge frequencies for multi-band designs. An unconstrained quasi-Newton algorithm is employed and any poles or zeros that lie outside of the unit circle are reflected back inside.  $n$  and  $d$  should be chosen so that the zeros and poles are used effectively. See the “Hints” on page 5-1007 section. Always use `freqz` to check the resulting filter.

`[num,den] = iirlpnorm(n,d,f,edges,a,w)` uses the weights in  $w$  to weight the error.  $w$  has one entry per frequency point (the same length as  $f$  and  $a$ ) which tells `iirlpnorm` how much emphasis to put on minimizing the error in the vicinity of each frequency point relative to the other points.  $f$  and  $a$  must have the same number of elements, which may exceed the number of elements in `edges`. This allows for the specification of filters having any gain contour within each band. The frequencies specified in `edges` must also appear in the vector  $f$ . For example,

```
[num,den] = iirlpnorm(5,12,[0 .15 .4 .5 1],[0 .4 .5 1],...
[1 1.6 1 0 0],[1 1 1 10 10])
```

is a lowpass filter with a peak of 1.6 within the passband.

`[num,den] = iirlpnorm(n,d,f,edges,a,w,p)` where  $p$  is a two-element vector `[pmin pmax]` allows for the specification of the minimum and maximum values of  $p$  used in the least- $p$ th algorithm. Default is `[2 128]` which essentially yields the L-infinity, or Chebyshev, norm.  $p_{min}$  and  $p_{max}$  should be even. If  $p$  is 'inspect', no optimization will occur. This can be used to inspect the initial pole/zero placement.

`[num,den] = iirlpnorm(n,d,f,edges,a,w,p,dens)` specifies the grid density `dens` used in the optimization. The number of grid points is  $(dens*(n+d+1))$ . The default is 20. `dens` can be specified as a single-element cell array. The grid is not equally spaced.

`[num,den] = iirlpnorm(n,d,f,edges,a,w,p,dens,initnum,initden)` allows for the specification of the initial estimate of the filter numerator and denominator coefficients in vectors `initnum` and `initden`. `initnum` should be of length  $n+1$ , and `initden` should be of length  $d+1$ . This may be useful for difficult optimization problems. The pole-zero editor in Signal Processing Toolbox software can be used for generating `initnum` and `initden`.

`[num,den,err] = iirlpnorm(...)` returns the least- $p^{\text{th}}$  approximation error, `err`.

`[num,den,err,sos,g] = iirlpnorm(...)` returns the second-order section representation in the matrix `sos` and gain `g`. For numerical reasons it may be beneficial to use `sos` and `g` in some cases.

## Examples

“Least Pth-Norm Optimal IIR Filter Design”

### Design a Lowpass Filter with a Peak of 4.015 dB in Passband

```
[num,den] = iirlpnorm(5,12,[0 .15 .4 .5 1],[0 .4 .5 1],...  
[1 1.6 1 0 0],[1 1 1 10 10])
```

```
num = 1×6
```

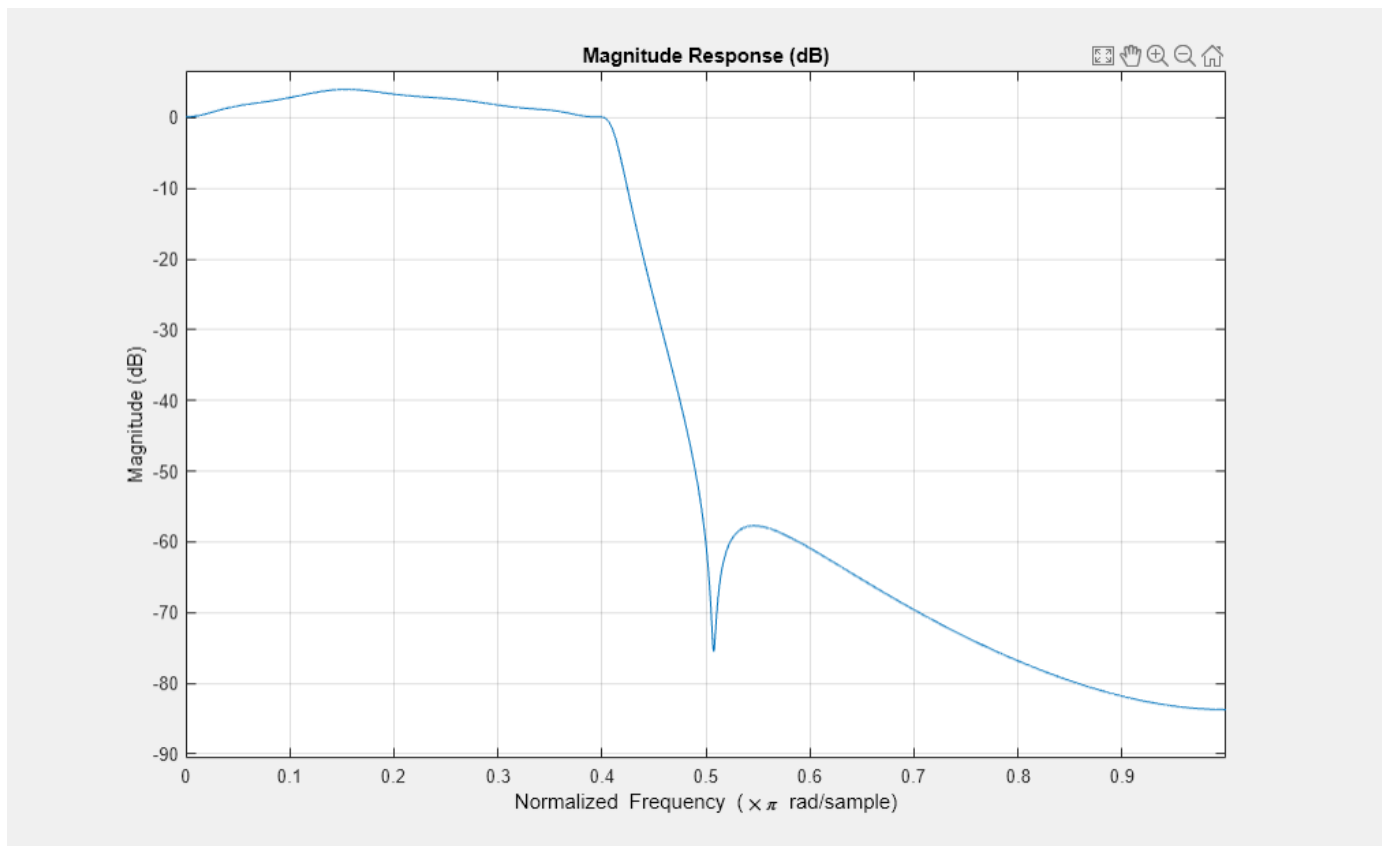
```
    -0.0128    0.0041   -0.0068    0.0048    0.0056   -0.0001
```

```
den = 1×13
```

```
    1.0000   -6.0264   18.6845  -38.7635   59.2365  -69.7345   64.5556  -47.2403   27.1100  -11.9
```

Display the magnitude response in `fvtool`

```
fvtool(num,den)
```



## Hints

- This is a weighted least- $p^{\text{th}}$  optimization.
- Check the radii and locations of the poles and zeros for your filter. If the zeros are on the unit circle and the poles are well inside the unit circle, try increasing the order of the numerator or reducing the error weighting in the stopband.
- Similarly, if several poles have a large radii and the zeros are well inside of the unit circle, try increasing the order of the denominator or reducing the error weighting in the passband.

## Version History

Introduced in R2011a

## References

Antoniou, A., *Digital Filters: Analysis, Design, and Applications*, Second Edition, McGraw-Hill, Inc. 1993.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- All inputs must be constant. Expressions or variables are allowed if their values do not change.
- Does not support syntaxes that have cell array input.

### **See Also**

`iirlpnormc` | `filter` | `freqz` | `iirgrpdelay` | `zplane`



# iirlpnormc

Constrained least Pth-norm optimal IIR filter

## Syntax

```
[num,den] = iirlpnormc(n,d,f,edges,a)
[num,den] = iirlpnormc(n,d,f,edges,a,w)
[num,den] = iirlpnormc(n,d,f,edges,a,w,radius)
[num,den] = iirlpnormc(n,d,f,edges,a,w,radius,p)
[num,den] = iirlpnormc(n,d,f,edges,a,w,radius,p,dens)
[num,den] = iirlpnormc(n,d,f,edges,a,w,radius,p,dens,initnum,initden)
[num,den,err] = iirlpnormc(...)
[num,den,err,sos,g] = iirlpnormc(...)
```

## Description

`[num,den] = iirlpnormc(n,d,f,edges,a)` returns a filter having numerator order  $n$  and denominator order  $d$  which is the best approximation to the desired frequency response described by  $f$  and  $a$  in the least- $p$ th sense. The vector `edges` specifies the band-edge frequencies for multi-band designs. A constrained Newton-type algorithm is employed.  $n$  and  $d$  should be chosen so that the zeros and poles are used effectively. See the Hints on page 5-1010 section. Always check the resulting filter using `fvtool`.

`[num,den] = iirlpnormc(n,d,f,edges,a,w)` uses the weights in  $w$  to weight the error.  $w$  has one entry per frequency point (the same length as  $f$  and  $a$ ) which tells `iirlpnormc` how much emphasis to put on minimizing the error in the vicinity of each frequency point relative to the other points.  $f$  and  $a$  must have the same number of elements, which can exceed the number of elements in `edges`. This allows for the specification of filters having any gain contour within each band. The frequencies specified in `edges` must also appear in the vector  $f$ . For example,

```
[num,den] = iirlpnormc(5,5,[0 .15 .4 .5 1],[0 .4 .5 1],...
[1 1.6 1 0 0],[1 1 1 10 10])
```

designs a lowpass filter with a peak of 1.6 within the passband.

`[num,den] = iirlpnormc(n,d,f,edges,a,w,radius)` returns a filter having a maximum pole radius of `radius` where  $0 < \text{radius} < 1$ . `radius` defaults to 0.999999. Filters that have a reduced pole radius may retain better transfer function accuracy after you quantize them.

`[num,den] = iirlpnormc(n,d,f,edges,a,w,radius,p)` where  $p$  is a two-element vector `[pmin pmax]` allows for the specification of the minimum and maximum values of  $p$  used in the least- $p$ th algorithm. Default is `[2 128]` which essentially yields the L-infinity, or Chebyshev, norm. `pmin` and `pmax` should be even. If  $p$  is 'inspect', no optimization will occur. This can be used to inspect the initial pole/zero placement.

`[num,den] = iirlpnormc(n,d,f,edges,a,w,radius,p,dens)` specifies the grid density `dens` used in the optimization. The number of grid points is  $(\text{dens} * (n+d+1))$ . The default is 20. `dens` can be specified as a single-element cell array. The grid is not equally spaced.

`[num,den] = iirlpnormc(n,d,f,edges,a,w,radius,p,dens,initnum,initden)` allows for the specification of the initial estimate of the filter numerator and denominator coefficients in vectors

`initnum` and `initden`. This may be useful for difficult optimization problems. The pole-zero editor in Signal Processing Toolbox software can be used for generating `initnum` and `initden`.

`[num,den,err] = iirlpnormc(...)` returns the least-Pth approximation error `err`.

`[num,den,err,sos,g] = iirlpnormc(...)` returns the second-order section representation in the matrix `SOS` and gain `G`. For numerical reasons you may find `SOS` and `G` beneficial in some cases.

## Hints

- This is a weighted least-pth optimization.
- Check the radii and location of the resulting poles and zeros.
- If the zeros are all on the unit circle and the poles are well inside of the unit circle, try increasing the order of the numerator or reducing the error weighting in the stopband.
- Similarly, if several poles have a large radius and the zeros are well inside of the unit circle, try increasing the order of the denominator or reducing the error weight in the passband.
- If you reduce the pole radius, you might need to increase the order of the denominator.

The message

Poorly conditioned matrix. See the "help" file.

indicates that `iirlpnormc` cannot accurately compute the optimization because either:

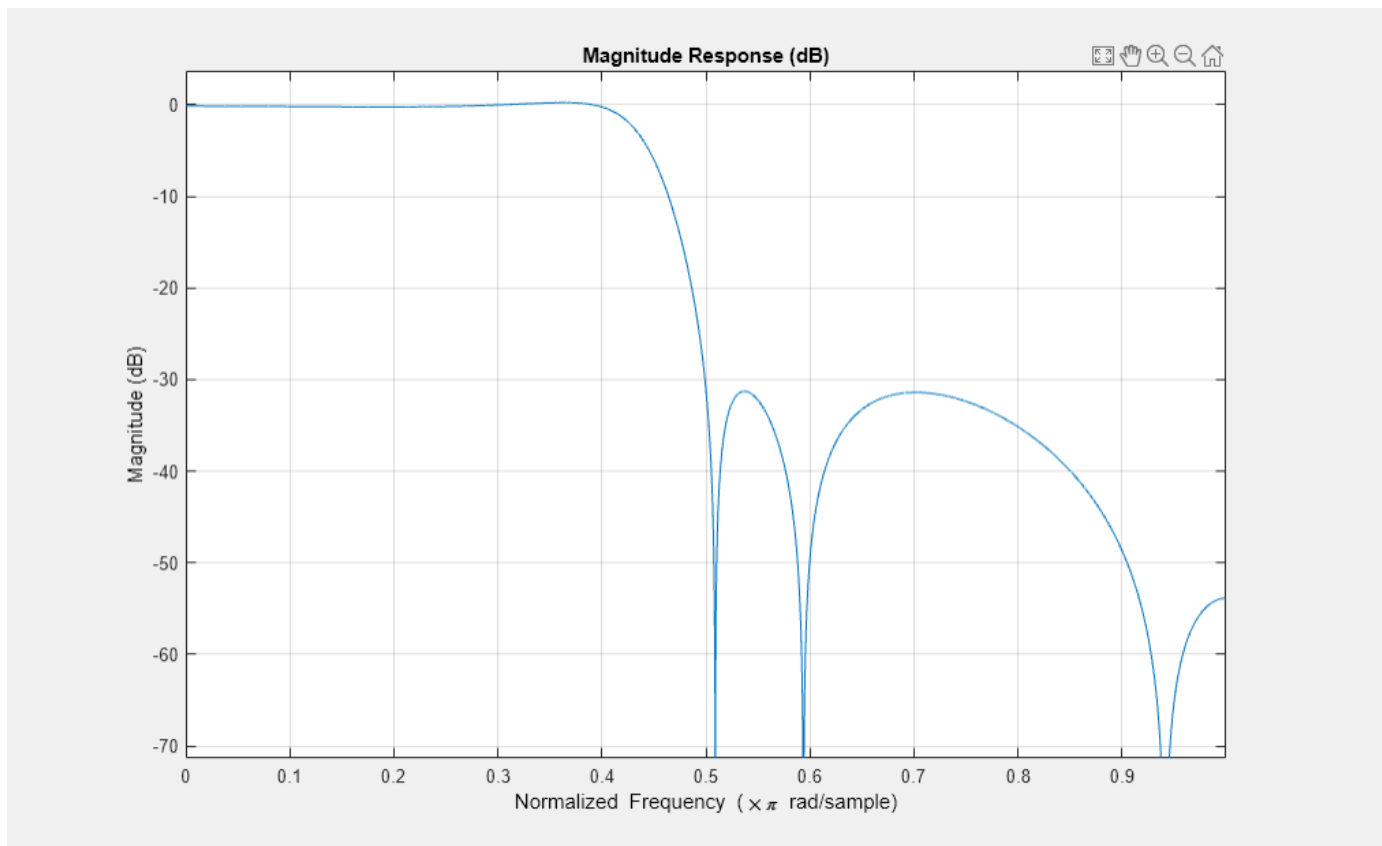
- a The approximation error is extremely small (try reducing the number of poles or zeros — refer to the hints above).
- b The filter specifications have huge variation, such as `a=[1 1e9 0 0]`.

## Examples

### Magnitude Response of Constrained Least Pth-norm Optimal IIR Filter

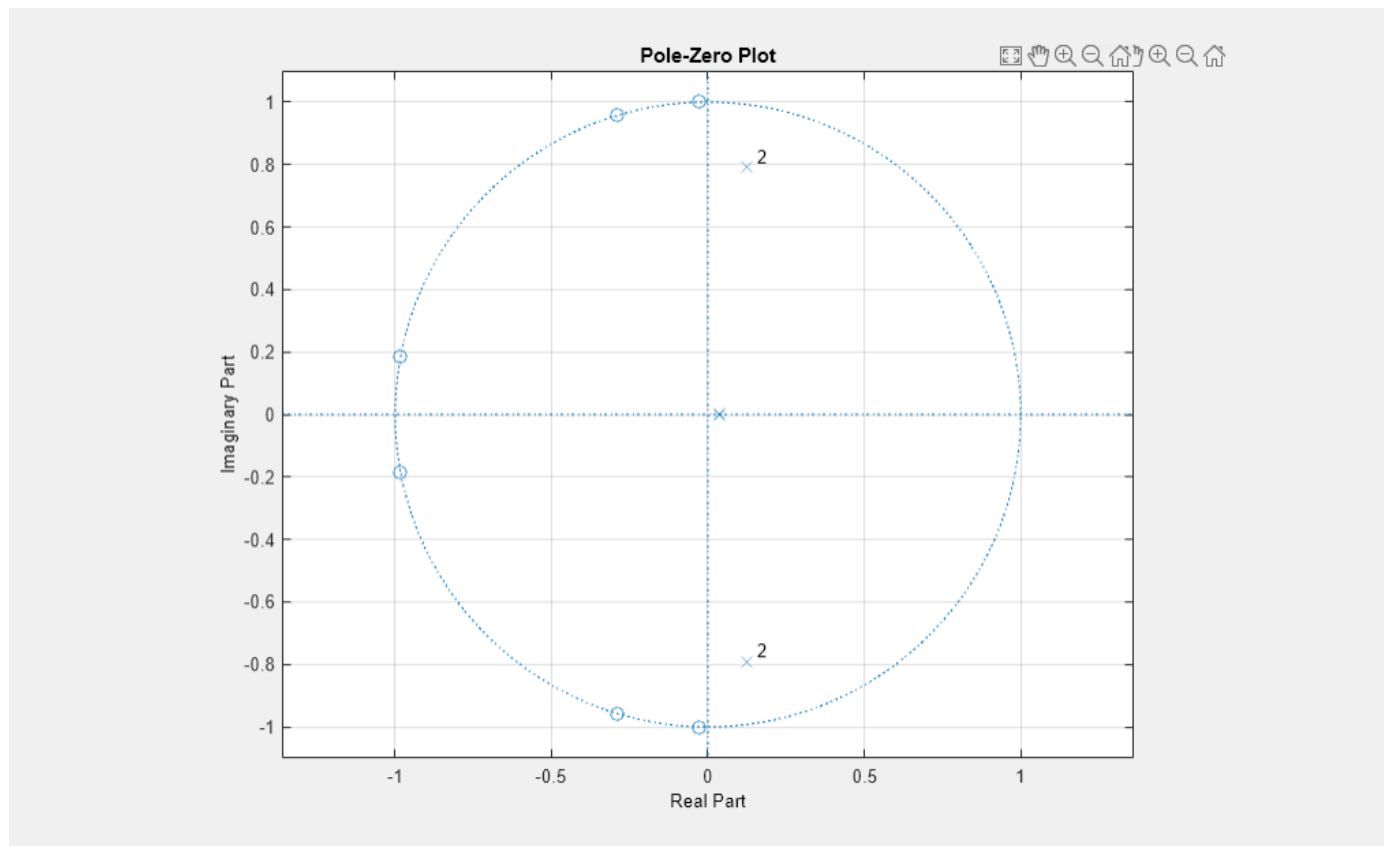
This example returns a lowpass filter whose pole radius is constrained to 0.8.

```
[b,a,err,s,g] = iirlpnormc(6,6,[0 .4 .5 1],[0 .4 .5 1],...  
[1 1 0 0],[1 1 1 1],.8);  
fvtool(b,a);
```



The magnitude response shows the lowpass nature of the filter. The pole/zero plot following shows that the poles are constrained to 0.8 as specified in the command.

```
fvtool(b,a,'polezero');
```



## Version History

Introduced in R2011a

## References

[1] Antoniou, A., *Digital Filters: Analysis, Design, and Applications*, Second Edition, McGraw-Hill, Inc. 1993.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- All inputs must be constant. Expressions or variables are allowed if their values do not change.
- Does not support syntaxes that have cell array input.

## See Also

### Functions

freqz | filter | iirgrpdelay | iirlpnorm | zplane

## iirls

Least-squares IIR filter from specification object

### Syntax

```
hd = design(d,'iirls','SystemObject',true)
hd =
design(d,'iirls',designoption,value,designoption,value,...,'SystemObject',true)
```

### Description

`hd = design(d,'iirls','SystemObject',true)` designs a least-squares filter specified by the filter specification object `d`.

---

**Note** The `iirls` algorithm might not be well behaved in all cases. Experience is your best guide to determining if the resulting filter meets your needs. When you use `iirls` to design a filter, review the filter carefully to ensure that it is appropriate for your use.

---

`hd = design(d,'iirls',designoption,value,designoption,value,...,'SystemObject',true)` returns a least-squares IIR filter where you specify design options as input arguments.

To determine the available design options, use `designopts` with the specification object and the design method as input arguments as shown.

```
designopts(d,'method')
```

For complete help about using `iirls`, refer to the command line help system. For example, to get specific information about using `iirls` with `d`, the specification object, enter the following at the MATLAB prompt.

```
help(d,'iirls')
```

### Examples

#### Design a Complex Bandpass Filter

Starting from an arbitrary magnitude and phase design object `d`, generate a complex bandpass filter of order = 5. To make the example a little easier to do, use the default values for `F`, and `H`, the frequency vector and the complex desired frequency response.

```
d = fdesign.arbmagnphase('N,F,H',5)
```

```
d =
arbmagnphase with properties:
```

```
Response: 'Arbitrary Magnitude and Phase'
```

```

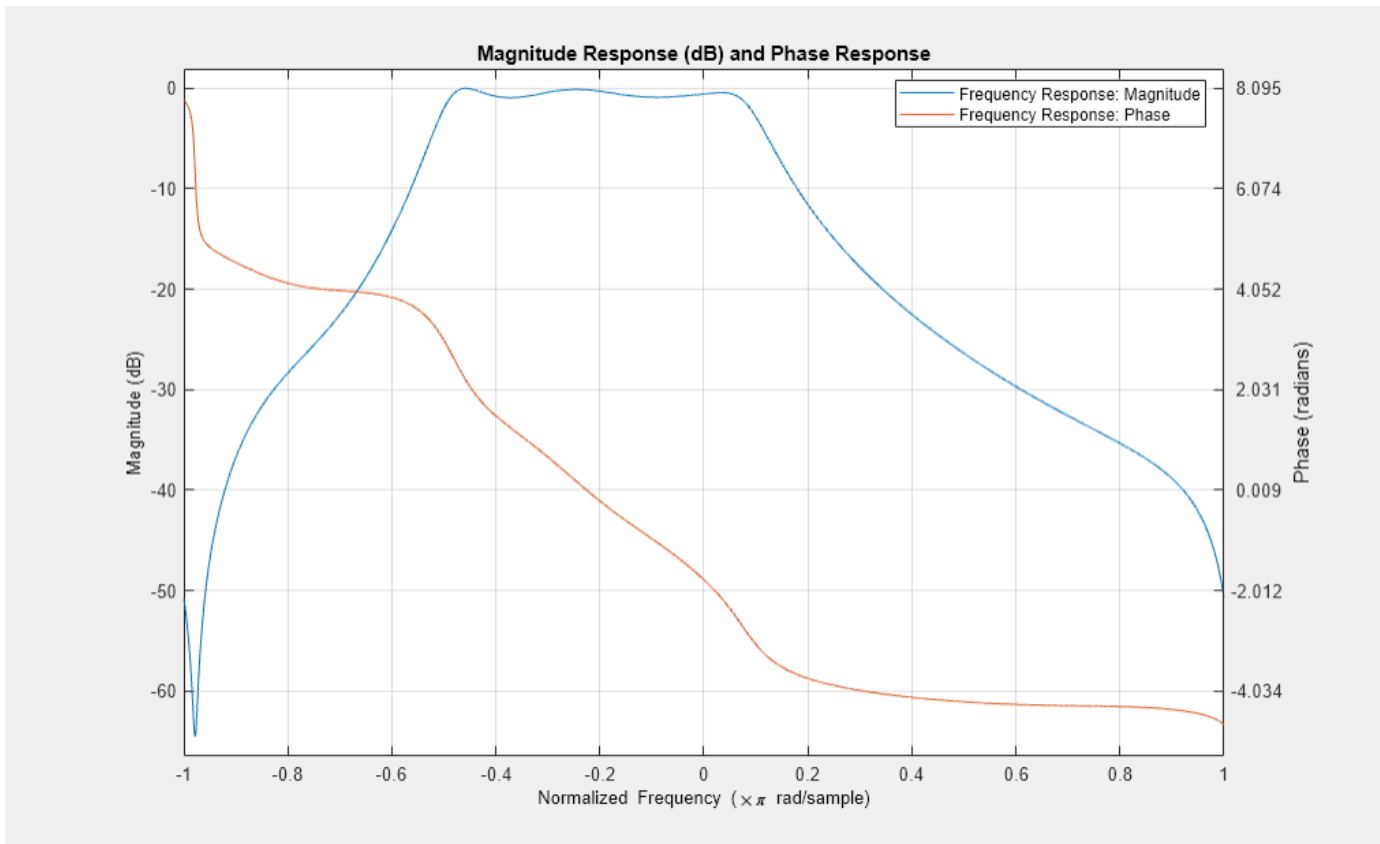
Specification: 'N,F,H'
Description: {3x1 cell}
NormalizedFrequency: 1
FilterOrder: 5
Frequencies: [-1 -0.9969 -0.9939 -0.9908 -0.9878 -0.9847 -0.9817 -0.9786 -0.9755 -0.9725]
FreqResponse: [-0.0030 - 0.0105i -0.0031 - 0.0107i -0.0032 - 0.0108i -0.0032 - 0.0110i]

```

```

filtObj = design(d,'iirls','SystemObject',true);
hvft = fvtool(filtObj,'freq');
legend(hvft,'Frequency Response')

```



## Version History

Introduced in R2011a

## See Also

fdesign.arbmag | fdesign.arbmagnphase | fir1s

## iirnotch

Second-order IIR notch filter

### Syntax

```
[num,den] = iirnotch(w0,bw)
[num,den] = iirnotch(w0,bw,ab)
```

### Description

`[num,den] = iirnotch(w0,bw)` returns the numerator coefficients, `num`, and the denominator coefficients, `den`, of the digital notching filter with the notch located at  $w_0$  and the bandwidth at the -3 dB point set to `bw`. To design the filter,  $w_0$  must meet the condition  $0.0 < w_0 < 1.0$ , where 1.0 corresponds to  $\pi$  radians per sample in the frequency range.

The quality factor (Q factor)  $q$  for the filter is related to the filter bandwidth by  $q = w_0/bw$ , where  $w_0$  is the notch frequency.

`[num,den] = iirnotch(w0,bw,ab)` returns the digital notching filter whose bandwidth, `bw`, is specified at a level of `-ab` decibels.

### Examples

#### Design IIR Notch Filter Using iirnotch

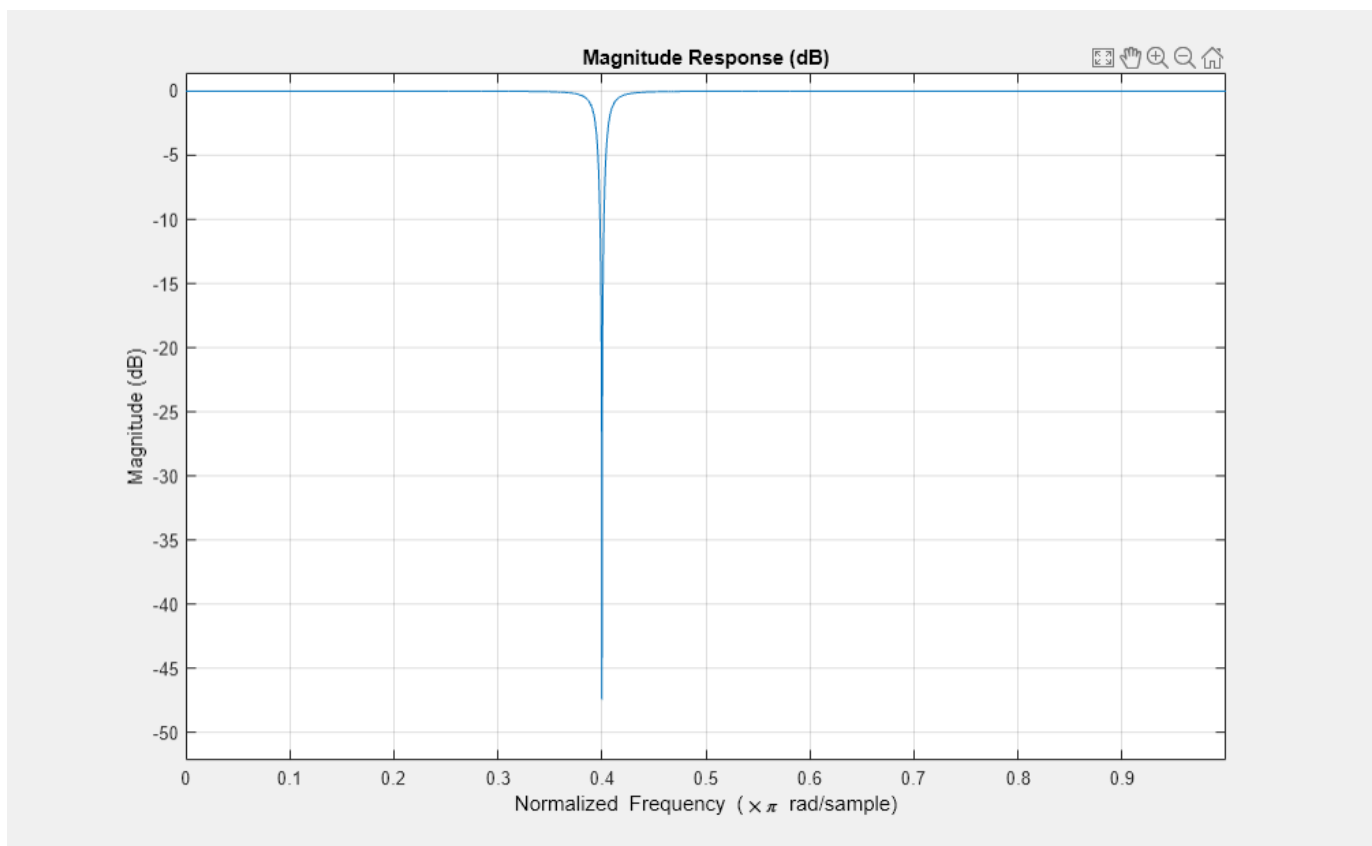
Design and plot an IIR notch filter that removes a 60 Hz tone ( $f_0$ ) from a signal at 300 Hz ( $f_s$ ). For this example, set the Q factor for the filter to 35 and use it to specify the filter bandwidth.

```
w0 = 60/(300/2);
bw = w0/35;
[b,a] = iirnotch(w0,bw);
```

The notch filter has the desired bandwidth with the notch located at 60 Hz, or  $0.4\pi$  radians per sample. Compare this plot to the comb filter plot shown for `iircomb`.

```
fvtool(b,a)
```





## Input Arguments

### **w0** — Notch frequency

positive scalar

Notch frequency, specified as a positive scalar in the range (0.0, 1.0), where 1.0 corresponds to  $\pi$  radians per sample in the frequency range.

Data Types: `single` | `double`

### **bw** — 3 dB bandwidth

positive scalar

Bandwidth at the -3 dB point, specified as a positive scalar in the range (0.0, 1.0).

The quality factor (Q factor)  $q$  for the filter is related to the filter bandwidth by  $q = w0/bw$ , where  $w0$  is the notch frequency.

Data Types: `single` | `double`

### **ab** — Custom decibel level

real scalar

Custom decibel level,  $-ab$ , at which the filter has a bandwidth of  $bw$ . Including the optional input argument  $ab$  lets you specify the magnitude response bandwidth at a level that is not the default -3 dB point, such as -6 dB or 0 dB. If not specified,  $ab$  defaults to the -3 dB width ( $10\log_{10}(1/2)$ ).

Data Types: `single` | `double`

## Output Arguments

### **num** — Numerator coefficients

row vector

Numerator coefficients of the designed notch filter, returned as a row vector.

Data Types: `double`

### **den** — Denominator coefficients

row vector

Denominator coefficients of the designed notch filter, returned as a row vector.

Data Types: `double`

## Version History

Introduced in R2011a

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

## See Also

### **Functions**

`firgr` | `iircomb` | `iirpeak`

### **Topics**

“Design of Peaking and Notching Filters”

# iirparameq

(To be removed) IIR biquad digital parametric equalizer filter

---

**Note** The `iirparameq` function will be removed in a future release. Existing code using the function continues to run. For new code, use the `designParamEQ` function instead. For more information on how to update your existing code, see “Compatibility Considerations”.

---

## Syntax

```
[SOS,SV] = iirparameq(N,G,Wo,BW)
```

```
[SOS,SV,B,A] = iirparameq(N,G,Wo,BW)
```

## Description

`[SOS,SV] = iirparameq(N,G,Wo,BW)` returns the coefficients of an  $N$ th order IIR biquad digital parametric equalizer with gain  $G$ , center frequency  $W_o$ , and bandwidth  $BW$ . The coefficients are returned in a second-order section matrix,  $SOS$ , and a vector of scale values between each biquad stage,  $SV$ .

`[SOS,SV,B,A] = iirparameq(N,G,Wo,BW)` additionally returns a matrix of numerator fourth-order sections,  $B$ , and a matrix  $A$  of denominator fourth-order sections,  $A$ . These can be used in lieu of a biquad implementation and are useful for the case  $W_o = 0.5$ .

## Examples

### Second-order section matrices of the parametric equalizer

Compute the second-order section matrix and scale values of a parametric equalizer.

```
[SOS,SV] = iirparameq(6,5,0.0042,0.0028)
```

SOS =

```

    1.0000    -1.9892     0.9894     1.0000    -1.9911     0.9912
    1.0000    -1.9926     0.9929     1.0000    -1.9941     0.9944
    1.0000    -1.9964     0.9965     1.0000    -1.9967     0.9968

```

SV =

```

    1.0009
    1.0000
    1.0009
    1.0000

```

### Fourth-order section matrices of the parametric equalizer

Compute the numerator and denominator coefficients of the fourth-order sections of the parametric equalizer.

```
[SOS,SV,B,A] = iirparameq(6,5,0.0042,0.0028)
```

SOS =

```
    1.0000    -1.9892    0.9894    1.0000    -1.9911    0.9912
    1.0000    -1.9926    0.9929    1.0000    -1.9941    0.9944
    1.0000    -1.9964    0.9965    1.0000    -1.9967    0.9968
```

SV =

```
    1.0009
    1.0000
    1.0009
    1.0000
```

B =

```
    1.0009    -1.9911    0.9903         0         0
    1.0009    -3.9927    5.9729    -3.9715    0.9903
```

A =

```
    1.0000    -1.9911    0.9912         0         0
    1.0000    -3.9908    5.9729    -3.9733    0.9912
```

### Two Equalizers Centered at Different Frequencies

Design two equalizers centered at 100 Hz and 1000 Hz respectively, both with a gain of 5 dB and a Q-factor of 1.5, for a system running at 48 kHz.

```
Fs = 48e3;
N = 6;
G = 5;
Q = 1.5;
Wo1 = 100/(Fs/2);
Wo2 = 1000/(Fs/2);
% Obtain the bandwidth of the equalizers from the center frequencies and
% Q-factors.
BW1 = Wo1/Q;
BW2 = Wo2/Q;
% Design the equalizers and obtain their SOS and SV values.
[SOS1,SV1] = iirparameq(N,G,Wo1,BW1);
[SOS2,SV2] = iirparameq(N,G,Wo2,BW2);
```

Design biquad filters using the obtained SOS and SV values.

```
BQ1 = dsp.BiquadFilter('SOSMatrix',SOS1,'ScaleValues',SV1);
BQ2 = dsp.BiquadFilter('SOSMatrix',SOS2,'ScaleValues',SV2);
```

Plot the magnitude response of both filters using a log scale.

```
fvtool(BQ1,BQ2,'Fs',Fs,'FrequencyScale','Log');
legend('Equalizer centered at 100 Hz','Equalizer centered at 1000 Hz');
```

## Compare Notch Filters Designed with Different Orders

Design an eighth-order notch filter and compare it to a traditional second-order notch filter designed with IIRNOTCH.

```
Fs = 44.1e3;
N = 8;
G = -inf;
Q = 1.8;
Wo = 60/(Fs/2); % Notch at 60 Hz
BW = Wo/Q; % Bandwidth occurs at -3 dB for this special case
[SOS1,SV1] = iirparameq(N,G,Wo,BW);
[NUM,DEN] = iirnotch(Wo,BW);
SOS2 = [NUM,DEN];
```

Design the notch filters using the SOS and SV values.

```
BQ1 = dsp.BiquadFilter('SOSMatrix',SOS1,'ScaleValues',SV1);
BQ2 = dsp.BiquadFilter('SOSMatrix',SOS2);
```

Plot the magnitude response of both filters. The filters intersect at -3 dB point.

```
FVT = fvtool(BQ1,BQ2,'Fs',Fs,'FrequencyScale','Log');
legend(FVT,'8th order notch filter','2nd order notch filter');
```

## Input Arguments

### N — Order of the parametric equalizer

even positive integer

Order of the parametric equalizer, specified as an even positive integer.

Example: 6

Example: 10

Data Types: single | double

### G — Gain of the parametric equalizer

real scalar

Gain of the parametric equalizer in dB, specified as a real scalar.

Example: 2

Example: -2.2

Data Types: single | double

### Wo — Center frequency of the parametric equalizer

real scalar

Center frequency of the parametric equalizer, specified as a real scalar in the range [0.0 1.0]. A value of 1.0 corresponds to  $\pi$  radians/sample.

Example: 0.0

Example: 1.0

Data Types: single | double

### **BW — Bandwidth of the parametric equalizer**

real scalar

Bandwidth of the parametric equalizer, specified as a real scalar in the range [0.0 1.0]. A value of 1.0 corresponds to  $\pi$  radians/sample.

Example: 0.0

Example: 1.0

Data Types: single | double

## **Output Arguments**

### **SOS — Second-order section matrix**

real-valued matrix

Second-order section matrix, returned as a real-valued  $L$ -by-6 matrix, where  $L$  is the number of second-order sections of the filter.

### **SV — Scale values between each biquad stage**

real-valued vector

Scale values between each biquad stage, returned as a real-valued vector of length  $L + 1$ .

### **B — Numerator coefficients of the fourth-order sections of the parametric equalizer**

real-valued matrix

Numerator coefficients of the fourth-order sections of the parametric equalizer, returned as a real-valued  $M$ -by-5 matrix.  $M$  is the number of fourth-order sections of the filter.

### **A — Denominator coefficients of the fourth-order sections of the parametric equalizer**

real-valued matrix

Denominator coefficients of the fourth-order sections of the parametric equalizer, returned as a real-valued  $M$ -by-5 matrix.  $M$  is the number of fourth-order sections of the filter.

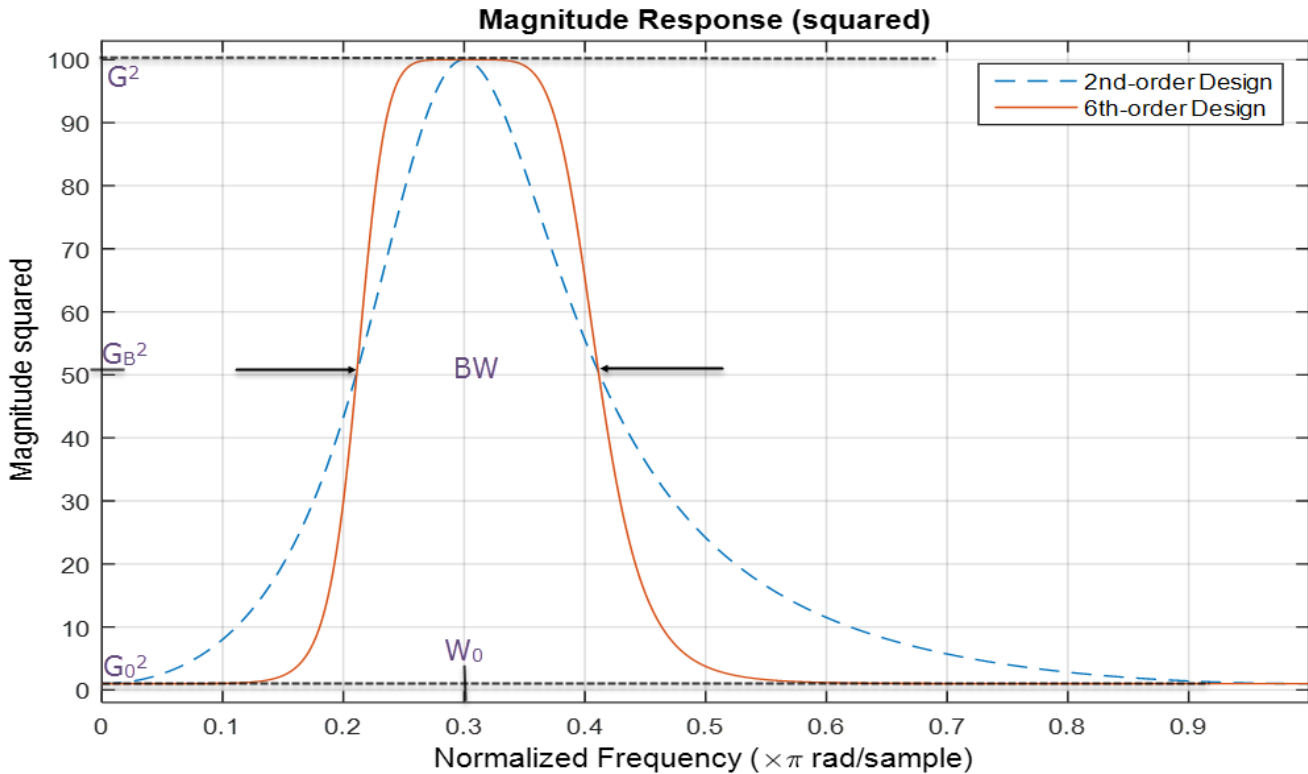
## **Algorithms**

### **High-order Parametric Equalizer**

Parametric equalizers are digital filters used in audio processing for adjusting the frequency content of a sound signal. Parametric equalizers provide capabilities beyond those of graphic equalizers by allowing the adjustment of gain, center frequency, and bandwidth of each filter. In contrast, graphic equalizers allow for the adjustment of the gain of each filter only. Digital parametric audio equalizers are commonly implemented as biquadratic (second-order IIR) filters. Due to low order, biquadratic filters can present relatively large ripple or transition regions. When several filters are connected in cascade, they can overlap with each other. In such circumstances, high-order filters are preferred. High-order filters provide flatter passbands, sharper band edges, and more control over the shape of

each filter. In addition, if the order of the filter is set to two, the design changes to a special case: a traditional second-order parametric equalizer.

The figure shows the magnitude response of a second-order parametric equalizer compared with a high-order parametric equalizer.



$W_0 = 0.3 \times \pi$  radians/sample is the center frequency of the equalizer,  $BW = 0.2$  radians/sample is the bandwidth of the equalizer,  $G = 10$  is the peak gain of the equalizer,  $G_0 = 1$ , and  $G_B = (G_0^2 + G^2) / 2$ .

### Algorithm

The first step in the filter design is to design a high-order analog lowpass shelving filter that meets the specified gain and bandwidth. A high-order Butterworth filter is used for this purpose. The analog Butterworth filter is then transformed into a digital lowpass shelving filter, and finally into a peaking equalizer that is centered at the specified peak frequency.

The design specifications for the digital equalizer are the order of the equalizer,  $N$ , the gain of the equalizer,  $G$ , the center frequency of the equalizer,  $W_0$ , and the bandwidth of the equalizer,  $BW$ .

The transfer function of the high-order parametric equalizer is given by:

$$H(z) = \left[ \frac{b_{00} + b_{01}z^{-1} + b_{02}z^{-2}}{1 + a_{01}z^{-1} + a_{02}z^{-2}} \right]^r \times \prod_{i=1}^L \left[ \frac{b_{i0} + b_{i1}z^{-1} + b_{i2}z^{-2} + b_{i3}z^{-3} + b_{i4}z^{-4}}{1 + a_{i1}z^{-1} + a_{i2}z^{-2} + a_{i3}z^{-3} + a_{i4}z^{-4}} \right]$$

where  $b_{00}$ ,  $b_{01}$ ,  $b_{02}$ ,  $a_{01}$ , and  $a_{02}$  are coefficients of the second-order section of the equalizer.  $b_{i0}$ ,  $b_{i1}$ ,  $b_{i2}$ ,  $b_{i3}$ ,  $b_{i4}$ ,  $a_{i1}$ ,  $a_{i2}$ , and  $a_{i3}$  are coefficients of the fourth-order sections of the equalizer.  $L = (N - r) / 2$ ,

where  $r = 1$  when  $N$  is odd, and  $r = 0$  when  $N$  is even. The fourth-order sections are factored into second-order sections so that you can implement them using biquad filters.

For more information on how coefficients are computed in terms of the design specifications, see the "Butterworth Designs" section in [1].

## Version History

### Introduced in R2015a

#### R2022a: `iirparameq` will be removed

*Warns starting in R2022a*

The `iirparameq` function will be removed in a future release. Existing instances of the function continue to run. For new code, use the `designParamEQ` function instead.

#### Update Code

This table shows how the function is typically used and explains how to update the existing code to use the `designParamEQ` function.

| Discouraged Usage  | Recommended Replacement  |
|--|--|
| <b>Design based on filter bandwidth</b><br><pre> Fs = 48e3; N = 2; Q = 10; G = 9; % 9 dB  % Normalized center frequency Wo1 = 2000/(Fs/2); Wo2 = 12000/(Fs/2);  % Normalized bandwidth BW1 = Wo1/Q; BW2 = Wo2/Q;  [SOS1,SV1] = iirparameq(N,G,Wo1,BW1); [SOS2,SV2] = iirparameq(N,G,Wo2,BW2); BQ1 = dsp.BiquadFilter('SOSMatrix',SOS1,'Scale',SV1); BQ2 = dsp.BiquadFilter('SOSMatrix',SOS2,'Scale',SV2);  % Visualize the filters hfvt = fvtool(BQ1,BQ2,'Fs',Fs,'FrequencyScale','Log'); legend(hfvt,'BW1 = 200 Hz; Q = 10','BW2 = 1200 Hz; Q = 10');</pre> | <b>Design based on filter bandwidth</b><br><pre> Fs = 48e3; N = 2; Q = 10; G = 9; % 9 dB  % Normalized center frequency Wo1 = 2000/(Fs/2); Wo2 = 12000/(Fs/2);  % Normalized bandwidth BW1 = Wo1/Q; BW2 = Wo2/Q;  [B1,A1] = designParamEQ(N,G,Wo1,BW1); [B2,A2] = designParamEQ(N,G,Wo2,BW2); BQ1 = dsp.BiquadFilter('SOSMatrix',[B1.',[1,A1.]]); BQ2 = dsp.BiquadFilter('SOSMatrix',[B2.',[1,A2.]]);  % Visualize the filters hfvt = fvtool(BQ1,BQ2,'Fs',Fs,'FrequencyScale','Log'); legend(hfvt,'BW1 = 200 Hz; Q = 10','BW2 = 1200 Hz; Q = 10');</pre> |



| Discouraged Usage  | Recommended Replacement  |
|--|--|
| <p><b>Design based on quality factor</b></p> <pre> Fs = 48e3; N = 2; G = 15; % 15 dB  % Quality factor Q1 = 0.48; Q2 = 1/sqrt(2);  % Normalized center frequency Wo = 6000/(Fs/2);  % Normalized bandwidth BW1 = Wo/Q1; BW2 = Wo/Q2;  [SOS1,SV1] = iirparameq(N,G,Wo,BW1); [SOS2,SV2] = iirparameq(N,G,Wo,BW2); BQ1 = dsp.BiquadFilter('SOSMatrix',SOS1,'Scale',1); BQ2 = dsp.BiquadFilter('SOSMatrix',SOS2,'Scale',1);  % Visualize the filters hfvt = fvtool(BQ1,BQ2,'Fs',Fs,'Color','white'); legend(hfvt,'Q = 0.48','Q = 0.7071');</pre>   | <p><b>Design based on quality factor</b></p> <pre> Fs = 48e3; N = 2; G = 15; % 15 dB  % Quality factor Q1 = 0.48; Q2 = 1/sqrt(2);  % Normalized center frequency Wo = 6000/(Fs/2);  % Normalized bandwidth BW1 = Wo/Q1; BW2 = Wo/Q2;  [B1,A1] = designParamEQ(N,G,Wo,BW1); [B2,A2] = designParamEQ(N,G,Wo,BW2); BQ1 = dsp.BiquadFilter('SOSMatrix',[B1.',[1,A1.']],1); BQ2 = dsp.BiquadFilter('SOSMatrix',[B2.',[1,A2.']],1);  % Visualize the filters hfvt = fvtool(BQ1,BQ2,'Fs',Fs,'Color','white'); legend(hfvt,'Q = 0.48','Q = 0.7071');</pre>   |
| <p><b>Low shelf and high shelf filters</b></p> <pre> Fs = 48e3; N = 4; G = 10; % 10 dB  % Normalized center frequency Wo1 = 0; % Lowpass filter % Corresponds to Fs/2 (Hz) or pi (rad/sample) Wo2 = 1; % Highpass filter  % Bandwidth occurs at 7.4 dB in this case BW = 1000/(Fs/2);  [SOS1,SV1] = iirparameq(N,G,Wo1,BW); [SOS2,SV2] = iirparameq(N,G,Wo2,BW); BQ1 = dsp.BiquadFilter('SOSMatrix',SOS1,'Scale',1); BQ2 = dsp.BiquadFilter('SOSMatrix',SOS2,'Scale',1);  % Visualize the filters hfvt = fvtool(BQ1,BQ2,'Fs',Fs,'Color','white'); legend(hfvt,'Low Shelf Filter','High Shelf Filter');</pre> | <p><b>Low shelf and high shelf filters</b></p> <pre> Fs = 48e3; N = 4; G = 10; % 10 dB  % Normalized center frequency Wo1 = 0; % Lowpass filter % Corresponds to Fs/2 (Hz) or pi (rad/sample) Wo2 = 1; % Highpass filter  % Bandwidth occurs at 7.4 dB in this case BW = 1000/(Fs/2);  [B1,A1] = designParamEQ(N,G,Wo1,BW); [B2,A2] = designParamEQ(N,G,Wo2,BW); BQ1 = dsp.BiquadFilter('SOSMatrix',[B1.',[ones(2,1),A1.']],1); BQ2 = dsp.BiquadFilter('SOSMatrix',[B2.',[ones(2,1),A2.']],1);  % Visualize the filters hfvt = fvtool(BQ1,BQ2,'Fs',Fs,'Color','white'); legend(hfvt,'Low Shelf Filter','High Shelf Filter');</pre> |

## References

- [1] Sophocles J. Orphanidis. "High-Order Digital Parametric Equalizer Design." *J. Audio Eng. Soc.* Vol. 53, November 2005, pp. 1026-1046.

**See Also**

designParamEQ

# iirpeak

Second-order IIR peak or resonator filter

## Syntax

```
[num,den] = iirpeak(w0,bw)  
[num,den] = iirpeak(w0,bw,ab)
```

## Description

`[num,den] = iirpeak(w0,bw)` returns a second-order digital peaking filter with the peak located at  $w_0$ , and with the bandwidth at the +3 dB point set to  $bw$ .

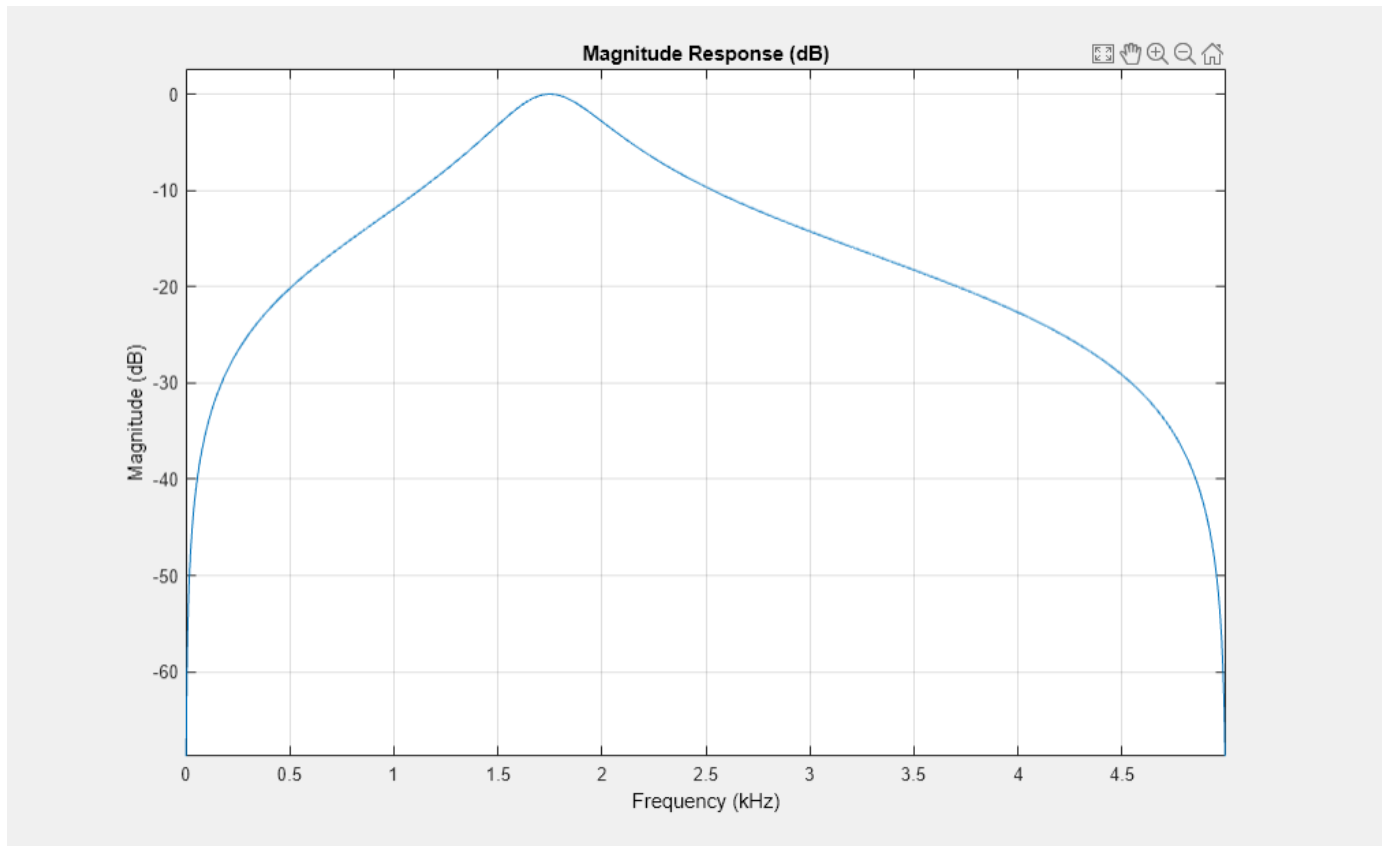
`[num,den] = iirpeak(w0,bw,ab)` returns a digital peaking filter whose bandwidth,  $bw$ , is specified at a level of  $+ab$  decibels.

## Examples

### Design IIR Peaking Filter

Design and plot an IIR peaking filter that boosts the frequency at 1.75 KHz in a signal and has a bandwidth of 500 Hz at the -3 dB point.

```
fs = 10000;  
wo = 1750/(fs/2);  
bw = 500/(fs/2);  
[b,a] = iirpeak(wo,bw);  
fvtool(b,a,Fs=fs)
```



The peak filter has the desired gain and bandwidth at 1.75 KHz.

## Input Arguments

### **w0 — Peak frequency**

positive scalar in the range (0.0, 1.0)

Peak frequency of the IIR filter, specified as a positive scalar in the range (0.0, 1.0), where 1.0 corresponds to  $\pi$  radians per sample in the frequency range.

Data Types: single | double

### **bw — Bandwidth of filter**

positive scalar in the range (0.0, 1.0)

Bandwidth of the filter at a level of +ab decibels, specified as a positive scalar in the range (0.0, 1.0).

The quality factor (Q factor)  $q$  for the filter is related to the filter bandwidth by  $q = \omega_0 / bw$  where  $\omega_0$  is the signal frequency to boost.

Data Types: single | double

### **ab — Magnitude response at filter bandwidth**

3 (default) | scalar

Magnitude response at the filter bandwidth `bw`, specified as a scalar in dB. Including the optional input argument `ab` lets you specify the magnitude response bandwidth at a level that is not the default +3 dB point, such as +6 dB or 0 dB.

Data Types: `single` | `double`

## Output Arguments

### **num** — Numerator coefficients

real-valued three-element row vector

Numerator coefficients of the second order IIR peak filter transfer function, returned as a real-valued three-element row vector.

Data Types: `single` | `double`

### **den** — Denominator coefficients

real-valued three-element row vector

Denominator coefficients of the second order IIR peak filter transfer function, returned as a real-valued three-element row vector.

Data Types: `single` | `double`

## Version History

Introduced in R2011a

## References

[1] S.J.Orfanidis. *Introduction To Signal Processing*. Englewood Cliffs, New Jersey: Prentice-Hall, 1996.

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

## See Also

### **Functions**

`firgr` | `iircomb` | `iirnotch`

## iirpowcomp

Power complementary IIR filter

### Syntax

```
[bp,ap] = iirpowcomp(b,a)
[bp,ap] = iirpowcomp(b,a,c)
```

### Description

`[bp,ap] = iirpowcomp(b,a)` returns the coefficients vectors `bp` and `ap`, of the power complementary IIR filter  $g(z) = bp(z) / ap(z)$ , given the coefficients vectors `b` and `a` of the IIR filter  $h(z) = b(z) / a(z)$  in `.` The vector `b` must be symmetric (Hermitian) or antisymmetric (antihermitian) and of the same length as the vector `a`. The two power complementary filters satisfy the relation

$$|H(w)|^2 + |G(w)|^2 = 1.$$

`[bp,ap] = iirpowcomp(b,a,c)` allow you to specify a complex scalar of unity magnitude, `c`, which forces `bp` to satisfy the generalized Hermitian property:

$$\text{conj}(bp(\text{end}:-1:1)) = c*bp$$

### Examples

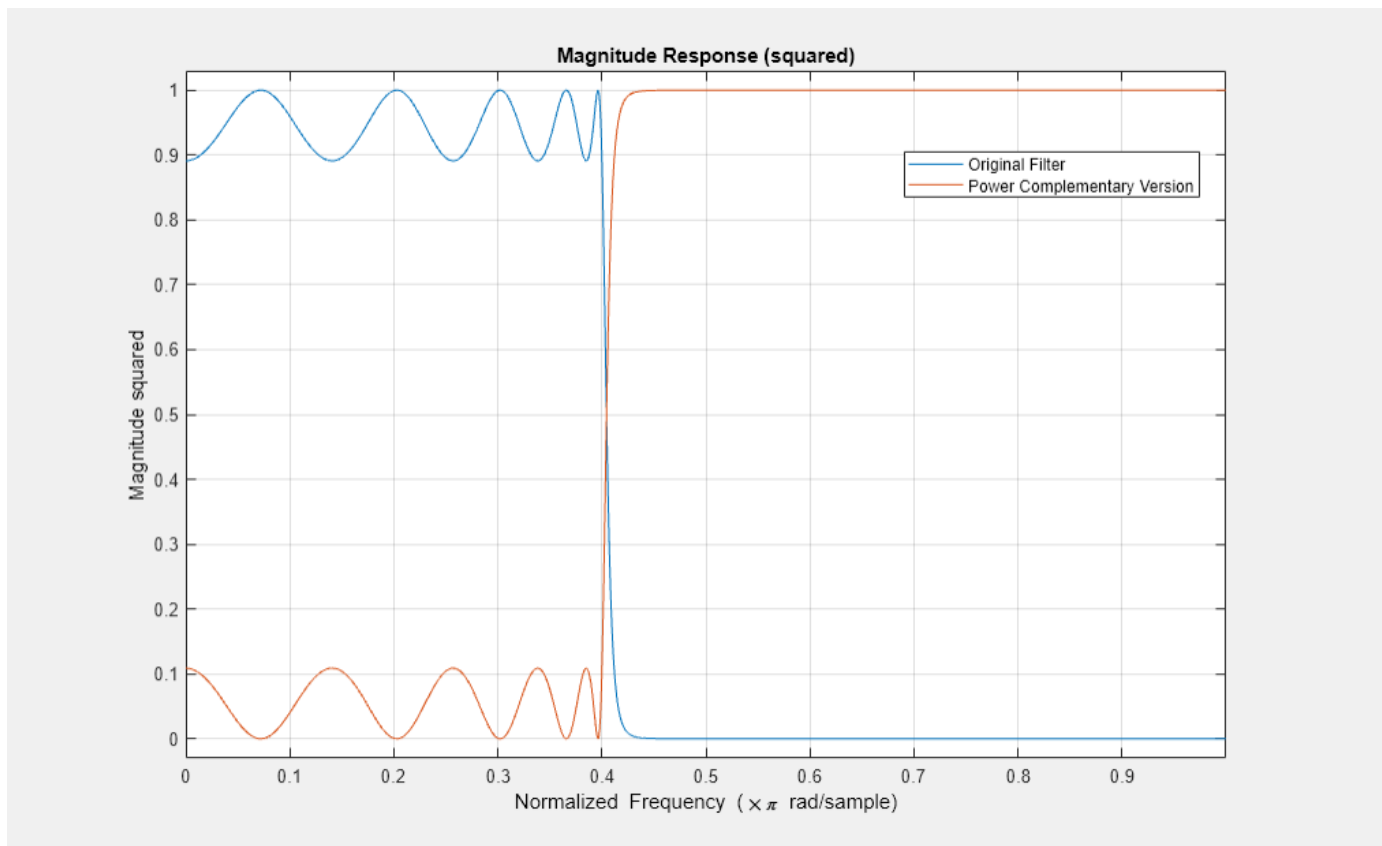
#### Power Complementary IIR Filter

Calculate the coefficients of the power complementary IIR filter.

```
[b,a]=cheby1(10,.5,.4);
[bp,ap]=iirpowcomp(b,a);
```

Compare the magnitude responses of the filters using FVTool.

```
hvft = fvtool(b,a,bp,ap,'MagnitudeDisplay','Magnitude squared');
legend(hvft,'Original Filter','Power Complementary Version');
```



## Input Arguments

### **b** — Numerator of IIR filter

row vector

Numerator coefficients of the IIR filter, specified as a row vector. The vector **b** must be symmetric (Hermitian) or antisymmetric (antihermitian) and of the same length as the vector **a**.

Data Types: `single` | `double`

### **a** — Denominator of IIR filter

row vector

Denominator coefficients of the IIR filter, specified as a row vector.

Data Types: `single` | `double`

### **c** — Complex scalar of unity magnitude

complex scalar

Complex scalar of unity magnitude, which forces the **bp** to satisfy generalized Hermitian property.

The generalized Hermitian property is given by:

$$\text{conj}(\text{bp}(\text{end}:-1:1)) = \text{c}*\text{bp}$$

When *c* is omitted, the function chooses *c* as follows:

- When *b* is real, the function chooses *c* as 1 or -1, whichever yields *bp* as real.
- When *b* is complex, *c* defaults to 1.

*ap* is always equal to *a*.

Data Types: `single` | `double`

## Output Arguments

### **bp — Numerator of power complementary IIR filter**

row vector

Numerator coefficients of the power complementary IIR filter, returned as a row vector.

Data Types: `single` | `double`

### **ap — Denominator of power complementary IIR filter**

row vector

Denominator coefficients of the power complementary IIR filter, returned as a row vector.

Data Types: `single` | `double`

## Version History

**Introduced in R2011a**

## See Also

### **Functions**

`tf2ca` | `tf2cl` | `ca2tf` | `cl2tf`



# iirrateup

Upsample IIR filter by integer factor

## Syntax

```
[Num,Den,AllpassNum,AllpassDen] = iirrateup(B,A,N)
```

## Description

`[Num,Den,AllpassNum,AllpassDen] = iirrateup(B,A,N)` returns the numerator and denominator vectors, `Num` and `Den` respectively, of the target filter being transformed from any prototype by applying an `N`th-order rateup frequency transformation, where `N` is the upsample ratio. Transformation creates `N` equal replicas of the prototype filter frequency response.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with a numerator specified by `B` and a denominator specified by `A`.

The relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter,  $F_1$  and  $F_2$ , with  $F_1$  preceding  $F_2$ . Feature  $F_1$  will still precede  $F_2$  after the transformation. However, the distance between  $F_1$  and  $F_2$  will not be the same before and after the transformation.

## Examples

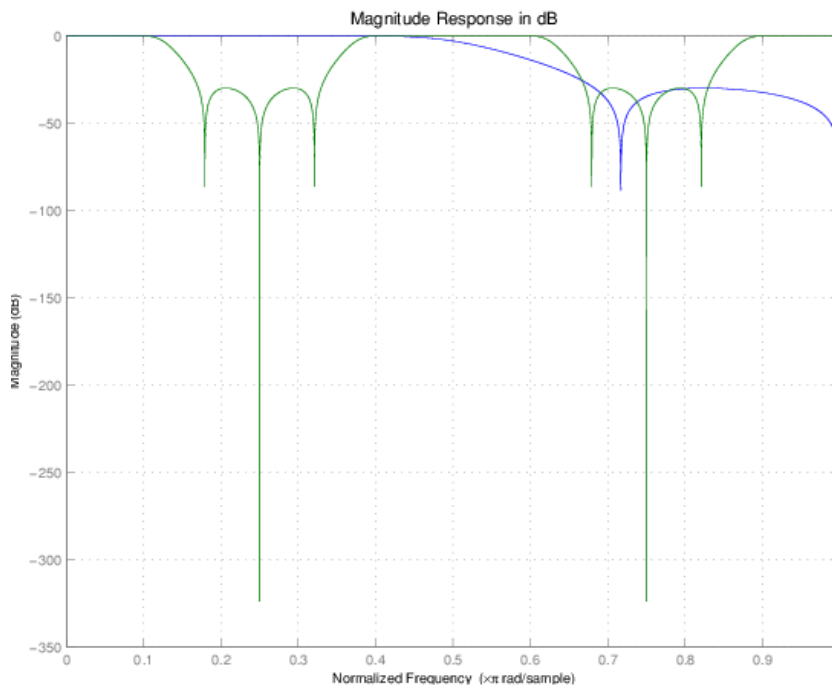
Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3, 0.1, 30, 0.409);  
[num, den] = iirrateup(b, a, 4);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, num, den);
```

As shown in the figure produced by FVTool, the transformed filter appears as expected.



## Arguments

| Variable     | Description                                 |
|--------------|---|
| $B$          | Numerator of the prototype lowpass filter   |
| $A$          | Denominator of the prototype lowpass filter |
| $N$          | Frequency multiplication ratio              |
| $Num$        | Numerator of the target filter              |
| $Den$        | Denominator of the target filter            |
| $AllpassNum$ | Numerator of the mapping filter             |
| $AllpassDen$ | Denominator of the mapping filter           |

## Version History

Introduced in R2011a

### See Also

`iirfttransf` | `allpassrateup` | `zpkrateup`

## iirshift

Shift frequency response of IIR filter

### Syntax

```
[Num,Den,AllpassNum,AllpassDen] = iirshift(B,A,Wo,Wt)
```

### Description

`[Num,Den,AllpassNum,AllpassDen] = iirshift(B,A,Wo,Wt)` returns the numerator and denominator vectors, `Num` and `Den` respectively, of the target filter transformed from the real lowpass prototype by applying a second-order real shift frequency mapping.

It also returns the numerator, `AllpassNum`, and the denominator of the allpass mapping filter, `AllpassDen`. The prototype lowpass filter is given with the numerator specified by `B` and the denominator specified by `A`.

This transformation places one selected feature of an original filter located at frequency  $W_0$  to the required target frequency location,  $W_t$ . This transformation implements the "DC Mobility," which means that the Nyquist feature stays at Nyquist, but the DC feature moves to a location dependent on the selection of  $W_0$  and  $W_t$ .

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter,  $F_1$  and  $F_2$ , with  $F_1$  preceding  $F_2$ . Feature  $F_1$  will still precede  $F_2$  after the transformation. However, the distance between  $F_1$  and  $F_2$  will not be the same before and after the transformation.

Choice of the feature subject to the real shift transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can change their position in a simple way without designing them from the beginning.

### Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

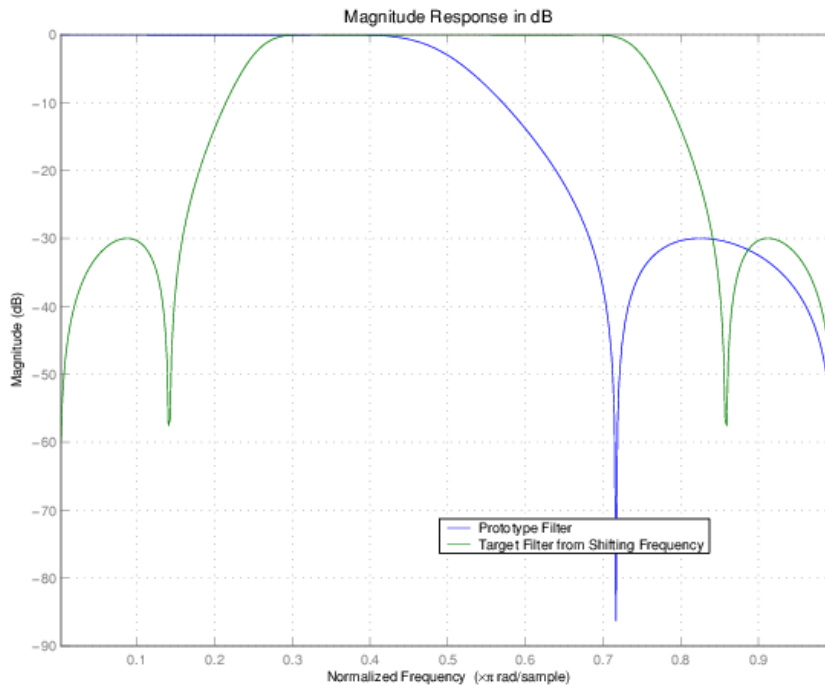
Perform the real frequency shift by defining where the selected feature of the prototype filter, originally at  $W_0=0.5$ , should be placed in the target filter,  $W_t=0.75$ :

```
Wo = 0.5; Wt = 0.75;
[num, den] = iirshift(b, a, Wo, Wt);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, num, den);
```

Shifting the specified feature from the prototype to the target generates the response shown in the figure.



## Arguments

| Variable     | Description   |
|--------------|---|
| $B$          | Numerator of the prototype lowpass filter                   |
| $A$          | Denominator of the prototype lowpass filter                 |
| $\omega_0$   | Frequency value to be transformed from the prototype filter |
| $\omega_t$   | Desired frequency location in the transformed target filter |
| $Num$        | Numerator of the target filter                              |
| $Den$        | Denominator of the target filter                            |
| $AllpassNum$ | Numerator of the mapping filter                             |
| $AllpassDen$ | Denominator of the mapping filter                           |

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

## Version History

Introduced in R2011a

## See Also

`iirftransf` | `allpassshift` | `zpkshift`

# iirshiftc

Shift frequency response of IIR complex filter

## Syntax

```
[Num,Den,AllpassNum,AllpassDen] = iirshiftc(B,A,Wo,Wc)
[Num,Den,AllpassNum,AllpassDen] = iirshiftc(B,A,0,0.5)
[Num,Den,AllpassNum,AllpassDen] = iirshiftc(B,A,0,-0.5)
```

## Description

`[Num,Den,AllpassNum,AllpassDen] = iirshiftc(B,A,Wo,Wc)` returns the numerator and denominator vectors, `Num` and `Den` respectively, of the target filter transformed from the real lowpass prototype by applying a first-order complex frequency shift transformation. This transformation rotates all the features of an original filter by the same amount specified by the location of the selected feature of the prototype filter, originally at  $W_o$ , placed at  $W_t$  in the target filter.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with the numerator specified by `B` and the denominator specified by `A`.

`[Num,Den,AllpassNum,AllpassDen] = iirshiftc(B,A,0,0.5)` calculates the allpass filter for doing the Hilbert transformation, i.e. a 90 degree counterclockwise rotation of an original filter in the frequency domain.

`[Num,Den,AllpassNum,AllpassDen] = iirshiftc(B,A,0,-0.5)` calculates the allpass filter for doing an inverse Hilbert transformation, i.e. a 90 degree clockwise rotation of an original filter in the frequency domain.

## Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

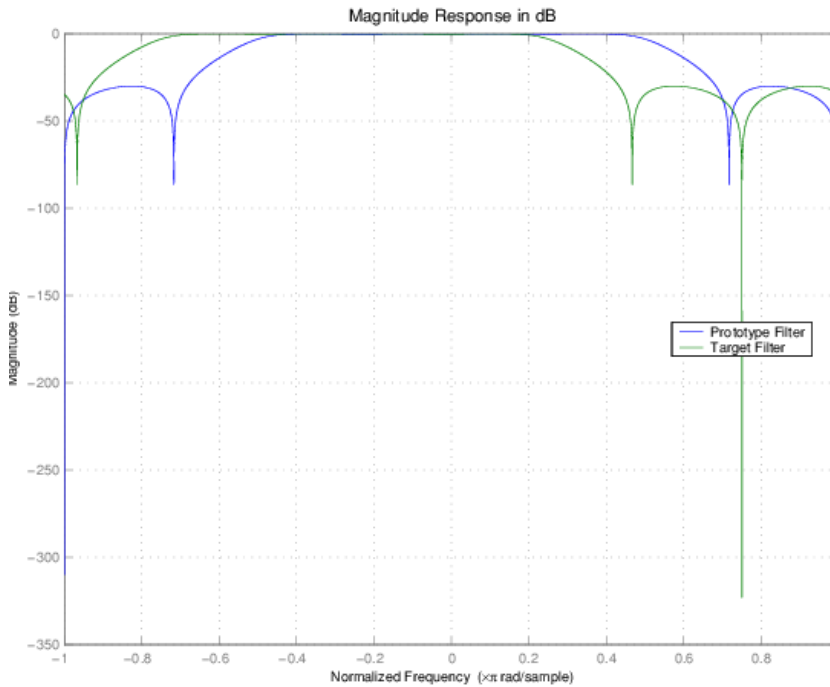
Rotate all features of the prototype filter in the frequency domain by the same amount by specifying where the selected feature of an original filter,  $W_o = 0.5$ , should appear in the target filter,  $W_t = 0.25$ :

```
[num, den] = iirshiftc(b, a, 0.5, 0.25);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, num, den);
```

After applying the shift, the selected feature from the original filter is just where it should be, at  $W_t = 0.25$ .



## Arguments

| Variable     | Description   |
|--------------|---|
| $B$          | Numerator of the prototype lowpass filter                   |
| $A$          | Denominator of the prototype lowpass filter                 |
| $\omega_0$   | Frequency value to be transformed from the prototype filter |
| $\omega_t$   | Desired frequency location in the transformed target filter |
| $Num$        | Numerator of the target filter                              |
| $Den$        | Denominator of the target filter                            |
| $AllpassNum$ | Numerator of the mapping filter                             |
| $AllpassDen$ | Denominator of the mapping filter                           |

Frequencies must be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

## Version History

Introduced in R2011a

## References

Oppenheim, A.V., R.W. Schaffer and J.R. Buck, *Discrete-Time Signal Processing*, Prentice-Hall International Inc., 1989.

Dutta-Roy, S.C. and B. Kumar, "On digital differentiators, Hilbert transformers, and half-band low-pass filters," *IEEE Transactions on Education*, vol. 32, pp. 314-318, August 1989.

**See Also**

iirftransf | allpassshiftc | zpkshiftc

# impz

**Package:** dsp

Impulse response of discrete-time filter System object

## Syntax

```
[impResp,t] = impz(sysobj)
[impResp,t] = impz(sysobj,n)
[impResp,t] = impz(sysobj,n,fs)
[impResp,t] = impz(sysobj,[],fs)
[impResp,t] = impz(sysobj,'Arithmetic',arithType)
impz(sysobj)
```

## Description

`[impResp,t] = impz(sysobj)` computes the impulse response of the filter System object and returns the response in column vector `impResp`, and a vector of times (or sample intervals) in `t`, where `t = [0 1 2 ...k-1]'`. `k` is the number of filter coefficients.

`[impResp,t] = impz(sysobj,n)` computes the impulse response at `floor(n)` one-second intervals. The time vector `t` equals `(0:floor(n)-1)'`.

`[impResp,t] = impz(sysobj,n,fs)` computes the impulse response at `floor(n)` `1/fs`-second intervals. The time vector `t` equals `(0:floor(n)-1)/fs`.

`[impResp,t] = impz(sysobj,[],fs)` computes the impulse response at `k` `1/fs`-second intervals. `k` is the number of filter coefficients. The time vector `t` equals `(0:k-1)/fs`.

`[impResp,t] = impz(sysobj,'Arithmetic',arithType)` computes the impulse response based on the arithmetic specified in `arithType`, using either of the previous syntaxes.

`impz(sysobj)` uses `fvtool` to plot the impulse response of the filter System object `sysobj`.

You can use `impz` for both real and complex filters. When you omit the output arguments, `impz` plots only the real part of the impulse response.

For more input options, refer to `impz` in Signal Processing Toolbox.

## Examples

### Plot the Impulse Response of a Lowpass Elliptic Filter

Create a discrete-time filter for a fourth-order, lowpass elliptic filter with a cutoff frequency of 0.4 times the Nyquist frequency. Use a second-order sections structure to resist quantization errors. Plot the first 50 samples of the impulse response, along with the reference impulse response.

```
d = fdesign.lowpass(.4,.5,1,80);
```



Create a design object for the prototype filter. Use `ellip` to design a minimum order discrete-time SOS filter.

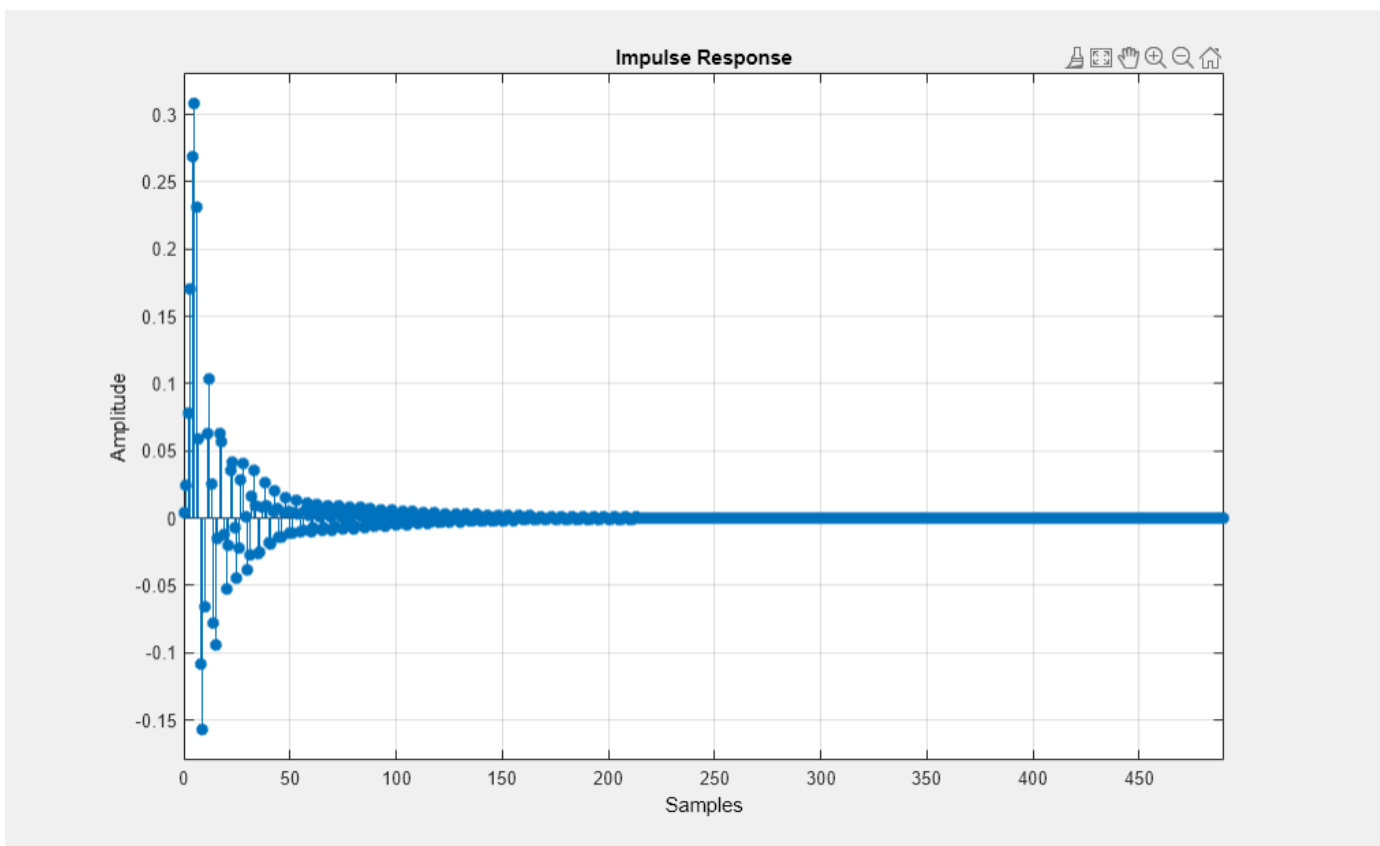
```
sosFilt = design(d,'ellip','Systemobject',true)
```

```
sosFilt =  
  dsp.SOSFilter with properties:  
    Structure: 'Direct form II'  
  CoefficientSource: 'Property'  
    Numerator: [4x3 double]  
    Denominator: [4x3 double]  
  HasScaleValues: true  
    ScaleValues: [0.7277 0.8290 1.3494 0.0048 1]
```

Show all properties

Plot the impulse response.

```
impz(sosFilt);
```



## Input Arguments

**sysobj** — Input filter

filter System object

Input filter, specified as one of the following filter System objects:

- `dsp.AllpassFilter`
- `dsp.AllpoleFilter`
- `dsp.BiquadFilter`
- `dsp.CICCompensationDecimator`
- `dsp.CICCompensationInterpolator`
- `dsp.CICDecimator`
- `dsp.CICInterpolator`
- `dsp.CoupledAllpassFilter`
- `dsp.Differentiator`
- `dsp.FarrowRateConverter`
- `dsp.FilterCascade`
- `dsp.FIRDecimator`
- `dsp.FIRFilter`
- `dsp.FIRHalfbandDecimator`
- `dsp.FIRHalfbandInterpolator`
- `dsp.FIRInterpolator`
- `dsp.FIRRateConverter`
- `dsp.FourthOrderSectionFilter`
- `dsp.HighpassFilter`
- `dsp.IIRFilter`
- `dsp.IIRHalfbandDecimator`
- `dsp.IIRHalfbandInterpolator`
- `dsp.LowpassFilter`
- `dsp.NotchPeakFilter`
- `dsp.SOSFilter`
- `dsp.VariableBandwidthFIRFilter`
- `dsp.VariableBandwidthIIRFilter`

**n — Number of filter coefficients**

positive integer

Length of the impulse response vector, specified as a positive integer.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**fs — Sampling frequency**

1 (default) | positive scalar

Sampling frequency used in computing the impulse response, specified as a positive scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**arithType — Arithmetic type**

'double' (default) | 'single' | 'Fixed'

Arithmetic used in the filter analysis, specified as 'double', 'single', or 'Fixed'. When the arithmetic input is not specified and the filter System object is unlocked, the analysis tool assumes a double-precision filter. When the arithmetic input is not specified and the System object is locked, the function performs the analysis based on the data type of the locked input.

The 'Fixed' value applies to filter System objects with fixed-point properties only.

When the 'Arithmetic' input argument is specified as 'Fixed' and the filter object has the data type of the coefficients set to 'Same word length as input', the arithmetic analysis depends on whether the System object is unlocked or locked.

- unlocked -- The analysis object function cannot determine the coefficients data type. The function assumes that the coefficients data type is signed, has a 16-bit word length, and is auto scaled. The function performs fixed-point analysis based on this assumption.
- locked -- When the input data type is 'double' or 'single', the analysis object function cannot determine the coefficients data type. The function assumes that the data type of the coefficients is signed, has a 16-bit word length, and is auto scaled. The function performs fixed-point analysis based on this assumption.

To check if the System object is locked or unlocked, use the `isLocked` function.

When the arithmetic input is specified as 'Fixed' and the filter object has the data type of the coefficients set to a custom numeric type, the object function performs fixed-point analysis based on the custom numeric data type.

## Output Arguments

### **impResp** — Impulse response

vector

Impulse response, returned as an n-element vector. If n is not specified, the length of the impulse response vector equals the number of coefficients in the filter.

Data Types: double

### **t** — Time vector

vector

Time vector of length n, in seconds. **t** consists of n equally spaced points in the range  $(0:\text{floor}(n)-1)/f_s$ . If n is not specified, the function uses the number of coefficients of the filter.

Data Types: double

## Version History

Introduced in R2011a

## See Also

### Functions

filter | impz

**Topics**

“Analysis Methods for Filter System Objects” on page 3-2

# impzlength

**Package:** dsp

Length of impulse response of discrete-time filter System object

## Syntax

```
len = impzlength(sysobj)
len = impzlength(sysobj,tol)
len = impzlength( ____, 'Arithmetic', arithType)
```

## Description

`len = impzlength(sysobj)` returns the length of the impulse response of the filter System object.

`len = impzlength(sysobj,tol)` specifies the tolerance to increase or decrease the length accuracy.

`len = impzlength( ____, 'Arithmetic', arithType)` analyzes the filter System object based on the arithmetic specified in `arithType`, using either of the previous syntaxes.

For more input options, see `impzlength` in Signal Processing Toolbox.

## Examples

### Impulse Response Length of Filter System object

This example requires DSP System Toolbox™ software.

Design a 4th-order lowpass elliptic filter with a cutoff frequency of  $0.4\pi$  rad/sample. Specify 1 dB of passband ripple and 60 dB of stopband attenuation. Design the filter in pole-zero-gain form and obtain the second order section matrix using the `zp2sos` function. Create an SOS filter object and input the object to `impzlength`.

```
[z,p,k] = ellip(4,1,60,.4);
[sos,g] = zp2sos(z,p,k);
hSosFilt = dsp.SOSFilter(Structure='Direct form I',...
    Numerator=sos(:,1:3),Denominator=sos(:,4:6),...
    HasScaleValues=true,ScaleValues=g);
len = impzlength(hSosFilt)
```

```
len = 80
```

### Impulse Response Length for Filter System Objects

Design an IIR Butterworth and an FIR equiripple filter for data sampled at 1 kHz. Set the passband frequency to 100 Hz and the stopband frequency to 150 Hz. Set the passband ripple to 0.5 dB and the

stopband attenuation to 60 dB. Create System objects for the filters and compare the filter impulse response sequence lengths.

```
d = fdesign.lowpass('Fp,Fst,Ap,Ast',100,150,0.5,60,1000);
Hd1 = design(d,'butter',SystemObject=true)
```

```
Hd1 =
  dsp.SOSFilter with properties:
    Structure: 'Direct form II'
    CoefficientSource: 'Property'
    Numerator: [9x3 double]
    Denominator: [9x3 double]
    HasScaleValues: true
    ScaleValues: [0.1020 0.0927 0.0852 0.0793 0.0748 0.0713 0.0689 0.0673 0.0665 1]
```

Show all properties

```
Hd2 = design(d,'equiripple',SystemObject=true)
```

```
Hd2 =
  dsp.FIRFilter with properties:
    Structure: 'Direct form'
    NumeratorSource: 'Property'
    Numerator: [-7.9808e-04 -2.8559e-04 0.0011 0.0036 0.0064 0.0079 0.0063 0.0012 -0.0055]
    InitialConditions: 0
```

Show all properties

```
len = [impzlength(Hd1) impzlength(Hd2)]
```

```
len = 1×2
    183    49
```

## Input Arguments

### **sysobj** — Filter System object

filter System object

Input filter, specified as one of the following filter System objects:

- `dsp.AllpassFilter`
- `dsp.AllpoleFilter`
- `dsp.BiquadFilter`
- `dsp.CICCompensationDecimator`
- `dsp.CICCompensationInterpolator`
- `dsp.CICDecimator`
- `dsp.CICInterpolator`

- `dsp.CoupledAllpassFilter`
- `dsp.Differentiator`
- `dsp.FarrowRateConverter`
- `dsp.FilterCascade`
- `dsp.FIRDecimator`
- `dsp.FIRFilter`
- `dsp.FIRHalfbandDecimator`
- `dsp.FIRHalfbandInterpolator`
- `dsp.FIRInterpolator`
- `dsp.FIRRateConverter`
- `dsp.FourthOrderSectionFilter`
- `dsp.HighpassFilter`
- `dsp.IIRFilter`
- `dsp.IIRHalfbandDecimator`
- `dsp.IIRHalfbandInterpolator`
- `dsp.LowpassFilter`
- `dsp.NotchPeakFilter`
- `dsp.SOSFilter`
- `dsp.VariableBandwidthFIRFilter`
- `dsp.VariableBandwidthIIRFilter`

### **tol — Tolerance for IIR filter effective impulse response length**

5e-5 (default) | positive scalar

Tolerance for IIR filter effective impulse response length, specified as a positive number. The tolerance determines the term in the absolutely summable sequence after which subsequent terms are considered to be 0. Increasing the tolerance returns a shorter effective impulse response sequence length. Decreasing the tolerance returns a longer effective impulse response sequence length.

### **arithType — Arithmetic type**

'double' (default) | 'single' | 'Fixed'

Arithmetic used in the filter analysis, specified as 'double', 'single', or 'Fixed'. When the arithmetic input is not specified and the filter System object is unlocked, the analysis tool assumes a double-precision filter. When the arithmetic input is not specified and the System object is locked, the function performs the analysis based on the data type of the locked input.

The 'Fixed' value applies to filter System objects with fixed-point properties only.

When the 'Arithmetic' input argument is specified as 'Fixed' and the filter object has the data type of the coefficients set to 'Same word length as input', the arithmetic analysis depends on whether the System object is unlocked or locked.

- unlocked -- The analysis object function cannot determine the coefficients data type. The function assumes that the coefficients data type is signed, has a 16-bit word length, and is auto scaled. The function performs fixed-point analysis based on this assumption.

- `locked` -- When the input data type is `'double'` or `'single'`, the analysis object function cannot determine the coefficients data type. The function assumes that the data type of the coefficients is signed, has a 16-bit word length, and is auto scaled. The function performs fixed-point analysis based on this assumption.

To check if the System object is locked or unlocked, use the `isLocked` function.

When the arithmetic input is specified as `'Fixed'` and the filter object has the data type of the coefficients set to a custom numeric type, the object function performs fixed-point analysis based on the custom numeric data type.

## Output Arguments

### **len** — Length of impulse response

positive integer

Length of the impulse response, specified as a positive integer. For stable IIR filters with absolutely summable impulse responses, `impzlength` returns an effective length for the impulse response beyond which the coefficients are essentially zero. You can control this cutoff point by specifying the optional `tol` input argument.

## Version History

Introduced in R2013a

## See Also

### Functions

`impz` | `fvtool` | `impzlength` | `freqz` | `stepz` | `zerophase`



# info

**Package:** dsp

Information about filter System object

## Syntax

```
s = info(sysobj)
s = info(sysobj,infoType)
s = info( ____, 'Arithmetic', arithType)
```

## Description

`s = info(sysobj)` returns very basic information about the filter System object. The particulars depend on the filter type and structure.

`s = info(sysobj,infoType)` returns the amount of filter information as specified by the `infoType`.

`s = info( ____, 'Arithmetic', arithType)` analyzes the filter System object, based on the arithmetic specified in `arithType`, using either of the previous syntaxes.

For more input options, see `info` in Signal Processing Toolbox.

## Examples

### Obtain Filter Information

Obtain short-format and long-format information about a filter.

```
d = fdesign.lowpass;
f = design(d,'SystemObject',true);
info(f)

ans = 6x35 char array
'Discrete-Time FIR Filter (real)      '
'-----'
'Filter Structure   : Direct-Form FIR'
'Filter Length     : 43              '
'Stable            : Yes             '
'Linear Phase      : Yes (Type 1)    '
```

```
info(f,'long')
```

```
ans = 45x45 char array
'Discrete-Time FIR Filter (real)      '
'-----'
'Filter Structure   : Direct-Form FIR'
'Filter Length     : 43              '
'Stable            : Yes             '
'Linear Phase      : Yes (Type 1)    '
```

```

'Linear Phase      : Yes (Type 1)
'
'Design Method Information
'Design Algorithm : equiripple
'
'Design Options
'Density Factor : 16
'Maximum Phase : false
'Minimum Order : any
'Minimum Phase : false
'Stopband Decay : 0
'Stopband Shape : flat
'SystemObject : true
'Uniform Grid : true
'
'Design Specifications
'Sample Rate : N/A (normalized frequency)
'Response : Lowpass
'Specification : Fp,Fst,Ap,Ast
'Passband Ripple : 1 dB
'Stopband Edge : 0.55
'Passband Edge : 0.45
'Stopband Atten. : 60 dB
'
'Measurements
'Sample Rate : N/A (normalized frequency)
'Passband Edge : 0.45
'3-dB Point : 0.46957
'6-dB Point : 0.48314
'Stopband Edge : 0.55
'Passband Ripple : 0.89042 dB
'Stopband Atten. : 60.945 dB
'Transition Width : 0.1
'
'Implementation Cost
'Number of Multipliers : 43
'Number of Adders : 42
'Number of States : 42
'Multiplications per Input Sample : 43
'Additions per Input Sample : 42

```

### Decimate a Signal Using a CICDecimator Object

Create a `dsp.CICDecimator` System object™ with `DecimationFactor` set to 4. Decimate a signal from 44.1 kHz to 11.025 kHz.

```

cicdec = dsp.CICDecimator(4);
cicdec.FixedPointDataType = 'Minimum section word lengths';
cicdec.OutputWordLength = 16;

```

Create a fixed-point sinusoidal input signal of 1024 samples, with a sampling frequency of 44.1e3 Hz.

```

Fs = 44.1e3;
% 0.0232 sec signal

```

```
n = (0:1023)';  
x = fi(sin(2*pi*1e3/Fs*n),true,16,15);
```

Create a `dsp.SignalSource` object.

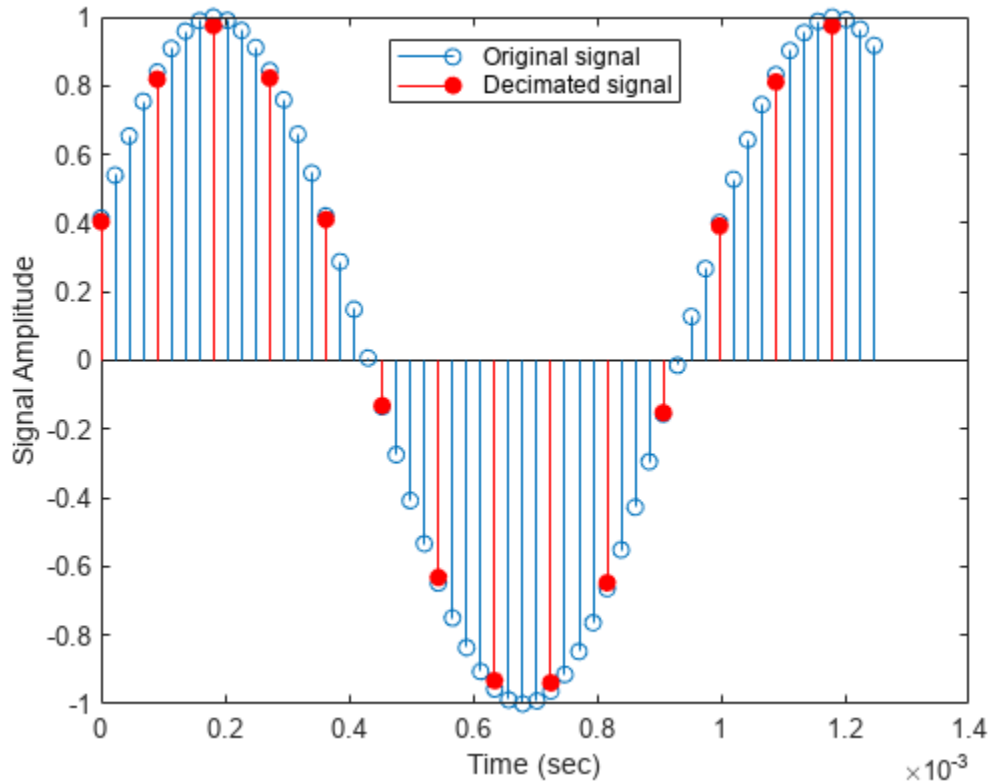
```
src = dsp.SignalSource(x,64);
```

Decimate the output with 16 samples per frame.

```
y = zeros(16,16);  
for ii = 1:16  
    y(ii,:) = cicdec(src());  
end
```

Plot the first frame of the original and decimated signals. Output latency is 2 samples.

```
D = cicdec.DecimationFactor;  
diffDelay = cicdec.DifferentialDelay;  
NumSect = cicdec.NumSections;  
gainCIC = ...  
    (D*diffDelay)^NumSect;  
stem(n(1:56)/Fs,double(x(4:59)))  
hold on;  
stem(n(1:14)/(Fs/D),double(y(1,3:end))/gainCIC,...  
    'r','filled')  
xlabel('Time (sec)')  
ylabel('Signal Amplitude')  
legend('Original signal',...  
    'Decimated signal',...  
    'Location','north')  
hold off;
```



Using the `info` method in 'long' format, obtain the word lengths and fraction lengths of the fixed-point filter sections and the filter output.

```
info(cicdec, 'long')
```

```
ans =
'Discrete-Time FIR Multirate Filter (real)
-----
Filter Structure      : Cascaded Integrator-Comb Decimator
Decimation Factor    : 4
Differential Delay    : 1
Number of Sections   : 2
Stable                : Yes
Linear Phase         : Yes (Type 1)

Implementation Cost
Number of Multipliers      : 0
Number of Adders           : 4
Number of States           : 4
Multiplications per Input Sample : 0
Additions per Input Sample : 2.5

Fixed-Point Info
Section word lengths      : 20  19  19  18
Section fraction lengths  : 15  14  14  13
Output word length       : 16
```

```
Output fraction length : 11
```

## Interpolate Signal Using CICInterpolator System object

Create a `dsp.CICInterpolator System` object™ with `InterpolationFactor` set to 2. Interpolate a fixed-point signal by a factor of 2 from 22.05 kHz to 44.1 kHz.

```
cicint = dsp.CICInterpolator(2)

cicint =
  dsp.CICInterpolator with properties:

    InterpolationFactor: 2
    DifferentialDelay: 1
    NumSections: 2
    FixedPointDataType: 'Full precision'
```

Create a `dsp.SinWave` object with `SampleRate` set to 22.05 kHz, `SamplesPerFrame` set to 32, and `OutputDataType` set to 'Custom'. To generate a fixed-point signal, set the `CustomOutputDataType` property to a `numericType` object. For the purpose of this example, set the value to `numericType([], 16)`. The fraction length is computed based on the values of the generated sinusoidal signal to give the best possible precision.

To generate a fixed-point signal, set the `Method` property of the `dsp.SinWave` object to 'Table lookup'. This method of generating the sinusoidal signal requires that the period of every sinusoid in the output be evenly divisible by the sample period. That is,  $1/f_i T_s = k_i$  must be an integer value for every channel  $i = 1, 2, \dots, N$ . The value of  $T_s$  equals  $1/F_s$ , the variable  $f_i$  is the frequency of the sinusoidal signal, and  $F_s$  is the sample rate of the signal. In other words, the ratio  $F_s/f_i$  must be an integer. For more details, see the "Algorithms" on page 4-1009 section on the `dsp.SinWave` object page.

In this example,  $F_s$  is set to 22050 Hz and  $f_i$  is set to 1050 Hz.

```
Fs = 22.05e3;
sine = dsp.SinWave('Frequency', 1050, ...
    'SampleRate', Fs, ...
    'SamplesPerFrame', 32, ...
    'Method', 'Table lookup', ...
    'OutputDataType', 'Custom')

sine =
  dsp.SinWave with properties:

    Amplitude: 1
    Frequency: 1050
    PhaseOffset: 0
    ComplexOutput: false
    Method: 'Table lookup'
    TableOptimization: 'Speed'
    SamplesPerFrame: 32
    SampleRate: 22050
```

```
OutputDataType: 'Custom'
```

```
Show all properties
```

In each loop of the iteration, stream in a frame of the fixed-point sinusoidal signal sampled at 22.05 kHz. Interpolate the streamed signal by a factor of 2. The interpolated output has 64 samples per frame.

```
for i = 1:16
    x = sine();
    y = cicint(x);
end
```

The output of the CIC interpolation filter is amplified by a specific gain value. You can determine this value using the `gain` function. This gain equals the gain of the  $2N^{\text{th}}$  stage of the CIC interpolation filter and equals  $(I \times D)^N / I$ , where  $I$  is the interpolation factor,  $D$  is the differential delay, and  $N$  is the number of sections of the CIC interpolator.

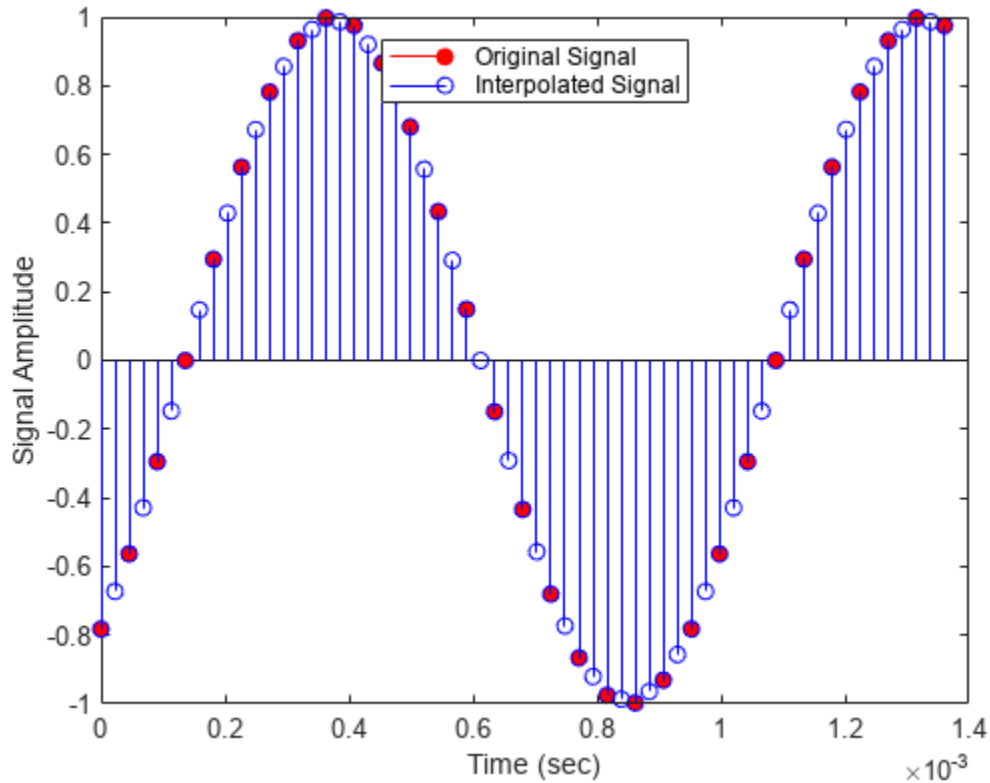
```
gainCIC = gain(cicint)
```

```
gainCIC = 2
```

To adjust this amplified output and to match it to the amplitude of the original signal, divide the CIC interpolated signal with the computed gain value.

Compare the last frames of the original and the interpolated signals. While plotting, account for the output latency of 2 samples.

```
n = (0:63)';
stem(n(1:31)/Fs, double(x(1:31)), 'r', 'filled')
hold on;
I = cicint.InterpolationFactor;
stem(n(1:61)/(Fs*I), ...
     double(y(4:end))/gainCIC, 'b')
xlabel('Time (sec)')
ylabel('Signal Amplitude')
legend('Original Signal', ...
       'Interpolated Signal', ...
       'location', 'north')
hold off;
```



Using the `info` function in the 'long' format, obtain the word lengths and fraction lengths of the fixed-point filter sections and the filter output.

```
info(cicint, 'long')
```

```
ans =
'Discrete-Time FIR Multirate Filter (real)
-----
Filter Structure      : Cascaded Integrator-Comb Interpolator
Interpolation Factor : 2
Differential Delay   : 1
Number of Sections  : 2
Stable               : Yes
Linear Phase        : Yes (Type 1)

Implementation Cost
Number of Multipliers      : 0
Number of Adders          : 4
Number of States          : 4
Multiplications per Input Sample : 0
Additions per Input Sample : 6

Fixed-Point Info
Section word lengths      : 17 17 17 17
Section fraction lengths  : 14 14 14 14
Output word length       : 17
```

```
Output fraction length : 14
```

## Input Arguments

### **sysobj** — Input filter

filter System object

One of the following types of filter System objects:

- `dsp.AllpassFilter`
- `dsp.AllpoleFilter`
- `dsp.BiquadFilter`
- `dsp.CICCompensationDecimator`
- `dsp.CICCompensationInterpolator`
- `dsp.CICDecimator`
- `dsp.CICInterpolator`
- `dsp.ComplexBandpassDecimator`
- `dsp.CoupledAllpassFilter`
- `dsp.FarrowRateConverter`
- `dsp.FilterCascade`
- `dsp.FIRDecimator`
- `dsp.FIRFilter`
- `dsp.FIRHalfbandDecimator`
- `dsp.FIRHalfbandInterpolator`
- `dsp.FIRInterpolator`
- `dsp.FIRRateConverter`
- `dsp.FourthOrderSectionFilter`
- `dsp.HighpassFilter`
- `dsp.IIRFilter`
- `dsp.IIRHalfbandDecimator`
- `dsp.IIRHalfbandInterpolator`
- `dsp.LowpassFilter`
- `dsp.SOSFilter`
- `dsp.VariableBandwidthFIRFilter`
- `dsp.VariableBandwidthIIRFilter`

### **infoType** — Amount of information to display

'short' (default) | 'long'

Amount of filter information to be displayed. When this property is set to:



- 'short' -- The function displays the same information as `info(sysobj)`, which is the basic filter information.
- 'long' -- The function returns the following information about the filter:
  - Specifications such as the filter structure and filter order.
  - Information about the design method and options.
  - Performance measurements for the filter response, such as the passband cutoff or stopband attenuation, included in the `measure` method
  - .
  - Cost of implementing the filter in terms of operations required to apply the filter to data, included in the `cost` method.

When the filter uses fixed-point arithmetic, the function returns additional information about the filter, including the arithmetic setting and details about the filter internals.

Data Types: `char` | `string`

### **arithType — Arithmetic type**

'double' (default) | 'single' | 'Fixed'

Arithmetic used in the filter analysis, specified as 'double', 'single', or 'Fixed'. When the arithmetic input is not specified and the filter System object is unlocked, the analysis tool assumes a double-precision filter. When the arithmetic input is not specified and the System object is locked, the function performs the analysis based on the data type of the locked input.

The 'Fixed' value applies to filter System objects with fixed-point properties only.

When the 'Arithmetic' input argument is specified as 'Fixed' and the filter object has the data type of the coefficients set to 'Same word length as input', the arithmetic analysis depends on whether the System object is unlocked or locked.

- unlocked -- The analysis object function cannot determine the coefficients data type. The function assumes that the coefficients data type is signed, has a 16-bit word length, and is auto scaled. The function performs fixed-point analysis based on this assumption.
- locked -- When the input data type is 'double' or 'single', the analysis object function cannot determine the coefficients data type. The function assumes that the data type of the coefficients is signed, has a 16-bit word length, and is auto scaled. The function performs fixed-point analysis based on this assumption.

To check if the System object is locked or unlocked, use the `isLocked` function.

When the arithmetic input is specified as 'Fixed' and the filter object has the data type of the coefficients set to a custom numeric type, the object function performs fixed-point analysis based on the custom numeric data type.

## **Output Arguments**

### **s — store filter information**

character array

Filter information, returned as a character array.

When the `infoType` is 'short', the function displays basic filter information. When the `infoType` is 'long', the function displays the following information:

- Specifications such as the filter structure and filter order
- Information about the design method and options
- Performance measurements for the filter response, such as the passband cutoff or stopband attenuation, included in the `measure` method
- Cost of implementing the filter in terms of operations required to apply the filter to data, included in the `cost` method

When the filter uses fixed-point arithmetic, the function returns additional information about the filter, including the arithmetic setting and details about the filter internals.

## **Version History**

**Introduced in R2011a**

### **See Also**

#### **Functions**

`info`

#### **Topics**

“Analysis Methods for Filter System Objects” on page 3-2

# info

**Package:** dsp

Characteristic information about valid delay range

## Syntax

```
S = info(obj)
```

## Description

`S = info(obj)` returns a structure that contains the valid delay range values of the `dsp.VariableFractionalDelay` object.

## Examples

### Obtain the Valid Delay Range

Create a `dsp.VariableFractionalDelay` object. Set the interpolation method to 'Farrow', and the maximum delay to 9. Obtain the valid delay range for these settings using the `info` method.

```
vfd = dsp.VariableFractionalDelay('InterpolationMethod','Farrow','MaximumDelay',9);
info(vfd)
```

```
ans = struct with fields:
    ValidDelayRange: '[1, 9]'
```

Specify the delay vector to have two taps - [0.95 10]. These values are out of the valid range. The object clips these values to [1 9] and concurrently applies the delay values to the input channel.

```
in = randn(10,1)
```

```
in = 10×1
    0.5377
    1.8339
   -2.2588
    0.8622
    0.3188
   -1.3077
   -0.4336
    0.3426
    3.5784
    2.7694
```

```
delayVec = [0.95 10]
```

```
delayVec = 1×2
```

```
0.9500    10.0000
```

```
vfdout = vfd(in,delayVec)
```

```
vfdout = 10x2
```

```
         0         0
    0.5377         0
    1.8339         0
   -2.2588         0
    0.8622         0
    0.3188         0
   -1.3077         0
   -0.4336         0
    0.3426         0
    3.5784    0.5377
```

The output contains two channels, each being a delayed version of the input channel. The first channel is delayed by 1 sample and the second channel is delayed by 9 samples.

## Input Arguments

### **obj** — Input System object

`dsp.VariableFractionalDelay`

Input object, specified as a `dsp.VariableFractionalDelay` System object.

Example: `vfd = dsp.VariableFractionalDelay; info(vfd);`

## Output Arguments

### **S** — Valid delay range information

structure

Valid delay range information of the input `dsp.VariableFractionalDelay` object, returned as the `ValidDelayRange` field in the output structure `S`. `ValidDelayRange` contains the possible range of delay values based on the current property values of the object. The `ValidDelayRange` is in the format `[MinValidDelay, MaxValidDelay]`. The object clips all input delay values to be within this `ValidDelayRange`.

## Version History

Introduced in R2012a

## See Also

### Objects

`dsp.VariableFractionalDelay`

# info

Characteristic information about audio device writer

## Syntax

```
S = info(adw)
```

## Description

`S = info(adw)` returns a structure, `S`, containing characteristic information of the audio device writer object, `adw`.

## Examples

### Read from File and Write to Audio Device

Read an MP3 audio file and play it through your default audio output device.

Create a `dsp.AudioFileReader` object with default settings. Use the `audioinfo` function to return a structure containing information about the audio file.

```
fileReader = dsp.AudioFileReader('speech_dft.mp3');
fileInfo = audioinfo('speech_dft.mp3')

fileInfo = struct with fields:
    Filename: 'B:\matlab\toolbox\dsp\samples\speech_dft.mp3'
    CompressionMethod: 'MP3'
    NumChannels: 1
    SampleRate: 22050
    TotalSamples: 112893
    Duration: 5.1199
    Title: []
    Comment: []
    Artist: []
    BitRate: 64
```

Create an `audioDeviceWriter` object and specify the sample rate.

```
deviceWriter = audioDeviceWriter('SampleRate', fileInfo.SampleRate);
```

Call `setup` to reduce the computational load of initialization in an audio stream loop.

```
setup(deviceWriter, zeros(fileReader.SamplesPerFrame, ...
    fileInfo.NumChannels))
```

Use the `info` function to obtain the characteristic information about the device writer.

```
info(deviceWriter)

ans = struct with fields:
    Driver: 'DirectSound'
```

```
        DeviceName: 'Primary Sound Driver'  
MaximumOutputChannels: 2
```

In an audio stream loop, read an audio signal frame from the file, and write the frame to your device.

```
while ~isDone(fileReader)  
    audioData = fileReader();  
    deviceWriter(audioData);  
end
```

Close the input file and release the device.

```
release(fileReader)  
release(deviceWriter)
```

## Input Arguments

**adw** — Audio device writer object

audioDeviceWriter

Audio device writer object, specified as audioDeviceWriter System object.

## Output Arguments

**S** — Characteristic information

struct

Characteristic information of the audio device writer object, returned as a structure. The fields of the structure vary depending on the System object.

## Version History

Introduced in R2016a

## See Also

### Functions

getAudioDevices

### Objects

audioDeviceWriter

# info

**Package:** dsp

Information about specific audio file

## Syntax

```
S = info(afr)
```

## Description

`S = info(afr)` returns a MATLAB structure, `S`, with information about the audio file specified in the `Filename` property.

## Examples

### Obtain Information About Audio File

Read an audio file using the `dsp.AudioFileReader` object. Obtain information related to the audio content of the file.

The `dsp.AudioFileReader` objects reads the default shipped file 'speech\_dft.mp3'.

```
afr = dsp.AudioFileReader
```

```
afr =  
dsp.AudioFileReader with properties:  
  
    Filename: 'B:\matlab\toolbox\dsp\samples\speech_dft.mp3'  
    PlayCount: 1  
    ReadRange: [1 Inf]  
    SamplesPerFrame: 1024  
    SampleRate: 22050  
    OutputDataType: 'double'
```

```
Show all properties
```

The output structure displays the sample rate of the audio signal in Hz, number of bits used to encode the audio stream, and the number of audio channels.

```
S = info(afr)
```

```
S = struct with fields:  
    SampleRate: 22050  
    NumBits: 32  
    NumChannels: 1
```

## Input Arguments

### **afr** — File reader

`dsp.AudioFileReader` (default)

File reader, specified as a `dsp.AudioFileReader` System object.

## Output Arguments

### **S** — Audio file information

structure

Information about the audio file specified in the `Filename` property, returned as a structure. The number of fields in the structure `S` varies depending on the audio content of the file. This table shows some of the fields that can appear in the structure `S`.

| Field                    | Value   |
|--------------------------|---|
| <code>SampleRate</code>  | Audio sampling rate of the audio file in Hz.    |
| <code>NumBits</code>     | Number of bits used to encode the audio stream. |
| <code>NumChannels</code> | Number of audio channels.                       |

## Version History

Introduced in R2012a

## See Also

### Functions

`isDone`

### Objects

`dsp.AudioFileReader`



# isDone

**Package:** dsp

End-of-file status (logical)

## Syntax

```
STATUS = isDone(afr)
```

## Description

`STATUS = isDone(afr)` returns a logical value, `STATUS`. The value of `STATUS` is true when the file has been read `PlayCount` number of times. The `PlayCount` property of the `dsp.AudioFileReader` System object determines the number of times the audio file plays.

## Examples

### Read and Play Back Audio File

Read and play back an audio file using the standard audio output device.

You can choose to read the entire data or specify a range of data to read from using the `ReadRange` property. By default, `ReadRange` is set to `[1 inf]`, indicating the file reader to read the entire data from the source. In this example, set `ReadRange` to `3Fs`, indicating the file reader to read the first 3 seconds of the data.

```
afr = dsp.AudioFileReader('speech_dft.mp3','ReadRange',[1 3*22050]);  
adw = audioDeviceWriter('SampleRate', afr.SampleRate);
```

```
while ~isDone(afr)  
    audio = afr();  
    adw(audio);  
end  
release(afr);  
release(adw);
```

## Input Arguments

**afr — Audio file reader**

`dsp.AudioFileReader`

Audio file reader, specified as a `dsp.AudioFileReader` System object.

## Output Arguments

**STATUS — Indicates EOF**

false | true

Logical value that indicates if the reader has reached the EOF, returned as:

- `true` -- The `STATUS` is `true` when the EOF is reached. If the `PlayCount` property is set to a value greater than 1, `STATUS` is returned as `true` only once the reader reaches the EOF `PlayCount` number of times.
- `false` -- The `STATUS` is `false` when the EOF has not reached. If `PlayCount` property is greater than 1, `STATUS` is returned as `false` until the EOF has reached `PlayCount` number of times.

Data Types: `logical`

## **Version History**

**Introduced in R2012a**

### **See Also**

#### **Functions**

`info`

#### **Objects**

`dsp.AudioFileReader`

# info

**Package:** dsp

Get cumulative overrun and underrun

## Syntax

```
S = info(asyncBuff)
```

## Description

`S = info(asyncBuff)` returns a structure, `S`, containing the cumulative overrun and underrun information of the `dsp.AsyncBuffer` System object, `asyncBuff`.

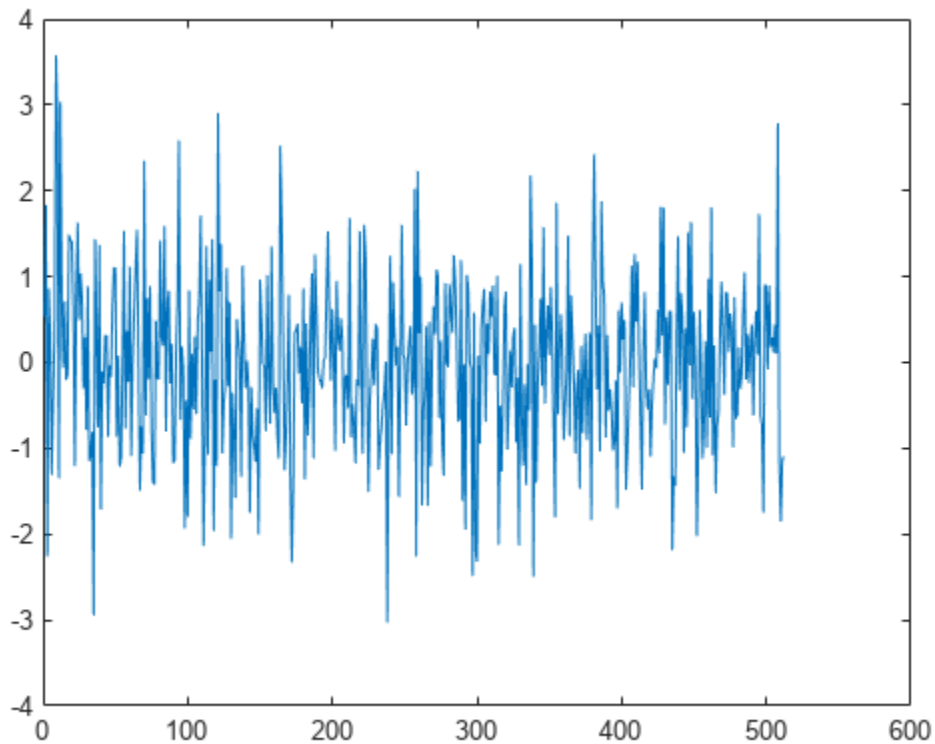
## Examples

### Read Variable Frame Sizes from Buffer

The `dsp.AsyncBuffer` System object™ supports reading variable frame sizes from the buffer.

Create a `dsp.AsyncBuffer` System object. The input is white Gaussian noise with a mean of 0, a standard deviation of 1, and a frame size of 512 samples. Write the input to the buffer using the `write` method.

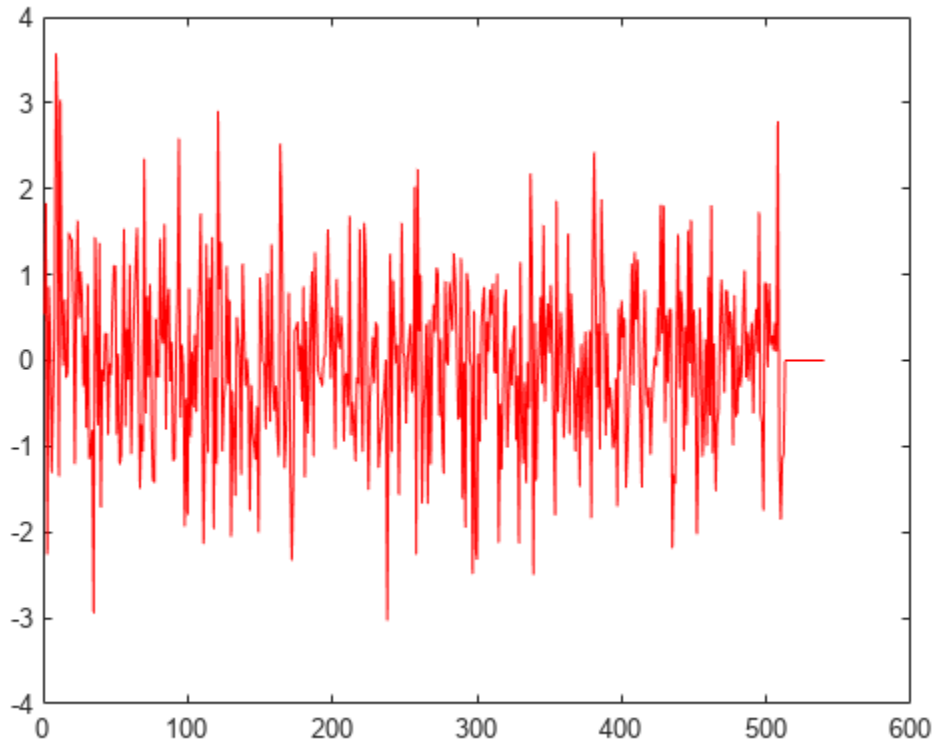
```
asyncBuff = dsp.AsyncBuffer;  
input = randn(512,1);  
write(asyncBuff,input);  
plot(input)  
hold on
```



Store the data that is read from the buffer in `outTotal`.

Plot the input signal and data that is read from the buffer in the same plot. Read data from the buffer until all samples are read. In each iteration of the loop, `randi` determines the number of samples to read. Therefore, the signal is read in as a variable-size signal. The `prevIndex` variable keeps track of the previous index value that contains the data.

```
outTotal = zeros(size(input));
prevIndex = 0;
while asyncBuff.NumUnreadSamples ~= 0
    numToRead = randi([1,64]);
    out = read(asyncBuff,numToRead);
    outTotal(prevIndex+1:prevIndex+numToRead) = out;
    prevIndex = prevIndex+numToRead;
end
plot(outTotal,'r')
hold off
```



Verify that the input data and the data read from the buffer (excluding the underrun samples, if any) are the same. The cumulative number of overrun and underrun samples in the buffer is determined by the `info` function.

```
S = info(asyncBuff)
```

```
S = struct with fields:
    CumulativeOverrun: 0
    CumulativeUnderrun: 28
```

The `CumulativeUnderrun` field shows the number of samples underrun per channel. Underrun occurs if you attempt to read more samples than available.

## Input Arguments

### **asyncBuff** — Async buffer

`dsp.AsyncBuffer` System object

Async buffer, specified as a `dsp.AsyncBuffer` System object.

## Output Arguments

### **S** — Cumulative overrun and underrun information

structure

Cumulative overrun and underrun information, returned as a structure. The fields for `S` are described in the table.

| <b>Field</b>                    | <b>Value</b>   |
|---------------------------------|--|
| <code>CumulativeOverrun</code>  | Number of samples overrun per channel since last call to <code>reset</code> . The number of samples overrun is the number of unread samples overwritten. |
| <code>CumulativeUnderrun</code> | Number of samples underrun per channel since last call to <code>reset</code> . Underrun occurs if you attempt to read more samples than available.       |

The `CumulativeOverrun` and `CumulativeUnderrun` properties are data type `int32`.

## **Version History**

**Introduced in R2017a**

### **See Also**

#### **Functions**

`read` | `write` | `peek`

#### **Objects**

`dsp.AsyncBuffer`

# read

**Package:** dsp

Read data from buffer

## Syntax

```
out = read(asyncBuff)
out = read(asyncBuff, NumRows)
out = read(asyncBuff, NumRows, Overlap)
[out, nUnderrun] = read( ___ )
```

## Description

`out = read(asyncBuff)` returns all unread samples from the buffer, `asyncBuff`.

`out = read(asyncBuff, NumRows)` returns `NumRows` samples from each channel (column) of the buffer.

`out = read(asyncBuff, NumRows, Overlap)` returns `NumRows` samples from each channel and overlaps previously read samples by `Overlap`.

`[out, nUnderrun] = read( ___ )` also returns the number of rows zero-padded if underrun occurred, using any of the previous arguments.

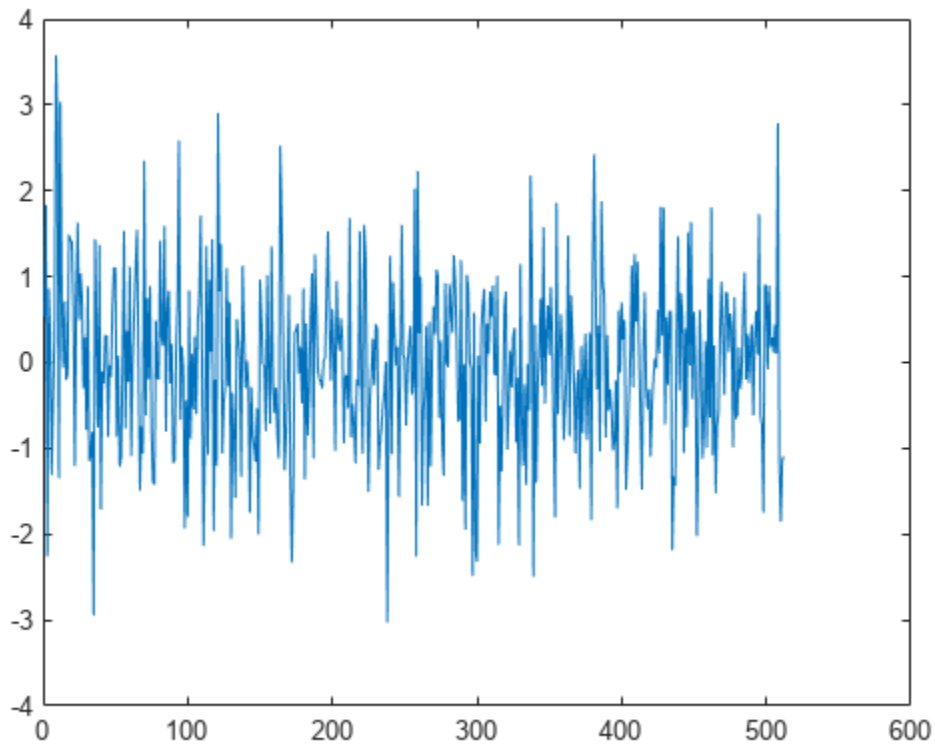
## Examples

### Read Variable Frame Sizes from Buffer

The `dsp.AsyncBuffer System` object™ supports reading variable frame sizes from the buffer.

Create a `dsp.AsyncBuffer System` object. The input is white Gaussian noise with a mean of 0, a standard deviation of 1, and a frame size of 512 samples. Write the input to the buffer using the `write` method.

```
asyncBuff = dsp.AsyncBuffer;
input = randn(512,1);
write(asyncBuff,input);
plot(input)
hold on
```

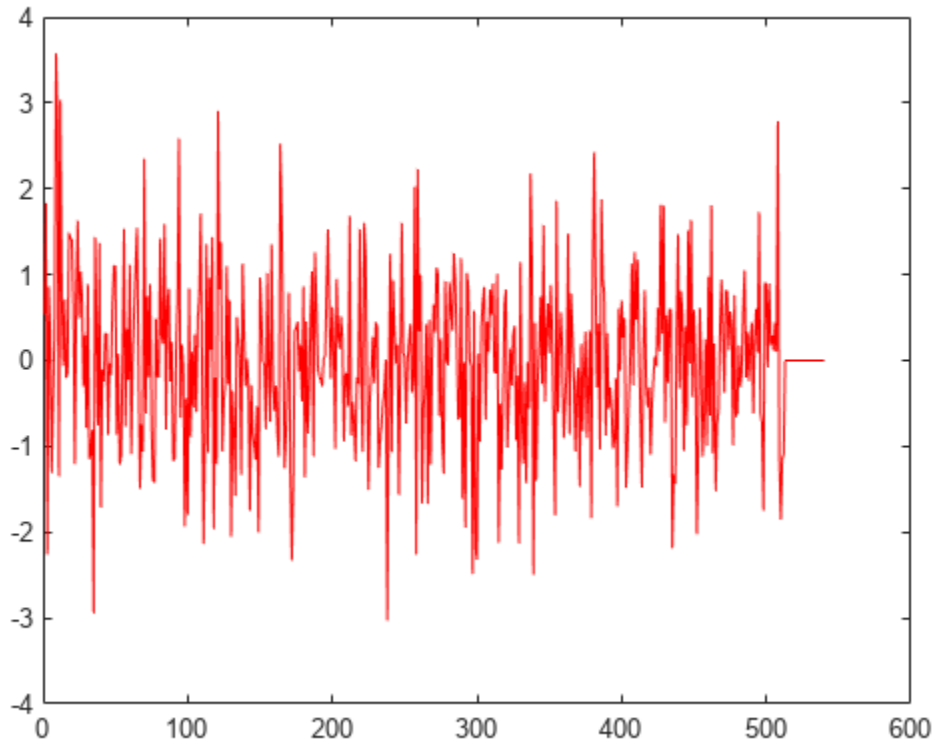


Store the data that is read from the buffer in `outTotal`.

Plot the input signal and data that is read from the buffer in the same plot. Read data from the buffer until all samples are read. In each iteration of the loop, `randi` determines the number of samples to read. Therefore, the signal is read in as a variable-size signal. The `prevIndex` variable keeps track of the previous index value that contains the data.

```
outTotal = zeros(size(input));
prevIndex = 0;
while asyncBuff.NumUnreadSamples ~= 0
    numToRead = randi([1,64]);
    out = read(asyncBuff,numToRead);
    outTotal(prevIndex+1:prevIndex+numToRead) = out;
    prevIndex = prevIndex+numToRead;
end
plot(outTotal,'r')
hold off
```





Verify that the input data and the data read from the buffer (excluding the underrun samples, if any) are the same. The cumulative number of overrun and underrun samples in the buffer is determined by the `info` function.

```
S = info(asyncBuff)
```

```
S = struct with fields:
    CumulativeOverrun: 0
    CumulativeUnderrun: 28
```

The `CumulativeUnderrun` field shows the number of samples underrun per channel. Underrun occurs if you attempt to read more samples than available.

## Input Arguments

### **asyncBuff** — Async buffer

`dsp.AsyncBuffer` System object

Async buffer, specified as a `dsp.AsyncBuffer` System object.

### **NumRows** — Number of samples read from each channel

positive integer

Number of samples read from each channel (column) of the buffer, specified as a positive integer. If the requested number of samples is greater than the number of unread samples, the output is zero-padded.

**Overlap — Number of samples overlapped**

integer

The function returns `NumRows` samples from each channel and overlaps previously read samples by `Overlap`. The total number of samples read is  $\text{NumRows} \times \text{NumChann}$ , where *NumChann* is the number of channels in the buffer. The total number of new samples read is  $(\text{NumRows} - \text{Overlap}) \times \text{NumChann}$ . If the overlap portion contains samples that are overwritten, and are therefore not contiguously written, the output is zero-padded.

**Output Arguments****out — Data read from buffer**

vector | matrix

Data read from the buffer, returned as an array of  $\text{NumRows} \times \text{NumChann}$  samples. If `Overlap` is specified, the function returns  $(\text{NumRows} - \text{Overlap}) \times \text{NumChann}$  samples. If the requested number of samples is greater than the number of unread samples, the output is zero-padded.

Data Types: double

**nUnderrun — Number of samples zero-padded in each channel**

vector | matrix

Number of samples zero-padded in each channel (column) if underrun occurred. Underrun occurs if you attempt to read more samples than available. Samples that are zero-padded in overlapped portions are not counted as underrun.

Data Types: int32

**Version History**

Introduced in R2017a

**See Also****Functions**

info | write | peek

**Objects**

dsp.AsyncBuffer

# write

**Package:** dsp

Write data to buffer

## Syntax

```
nOverrun = write(asyncBuff,x)
```

## Description

`nOverrun = write(asyncBuff,x)` writes the input array, `x`, to the buffering object, `asyncBuff`, and returns the number of samples overrun, `nOverrun`.

## Examples

### Write Variable Frame Sizes to Buffer

Write a sine wave of variable frame size to the buffer. Compute the FFT of the sine wave and visualize the result on an array plot.

Initialize the `dsp.AsyncBuffer`, `dsp.ArrayPlot`, and `dsp.FFT System` objects.

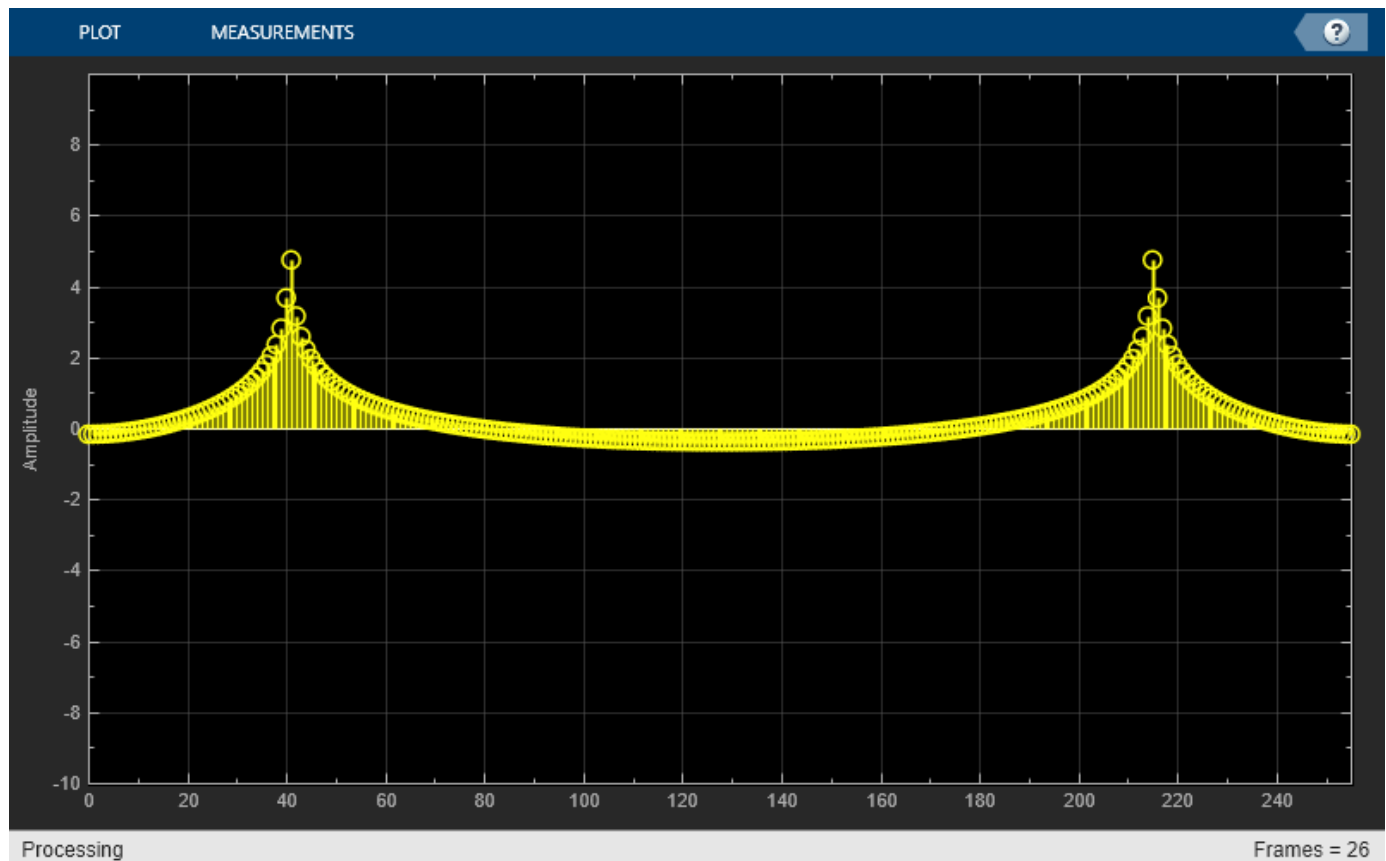
```
asynBuff = dsp.AsyncBuffer;
plotter = dsp.ArrayPlot;
fftObj = dsp.FFT('FFTLenghSource', 'Property', 'FFTLengh', 256);
```

The sine wave is generated using the `sin` function in MATLAB. The `start` and `finish` variables mark the start and finish indices of each frame. If enough data is cached, read from the buffer and perform the FFT. View the FFT on an array plot.

```
start = 1;
for Iter = 1 : 2000
    numToWrite = randi([200,800]);
    finish = start + numToWrite;

    inputData = sin(start:finish)';
    start = finish + 1;

    write(asynBuff,inputData);
    while asynBuff.NumUnreadSamples >= 256
        x = read(asynBuff,256);
        X = abs(fftObj(x));
        plotter(log(X));
    end
end
```



## Input Arguments

### **asyncBuff** — Async buffer

`dsp.AsyncBuffer` System object

Async buffer, specified as a `dsp.AsyncBuffer` System object.

### **x** — Data input

vector | matrix

Data written to the buffer, specified as a vector or a matrix. The maximum number of rows in the buffer is determined by the `Capacity` property of `asyncBuff`. The number of channels in the buffer is determined by the second dimension of the first data written to the buffer. Successive data inputs can vary in the number of rows, but the number of channels must remain fixed. To change the number of channels, you must call `release` on the buffer.

For example, the following is accepted:

```
asyncBuff = dsp.AsyncBuffer;
% First call to write
write(asyncBuff, randn(15,5));
% Add more data with a different number of rows
write(asyncBuff, randn(25,5));
write(asyncBuff, randn(5,5));
```

The following is not accepted and errors out:

```
asyncBuff = dsp.AsyncBuffer;  
% First call to write  
write(asyncBuff,randn(15,5));  
% Add more data with a different number of columns  
write(asyncBuff,randn(15,15));
```

To change the number of channels, call `release` on the buffer.

```
asyncBuff = dsp.AsyncBuffer;  
% First call to write  
write(asyncBuff,randn(15,5));  
release(asyncBuff)  
% Add more data with a different number of columns  
write(asyncBuff,randn(15,15));
```

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

Complex Number Support: Yes

## Output Arguments

### **nOverrun** — Number of samples overrun

scalar

Number of samples overrun in the current call to `write`. The number of samples overrun is the number of unread samples overwritten. If `x` is a multichannel input, then `nOverrun` is the number of rows of data overrun.

Data Types: `int32`

## Version History

Introduced in R2017a

## See Also

### Functions

`info` | `read` | `peek`

### Objects

`dsp.AsyncBuffer`

## peek

**Package:** dsp

Read data from buffer without changing number of unread samples

### Syntax

```
out = peek(asyncBuff)
out = peek(asyncBuff,numRows)
out = peek(asyncBuff,numRows,overlap)
[out,nUnderrun] = peek( ___ )
```

### Description

`out = peek(asyncBuff)` returns all unread samples from the buffer, `asyncBuff`, without changing the number of unread samples in the buffer.

`out = peek(asyncBuff,numRows)` returns `numRows` samples from each channel (column) of the buffer.

`out = peek(asyncBuff,numRows,overlap)` returns `numRows` samples from each channel and overlaps previously read samples by `overlap`.

`[out,nUnderrun] = peek( ___ )` also returns the number of zero-padded rows if underrun occurred, using any of the previous arguments.

### Examples

#### Peek Data from Async Buffer

Read data from the async buffer without changing the number of unread samples using the `peek` function.

Create a `dsp.AsyncBuffer` System object™. The input is a column vector of 100 samples, 1 to 100. Write the data to the buffer.

```
asyncBuff = dsp.AsyncBuffer
asyncBuff =
  AsyncBuffer with properties:
      Capacity: 192000
  NumUnreadSamples: 0
```

```
input = (1:100)';
write(asyncBuff,input);
```

Peek at the first three samples. The output is `[1 2 3]'`.

```
out1 = peek(asyncBuff,3)
```

```
out1 = 3×1
```

```
 1
 2
 3
```

The `NumUnreadSamples` is 100, indicating that the `peek` function has not changed the number of unread samples in the buffer.

```
asyncBuff.NumUnreadSamples
```

```
ans = int32
     100
```

After peeking, read 50 samples using the `read` function. The output is `[1:50]'`.

```
out2 = read(asyncBuff,50)
```

```
out2 = 50×1
```

```
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
 ⋮
```

The `NumUnreadSamples` is 50, indicating that the `read` function has changed the number of unread samples in the buffer.

```
asyncBuff.NumUnreadSamples
```

```
ans = int32
     50
```

Now peek again at the first three samples. The output is `[51 52 53]'`. Verify that the `NumUnreadSamples` is still 50.

```
out3 = peek(asyncBuff,3)
```

```
out3 = 3×1
```

```
51
52
53
```

```
asyncBuff.NumUnreadSamples
```

```
ans = int32
     50
```

Read 50 samples again. The output now contains the sequence [51:100]'. Verify that `NumUnreadSamples` is 0.

```
out4 = read(asyncBuff)
```

```
out4 = 50×1
```

```
51  
52  
53  
54  
55  
56  
57  
58  
59  
60  
⋮
```

```
asyncBuff.NumUnreadSamples
```

```
ans = int32  
0
```

## Input Arguments

### **asyncBuff** — Async buffer

`dsp.AsyncBuffer` System object

Async buffer, specified as a `dsp.AsyncBuffer` System object.

### **numRows** — Number of samples peeked from each channel

positive integer

Number of samples peeked from each channel (column) of the buffer, specified as a positive integer. This operation does not change the number of unread samples in the buffer. If the requested number of samples is greater than the number of unread samples, the output is zero-padded.

### **overlap** — Number of samples overlapped

integer

Number of samples overlapped, specified as an integer. The function returns `numRows` samples from each channel and overlaps previously read samples by `overlap`. The total number of samples peeked is  $\text{numRows} \times \text{NumChann}$ , where `NumChann` is the number of channels in the buffer. The total number of new samples peeked is  $(\text{numRows} - \text{overlap}) \times \text{NumChann}$ . If the overlap portion contains samples that are overwritten, and are therefore not contiguously written, the output is zero-padded.

## Output Arguments

### **out** — Data peeked from buffer

vector | matrix



Data peeked from the buffer, returned as an array of  $\text{numRows} \times \text{NumChann}$  samples. If `overlap` is specified, the function returns  $(\text{numRows} - \text{overlap}) \times \text{NumChann}$  samples. If the requested number of samples is greater than the number of unread samples, the output is zero-padded.

Data Types: `double`

#### **nUnderrun — Number of zero-padded samples in each channel**

`vector` | `matrix`

Number of zero-padded samples in each channel (column) if underrun occurred. Underrun occurs if you attempt to peek more samples than available. Samples that are zero-padded in overlapped portions are not counted as underrun.

Data Types: `int32`

## **Version History**

**Introduced in R2018b**

### **See Also**

#### **Functions**

`info` | `write` | `read`

#### **Objects**

`dsp.AsyncBuffer`

## readHeader

**Package:** dsp

Read file header

### Syntax

```
header = readHeader(reader)
```

### Description

`header = readHeader(reader)` returns the header structure, `header`, from the file specified by the binary file reader, `reader`.

### Examples

#### Read Header Data

Read the header data from a binary file using the `readHeader` function.

Write a header, followed by the data to a binary file named `myfile.dat`. The header is a 1-by-4 matrix of double precision values, followed by a 5-by-1 vector of single-precision values. The data is a sequence of 1000 double-precision values.

```
fid = fopen('myfile.dat','w');  
fwrite(fid,[1 2 3 4],'double');  
fwrite(fid,single((1:5).'),'single');  
fwrite(fid,(1:1000).'),'double');  
fclose(fid);
```

Read the header using a `dsp.BinaryFileReader` object. Specify the expected header structure. This structure specifies only the format of the expected binary file header and does not contain the exact values.

```
reader = dsp.BinaryFileReader('myfile.dat');  
s = struct('A',zeros(1,4),'B',ones(5,1,'single'));  
reader.HeaderStructure = s;
```

Read the header using the `readHeader` function.

```
H = readHeader(reader);  
fprintf('H.A: ')
```

H.A:

```
fprintf('%d ',H.A);
```

1 2 3 4

```
fprintf('\nH.A datatype: %s\n',class(H.A))
```

H.A datatype: double

```
fprintf('H.B: ')
H.B:
fprintf('%d ',H.B);
1 2 3 4 5
fprintf('\nH.B datatype: %s\n',class(H.B))
H.B datatype: single
```

## Input Arguments

### **reader** — Binary file reader

`dsp.BinaryFileReader` System object

Binary file reader object, specified as a `dsp.BinaryFileReader` System object.

## Output Arguments

### **header** — Header structure

structure

Header structure of the binary file, returned as a structure. Each field of the structure is a real matrix of a built-in type. For example, if you specify the `HeaderStructure` property of the `dsp.BinaryFileReader` object to `struct('field1',1:10,'field2',single(1))`, the object writes a header formed by 10 double-precision values, (1:10), followed by one single precision value, `single(1)`. If you do not specify a header, the object returns an empty structure, `struct([])`.

Data Types: `struct`

## Version History

**Introduced in R2016b**

## See Also

### **Objects**

`dsp.BinaryFileReader`

## getFrequencyVector

**Package:** dsp

Vector of frequencies at which estimation is done

### Syntax

```
freq = getFrequencyVector(estimator)
freq = getFrequencyVector(estimator,Fs)
```

### Description

`freq = getFrequencyVector(estimator)` returns the vector of frequencies at which the estimation is done.

`freq = getFrequencyVector(estimator,Fs)` returns the frequency vector assuming an input sample rate, `Fs`.

### Examples

#### Power Spectrum of Multichannel Sinusoidal Signal

Compute the power spectrum of a multichannel sinusoidal signal using the `dsp.SpectrumEstimator` System object™. You can get the vector of frequencies at which the spectrum is estimated using the `getFrequencyVector` function. To compute the resolution bandwidth of the estimate (RBW), use the `getRBW` function.

Generate a three-channel sinusoid sampled at 1 kHz. Specify sinusoidal frequencies of 100, 200, and 300 Hz. The second and third channels have their phases offset from the first by  $\pi/2$  and  $\pi/4$ , respectively.

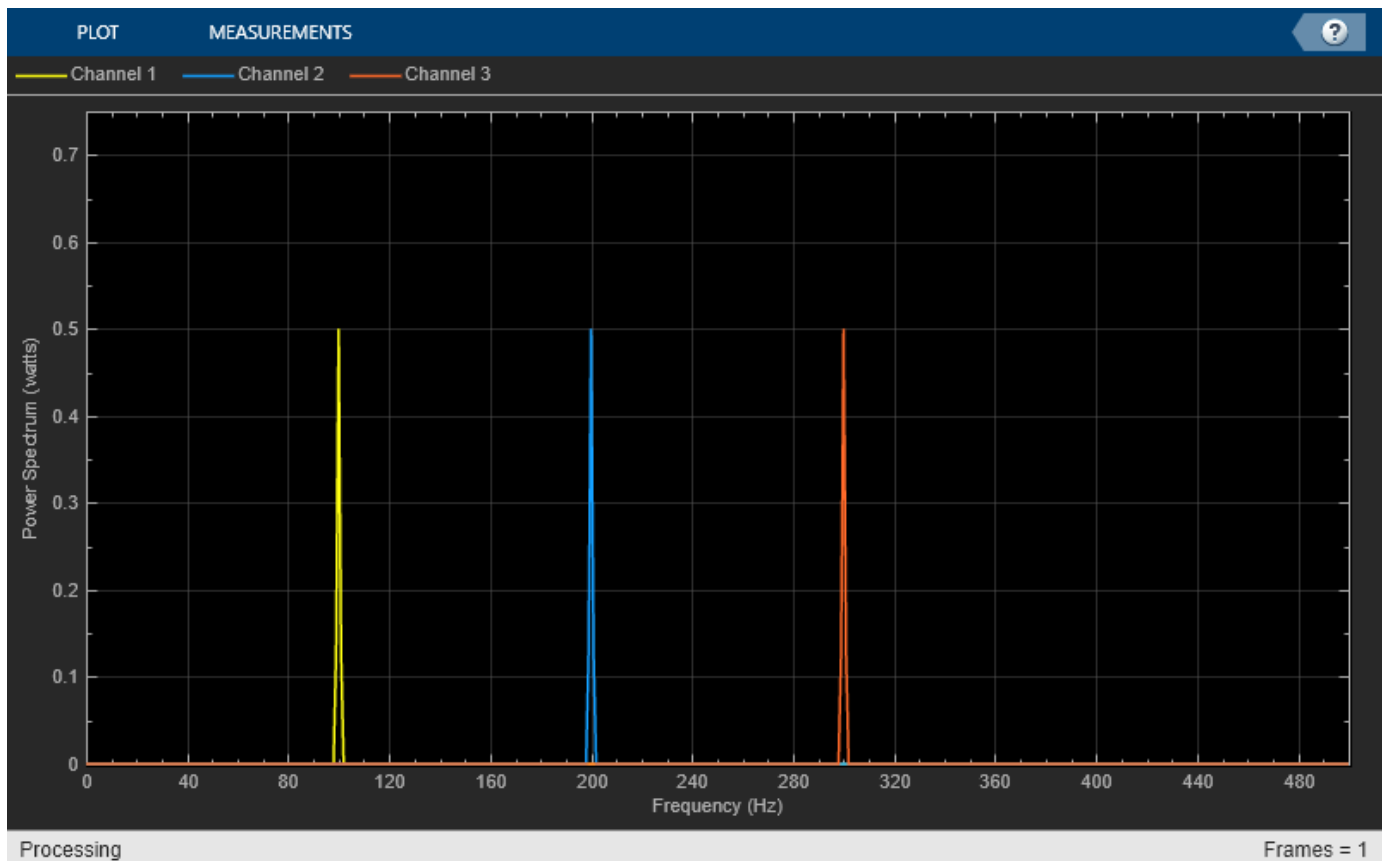
```
sineSignal = dsp.SineWave('SamplesPerFrame',1000,'SampleRate',1000, ...
    'Frequency',[100 200 300],'PhaseOffset',[0 pi/2 pi/4]);
```

Estimate and plot the one-sided spectrum of the signal. Use the `dsp.SpectrumEstimator` object for the computation and the `dsp.ArrayPlot` for the plotting.

```
estimator = dsp.SpectrumEstimator('FrequencyRange','onesided');
plotter = dsp.ArrayPlot('PlotType','Line','YLimits',[0 0.75], ...
    'YLabel','Power Spectrum (watts)','XLabel','Frequency (Hz)');
```

Step through to obtain the data streams and display the spectra of the three channels.

```
y = sineSignal();
pxx = estimator(y);
plotter(pxx)
```



Get the vector of frequencies at which the spectrum is estimated in Hz, using the `getFrequencyVector` function.

```
f = getFrequencyVector(estimator);
```

Compute the resolution bandwidth (RBW) of the estimate using the `getRBW` function.

```
rbw = getRBW(estimator)
```

```
rbw =
```

```
0.0015
```

The resolution bandwidth of the signal power spectrum is 0.0015 Hz. This frequency is the smallest frequency that can be resolved on the spectrum.

## Input Arguments

**estimator** — Estimator object

`dsp.SpectrumEstimator` | `dsp.CrossSpectrumEstimator` | `dsp.TransferFunctionEstimator`

Estimator object, specified as one of the following:

- `dsp.SpectrumEstimator` -- Estimates the power spectrum of the input signal.
- `dsp.CrossSpectrumEstimator` -- Estimates the cross-power spectrum of the input signal.
- `dsp.TransferFunctionEstimator` -- Estimates the transfer function of the system.

**Fs — Input sample rate**

positive scalar

Input sample rate, specified as a real positive scalar.

**Output Arguments****freq — Spectrum frequencies**

vector

Spectrum frequencies, returned as a column vector.

The length of the frequency vector is determined by the `FrequencyRange` and the FFT length.

If you set the `FrequencyRange` to `'onesided'` and the FFT length, `NFFT`, is even, the frequency vector is of length  $NFFT/2+1$ , and covers the interval  $[0, SampleRate/2]$ .

If you set the `FrequencyRange` to `'onesided'` and `NFFT` is odd, the frequency vector is of length  $(NFFT+1)/2$  and covers the interval  $[0, SampleRate/2]$ .

If you set the `FrequencyRange` to `'twosided'`, the frequency vector is of length `NFFT` and covers the interval  $[0, SampleRate]$ .

If you set the `FrequencyRange` to `'centered'`, the frequency vector is of length `NFFT` and covers the range  $[-SampleRate/2, SampleRate/2]$  and  $[-SampleRate/2, SampleRate/2]$  for even and odd length `NFFT`, respectively.

Data Types: `single` | `double`

**Version History****Introduced in R2013b****See Also****Functions**`getRBW`**Objects**`dsp.SpectrumEstimator` | `dsp.CrossSpectrumEstimator` |  
`dsp.TransferFunctionEstimator`**Topics**

“Fractional Delay Filters Using Farrow Structures”

# getFrequencyVector

**Package:** dsp

Get the vector of frequencies at which the short-time FFT is computed

## Syntax

```
freq = getFrequencyVector(stf)
freq = getFrequencyVector(stf,Fs)
```

## Description

`freq = getFrequencyVector(stf)` returns the frequency vector at which the short-time FFT is computed. The input sample rate used is  $2\pi$ .

`freq = getFrequencyVector(stf,Fs)` returns the frequency vector assuming an input sample rate,  $F_s$ .

## Examples

### Get Short-time FFT Frequencies

Get the frequency vector at which the short-time FFT is computed.

Create a `dsp.STFT` object. The STFT object is defined with a 'twosided' frequency range. The frequency vector is defined by the interval  $[0 F_s]$  and has the same length as the FFT length, where  $F_s$  is the input sample rate.

```
stf = dsp.STFT
stf =
  STFT with properties:
      Window: [512x1 double]
  OverlapLength: 256
      FFTLength: 512
  FrequencyRange: 'twosided'
```

When the input sample rate  $F_s$  is not defined, the frequencies are computed in the interval  $[0, 2\pi]$ .

Using the `getFrequencyVector` function, get the vector of frequencies at which the STFT is computed.

```
freq = getFrequencyVector(stf)
freq = 512x1
      0
  0.0123
  0.0245
```

```
0.0368
0.0491
0.0614
0.0736
0.0859
0.0982
0.1104
⋮
```

When the input sample rate  $F_s$  is defined, the frequency vector is defined by the interval  $[0 F_s]$ .

```
Fs = 44100;
freqFs = getFrequencyVector(stf,Fs)
```

```
freqFs = 512×1
104 ×
```

```
0
0.0086
0.0172
0.0258
0.0345
0.0431
0.0517
0.0603
0.0689
0.0775
⋮
```

## Input Arguments

### **stf** — STFT object

`dsp.STFT`

Short-time FFT object whose frequency vector is computed, specified as `dsp.STFT` object.

### **Fs** — Input sample rate

positive scalar

Input sample rate, specified as a real positive scalar. The STFT frequencies are computed in the interval determined by  $F_s$ . For more details, see `freq`.

Data Types: `single` | `double`

## Output Arguments

### **freq** — Frequencies

vector

Frequencies at which the short-time FFT is computed, returned as a column vector.

The length of the frequency vector is determined by the `FrequencyRange` property and the `FFTLlength`.



If you set `FrequencyRange` to `'onesided'` and `FFTlength` is even, the frequency vector is of length  $(\text{FFTlength}/2)+1$ . If you set the `FrequencyRange` to `'onesided'` and `FFTlength` is odd, the frequency vector is of length  $(\text{FFTlength}+1)/2$ .

The frequencies cover the interval  $[0, F_s/2]$ . When  $F_s$  is not specified, the frequencies range from  $[0, \pi]$ .

If you set the `FrequencyRange` property to `'twosided'`, the length of the frequency vector is equal to the value you specify in the `FFTlength` property. The frequencies cover the interval  $[0, F_s]$ . When  $F_s$  is not specified, the frequencies range from  $[0, 2\pi]$ .

Data Types: `single` | `double`

## Version History

Introduced in R2020b

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

`dsp.STFT`

## getRBW

**Package:** dsp

Resolution bandwidth of spectrum

### Syntax

```
RBW = getRBW(estimator)
RBW = getRBW(estimator,Fs)
```

### Description

`RBW = getRBW(estimator)` returns the resolution bandwidth of the spectral estimate.

`RBW = getRBW(estimator,Fs)` returns the resolution bandwidth assuming an input sample rate of `Fs`.

### Examples

#### Power Spectrum of Multichannel Sinusoidal Signal

Compute the power spectrum of a multichannel sinusoidal signal using the `dsp.SpectrumEstimator` System object™. You can get the vector of frequencies at which the spectrum is estimated using the `getFrequencyVector` function. To compute the resolution bandwidth of the estimate (RBW), use the `getRBW` function.

Generate a three-channel sinusoid sampled at 1 kHz. Specify sinusoidal frequencies of 100, 200, and 300 Hz. The second and third channels have their phases offset from the first by  $\pi/2$  and  $\pi/4$ , respectively.

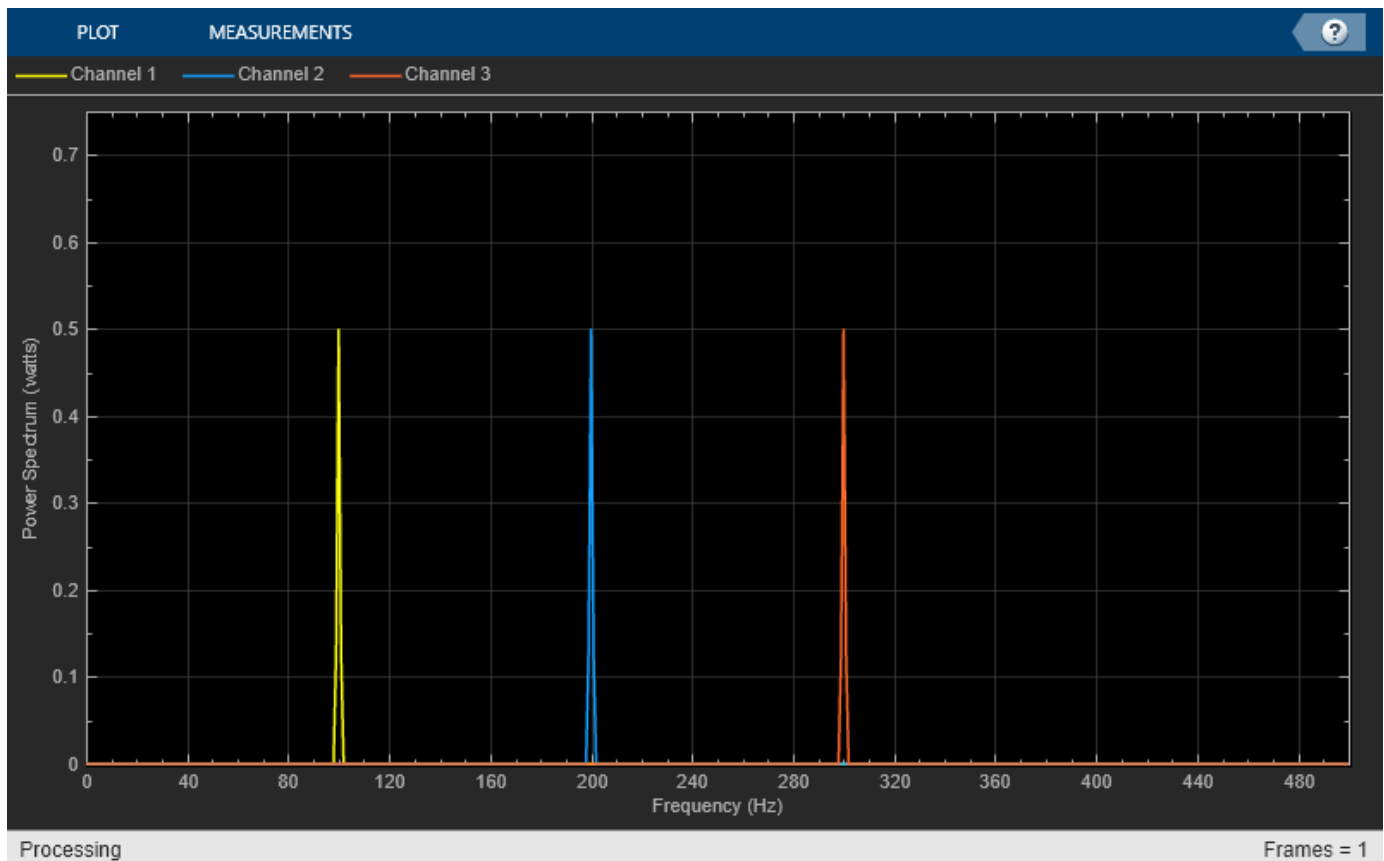
```
sineSignal = dsp.SineWave('SamplesPerFrame',1000,'SampleRate',1000, ...
    'Frequency',[100 200 300],'PhaseOffset',[0 pi/2 pi/4]);
```

Estimate and plot the one-sided spectrum of the signal. Use the `dsp.SpectrumEstimator` object for the computation and the `dsp.ArrayPlot` for the plotting.

```
estimator = dsp.SpectrumEstimator('FrequencyRange','onesided');
plotter = dsp.ArrayPlot('PlotType','Line','YLimits',[0 0.75], ...
    'YLabel','Power Spectrum (watts)','XLabel','Frequency (Hz)');
```

Step through to obtain the data streams and display the spectra of the three channels.

```
y = sineSignal();
pxx = estimator(y);
plotter(pxx)
```



Get the vector of frequencies at which the spectrum is estimated in Hz, using the `getFrequencyVector` function.

```
f = getFrequencyVector(estimator);
```

Compute the resolution bandwidth (RBW) of the estimate using the `getRBW` function.

```
rbw = getRBW(estimator)
```

```
rbw =
```

```
0.0015
```

The resolution bandwidth of the signal power spectrum is 0.0015 Hz. This frequency is the smallest frequency that can be resolved on the spectrum.

## Input Arguments

**estimator** — Estimator object

`dsp.SpectrumEstimator` | `dsp.CrossSpectrumEstimator` |  
`dsp.TransferFunctionEstimator`

Estimator object, specified as one of the following:

- `dsp.SpectrumEstimator` -- Estimates the power spectrum of the input signal.
- `dsp.CrossSpectrumEstimator` -- Estimates the cross-power spectrum of the input signal.
- `dsp.TransferFunctionEstimator` -- Estimates the transfer function of the system.

**Fs — Input sample rate**

positive scalar

Input sample rate, specified as a real positive scalar.

**Output Arguments****RBW — Resolution bandwidth**

scalar

Resolution bandwidth of the estimate, returned as a scalar.

The resolution bandwidth, RBW, is the smallest positive frequency, or frequency interval, that can be resolved. It is equal to  $NENBW * SampleRate / L$ , where  $L$  is the input length, and  $NENBW$  is the normalized effective noise bandwidth of the window.

The data type of RBW matches the data type of the input.

Data Types: `single` | `double`**Version History**

Introduced in R2013b

**See Also****Functions**`getFrequencyVector`**Objects**`dsp.SpectrumEstimator` | `dsp.CrossSpectrumEstimator` |  
`dsp.TransferFunctionEstimator`**Topics**

“Generate a Multithreaded MEX File from a MATLAB Function Using Unfolding”

# int

States from CIC filter

## Compatibility

`mfilt` will be removed in a future release. Refer to the reference page for a specific `mfilt` object to see its recommended replacement.

## Syntax

```
integerstates = int(hm.states)
```

## Description

`integerstates = int(hm.states)` returns the states of a CIC filter in matrix form, rather than as the native `filtstates` object. An important point about `int` is that it quantizes the state values to the smallest number of bits possible while maintaining the values accurately.

## Examples

For many users, the states of multirate filters are most useful as a matrix, but the CIC filters store the states as objects. Here is how you get the states from you CIC filter as a matrix.

```
hm = mfilt.cicinterp;  
hs = hm.states; % Returns a FILTSTATES.CIC object hs.  
states = int(hs); % Convert object hs to a signed integer matrix.
```

After using `int` to convert the states object to a matrix, here is what you get.

Before converting:

```
hm.states  
ans =  
    Integrator: [2x1 States]  
    Comb: [2x1 States]
```

After the conversion and assigning the states to `states`:

```
states  
states =  
     0     0  
     0     0
```

## Version History

Introduced in R2011a

**See Also**

`filtstates.cic` | `dsp.CICDecimator` | `dsp.CICInterpolator`

# isallpass

**Package:** dsp

Verify that discrete-time filter System object is allpass

## Syntax

```
flag = isallpass(sysobj)
flag = isallpass(sysobj,tol)
flag = isallpass( ____, 'Arithmetic', arithType)
```

## Description

`flag = isallpass(sysobj)` returns `true` if the filter System object is allpass.

`flag = isallpass(sysobj,tol)` uses the tolerance `tol` to determine when two numbers are close enough to be considered equal.

`flag = isallpass( ____, 'Arithmetic', arithType)` analyzes the filter System object based on the arithmetic specified in `arithType`, using either of the previous syntaxes.

For more input options, see `isallpass` in Signal Processing Toolbox.

## Examples

### Verify if Filter is Allpass

Create an allpass filter using the `dsp.AllpassFilter` object.

```
apass = dsp.AllpassFilter
```

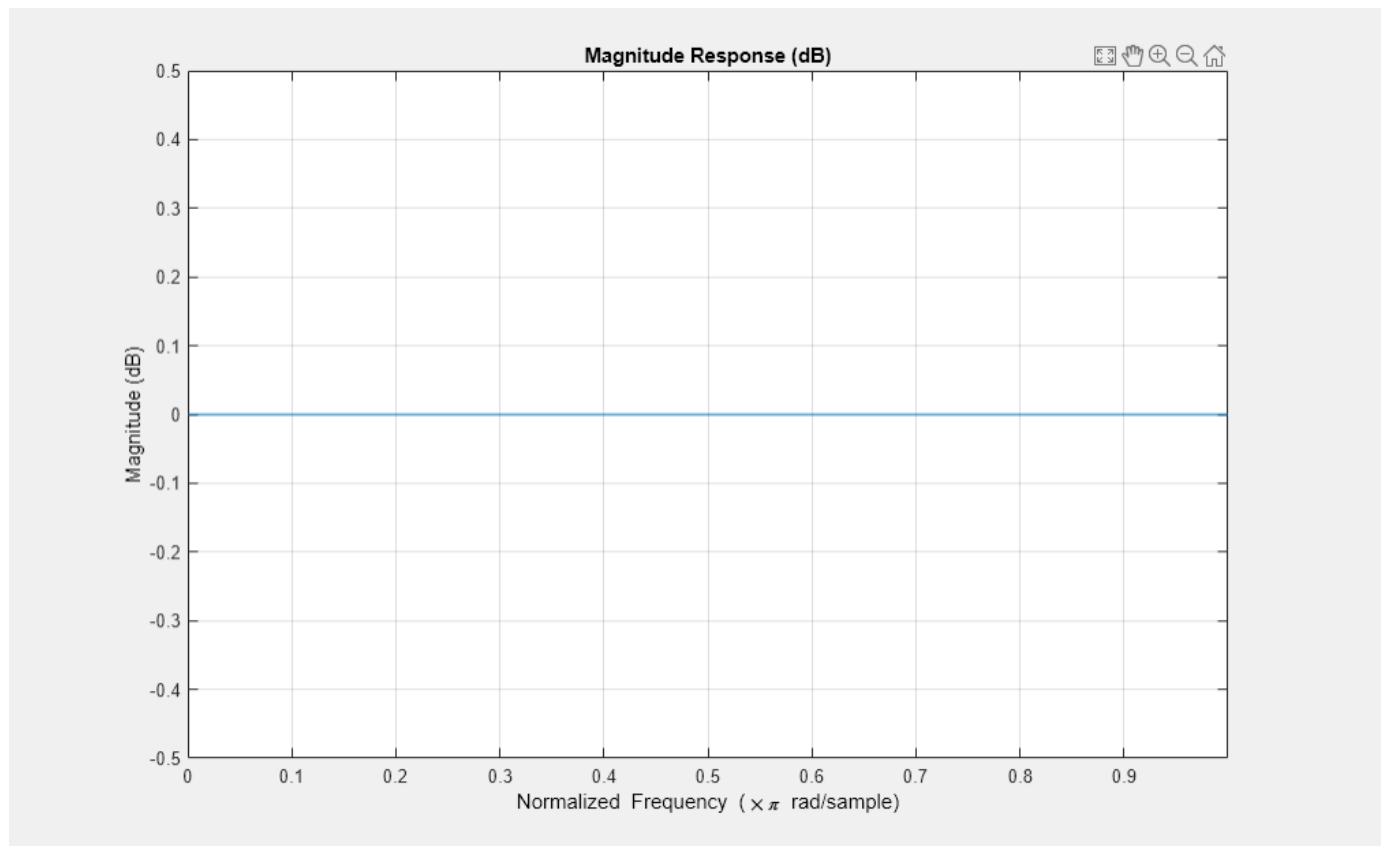
```
apass =
```

```
  dsp.AllpassFilter with properties:
```

```
          Structure: 'Minimum multiplier'
    AllpassCoefficients: [-0.7071 0.5000]
    TrailingFirstOrderSection: false
```

Verify that the filter is an allpass filter by visualizing the magnitude response in `fvtool`.

```
fvtool(apass)
```



Now verify that the filter is an allpass filter using the `isallpass` function.

```
isallpass(apass)
```

```
ans = logical  
     1
```

## Input Arguments

### **sysobj** — Filter System object

filter System object

Input filter, specified as one of the following filter System objects:

- `dsp.AllpassFilter`
- `dsp.AllpoleFilter`
- `dsp.BiquadFilter`
- `dsp.CICCompensationDecimator`
- `dsp.CICCompensationInterpolator`
- `dsp.CICDecimator`
- `dsp.CICInterpolator`



- `dsp.CoupledAllpassFilter`
- `dsp.Differentiator`
- `dsp.FarrowRateConverter`
- `dsp.FilterCascade`
- `dsp.FIRDecimator`
- `dsp.FIRFilter`
- `dsp.FIRHalfbandDecimator`
- `dsp.FIRHalfbandInterpolator`
- `dsp.FIRInterpolator`
- `dsp.FIRRateConverter`
- `dsp.FourthOrderSectionFilter`
- `dsp.FrequencyDomainFIRFilter`
- `dsp.HighpassFilter`
- `dsp.IIRFilter`
- `dsp.IIRHalfbandDecimator`
- `dsp.IIRHalfbandInterpolator`
- `dsp.LowpassFilter`
- `dsp.NotchPeakFilter`
- `dsp.SOSFilter`
- `dsp.VariableBandwidthFIRFilter`
- `dsp.VariableBandwidthIIRFilter`

#### **tol — Tolerance value**

`eps^(2/3)` (default) | positive scalar

Tolerance value to determine when two numbers are close enough to be considered equal, specified as a positive scalar. If not specified, `tol`, defaults to `eps^(2/3)`. Specifying a tolerance may be most helpful in fixed-point allpass filters.

#### **arithType — Arithmetic type**

'double' (default) | 'single' | 'Fixed'

Arithmetic used in the filter analysis, specified as 'double', 'single', or 'Fixed'. When the arithmetic input is not specified and the filter System object is unlocked, the analysis tool assumes a double-precision filter. When the arithmetic input is not specified and the System object is locked, the function performs the analysis based on the data type of the locked input.

The 'Fixed' value applies to filter System objects with fixed-point properties only.

When the 'Arithmetic' input argument is specified as 'Fixed' and the filter object has the data type of the coefficients set to 'Same word length as input', the arithmetic analysis depends on whether the System object is unlocked or locked.

- unlocked -- The analysis object function cannot determine the coefficients data type. The function assumes that the coefficients data type is signed, has a 16-bit word length, and is auto scaled. The function performs fixed-point analysis based on this assumption.

- `locked` -- When the input data type is `'double'` or `'single'`, the analysis object function cannot determine the coefficients data type. The function assumes that the data type of the coefficients is signed, has a 16-bit word length, and is auto scaled. The function performs fixed-point analysis based on this assumption.

To check if the System object is locked or unlocked, use the `isLocked` function.

When the arithmetic input is specified as `'Fixed'` and the filter object has the data type of the coefficients set to a custom numeric type, the object function performs fixed-point analysis based on the custom numeric data type.

## Output Arguments

### **`flag` — Flag to determine if filter is allpass**

`true` or `1` | `false` or `0`

Flag to determine if the filter is allpass, returned as a logical:

- `1` -- Filter is allpass.
- `0` -- Filter is not allpass.

Data Types: `logical`

## Version History

**Introduced in R2013a**

### **See Also**

`isallpass`

# isfir

**Package:** dsp

Verify if filter System object is FIR

## Syntax

```
flag = isfir(sysobj)
```

## Description

`flag = isfir(sysobj)` determines whether the filter System object is an FIR filter. If the filter is an FIR filter, `isfir` returns 1.

To determine whether `sysobj` is an FIR filter, the `isfir` function inspects if the filter in the transfer function form has a scalar denominator. If it does, it is an FIR filter.

## Examples

### Determine if Filter is FIR

Design a Lowpass FIR Filter.

```
d = fdesign.lowpass;
h = design(d,'Systemobject',true)
```

```
h =
dsp.FIRFilter with properties:
```

```
    Structure: 'Direct form'
  NumeratorSource: 'Property'
    Numerator: [-0.0024 -0.0021 0.0068 0.0167 0.0111 -0.0062 -0.0084 0.0093 0.0130 -0.0130]
  InitialConditions: 0
```

Show all properties

Determine if the filter is an FIR filter using the `isfir` function.

```
isfir(h)
```

```
ans = logical
     1
```

`isfir` returns 1 to indicate that the filter is an FIR filter.

## Input Arguments

### **sysobj** — Input filter

filter System object

Input filter, specified as one of the following filter System objects:

- `dsp.AllpassFilter`
- `dsp.AllpoleFilter`
- `dsp.BiquadFilter`
- `dsp.CICCompensationDecimator`
- `dsp.CICCompensationInterpolator`
- `dsp.CICDecimator`
- `dsp.CICInterpolator`
- `dsp.CoupledAllpassFilter`
- `dsp.Differentiator`
- `dsp.FarrowRateConverter`
- `dsp.FilterCascade`
- `dsp.FIRDecimator`
- `dsp.FIRFilter`
- `dsp.FIRHalfbandDecimator`
- `dsp.FIRHalfbandInterpolator`
- `dsp.FIRInterpolator`
- `dsp.FIRRateConverter`
- `dsp.FourthOrderSectionFilter`
- `dsp.HighpassFilter`
- `dsp.IIRFilter`
- `dsp.IIRHalfbandDecimator`
- `dsp.IIRHalfbandInterpolator`
- `dsp.LowpassFilter`
- `dsp.NotchPeakFilter`
- `dsp.SOSFilter`
- `dsp.VariableBandwidthFIRFilter`
- `dsp.VariableBandwidthIIRFilter`

## Output Arguments

### **flag** — Flag to determine whether filter is FIR

1 | 0

Flag to determine whether the filter is FIR, returned as a logical scalar. If the filter is FIR, `isfir` returns a logical 1, else it returns a logical 0.

Data Types: `logical`

## **Version History**

**Introduced in R2011a**

## **See Also**

### **Functions**

`isallpass` | `islinphase` | `ismaxphase` | `isminphase` | `isreal` | `issos` | `isstable`

## isDone

**Package:** dsp

End-of-file status for signal reader object

### Syntax

```
isDone(src)
```

### Description

`isDone(src)` returns a logical value indicating whether or not the `SignalSource` object, `src`, has reached the end of the imported signal. If the `SignalEndAction` property of `src` is set to `Cyclic` repetition, this method returns `true` every time the reader reaches the end.

### Examples

#### Create Signal Source

Create a signal source to output one sample at a time.

```
src1 = dsp.SignalSource;  
src1.Signal = randn(1024,1);  
y1 = zeros(1024,1);  
idx = 1;  
while(~isDone(src1))  
    y1(idx) = src1();  
    idx = idx + 1;  
end
```

Create a signal source to output vectors.

```
src2 = dsp.SignalSource(randn(1024,1),128);  
y2 = src2(); % y2 is a 128-by-1 frame of samples
```

### Input Arguments

#### **src** — Signal reader object

`dsp.SignalReader` System object

Signal reader object, specified as a `dsp.SignalSource` System object.

### Version History

**Introduced in R2012b**

## See Also

### Objects

`dsp.SignalSource`

# islinphase

**Package:** dsp

Verify that discrete-time filter System object is linear phase

## Syntax

```
flag = islinphase(sysobj)
flag = islinphase(sysobj,tol)
flag = islinphase( ____, 'Arithmetic', arithType)
```

## Description

`flag = islinphase(sysobj)` returns `true` if the filter System object has linear phase.

`flag = islinphase(sysobj,tol)` uses the tolerance `tol` to determine when two numbers are close enough to be considered equal. If not specified, `tol` defaults to  $\text{eps}^{(2/3)}$ .

`flag = islinphase( ____, 'Arithmetic', arithType)` analyzes the filter System object based on the arithmetic specified in the `arithType` input using either of the previous syntaxes.

For more input options, see `islinphase` in Signal Processing Toolbox.

## Examples

### Linear and Nonlinear Phase Filters

Use the window method to design a 10th order lowpass FIR filter with the normalized cutoff frequency of 0.55. Verify that the filter has linear phase.

```
firSpecs = fdesign.lowpass('N,Fc',10,0.55);
lpFIR = design(firSpecs,'window','SystemObject',true);

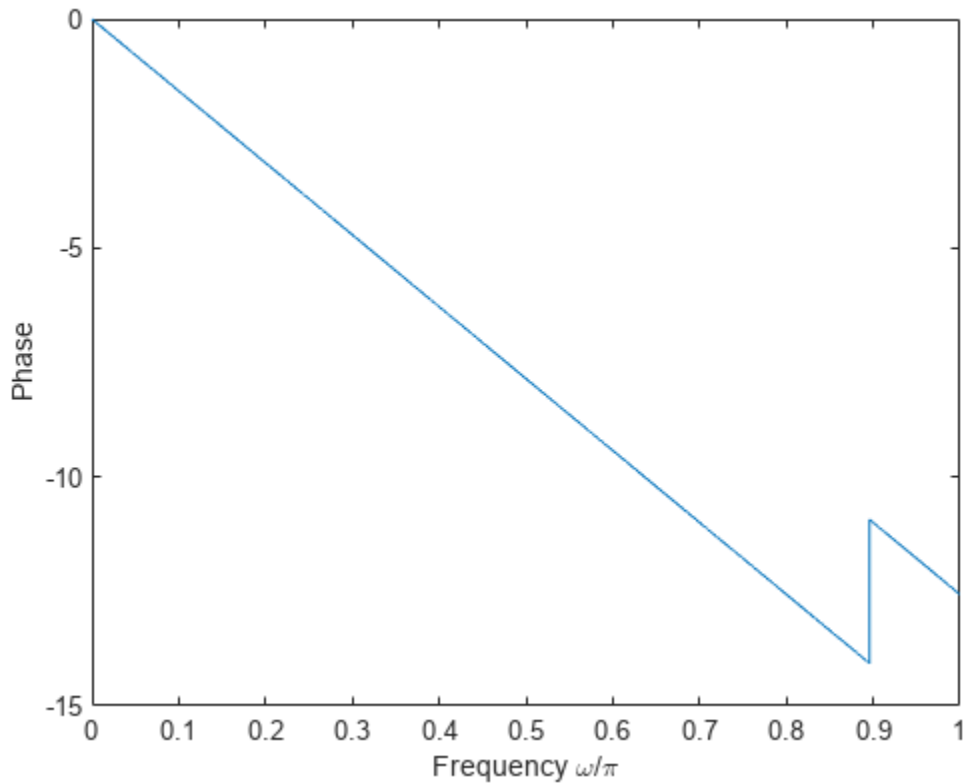
flag = islinphase(lpFIR)

flag = logical
     1
```

Plot the phase response of the filter and verify that it is linear.

```
[phs,w] = phasez(lpFIR);
plot(w/pi,phs)
xlabel('Frequency \omega/\pi')
ylabel('Phase')
```





IIR filters in general do not have linear phase. Verify this by constructing Butterworth, Chebyshev, and elliptic filters with similar specifications. Set the passband frequency to 0.35, stopband frequency to 0.4, passband ripple to 1 dB, and stopband attenuation to 20 dB.

```
Wp = 0.35;
Wst = 0.4;
atten = 20;
rippl = 1;
```

```
buttSpecs = fdesign.lowpass('Fp,Fst,Ap,Ast',Wp,Wst,rippl,atten);
buttIIR = design(buttSpecs,'butter','SystemObject',true);
```

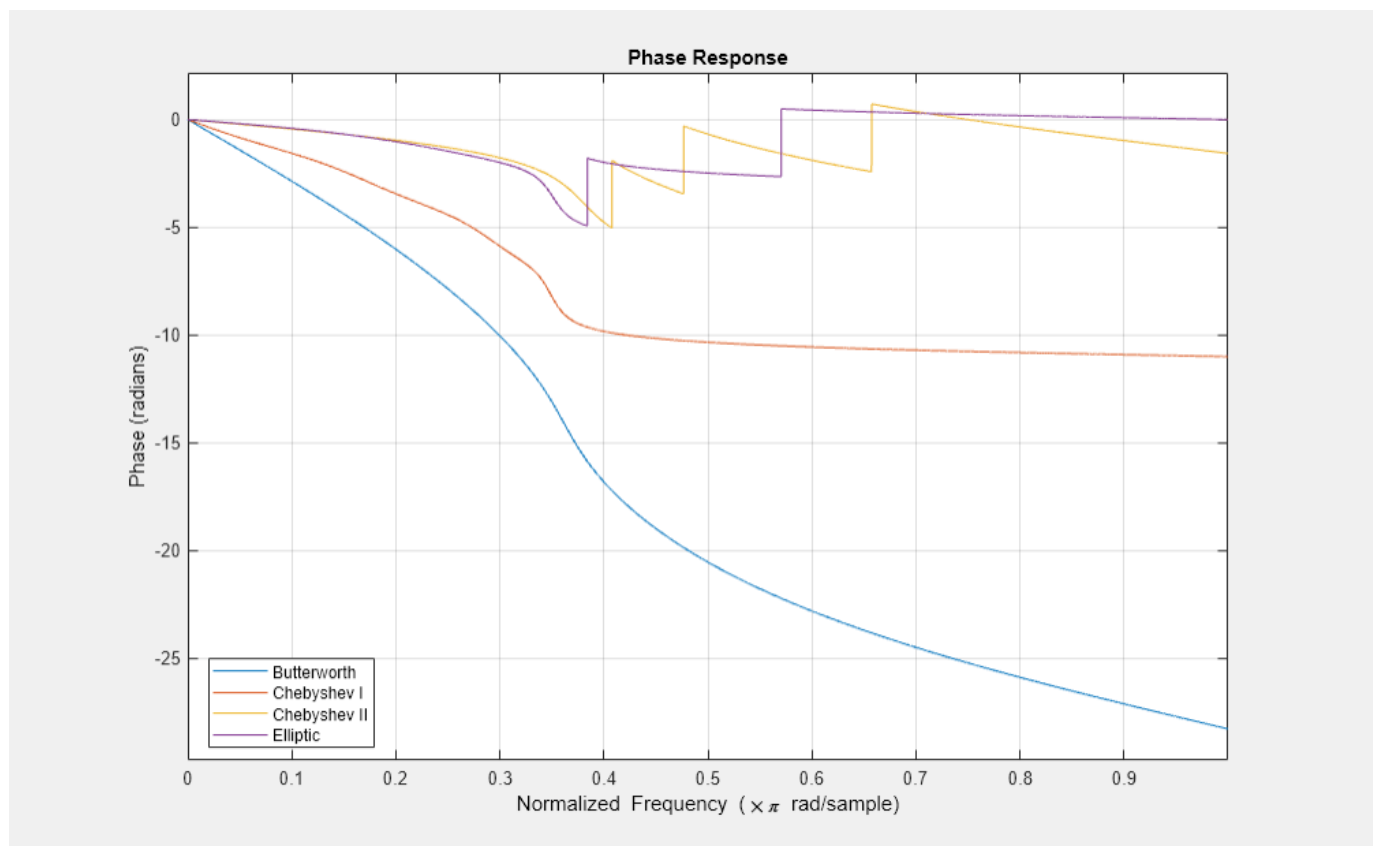
```
chb1Specs = fdesign.lowpass('Fp,Fst,Ap,Ast',Wp,Wst,rippl,atten);
chb1IIR = design(chb1Specs,'cheby1','SystemObject',true);
```

```
chb2Specs = fdesign.lowpass('Fp,Fst,Ap,Ast',Wp,Wst,rippl,atten);
chb2IIR = design(chb2Specs,'cheby2','SystemObject',true);
```

```
ellpSpecs = fdesign.lowpass('Fp,Fst,Ap,Ast',Wp,Wst,rippl,atten);
ellpIIR = design(ellpSpecs,'ellip','SystemObject',true);
```

Plot the phase responses of the filters. Determine whether they have linear phase.

```
fv = fvtool(buttIIR,chb1IIR,chb2IIR,ellpIIR,'Analysis','phase');
legend(fv,'Butterworth','Chebyshev I','Chebyshev II','Elliptic')
```



```
phs = [islinphase(buttIIR) islinphase(chb1IIR) ...
       islinphase(chb2IIR) islinphase(ellpIIR)]
```

*phs = 1x4 logical array*

```
0 0 0 0
```

## Input Arguments

### **sysobj** — Filter System object

filter System object

Input filter, specified as one of the following filter System objects:

- `dsp.AllpassFilter`
- `dsp.AllpoleFilter`
- `dsp.BiquadFilter`
- `dsp.CICCompensationDecimator`
- `dsp.CICCompensationInterpolator`
- `dsp.CICDecimator`

- `dsp.CICInterpolator`
- `dsp.CoupledAllpassFilter`
- `dsp.Differentiator`
- `dsp.FarrowRateConverter`
- `dsp.FilterCascade`
- `dsp.FIRDecimator`
- `dsp.FIRFilter`
- `dsp.FIRHalfbandDecimator`
- `dsp.FIRHalfbandInterpolator`
- `dsp.FIRInterpolator`
- `dsp.FIRRateConverter`
- `dsp.FourthOrderSectionFilter`
- `dsp.HighpassFilter`
- `dsp.IIRFilter`
- `dsp.IIRHalfbandDecimator`
- `dsp.IIRHalfbandInterpolator`
- `dsp.LowpassFilter`
- `dsp.NotchPeakFilter`
- `dsp.SOSFilter`
- `dsp.VariableBandwidthFIRFilter`
- `dsp.VariableBandwidthIIRFilter`

#### **tol — Tolerance value**

`eps^(2/3)` (default) | positive scalar

Tolerance value to determine when two numbers are close enough to be considered equal, specified as a positive scalar. If not specified, `tol`, defaults to `eps^(2/3)`.

#### **arithType — Arithmetic type**

'double' (default) | 'single' | 'Fixed'

Arithmetic used in the filter analysis, specified as 'double', 'single', or 'Fixed'. When the arithmetic input is not specified and the filter System object is unlocked, the analysis tool assumes a double-precision filter. When the arithmetic input is not specified and the System object is locked, the function performs the analysis based on the data type of the locked input.

The 'Fixed' value applies to filter System objects with fixed-point properties only.

When the 'Arithmetic' input argument is specified as 'Fixed' and the filter object has the data type of the coefficients set to 'Same word length as input', the arithmetic analysis depends on whether the System object is unlocked or locked.

- unlocked -- The analysis object function cannot determine the coefficients data type. The function assumes that the coefficients data type is signed, has a 16-bit word length, and is auto scaled. The function performs fixed-point analysis based on this assumption.
- locked -- When the input data type is 'double' or 'single', the analysis object function cannot determine the coefficients data type. The function assumes that the data type of the coefficients is

signed, has a 16-bit word length, and is auto scaled. The function performs fixed-point analysis based on this assumption.

To check if the System object is locked or unlocked, use the `isLocked` function.

When the arithmetic input is specified as 'Fixed' and the filter object has the data type of the coefficients set to a custom numeric type, the object function performs fixed-point analysis based on the custom numeric data type.

## Output Arguments

### **flag** — Flag to determine if filter has linear phase

`true` or `1` | `false` or `0`

Flag to determine if the filter has linear phase, returned as a logical:

- `1` -- Filter has linear phase.
- `0` -- Filter has nonlinear phase.

Data Types: `logical`

## Version History

**Introduced in R2013a**

### **See Also**

`islinphase`

# ismaxphase

**Package:** dsp

Verify that discrete-time filter System object is maximum phase

## Syntax

```
flag = ismaxphase(sysobj)
flag = ismaxphase(sysobj,tol)
flag = ismaxphase( ____, 'Arithmetic', arithType)
```

## Description

`flag = ismaxphase(sysobj)` returns `true` if the filter System object has maximum phase.

`flag = ismaxphase(sysobj,tol)` uses the tolerance `tol` to determine when two numbers are close enough to be considered equal. If not specified, `tol` defaults to  $\text{eps}^{(2/3)}$ .

`flag = ismaxphase( ____, 'Arithmetic', arithType)` analyzes the filter System object based on the arithmetic specified in the `arithType` input using either of the previous syntaxes.

For more input options, see `ismaxphase` in Signal Processing Toolbox.

## Examples

### Determine if Filter has Maximum Phase

Design an allpass filter and determine if the filter has maximum phase.

Using the `dsp.AllpassFilter` System object™, design an allpass filter that uses the minimum multiplier structure.

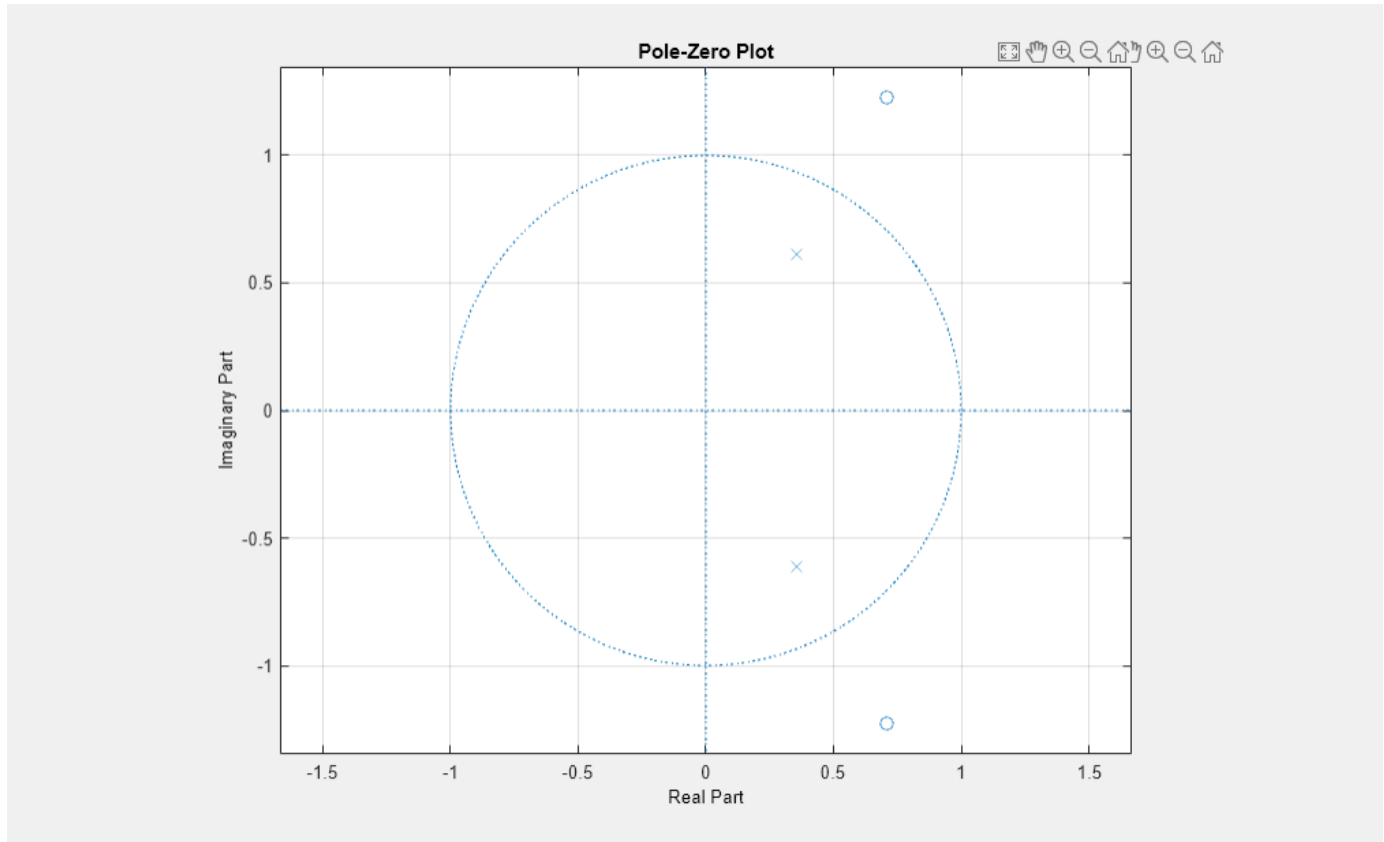
```
a = dsp.AllpassFilter
a =
    dsp.AllpassFilter with properties:
        Structure: 'Minimum multiplier'
        AllpassCoefficients: [-0.7071 0.5000]
        TrailingFirstOrderSection: false
```

Using the `ismaxphase` function, determine if the filter has maximum phase.

```
ismaxphase(a)
ans = logical
     1
```

Verify the location of poles and zeros of the filter transfer function on the z-plane. By definition, the zeros of the maximum phase filter must be outside the unit circle.

`zplane(a)`



## Input Arguments

### **sysobj** — Filter System object

filter System object

Input filter, specified as one of the following filter System objects:

- `dsp.AllpassFilter`
- `dsp.AllpoleFilter`
- `dsp.BiquadFilter`
- `dsp.CICCompensationDecimator`
- `dsp.CICCompensationInterpolator`
- `dsp.CICDecimator`
- `dsp.CICInterpolator`
- `dsp.CoupledAllpassFilter`
- `dsp.Differentiator`

- `dsp.FarrowRateConverter`
- `dsp.FilterCascade`
- `dsp.FIRDecimator`
- `dsp.FIRFilter`
- `dsp.FIRHalfbandDecimator`
- `dsp.FIRHalfbandInterpolator`
- `dsp.FIRInterpolator`
- `dsp.FIRRateConverter`
- `dsp.FourthOrderSectionFilter`
- `dsp.HighpassFilter`
- `dsp.IIRFilter`
- `dsp.IIRHalfbandDecimator`
- `dsp.IIRHalfbandInterpolator`
- `dsp.LowpassFilter`
- `dsp.NotchPeakFilter`
- `dsp.SOSFilter`
- `dsp.VariableBandwidthFIRFilter`
- `dsp.VariableBandwidthIIRFilter`

#### **tol — Tolerance**

`eps^(2/3)` (default) | positive scalar

Tolerance value to determine when two numbers are close enough to be considered equal, specified as a positive scalar. If not specified, `tol` defaults to `eps^(2/3)`.

#### **arithType — Arithmetic type**

'double' (default) | 'single' | 'Fixed'

Arithmetic used in the filter analysis, specified as 'double', 'single', or 'Fixed'. When the arithmetic input is not specified and the filter System object is unlocked, the analysis tool assumes a double-precision filter. When the arithmetic input is not specified and the System object is locked, the function performs the analysis based on the data type of the locked input.

The 'Fixed' value applies to filter System objects with fixed-point properties only.

When the 'Arithmetic' input argument is specified as 'Fixed' and the filter object has the data type of the coefficients set to 'Same word length as input', the arithmetic analysis depends on whether the System object is unlocked or locked.

- unlocked -- The analysis object function cannot determine the coefficients data type. The function assumes that the coefficients data type is signed, has a 16-bit word length, and is auto scaled. The function performs fixed-point analysis based on this assumption.
- locked -- When the input data type is 'double' or 'single', the analysis object function cannot determine the coefficients data type. The function assumes that the data type of the coefficients is signed, has a 16-bit word length, and is auto scaled. The function performs fixed-point analysis based on this assumption.

To check if the System object is locked or unlocked, use the `isLocked` function.

When the arithmetic input is specified as 'Fixed' and the filter object has the data type of the coefficients set to a custom numeric type, the object function performs fixed-point analysis based on the custom numeric data type.

## Output Arguments

### **flag** — Flag to determine if filter has maximum phase

true or 1 | false or 0

Flag to determine if the filter has maximum phase, returned as a logical:

- 1 -- Filter has maximum phase.
- 0 -- Filter has non maximum phase.

Data Types: logical

## More About

### Maximum Phase Filters

A causal and stable discrete-time system is said to be *strictly maximum-phase* when all its zeros are outside the unit circle. A causal and stable LTI system is a *maximum-phase* system if its inverse is causal and unstable.

Such a system is called a *maximum-phase system* because it has the maximum group delay (`grpdelay`) of the set of systems that have the same magnitude response.

## Version History

Introduced in R2013a

### See Also

`ismaxphase` | `isminphase`



# isminphase

**Package:** dsp

Verify that discrete-time filter System object is minimum phase

## Syntax

```
flag = isminphase(sysobj)
flag = isminphase(sysobj,tol)
flag = isminphase( ____, 'Arithmetic', arithType)
```

## Description

`flag = isminphase(sysobj)` returns `true` if the filter System object has minimum phase.

`flag = isminphase(sysobj,tol)` uses the tolerance `tol` to determine when two numbers are close enough to be considered equal. If not specified, `tol` defaults to  $\text{eps}^{(2/3)}$ .

`flag = isminphase( ____, 'Arithmetic', arithType)` analyzes the filter System object based on the arithmetic specified in the `arithType` input using either of the previous syntaxes.

For more input options, see `isminphase` in Signal Processing Toolbox.

## Examples

### Determine if Filter Has Minimum Phase and is Stable

Design a Chebyshev Type I IIR filter and determine if the filter has minimum phase and is stable.

Using the `fdesign.lowpass` and `design` functions, design a Chebyshev Type I IIR filter with a passband ripple of 0.5 dB and a 3 dB cutoff frequency at 9600 Hz.

```
Fs = 48000; % Sampling frequency of input signal
d = fdesign.lowpass('N,F3dB,Ap', 10, 9600, .5, Fs);
filt = design(d, 'cheby1', 'Systemobject', true)

filt =
    dsp.SOSFilter with properties:
        Structure: 'Direct form II'
        CoefficientSource: 'Property'
        Numerator: [5x3 double]
        Denominator: [5x3 double]
        HasScaleValues: true
        ScaleValues: [0.3318 0.2750 0.1876 0.0904 0.0225 0.9441]
```

Show all properties

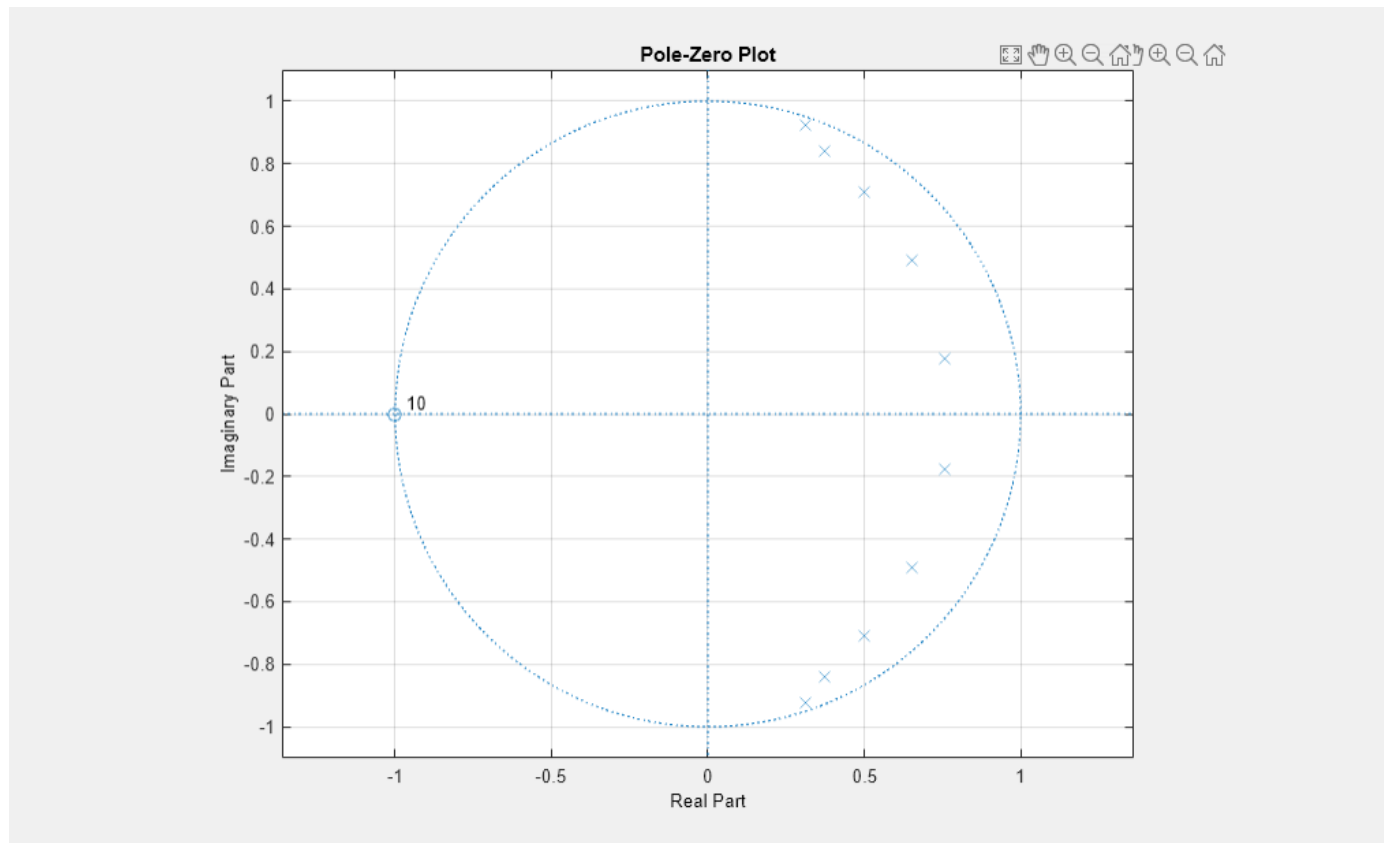
Using the `isminphase` function, determine if the filter has minimum phase.

```
isminphase(filt)
```

```
ans = logical
      1
```

Verify the location of poles and zeros of the filter transfer function on the z-plane. By definition, the poles and zeros of the minimum phase filter must be on or inside the unit circle.

```
zplane(filt)
```



All minimum phase filters are stable. To verify if the designed filter is stable, use the `isstable` function.

```
isstable(filt)
```

```
ans = logical
      1
```

## Input Arguments

**sysobj** — Filter System object

filter System object

Input filter, specified as one of the following filter System objects:

- `dsp.AllpassFilter`
- `dsp.AllpoleFilter`
- `dsp.BiquadFilter`
- `dsp.CICCompensationDecimator`
- `dsp.CICCompensationInterpolator`
- `dsp.CICDecimator`
- `dsp.CICInterpolator`
- `dsp.CoupledAllpassFilter`
- `dsp.Differentiator`
- `dsp.FarrowRateConverter`
- `dsp.FilterCascade`
- `dsp.FIRDecimator`
- `dsp.FIRFilter`
- `dsp.FIRHalfbandDecimator`
- `dsp.FIRHalfbandInterpolator`
- `dsp.FIRInterpolator`
- `dsp.FIRRateConverter`
- `dsp.FourthOrderSectionFilter`
- `dsp.HighpassFilter`
- `dsp.IIRFilter`
- `dsp.IIRHalfbandDecimator`
- `dsp.IIRHalfbandInterpolator`
- `dsp.LowpassFilter`
- `dsp.NotchPeakFilter`
- `dsp.SOSFilter`
- `dsp.VariableBandwidthFIRFilter`
- `dsp.VariableBandwidthIIRFilter`

**tol — Tolerance value**

`eps^(2/3)` (default) | positive scalar

Tolerance value to determine when two numbers are close enough to be considered equal, specified as a positive scalar. If not specified, `tol` defaults to `eps^(2/3)`.

**arithType — Arithmetic type**

'double' (default) | 'single' | 'Fixed'

Arithmetic used in the filter analysis, specified as 'double', 'single', or 'Fixed'. When the arithmetic input is not specified and the filter System object is unlocked, the analysis tool assumes a double-precision filter. When the arithmetic input is not specified and the System object is locked, the function performs the analysis based on the data type of the locked input.

The 'Fixed' value applies to filter System objects with fixed-point properties only.

When the 'Arithmetic' input argument is specified as 'Fixed' and the filter object has the data type of the coefficients set to 'Same word length as input', the arithmetic analysis depends on whether the System object is unlocked or locked.

- unlocked -- The analysis object function cannot determine the coefficients data type. The function assumes that the coefficients data type is signed, has a 16-bit word length, and is auto scaled. The function performs fixed-point analysis based on this assumption.
- locked -- When the input data type is 'double' or 'single', the analysis object function cannot determine the coefficients data type. The function assumes that the data type of the coefficients is signed, has a 16-bit word length, and is auto scaled. The function performs fixed-point analysis based on this assumption.

To check if the System object is locked or unlocked, use the `isLocked` function.

When the arithmetic input is specified as 'Fixed' and the filter object has the data type of the coefficients set to a custom numeric type, the object function performs fixed-point analysis based on the custom numeric data type.

## Output Arguments

### **flag** — Flag to determine if filter has minimum phase

true or 1 | false or 0

Flag to determine if the filter has minimum phase, returned as a logical:

- 1 -- Filter has minimum phase.
- 0 -- Filter has non minimum phase.

Data Types: logical

## More About

### Minimum Phase Filters

A causal and stable discrete-time system is said to be *strictly minimum-phase* when all its zeros are inside the unit circle. A causal and stable LTI system is a *minimum-phase* system if its inverse is causal and stable as well.

Such a system is called a *minimum-phase system* because it has the minimum group delay (`grpdelay`) of the set of systems that have the same magnitude response.

## Version History

Introduced in R2013a

### See Also

`isminphase` | `ismaxphase`

# isNewDataReady

Check spectrum analyzer for new data

## Syntax

```
flag = isNewDataReady(scope)
```

## Description

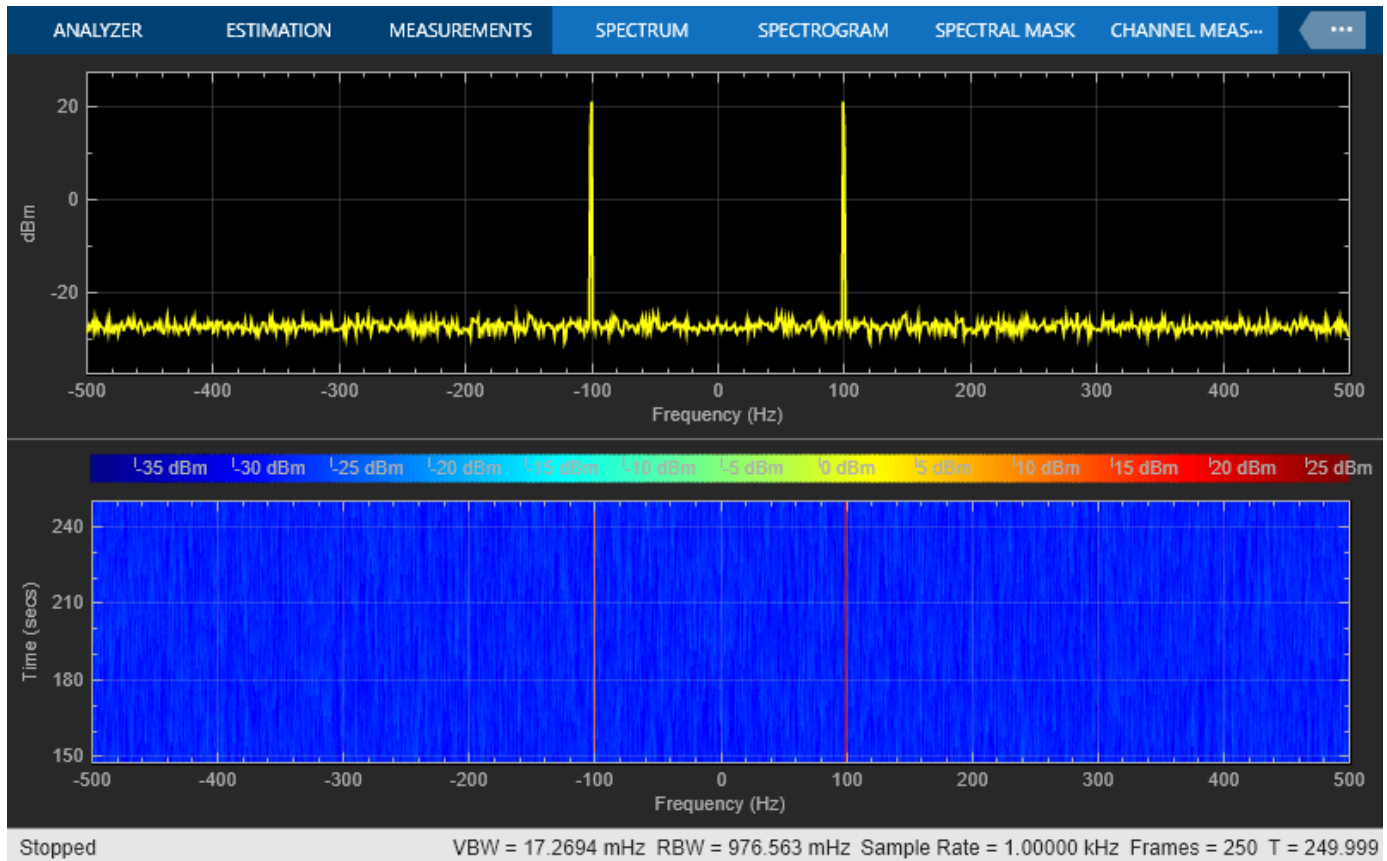
`flag = isNewDataReady(scope)` indicates whether the spectrum analyzer `scope` displays new spectral estimates. When logging spectrum analyzer data from the `spectrumAnalyzer` scope, use this function to ignore duplicate spectra from the `getSpectrumData` function.

## Examples

### Log Spectrum Data

Generate a discrete sine wave. Then display the sine wave signal using the Spectrum Analyzer. While the Spectrum Analyzer is running, save the spectrum data to a table. The Spectrum Analyzer does not update at every time step. To avoid saving that redundant spectrum data, use the `isNewDataReady` function.

```
wave = dsp.SineWave(Frequency=100,SampleRate=1000);  
wave.SamplesPerFrame = 1000;  
scope = spectrumAnalyzer(SampleRate=wave.SampleRate,...  
    ViewType="spectrum-and-spectrogram");  
data = [];  
  
for ii = 1:250  
    x = wave() + 0.05*randn(1000,1);  
    scope(x);  
    if scope.isNewDataReady  
        data = [data;getSpectrumData(scope)];  
    end  
end  
  
release(scope);
```



Show the first five rows in the table. You can see gaps in the simulation time in the data table. The missing rows indicate times where the Spectrum Analyzer was waiting for additional samples to update the spectrum. The `isNewDataReady` function prevented the script from saving that redundant data.

```
data(1:5, :)
```

ans =

5x4 table

| SimulationTime | Spectrum        | Spectrogram       | FrequencyVector |
|----------------|-----------------|-------------------|-----------------|
| {[1.9990]}     | {1024x1 double} | {100x1024 double} | {1024x1 double} |
| {[2.9990]}     | {1024x1 double} | {100x1024 double} | {1024x1 double} |
| {[3.9990]}     | {1024x1 double} | {100x1024 double} | {1024x1 double} |
| {[4.9990]}     | {1024x1 double} | {100x1024 double} | {1024x1 double} |
| {[5.9990]}     | {1024x1 double} | {100x1024 double} | {1024x1 double} |

## Input Arguments

**scope** — Spectrum analyzer  
object name

Spectrum analyzer from which you want to save data, specified as a `spectrumAnalyzer` object.

## Output Arguments

### flag — Flag indicating new data

true | false

Flag indicating new data, returned as one of these:

- true -- The Spectrum Analyzer shows new data.
- false -- The Spectrum Analyzer shows the same spectrum and no new data is available.

## Version History

### Introduced in R2017b

### R2022a: Support for `dsp.SpectrumAnalyzer` object will be removed

*Not recommended starting in R2022a*

The `isNewDataReady` function will stop supporting the `dsp.SpectrumAnalyzer` object in a future release. Use the `spectrumAnalyzer` object instead.

### Update Code

No updates to your code are required except replacing instances of `dsp.SpectrumAnalyzer` with `spectrumAnalyzer`.

This table shows how the `isNewDataReady` function typically uses the `dsp.SpectrumAnalyzer` System object and explains how to update existing code to use the `spectrumAnalyzer` object.

| Discouraged Usage  | Recommended Replacement  |
|--|--|
| <pre> wave = dsp.SineWave('Frequency',100,'SampleRate',1000); wave.SamplesPerFrame = 1000; scope = dsp.SpectrumAnalyzer('SampleRate',wave.SampleRate,     'ReducePlotRate',false,...     'ViewType','Spectrum and spectrogram'); data = [];  for ii = 1:250     x = wave() + 0.05*randn(1000,1);     scope(x);     if scope.isNewDataReady         data = [data;getSpectrumData(scope)];     end end  release(scope); </pre> | <pre> wave = spectrumAnalyzer(SineWave(Frequency=100,SampleRate=1000); wave.SamplesPerFrame = 1000; scope = spectrumAnalyzer(SampleRate=wave.SampleRate,     ViewType="spectrum-and-spectrogram"); data = [];  for ii = 1:250     x = wave() + 0.05*randn(1000,1);     scope(x);     if scope.isNewDataReady         data = [data;getSpectrumData(scope)];     end end  release(scope); </pre> |

## See Also

`spectrumAnalyzer` | `getSpectrumData` | `getMeasurementsData`

## isreal

**Package:** dsp

Verify that discrete-time filter System object is real

### Syntax

```
flag = isreal(sysobj)
```

### Description

`flag = isreal(sysobj)` determines if the filter coefficients are real. If the filter coefficients are real, `isreal` returns a logical 1. If the filter coefficients are complex, `isreal` returns a logical 0. Complex filters have one or more coefficients with nonzero imaginary parts.

### Examples

#### Check if Filter Coefficients Are Real

Create a `dsp.SOSFilter` System object™. Pass a fixed-point input to the object. Test the coefficients of the fixed-point filter to see if they are strictly real.

```
d = fdesign.lowpass('n,fp,ap,ast',5,0.4,0.5,20);  
sosFilter = design(d,'ellip',SystemObject=true)
```

```
sosFilter =  
  dsp.SOSFilter with properties:  
  
      Structure: 'Direct form II'  
CoefficientSource: 'Property'  
      Numerator: [3x3 double]  
      Denominator: [3x3 double]  
HasScaleValues: true  
ScaleValues: [0.6252 0.5432 0.4861 1]
```

Show all properties

```
IsRealBefore = isreal(sosFilter)
```

```
IsRealBefore = logical  
             1
```

Pass a fixed-point input to the object.

```
fiInput = fi(randn(1000,2),1,32,16);  
fiOutput = sosFilter(fiInput);  
IsRealAfter = isreal(sosFilter)
```



```
IsRealAfter = logical  
1
```

The `isreal` function returns a value of 1, indicating that the filter coefficients are real.

## Input Arguments

### **sysobj** — Input filter

filter System object

Input filter, specified as one of the following filter System objects:

- `dsp.AllpassFilter`
- `dsp.AllpoleFilter`
- `dsp.BiquadFilter`
- `dsp.CICCompensationDecimator`
- `dsp.CICCompensationInterpolator`
- `dsp.CICDecimator`
- `dsp.CICInterpolator`
- `dsp.CoupledAllpassFilter`
- `dsp.Differentiator`
- `dsp.FarrowRateConverter`
- `dsp.FilterCascade`
- `dsp.FIRDecimator`
- `dsp.FIRFilter`
- `dsp.FIRHalfbandDecimator`
- `dsp.FIRHalfbandInterpolator`
- `dsp.FIRInterpolator`
- `dsp.FIRRateConverter`
- `dsp.FourthOrderSectionFilter`
- `dsp.HighpassFilter`
- `dsp.IIRFilter`
- `dsp.IIRHalfbandDecimator`
- `dsp.IIRHalfbandInterpolator`
- `dsp.LowpassFilter`
- `dsp.NotchPeakFilter`
- `dsp.SOSFilter`
- `dsp.VariableBandwidthFIRFilter`
- `dsp.VariableBandwidthIIRFilter`

## Output Arguments

**flag** — Flag to determine whether filter coefficients are real

1 | 0

Flag to determine whether the filter coefficients are real, returned as a logical scalar. If the filter coefficients are real, `isreal` returns a logical 1, else it returns a logical 0.

Data Types: `logical`

## Version History

Introduced in R2011a

## See Also

### Functions

`isfir` | `islinphase` | `ismaxphase` | `isminphase` | `issos` | `isstable` | `isallpass`

# issos

**Package:** dsp

Verify if discrete-time System object filter is in second-order sections form

## Syntax

```
flag = issos(sysobj)
```

## Description

`flag = issos(sysobj)` determines whether the filter System object is in second-order sections (SOS) form, returning a logical 1 if true and a logical 0 if false.

## Examples

### Design Lowpass SOS Filter

By default, `fdesign` and `design` functions return SOS filters when possible. This example designs a lowpass SOS filter that uses fixed-point arithmetic.

```
d = fdesign.lowpass('n,fp,ap,ast',40,0.55,0.1,60);
hd = design(d,'ellip','SystemObject',true);
```

Using the `issos` function, verify whether the designed filter is in SOS form.

```
IsSOS = issos(hd)
```

```
IsSOS = logical
       1
```

The filter is in second-order section form.

## Input Arguments

### **sysobj** — Input filter

filter System object

Input filter, specified as one of the following filter System objects:

- `dsp.AllpassFilter`
- `dsp.AllpoleFilter`
- `dsp.BiquadFilter`
- `dsp.CICCompensationDecimator`
- `dsp.CICCompensationInterpolator`

- `dsp.CICDecimator`
- `dsp.CICInterpolator`
- `dsp.CoupledAllpassFilter`
- `dsp.Differentiator`
- `dsp.FarrowRateConverter`
- `dsp.FilterCascade`
- `dsp.FIRDecimator`
- `dsp.FIRFilter`
- `dsp.FIRHalfbandDecimator`
- `dsp.FIRHalfbandInterpolator`
- `dsp.FIRInterpolator`
- `dsp.FIRRateConverter`
- `dsp.FourthOrderSectionFilter`
- `dsp.HighpassFilter`
- `dsp.IIRFilter`
- `dsp.IIRHalfbandDecimator`
- `dsp.IIRHalfbandInterpolator`
- `dsp.LowpassFilter`
- `dsp.NotchPeakFilter`
- `dsp.SOSFilter`
- `dsp.VariableBandwidthFIRFilter`
- `dsp.VariableBandwidthIIRFilter`

## Output Arguments

**flag** — Flag to determine if filter is in SOS form

1 | 0

Flag to determine whether the filter is in second-order sections (SOS) form, returned as a logical scalar. If the filter is in SOS form, `issos` returns a logical 1, else it returns a logical 0.

Data Types: `logical`

## Version History

Introduced in R2011a

## See Also

`isallpass` | `isfir` | `islinphase` | `ismaxphase` | `isminphase` | `isreal` | `isstable`

# isVisible

## Package:

Determine visibility of scope

## Syntax

```
visibility = isVisible(scope)
```

## Description

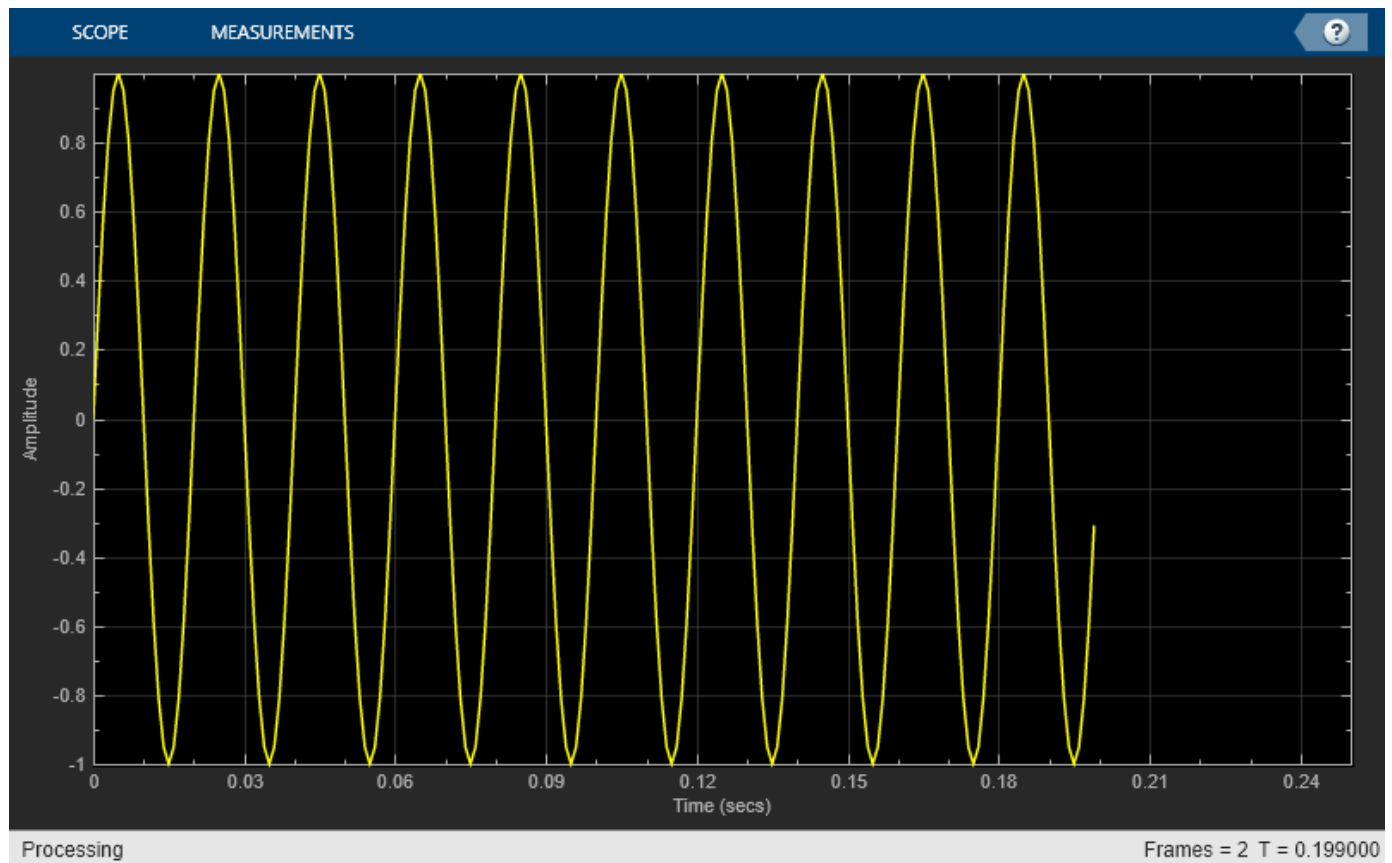
`visibility = isVisible(scope)` returns the visibility of the scope as logical, with 1 (true) for visible.

## Examples

### Hide and Show Time Scope

Create a sine wave signal and view it in the scope.

```
Fs = 1000; % Sample rate
signal = dsp.SineWave(Frequency=50, SampleRate=Fs, ...
    SamplesPerFrame=100);
scope = timescope(SampleRate=Fs, TimeSpanSource="property", ...
    TimeSpan=0.25, YLimits=[-1 1]);
for ii = 1:2
    xsine = signal();
    scope(xsine)
end
```

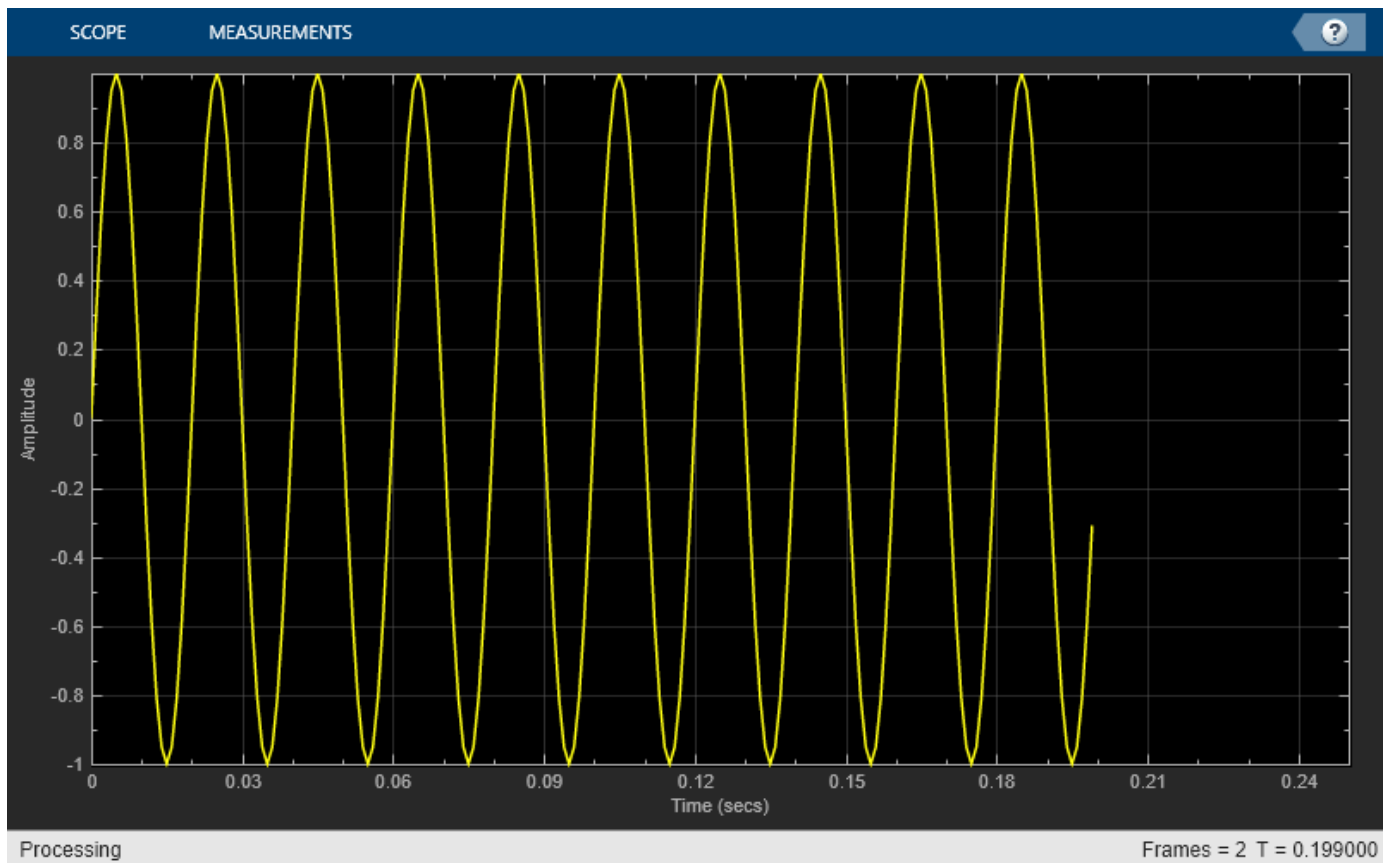


Hide the scope window.

```
if(isVisible(scope))  
    hide(scope)  
end
```

Show the scope window.

```
if(~isVisible(scope))  
    show(scope)  
end
```



Clean up workspace variables.

```
clear scope Fs sine ii xsine
```

## Input Arguments

### scope — Scope object

scope object

Scope object, specified as one of the following:

- spectrumAnalyzer object
- dsp.ArrayPlot object
- dsp.LogicAnalyzer System object
- timescope object
- dsp.DynamicFilterVisualizer object

Example: myScope = spectrumAnalyzer; visibility = isVisible(myScope)

Example: myTS = timescope; visibility = isVisible(myTS)

## Output Arguments

### visibility — Scope visibility

1 | 0

Scope visibility, returned as a logical 1 or a logical 0. If the scope display is showing, the `isVisible` function returns 1 (true). Otherwise, the function returns 0 (false).

## Version History

### Introduced in R2016b

#### R2022a: Support for `dsp.SpectrumAnalyzer` object will be removed

*Not recommended starting in R2022a*

The `isVisible` function will stop supporting the `dsp.SpectrumAnalyzer` object in a future release. Use the `spectrumAnalyzer` object instead.

#### Update Code

No updates to your code are required except replacing instances of `dsp.SpectrumAnalyzer` with `spectrumAnalyzer`.

This table shows how the `isVisible` function typically uses the `dsp.SpectrumAnalyzer` System object and explains how to update existing code to use the `spectrumAnalyzer` object.

| Discouraged Usage  | Recommended Replacement  |
|--|--|
| <pre>scope = dsp.SpectrumAnalyzer; scope(randn(100000,1)) if(isVisible(scope))     hide(scope) end</pre> | <pre>scope = spectrumAnalyzer; scope(randn(100000,1)) if(isVisible(scope))     hide(scope) end</pre> |

#### R2022b: New support for `dsp.DynamicFilterVisualizer` object

Starting in R2022b, the `isVisible` function supports the `dsp.DynamicFilterVisualizer` object.

## See Also

### Functions

`show` | `hide`

### Objects

`timescope` | `dsp.ArrayPlot` | `spectrumAnalyzer` | `dsp.DynamicFilterVisualizer` | `dsp.LogicAnalyzer`



# isstable

**Package:** dsp

Verify that discrete-time filter System object is stable

## Syntax

```
flag = isstable(sysobj)
flag = isstable(sysobj,'Arithmetic',arithType)
```

## Description

`flag = isstable(sysobj)` returns `true` if filter System object is stable. The function returns `false` if filter System object is not stable.

`flag = isstable(sysobj,'Arithmetic',arithType)` analyzes the filter System object based on the arithmetic specified in the `arithType` input.

For more input options, see `isstable` in Signal Processing Toolbox.

## Examples

### Determine if Filter Has Minimum Phase and is Stable

Design a Chebyshev Type I IIR filter and determine if the filter has minimum phase and is stable.

Using the `fdesign.lowpass` and `design` functions, design a Chebyshev Type I IIR filter with a passband ripple of 0.5 dB and a 3 dB cutoff frequency at 9600 Hz.

```
Fs = 48000; % Sampling frequency of input signal
d = fdesign.lowpass('N,F3dB,Ap', 10, 9600, .5, Fs);
filt = design(d,'cheby1','Systemobject',true)

filt =
    dsp.SOSFilter with properties:
        Structure: 'Direct form II'
        CoefficientSource: 'Property'
        Numerator: [5x3 double]
        Denominator: [5x3 double]
        HasScaleValues: true
        ScaleValues: [0.3318 0.2750 0.1876 0.0904 0.0225 0.9441]

    Show all properties
```

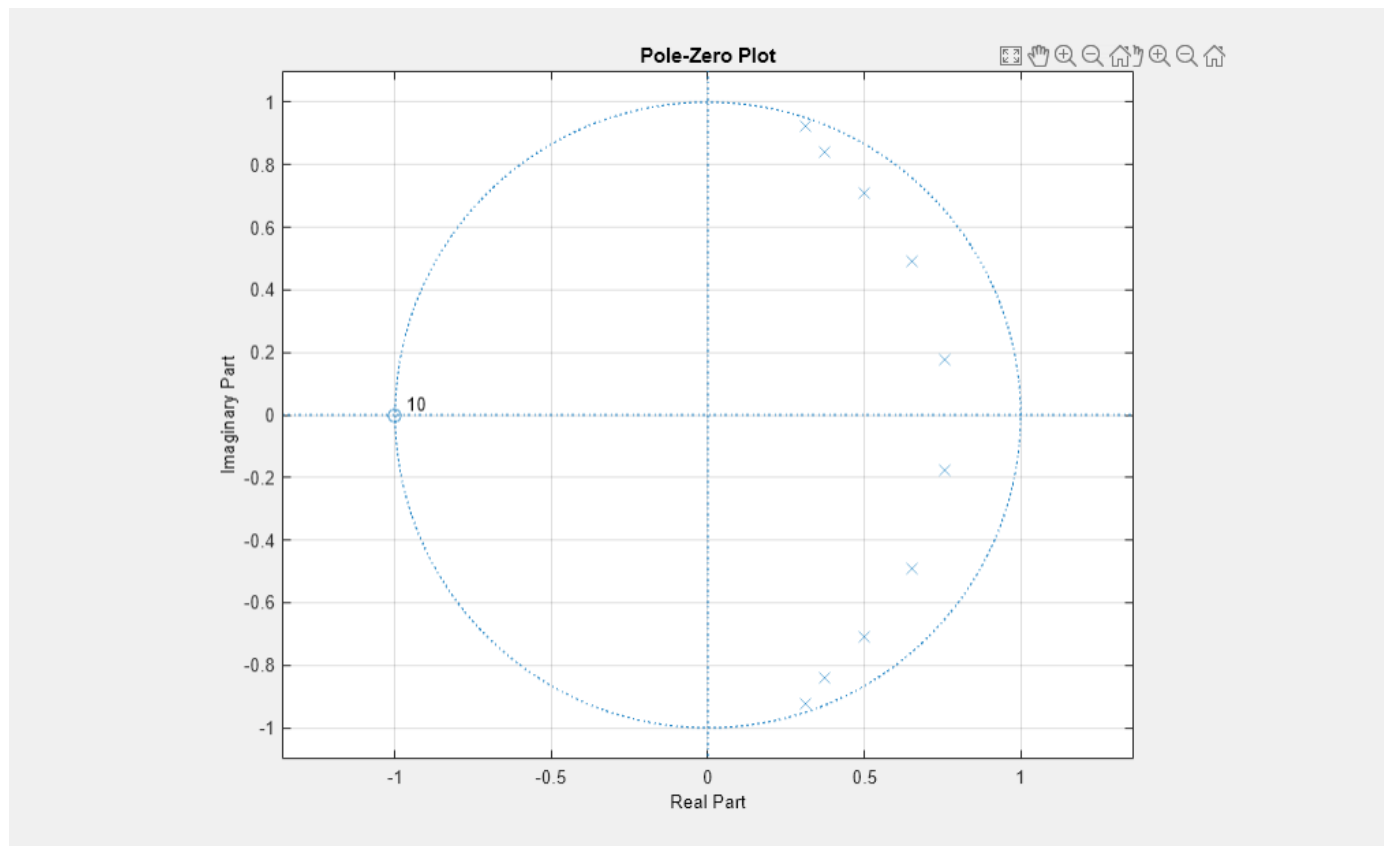
Using the `isminphase` function, determine if the filter has minimum phase.

```
isminphase(filt)
```

```
ans = logical
      1
```

Verify the location of poles and zeros of the filter transfer function on the z-plane. By definition, the poles and zeros of the minimum phase filter must be on or inside the unit circle.

```
zplane(filt)
```



All minimum phase filters are stable. To verify if the designed filter is stable, use the `isstable` function.

```
isstable(filt)
```

```
ans = logical
      1
```

## Input Arguments

### **sysobj** — Filter System object

filter System object

Input filter, specified as one of the following filter System objects:

- `dsp.AllpassFilter`
- `dsp.AllpoleFilter`
- `dsp.BiquadFilter`
- `dsp.CICCompensationDecimator`
- `dsp.CICCompensationInterpolator`
- `dsp.CICDecimator`
- `dsp.CICInterpolator`
- `dsp.CoupledAllpassFilter`
- `dsp.Differentiator`
- `dsp.FarrowRateConverter`
- `dsp.FilterCascade`
- `dsp.FIRDecimator`
- `dsp.FIRFilter`
- `dsp.FIRHalfbandDecimator`
- `dsp.FIRHalfbandInterpolator`
- `dsp.FIRInterpolator`
- `dsp.FIRRateConverter`
- `dsp.FourthOrderSectionFilter`
- `dsp.HighpassFilter`
- `dsp.IIRFilter`
- `dsp.IIRHalfbandDecimator`
- `dsp.IIRHalfbandInterpolator`
- `dsp.LowpassFilter`
- `dsp.NotchPeakFilter`
- `dsp.SOSFilter`
- `dsp.VariableBandwidthFIRFilter`
- `dsp.VariableBandwidthIIRFilter`

**arithType — Arithmetic type**

'double' (default) | 'single' | 'Fixed'

Arithmetic used in the filter analysis, specified as 'double', 'single', or 'Fixed'. When the arithmetic input is not specified and the filter System object is unlocked, the analysis tool assumes a double-precision filter. When the arithmetic input is not specified and the System object is locked, the function performs the analysis based on the data type of the locked input.

The 'Fixed' value applies to filter System objects with fixed-point properties only.

When the 'Arithmetic' input argument is specified as 'Fixed' and the filter object has the data type of the coefficients set to 'Same word length as input', the arithmetic analysis depends on whether the System object is unlocked or locked.

- unlocked -- The analysis object function cannot determine the coefficients data type. The function assumes that the coefficients data type is signed, has a 16-bit word length, and is auto scaled. The function performs fixed-point analysis based on this assumption.

- `locked` -- When the input data type is `'double'` or `'single'`, the analysis object function cannot determine the coefficients data type. The function assumes that the data type of the coefficients is signed, has a 16-bit word length, and is auto scaled. The function performs fixed-point analysis based on this assumption.

To check if the System object is locked or unlocked, use the `isLocked` function.

When the arithmetic input is specified as `'Fixed'` and the filter object has the data type of the coefficients set to a custom numeric type, the object function performs fixed-point analysis based on the custom numeric data type.

## Output Arguments

### `flag` — Flag to determine if filter is stable

1 | 0

Flag to determine if the filter is stable, returned as a logical:

- 1 -- Filter is stable.
- 0 -- Filter is not stable.

Data Types: `logical`

## Version History

**Introduced in R2013a**

### See Also

`isstable`

# kaiserwin

Kaiser window filter from specification object

## Syntax

```
kFilter = design(d,'kaiserwin','SystemObject',true)
kFilter = design(d,'kaiserwin',designoption,value,designoption,...
value,'SystemObject',true)
```

## Description

`kFilter = design(d,'kaiserwin','SystemObject',true)` designs a digital filter `kFilter` that uses a Kaiser window. For `kaiserwin` to work properly, the filter order in the specifications object must be even. In addition, higher order filters (filter order greater than 120) tend to be more accurate for smaller transition widths. `kaiserwin` returns a warning when your filter order may be too low to design your filter accurately.

`kFilter = design(d,'kaiserwin',designoption,value,designoption,... value,'SystemObject',true)` returns a filter where you specify design options as input arguments and the design process uses the Kaiser window technique.

To determine the available design options, use `designopts` with the specification object and the design method as input arguments as shown.

```
designopts(d,'method')
```

For complete help about using `kaiserwin`, refer to the command line help system. For example, to get specific information about using `kaiserwin` with `d`, the specification object, enter the following at the MATLAB prompt.

```
help(d,'kaiserwin')
```

## Examples

### Design a Direct Form FIR Filter

This example designs a direct form FIR filter from a halfband filter specification object.

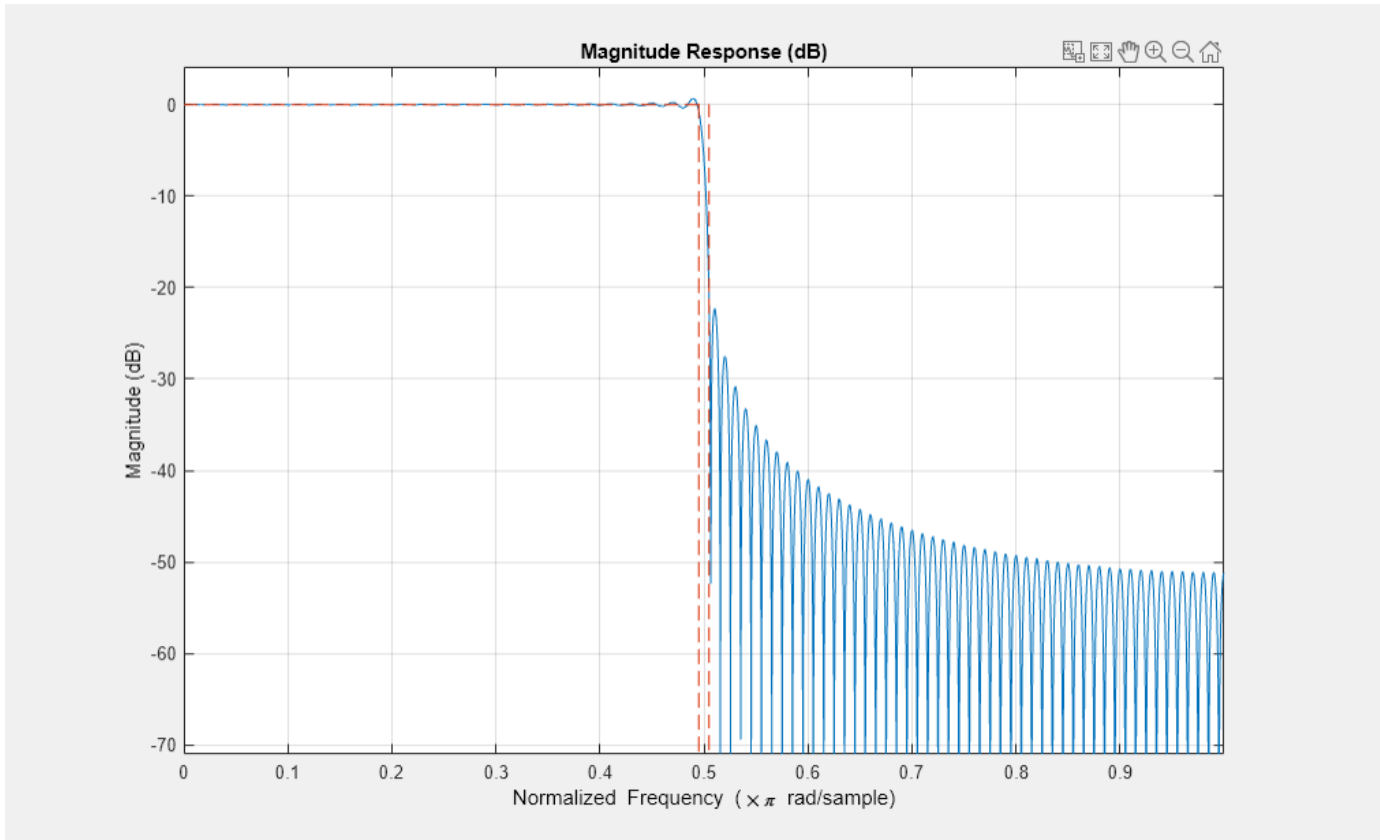
```
d = fdesign.halfband('n,tw',200,0.01);
hbFilter = design(d,'kaiserwin','filterstructure','dffir',...
'SystemObject',true)

hbFilter =
  dsp.FIRFilter with properties:

    Structure: 'Direct form'
  NumeratorSource: 'Property'
    Numerator: [0 -0.0028 0 0.0029 0 -0.0030 0 0.0030 0 -0.0031 0 0.0032 0 -0.0033 0 0.0034 0 -0.0035 0 0.0036 0 -0.0037 0 0.0038 0 -0.0039 0 0.0040 0 -0.0041 0 0.0042 0 -0.0043 0 0.0044 0 -0.0045 0 0.0046 0 -0.0047 0 0.0048 0 -0.0049 0 0.0050 0 -0.0051 0 0.0052 0 -0.0053 0 0.0054 0 -0.0055 0 0.0056 0 -0.0057 0 0.0058 0 -0.0059 0 0.0060 0 -0.0061 0 0.0062 0 -0.0063 0 0.0064 0 -0.0065 0 0.0066 0 -0.0067 0 0.0068 0 -0.0069 0 0.0070 0 -0.0071 0 0.0072 0 -0.0073 0 0.0074 0 -0.0075 0 0.0076 0 -0.0077 0 0.0078 0 -0.0079 0 0.0080 0 -0.0081 0 0.0082 0 -0.0083 0 0.0084 0 -0.0085 0 0.0086 0 -0.0087 0 0.0088 0 -0.0089 0 0.0090 0 -0.0091 0 0.0092 0 -0.0093 0 0.0094 0 -0.0095 0 0.0096 0 -0.0097 0 0.0098 0 -0.0099 0 0.0100 0]
  InitialConditions: 0
```

Show all properties

```
fvtool(hbFilter);
```



In this example, `kaiserwin` uses an interpolating filter specification object.

```
d = fdesign.interpolator(4,'lowpass');
interpFilter= design(d,'kaiserwin','SystemObject',true)
```

```
interpFilter =
  dsp.FIRInterpolator with properties:
```

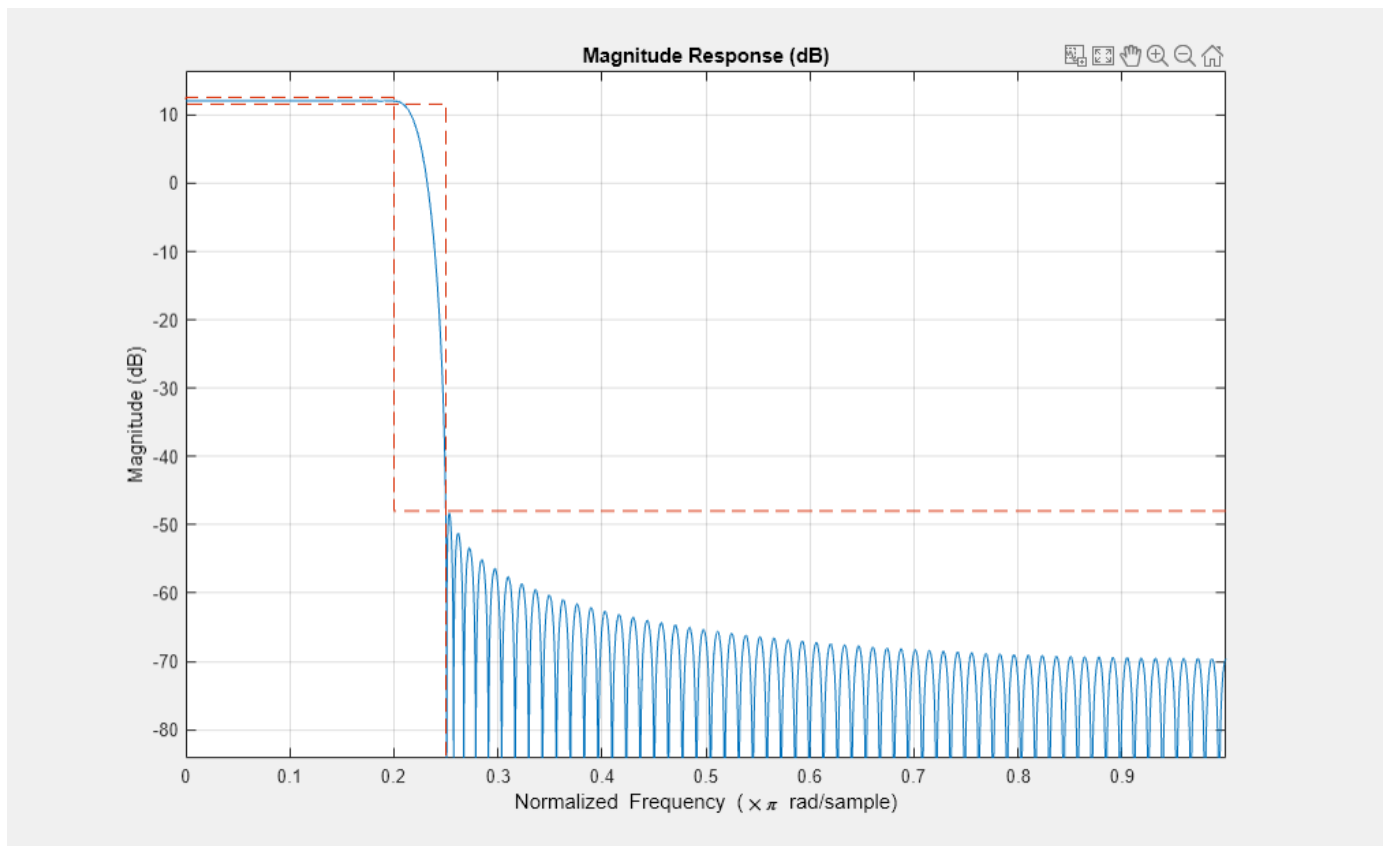
```
    InterpolationFactor: 4
```

```
    NumeratorSource: 'Property'
```

```
    Numerator: [3.4579e-04 2.6057e-04 -4.2478e-05 -4.6005e-04 -7.6928e-04 -7.3242e-04
```

Show all properties

```
fvtool(interpFilter);
```



## Version History

Introduced in R2011a

### See Also

`equiripple` | `firls`

# lagrange

Fractional delay filter from `fdesign.fracdelay` specification object

## Syntax

```
Hd = design(d,'lagrange')
hd = design(d,'lagrange',FilterStructure,structure)
```

## Description

`Hd = design(d,'lagrange')` designs a fractional delay filter using the Lagrange method based on the specifications in `d`.

`hd = design(d,'lagrange',FilterStructure,structure)` specifies the Lagrange design method and the `structure` filter structure for `hd`. The only valid filter structure is `fd`, describing the fractional delay structure.

## Examples

This example uses a fractional delay of 0.30 samples. The `help` and `designopts` commands provide the details about designing fractional delay filters.

```
d=fdesign.fracdelay(.30)

d =

    Response: 'Fractional Delay'
  Specification: 'N'
    Description: {'Filter Order'}
      FracDelay: 0.3
  NormalizedFrequency: true
      FilterOrder: 3

designmethods(d)

Design Methods for class fdesign.fracdelay (N):

lagrange

help(d,'lagrange')

DESIGN Design a Lagrange fractional delay filter.
HD = DESIGN(D, 'lagrange') designs a Lagrange filter specified by the
  FDESIGN object D, and returns the DFILT object HD.

HD = DESIGN(..., 'FilterStructure', STRUCTURE) returns a filter with the
  structure STRUCTURE. STRUCTURE is 'farrowfd' by default and can be any of
  the following:

'farrowfd'
'fd'

% Example #1 - Design a linear Lagrange fractional delay filter of 0.2 samples.
h = fdesign.fracdelay(0.2,'N',2);
Hd = design(h, 'lagrange', 'FilterStructure', 'farrowfd')

% Example #2 - Design a cubic Lagrange fractional delay filter
Fs = 8000; % Sampling frequency of 8kHz
fdelay = 50e-6; % Fractional delay of 50 microseconds.
h = fdesign.fracdelay(fdelay,'N',3,Fs);
Hd = design(h, 'lagrange', 'FilterStructure', 'farrowfd');
```

This example designs a linear Lagrange fractional delay filter where you set the delay to 0.2 seconds and the filter order `N` to 2.



```
h = fdesign.fracdelay(0.2,'N',2);  
hd = design(h,'lagrange','FilterStructure','farrowfd')
```

Design a cubic Lagrange fractional delay filter with filter order equal to 3.

```
Fs = 8000;          % Sampling frequency of 8 kHz.  
fdelay = 50e-6;    % Fractional delay of 50 microseconds.  
h = fdesign.fracdelay(fdelay,'N',3,Fs);  
hd = design(h,'lagrange','FilterStructure','farrowfd');
```

## Version History

Introduced in R2011a

## References

Laakso, T. I., V. Välimäki, M. Karjalainen, and Unto K. Laine, "Splitting the Unit Delay - Tools for Fractional Delay Filter Design," *IEEE Signal Processing Magazine*, Vol. 13, No. 1, pp. 30-60, January 1996.

## See Also

[design](#) | [designmethods](#) | [designopts](#) | [fdesign](#) | [fdesign.fracdelay](#)

## liblinks

Check model for blocks from specific DSP System Toolbox libraries

### Syntax

```
liblinks(lib)  
liblinks(lib,sys)  
liblinks(lib,sys,c)
```

### Description

`liblinks(lib)` returns a cell array of character vectors that lists the blocks in the current model that are linked to the specified libraries. The input `lib` provides a cell array of character vectors with the library names. Use the library name visible in the title bar when you open a library model.

`liblinks(lib,sys)` acts on the named model `sys`.

`liblinks(lib,sys,c)` changes the foreground color of the returned blocks to the color `c`. Possible values of `c` are 'blue', 'green', 'red', 'cyan', 'magenta', 'yellow', or 'black'.

### Examples

Check for blocks from the Sources library in the specified model:

```
rlsdemo  
liblinks('dspsrcs4',gcs)
```

## Version History

**Introduced before R2006a**

### See Also

`dsp_links`

# limitcycle

Response of single-rate, fixed-point IIR filter

## Syntax

```
report = limitcycle(hd)
report = limitcycle(hd,ntrials,inputlengthfactor,stopcriterion)
```

## Description

`report = limitcycle(hd)` returns the structure `report` that contains information about how filter `hd` responds to a zero-valued input vector. By default, the input vector has length equal to twice the impulse response length of the filter.

`limitcycle` returns a structure whose elements contain the details about the limit cycle testing. As shown in this table, the `report` includes the following details.

| Output Object Property      | Description   |
|-----------------------------|---|
| <code>LimitCycleType</code> | Contains one of the following results: <ul style="list-style-type: none"> <li><code>Granular</code> — indicates that a granular overflow occurred.</li> <li><code>Overflow</code> — indicates that an overflow limit cycle occurred.</li> <li><code>None</code> — indicates that the test did not find any limit cycles.</li> </ul> |
| <code>Zi</code>             | Contains the initial condition value(s) that caused the detected limit cycle to occur.  |
| <code>Output</code>         | Contains the output of the filter in the steady state.  |
| <code>Trial</code>          | Returns the number of the Monte Carlo trial on which the limit cycle testing stopped. For example, <code>Trial = 10</code> indicates that testing stopped on the tenth Monte Carlo trial.   |

Using an input vector longer than the filter impulse response ensures that the filter is in steady-state operation during the limit cycle testing. `limitcycle` ignores output that occurs before the filter reaches the steady state. For example, if the filter impulse length is 500 samples, `limitcycle` ignores the filter output from the first 500 input samples.

To perform limit cycle testing on your IIR filter, you must set the filter `Arithmetic` property to `fixed` and `hd` must be a fixed-point IIR filter of one of the following forms:

- `df1` — direct-form I
- `df1t` — direct-form I transposed
- `df1sos` — direct-form I with second-order sections
- `df1tsos` — direct-form I transposed with second-order sections
- `df2` — direct-form II
- `df2t` — direct-form II transposed
- `df2sos` — direct-form II with second-order sections

- `df2tsos` — direct-form II transposed with second-order sections

When you use `limitcycle` without optional input arguments, the default settings are

- Run 20 Monte Carlo trials
- Use an input vector twice the length of the filter impulse response
- Stop testing if the simulation process encounters either a granular or overflow limit cycle

To determine the length of the filter impulse response, use `impzlength`:

```
impzlength(hd)
```

During limit cycle testing, if the simulation runs reveal both overflow and granular limit cycles, the overflow limit cycle takes precedence and is the limit cycle that appears in the report.

Each time you run `limitcycle`, it uses a different sequence of random initial conditions, so the results can differ from run to run.

Each Monte Carlo trial uses a new set of randomly determined initial states for the filter. Test processing stops when `limitcycle` detects a zero-input limit cycle in filter `hd`.

`report = limitcycle(hd,ntrials,inputlengthfactor,stopcriterion)` returns a report of how filter `hd` responds to a zero-valued input vector, using the following optional input arguments:

- `ntrials` — Number of Monte Carlo trials (default is 20).
- `inputlengthfactor` — integer factor used to calculate the length of the input vector. The length of the input vector comes from  $(\text{impzlength}(\text{hd}) * \text{inputlengthfactor})$ , where `inputlengthfactor = 2` is the default value.
- `stopcriterion` — the criterion for stopping the Monte Carlo trial processing. `stopcriterion` can be set to **either** (the default), **granular**, **overflow**. This table describes the results of each stop criterion.

| stopcriterion Setting | Description   |
|-----------------------|---|
| <b>either</b>         | Stop the Monte Carlo trials when <code>limitcycle</code> detects either a granular or overflow limit cycle. |
| <b>granular</b>       | Stop the Monte Carlo trials when <code>limitcycle</code> detects a granular limit cycle.                    |
| <b>overflow</b>       | Stop the Monte Carlo trials when <code>limitcycle</code> detects an overflow limit cycle.                   |

**Note** An important feature is that if you specify a specific limit cycle stop criterion, such as **overflow**, the Monte Carlo trials do not stop when testing encounters a granular limit cycle. You receive a warning that no **overflow** limit cycle occurred, but consider that a **granular** limit cycle might have occurred.

## Examples

In this example, there is a region of initial conditions in which no limit cycles occur and a region where they do. If no limit cycles are detected before the Monte Carlo trials are over, the state sequence converges to zero. When a limit cycle is found, the states do not end at zero. Each time you

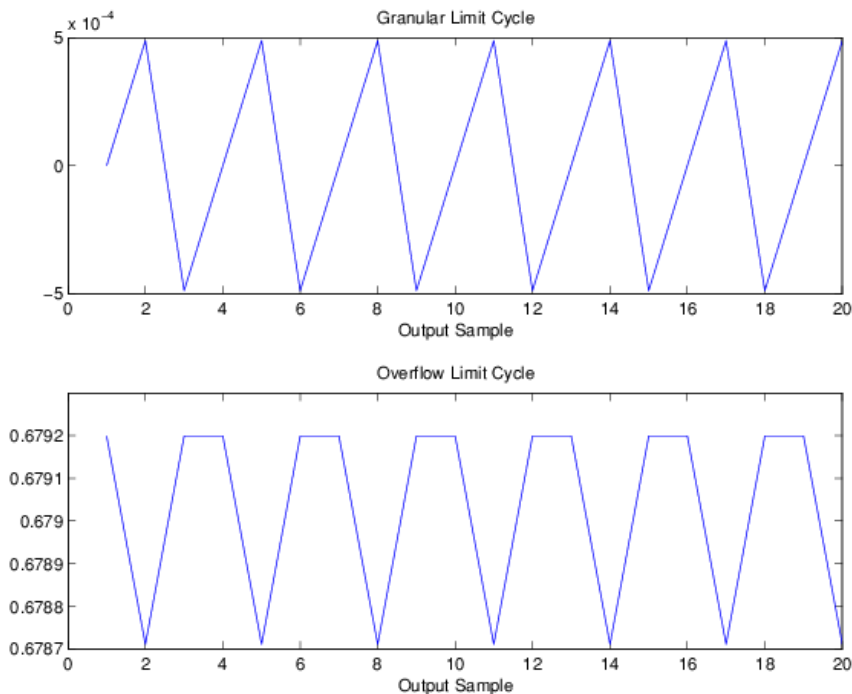
run this example, it uses a different sequence of random initial conditions, so the plot you get can differ from the one displayed in the following figure.

```
s = [1 0 0 1 0.9606 0.9849];
hd = dfilt.df2sos(s);
hd.arithmetic = 'fixed';
greport = limitcycle(hd,20,2,'granular')
oreport = limitcycle(hd,20,2,'overflow')
figure,
subplot(211),plot(greport.Output(1:20)), title('Granular Limit Cycle');
subplot(212),plot(oreport.Output(1:20)), title('Overflow Limit Cycle');

greport =
    LimitCycle: 'granular'
              Zi: [2x1 double]
              Output: [1303x1 embedded.fi]
              Trial: 1

oreport =
    LimitCycle: 'overflow'
              Zi: [2x1 double]
              Output: [1303x1 embedded.fi]
              Trial: 2
```

The plots shown in this figure present both limit cycle types — the first displays the small amplitude granular limit cycle, the second the larger amplitude overflow limit cycle.



As you see from the plots, and as is generally true, overflow limit cycles are much greater magnitude than granular limit cycles. This is why `limitcycle` favors overflow limit cycle detection and reporting.

## Version History

### Introduced in R2011a

**See Also**

freqz | noise PSD

# maxflat

Maxflat FIR filter

## Syntax

```
maxflatFilter = design(d,'maxflat','SystemObject',true)
maxflatFilter =
design(d,'maxflat','FilterStructure',structure,'SystemObject',true)
```

## Description

`maxflatFilter = design(d,'maxflat','SystemObject',true)` designs a maximally flat filter, `maxflatFilter`, from a filter specification object, `d`.

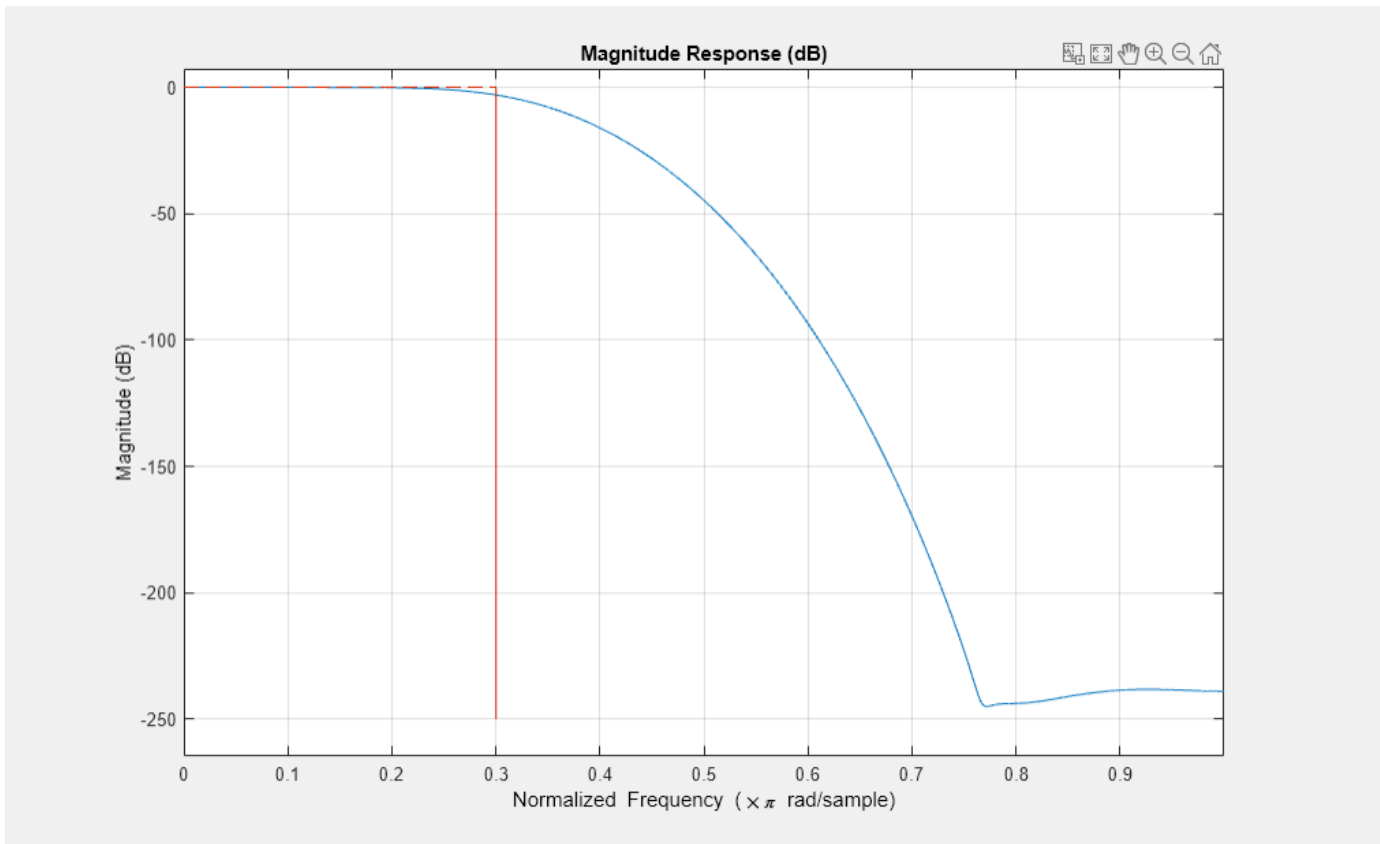
`maxflatFilter = design(d,'maxflat','FilterStructure',structure,'SystemObject',true)` designs a maximally flat filter where `structure` is one of the following:

- 'dffir', a discrete-time, direct-form FIR filter (the default value)
- 'dffirt', a discrete-time, direct-form FIR transposed filter
- 'dfsymfir', a discrete-time, direct-form symmetric FIR filter

## Examples

### Design a Lowpass Filter with a Maximally Flat FIR Structure

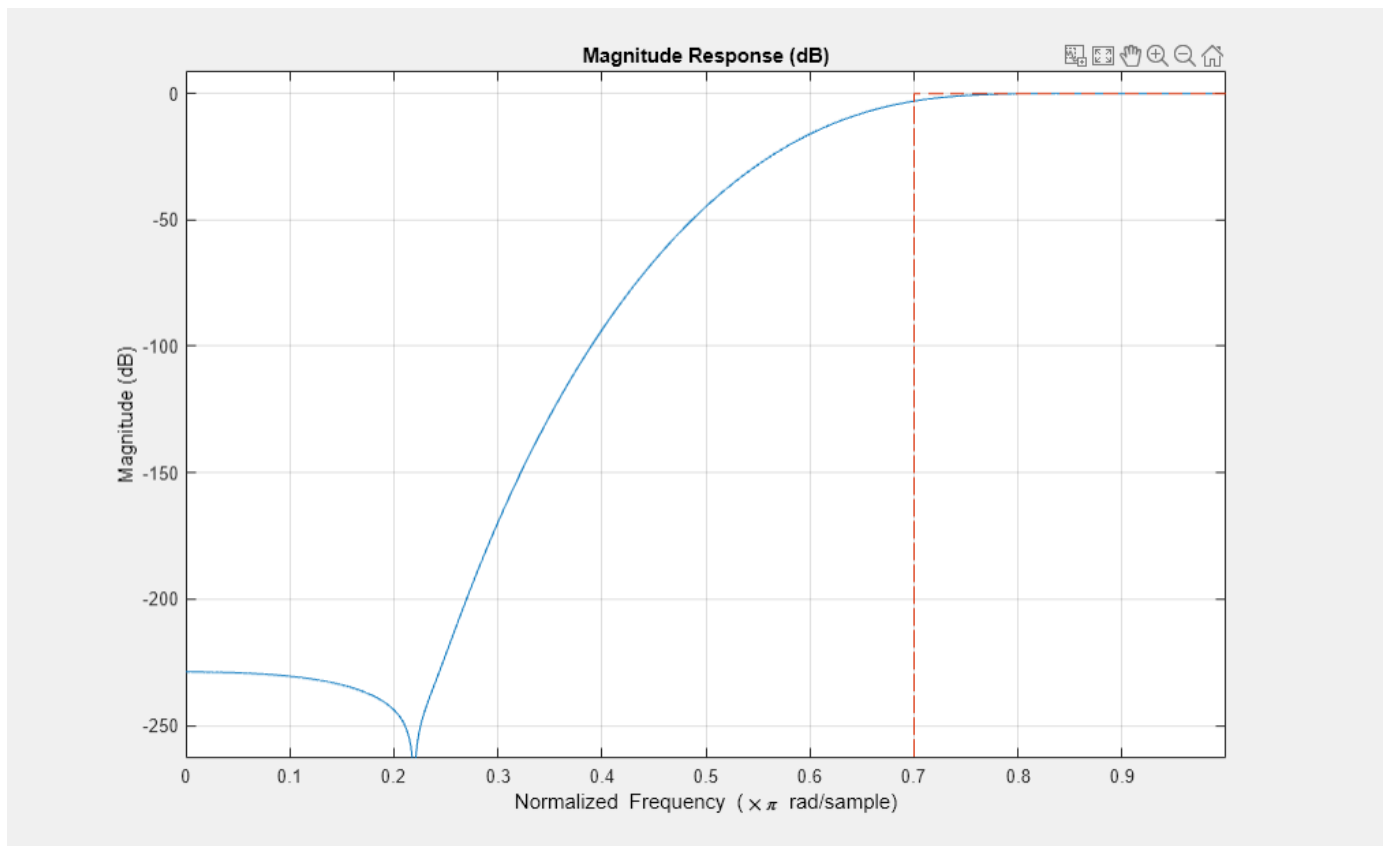
```
d = fdesign.lowpass('N,F3dB', 50, 0.3);
flatLowpass = design(d, 'maxflat','SystemObject',true);
fvtool(flatLowpass);
```



### Design a Highpass Filter With a Maximally Flat Symmetric FIR Structure

```
d = fdesign.highpass('N,F3dB', 50, 0.7);  
flatHighpass = design(d,'maxflat','FilterStructure','dfsymfir',...  
    'SystemObject',true);  
fvtool(flatHighpass)
```





## Version History

Introduced in R2011a

## maxstep

**Package:** dsp

Maximum step size for LMS adaptive filter convergence

### Syntax

```
mumax = maxstep(lmsFilt,x)
[mumax,mumaxmse] = maxstep(lmsFilt,x)
```

### Description

`mumax = maxstep(lmsFilt,x)` predicts a bound on the step size to provide convergence of the mean values of the coefficients of the adaptive filter, `lmsFilt`.

`[mumax,mumaxmse] = maxstep(lmsFilt,x)` predicts a bound, in mean squared sense, on the adaptive filter step size to provide convergence of the adaptive filter coefficients.

### Examples

#### Compute Maximum Step of LMS Adaptive Filter

The `maxstep` function computes the maximum step size of the adaptive filter. This step size keeps the filter stable at the maximum possible speed of convergence. Create the primary input signal, `x`, by passing a signed random signal to an IIR filter. Signal `x` contains 50 frames of 2000 samples each frame. Create an LMS filter with 32 taps and a step size of 0.1.

```
x = zeros(2000,50);
IIRFilter = dsp.IIRFilter('Numerator',sqrt(0.75),...
    'Denominator',[1 -0.5]);
for k = 1:size(x,2)
    x(:,k) = IIRFilter(sign(randn(size(x,1),1)));
end
mu = 0.1;
LMSFilter = dsp.LMSFilter('Length',32,...
    'StepSize',mu);
```

Compute the maximum adaptation step size and the maximum step size in mean-squared sense using the `maxstep` function.

```
[mumax,mumaxmse] = maxstep(LMSFilter,x)
```

```
mumax = 0.0625
```

```
mumaxmse = 0.0536
```

## Input Arguments

### **lmsfilt** — LMS adaptive filter System object

`dsp.LMSFilter` | `dsp.BlockLMSFilter`

LMS adaptive filter, specified as either a `dsp.LMSFilter` System object or a `dsp.BlockLMSFilter` System object.

### **x** — Input signal

scalar | column vector | matrix

Columns of the matrix `x` contain individual input signal sequences. The signal set is assumed to have zero mean or close to zero mean.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `logical`  
Complex Number Support: Yes

## Output Arguments

### **mumax** — Maximum step size

scalar

Maximum step size value, returned as a scalar. This is the step size you can specify for the adaptive filter without causing the filter to become unstable. For details on how this parameter is calculated, see “Algorithms” on page 5-1147.

Data Types: `double`

### **mumaxmse** — Maximum step size in mean squared sense

scalar

Maximum adaptive filter step size to provide convergence of the LMS adaptive filter coefficients in the mean squared sense, returned as a scalar. For details on how this parameter is calculated, see “Algorithms” on page 5-1147.

Data Types: `double`

## Algorithms

The step size of the adaptive filter must satisfy the following equation in order for the adaptive filter to be stable:

$$0 < \mu < \mu_{\max}$$

where,  $\mu_{\max}$  is the maximum step size.

The value of  $\mu_{\max}$  depends on the LMS filter System object and the adaptive filter algorithm the object uses.

### **dsp.LMSFilter**

#### **LMS**

When the `Method` property of the `dsp.LMSFilter` object is set to `'LMS'`, maximum step size  $\mu_{\max}$  is calculated using the following equation:

$$\mu_{\max} = \frac{2}{\text{mean}(xt \circ xt)L}$$

where,

- $xt$  -- Concatenated columns of the input matrix,  $x(:)$ .
- $xt \circ xt$  -- Hadamard or entrywise product of the two vectors.
- $L$  -- Length of the filter coefficients.

The maximum step size in mean-square sense,  $\mu_{\max\text{MSE}}$  is computed using the following equation:

$$\mu_{\max\text{MSE}} = \frac{2}{\lambda_{\max}(\text{Kurt} + 2) + \text{sum}(\lambda)}$$

where,

- $\text{sum}(\lambda)$  -- Sum of the eigenvalues of the input auto correlation matrix.
- $\lambda_{\max}$  -- Maximum eigenvalue of the input auto correlation matrix.
- $\text{Kurt}$  -- Average kurtosis value of eigenvector-filtered signals.

### Normalized LMS

When the Method property of the `dsp.LMSFilter` System object is set to 'Normalized LMS':

- Maximum step size,  $\mu_{\max} = 2$ .
- Maximum step size in mean-square sense,  $\mu_{\max\text{MSE}} = 2$ .

### For Other Methods

For all other methods such as Sign-Data LMS, Sign-Error LMS, and Sign-Sign LMS:

- $\mu_{\max} = \infty$ .
- $\mu_{\max\text{MSE}} = \infty$ .

### `dsp.BlockLMSFilter`

The maximum step size for `dsp.BlockLMSFilter` is computed using the following equation:

$$\mu_{\max} = \frac{2}{\text{mean}(xt \circ xt)L}$$

where,

- $xt$  -- Concatenated columns of the input matrix,  $x(:)$ .
- $xt \circ xt$  -- Hadamard or entrywise product of the two vectors.
- $L$  -- Length of the filter coefficients.

The maximum step size in mean-square sense,  $\mu_{\max\text{MSE}}$  is computed using the following equation:

$$\mu_{\max\text{MSE}} = \frac{\mu_{\max}}{3}$$

## Version History

Introduced in R2012a

## References

[1] Hayes, M.H. *Statistical Digital Signal Processing and Modeling*. New York: John Wiley & Sons, 1996.

## See Also

### Functions

msepred | msesim

### Objects

dsp.LMSFilter | dsp.BlockLMSFilter

### Topics

“Overview of Adaptive Filters and Applications”

## maximizestopband

Maximize stopband attenuation of fixed-point filter

### Syntax

```
Hq = maximizestopband(Hd,Wordlength)
Hq = maximizestopband(Hd,Wordlength,'Ntrials',N)
```

### Description

`Hq = maximizestopband(Hd,Wordlength)` quantizes the single-stage or multistage FIR filter `Hd` and returns the fixed-point filter `Hq` with wordlength `wordlength` that maximizes the stopband attenuation. `Hd` must be generated using `fdesign` and `design`. For multistage filters, `wordlength` can either be a scalar or vector. If `wordlength` is a scalar, the same wordlength is used for all stages. If `wordlength` is a vector, each stage uses the corresponding element in the vector. The vector length must equal the number of stages. `maximizestopband` uses a stochastic noise-shaping procedure by default to minimize the wordlength. To obtain repeatable results on successive function calls, initialize the uniform random number generator `rand`

`Hq = maximizestopband(Hd,Wordlength,'Ntrials',N)` specifies the number of Monte Carlo trials to use in the maximization. `Hq` is the fixed-point filter with the largest stopband attenuation among the trials. The number of Monte Carlo trials `N` defaults to 1.

You must have the Fixed-Point Designer software installed to use this function.

### Examples

#### Maximize Stopband Attenuation

Maximize stopband attenuation for 16-bit fixed-point filter.

```
Hf = fdesign.lowpass('Fp,Fst,Ap,Ast',0.4,0.45,0.5,60);
Hd = design(Hf,'equiripple');
```

Use 16 bits to represent coefficients.

```
WL = 16;
Hq = maximizestopband(Hd,WL);
```

Compare stopband attenuation

```
md = measure(Hd)

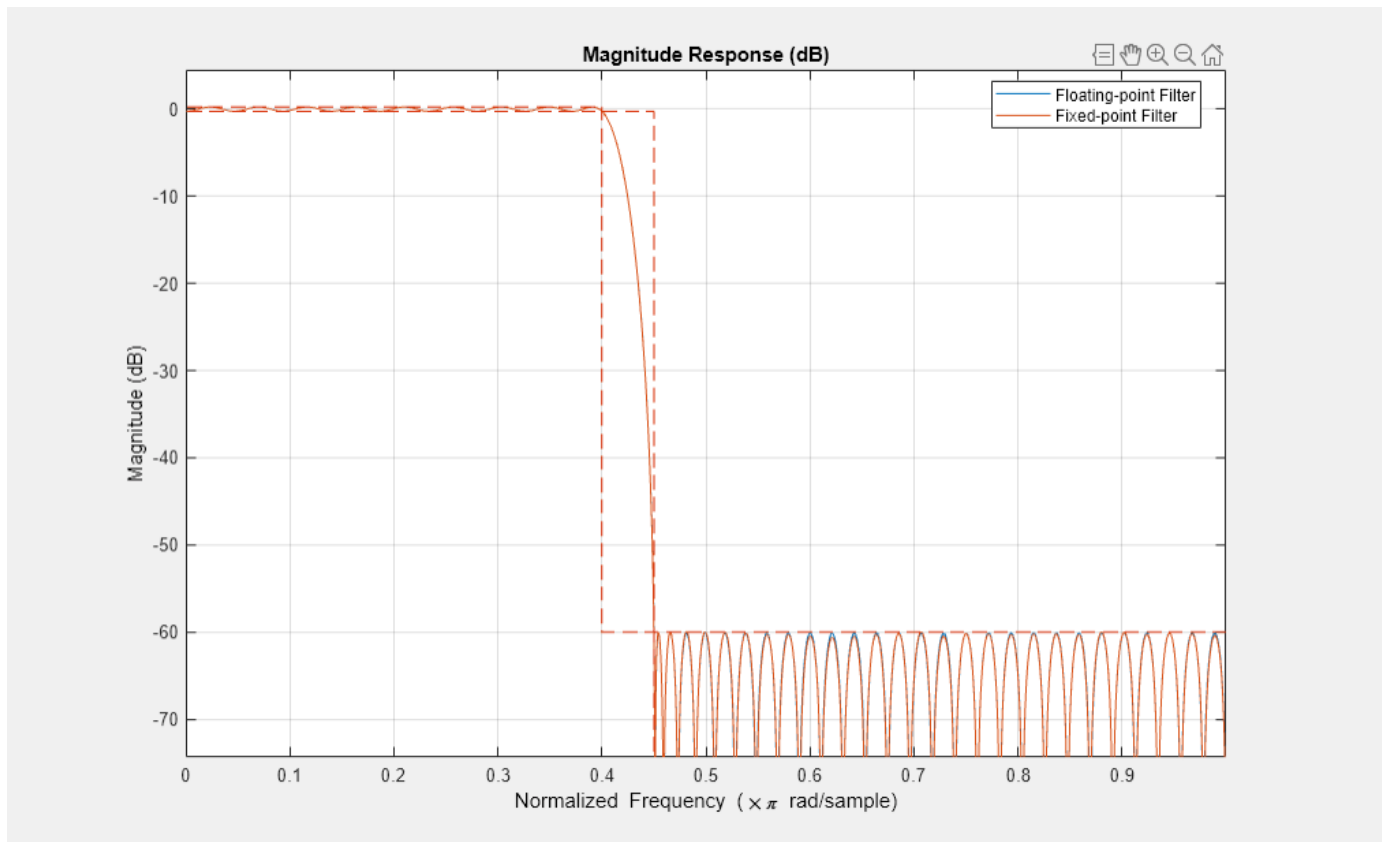
md =
Sample Rate      : N/A (normalized frequency)
Passband Edge    : 0.4
3-dB Point       : 0.41178
6-dB Point       : 0.41845
Stopband Edge    : 0.45
Passband Ripple  : 0.49369 dB
```

```
Stopband Atten. : 60.0697 dB
Transition Width : 0.05
```

```
mq = measure(Hq)
```

```
mq =
Sample Rate      : N/A (normalized frequency)
Passband Edge    : 0.4
3-dB Point       : 0.41178
6-dB Point       : 0.41845
Stopband Edge    : 0.45
Passband Ripple  : 0.49773 dB
Stopband Atten.  : 59.9728 dB
Transition Width  : 0.05
```

```
hfvf=fvtool(Hd,Hq,'ShowReference','off');
legend(hfvf,'Floating-point Filter','Fixed-point Filter');
```



## Version History

Introduced in R2011a

## See Also

[constraincoeffwl](#) | [design](#) | [fdesign](#) | [minimizecoeffwl](#) | [measure](#) | [rand](#)

**Topics**

“Fixed-Point Overview”



# measure

**Package:** dsp

Measure frequency response characteristics of filter System object

## Syntax

```
measure(sysobj)
M = measure(sysobj)
M = measure(sysobj, 'Arithmetic', arithType)
M = measure(sysobj, 'freqspec', freqspecvalue)
```

## Description

`measure(sysobj)` displays measurements of various quantities from the frequency response of the filter System object, `sysobj`. Measurements include the actual passband ripple, the minimum stopband attenuation, the frequency point at which the filter's gain is 3 dB below the nominal passband gain, etc. You must construct `sysobj` using `fdesign` and `design` with the name-value pair argument `'SystemObject', true`. You can optionally specify additional options by one or more `Name, Value` pair arguments.

`M = measure(sysobj)` returns the measurements, `M`, such that the measurements can be queried programmatically. For example, to query the 3 dB point, type `M.F3dB`. Type `get(M)` to see the full list of properties that can be queried. Note that different filter responses generate different measurements.

`M = measure(sysobj, 'Arithmetic', arithType)` analyzes the filter System object, `sysobj`, based on the arithmetic specified in the `arithType` input. `arithType` can be set to one of `'double'`, `'single'`, or `'fixed'`. When the arithmetic input is not specified and the filter System object is in an unlocked state, the analysis tool assumes a double precision filter.

`M = measure(sysobj, 'freqspec', freqspecvalue)` passes the frequency value as an input to `measure` in order to determine the corresponding magnitude measurements. For designs that do not specify some of the frequency constraints, you can determine the corresponding magnitude measurements using this option.

In the following example, the passband edge, passband ripple, and the transition width of the IIR filter are unknown.

```
designLowpass = fdesign.lowpass('N,F3dB,Ast',8,0.5,80);
chebFilter = design(designLowpass, 'cheby2');
measure(chebFilter)
```

```
Sample Rate      : N/A (normalized frequency)
Passband Edge    : Unknown
3-dB Point       : 0.5
6-dB Point       : 0.51823
Stopband Edge    : 0.68727
Passband Ripple  : Unknown
Stopband Atten.  : 79.9994 dB
Transition Width  : Unknown
```

Specify the passband edge to be 0.4, and measure the passband ripple and the transition width of this filter.

```
measure(chebFilter, 'Fpass', 0.4)

Sample Rate      : N/A (normalized frequency)
Passband Edge    : 0.4
3-dB Point       : 0.5
6-dB Point       : 0.51823
Stopband Edge    : 0.68727
Passband Ripple  : 0.013644 dB
Stopband Atten. : 79.9994 dB
Transition Width : 0.28727
```

## Examples

### Measure the Filter Specifications

Create a lowpass filter and check whether the actual filter meets the specifications. For this case, use normalized frequency for  $F_s$ , the default setting.

```
desigLowpass = fdesign.lowpass('Fp,Fst,Ap,Ast',0.45,0.55,0.1,80)
```

```
desigLowpass =
  lowpass with properties:

      Response: 'Lowpass'
  Specification: 'Fp,Fst,Ap,Ast'
  Description: {4x1 cell}
  NormalizedFrequency: 1
      Fpass: 0.4500
      Fstop: 0.5500
      Apass: 0.1000
      Astop: 80
```

```
designmethods(desigLowpass, 'SystemObject', true)
```

Design Methods that support System objects for class `fdesign.lowpass (Fp,Fst,Ap,Ast)`:

```
butter
cheby1
cheby2
ellip
equiripple
ifir
kaiserwin
multistage
```

Use the default equiripple design method.

```
equiFilter = design(desigLowpass, 'SystemObject', true)
```

```
equiFilter =
  dsp.FIRFilter with properties:
```

```

        Structure: 'Direct form'
    NumeratorSource: 'Property'
        Numerator: [-1.0186e-05 -6.8148e-04 -0.0017 -0.0014 7.3963e-04 0.0016 -7.5529e-04 -0
InitialConditions: 0

```

Show all properties

Measure the specifications of the designed lowpass filter.

```
measure(equiFilter)
```

```

ans =
Sample Rate      : N/A (normalized frequency)
Passband Edge    : 0.45
3-dB Point       : 0.47798
6-dB Point       : 0.48913
Stopband Edge    : 0.55
Passband Ripple  : 0.095021 dB
Stopband Atten.  : 80.1164 dB
Transition Width  : 0.1

```

Stopband Edge, Passband Edge, Passband Ripple, and Stopband Attenuation all meet the specifications.

Now, using  $F_s$  in linear frequency, create a bandpass filter, and measure the magnitude response characteristics.

```
designBandpass = fdesign.bandpass
```

```
designBandpass =
bandpass with properties:

```

```

        Response: 'Bandpass'
    Specification: 'Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2'
    Description: {7x1 cell}
NormalizedFrequency: 1
        Fstop1: 0.3500
        Fpass1: 0.4500
        Fpass2: 0.5500
        Fstop2: 0.6500
        Astop1: 60
        Apass: 1
        Astop2: 60

```

Convert to Linear Frequency.

```
normalizefreq(designBandpass,false,1.5e3)
```

```
bpFilter = design(designBandpass,'cheby2','SystemObject',true);
```

Measure the specifications of the designed bandpass filter.

```
measure(bpFilter)
```

```

ans =
Sample Rate      : 1.5 kHz

```

```
First Stopband Edge      : 262.5 Hz
First 6-dB Point        : 319.9585 Hz
First 3-dB Point        : 324.9744 Hz
First Passband Edge     : 337.5 Hz
Second Passband Edge    : 412.5 Hz
Second 3-dB Point       : 425.0256 Hz
Second 6-dB Point       : 430.0415 Hz
Second Stopband Edge    : 487.5 Hz
First Stopband Atten.   : 60 dB
Passband Ripple         : 0.17985 dB
Second Stopband Atten. : 60 dB
First Transition Width  : 75 Hz
Second Transition Width : 75 Hz
```

### Measure Frequency Response Characteristics of Highpass Filter

Measure the frequency response characteristics of a highpass filter. Create a `dsp.HighpassFilter` System object with default properties. Measure the frequency response characteristics of the filter.

```
HPF = dsp.HighpassFilter
```

```
HPF =
    dsp.HighpassFilter with properties:
```

```
        FilterType: 'FIR'
    DesignForMinimumOrder: true
        StopbandFrequency: 8000
        PassbandFrequency: 12000
    StopbandAttenuation: 80
        PassbandRipple: 0.1000
    NormalizedFrequency: false
        SampleRate: 44100
```

```
Show all properties
```

```
HPFMeas = measure(HPF)
```

```
HPFMeas =
    Sample Rate      : 44.1 kHz
    Stopband Edge    : 8 kHz
    6-dB Point       : 10.418 kHz
    3-dB Point       : 10.8594 kHz
    Passband Edge    : 12 kHz
    Stopband Atten.  : 81.8558 dB
    Passband Ripple  : 0.08066 dB
    Transition Width : 4 kHz
```

## Measure Frequency Response Characteristics of Lowpass Filter

Measure the frequency response characteristics of a lowpass filter. Create a `dsp.LowpassFilter` System object with default properties. Measure the frequency response characteristics of the filter.

```
LPF = dsp.LowpassFilter
```

```
LPF =
```

```
  dsp.LowpassFilter with properties:
```

```
          FilterType: 'FIR'
  DesignForMinimumOrder: true
    PassbandFrequency: 8000
    StopbandFrequency: 12000
      PassbandRipple: 0.1000
  StopbandAttenuation: 80
  NormalizedFrequency: false
        SampleRate: 44100
```

```
Show all properties
```

```
LPFMeas = measure(LPFF)
```

```
LPFMeas =
```

```
Sample Rate      : 44.1 kHz
Passband Edge    : 8 kHz
3-dB Point       : 9.1311 kHz
6-dB Point       : 9.5723 kHz
Stopband Edge    : 12 kHz
Passband Ripple  : 0.08289 dB
Stopband Atten.  : 81.6141 dB
Transition Width  : 4 kHz
```

## Input Arguments

### sysobj — Input filter

filter System object

Input filter, specified as one of the following filter System objects:

- `dsp.CICCompensationDecimator`
- `dsp.CICCompensationInterpolator`
- `dsp.FIRHalfbandDecimator`
- `dsp.FIRHalfbandInterpolator`
- `dsp.HighpassFilter`
- `dsp.IIRHalfbandDecimator`
- `dsp.IIRHalfbandInterpolator`
- `dsp.LowpassFilter`

When `sysobj` is a generic discrete-time filter, for example, a single-rate lowpass filter, `measure(sysobj)` returns the following filter specifications.

| Lowpass Filter Specification | Description  |
|------------------------------|--|
| Sample Rate                  | Filter sampling frequency.   |
| Passband Edge                | Location of the edge of the passband as it enters transition.  |
| 3-dB Point                   | Location of the -3 dB point on the response curve.   |
| 6-dB Point                   | Location of the -6 dB point on the response curve.   |
| Stopband Edge                | Location of the edge of the transition band as it enters the stopband.   |
| Passband Ripple              | Ripple in the passband.  |
| Stopband Atten               | Attenuation in the stopband.   |
| Transition Width             | Width of the transition between the passband and stopband, in normalized frequency or absolute frequency. Measured between $F_{pass}$ and $F_{stop}$ . |

When `sysobj` is a bandstop filter, `measure(sysobj)` returns these specifications for the resulting bandstop filter.

| Bandstop Filter Specification | Description  |
|-------------------------------|--|
| Sample Rate                   | Filter sampling frequency.   |
| First Passband Edge           | Location of the edge of the first passband.                                      |
| First 3-dB Point              | Location of the edge of the -3 dB point in the first transition band.            |
| First 6-dB Point              | Location of the edge of the -6 dB point in the first transition band.            |
| First Stopband Edge           | Location of the start of the stopband.   |
| Second Stopband Edge          | Location of the end of the stopband.   |
| Second 6-dB Point             | Location of the edge of the -6 dB point in the second transition band.           |
| Second 3-dB Point             | Location of the edge of the -3 dB point in the second transition band.           |
| Second Passband Edge          | Location of the start of the second passband.                                    |
| First Passband Ripple         | Ripple in the first passband.  |
| Stopband Atten                | Attenuation in the stopband.   |
| Second Passband Ripple        | Ripple in the second passband.   |
| First Transition Width        | Width of the first transition region. Measured between the -3 and -6 dB points.  |
| Second Transition Width       | Width of the second transition region. Measured between the -6 and -3 dB points. |

When `sysobj` is an interpolator, decimator, or a rate converter, `measure(sysobj)` returns these specifications for the resulting filter.

| Interpolator Filter Specification | Description                |
|-----------------------------------|----------------------------|
| Sample Rate                       | Filter sampling frequency. |

| Interpolator Filter Specification | Description  |
|-----------------------------------|--|
| First Passband Edge               | Location of the edge of the passband as it enters transition.  |
| 3-dB Point                        | Location of the -3 dB point on the response curve.   |
| 6-dB Point                        | Location of the -6 dB point on the response curve.   |
| Stopband Edge                     | Location of the edge of the transition band as it enters the stopband.   |
| Passband Ripple                   | Ripple in the passband.  |
| Stopband Atten                    | Attenuation in the stopband.   |
| Transition Width                  | Width of the transition between the passband and stopband, in normalized frequency or absolute frequency. Measured between $F_{pass}$ and $F_{stop}$ . |

### arithType – Arithmetic type

'double' (default) | 'single' | 'Fixed'

Arithmetic used in the filter analysis, specified as 'double', 'single', or 'Fixed'. When the arithmetic input is not specified and the filter System object is unlocked, the analysis tool assumes a double-precision filter. When the arithmetic input is not specified and the System object is locked, the function performs the analysis based on the data type of the locked input.

The 'Fixed' value applies to filter System objects with fixed-point properties only.

When the 'Arithmetic' input argument is specified as 'Fixed' and the filter object has the data type of the coefficients set to 'Same word length as input', the arithmetic analysis depends on whether the System object is unlocked or locked.

- unlocked -- The analysis object function cannot determine the coefficients data type. The function assumes that the coefficients data type is signed, has a 16-bit word length, and is auto scaled. The function performs fixed-point analysis based on this assumption.
- locked -- When the input data type is 'double' or 'single', the analysis object function cannot determine the coefficients data type. The function assumes that the data type of the coefficients is signed, has a 16-bit word length, and is auto scaled. The function performs fixed-point analysis based on this assumption.

To check if the System object is locked or unlocked, use the `isLocked` function.

When the arithmetic input is specified as 'Fixed' and the filter object has the data type of the coefficients set to a custom numeric type, the object function performs fixed-point analysis based on the custom numeric data type.

### freqspecvalue – Frequency specifications

scalar

Frequency specifications are input to `measure` in order to determine the corresponding magnitude measurements. For designs that do not specify some of the frequency constraints, you can determine the corresponding magnitude measurements using this option.

In the following example, the passband edge, passband ripple, and the transition width of the IIR filter are unknown.

```
designLowpass = fdesign.lowpass('N,F3dB,Ast',8,0.5,80);
chebFilter = design(designLowpass,'cheby2');
measure(chebFilter)
```

```
Sample Rate      : N/A (normalized frequency)
Passband Edge    : Unknown
3-dB Point       : 0.5
6-dB Point       : 0.51823
Stopband Edge    : 0.68727
Passband Ripple  : Unknown
Stopband Atten.  : 79.9994 dB
Transition Width  : Unknown
```

Specify the passband edge to be 0.4, and measure the passband ripple and the transition width of this filter.

```
measure(chebFilter,'Fpass',0.4)
```

```
Sample Rate      : N/A (normalized frequency)
Passband Edge    : 0.4
3-dB Point       : 0.5
6-dB Point       : 0.51823
Stopband Edge    : 0.68727
Passband Ripple  : 0.013644 dB
Stopband Atten.  : 79.9994 dB
Transition Width  : 0.28727
```

## Output Arguments

### M — Measurements

`fdesign` object

Measurements object, returned as an `fdesign` object. Here is a list of supported input filter objects with their corresponding `fdesign` measurements objects:

- `dsp.CICCompensationDecimator` -- `fdesign.isinclpmeas`.
- `dsp.CICCompensationInterpolator` -- `fdesign.isinclpmeas`
- `dsp.FIRHalfbandDecimator` -- `fdesign.lowpassmeas`
- `dsp.FIRHalfbandInterpolator` -- `fdesign.lowpassmeas`
- `dsp.HighpassFilter` -- `fdesign.highpassmeas`
- `dsp.IIRHalfbandDecimator` -- `fdesign.lowpassmeas`
- `dsp.IIRHalfbandInterpolator` -- `fdesign.lowpassmeas`
- `dsp.LowpassFilter` -- `fdesign.lowpassmeas`

The measurements, *M* can be queried programmatically. For example, to query the 3 dB point, type `M.F3dB`. Type `get(M)` to see the full list of properties that can be queried. Note that different filter responses generate different measurements.

## Tips

For designs that do not specify some of the frequency constraints, the function may not be able to determine corresponding magnitude measurements. In these cases, a constraint can be passed in to `measure` to determine such measurements. For example:



```
f = fdesign.lowpass('N,F3dB,Ast',8,0.5,80);  
H = design(f,'cheby2','SystemObject',true);  
measure(H)
```

returns values of Unknown for the passband edge, passband ripple, and transition width measurements, but

```
f = fdesign.lowpass('N,F3dB,Ast',8,0.5,80);  
H = design(f,'cheby2','SystemObject',true);  
measure(H,'Fpass',0.4)
```

provides measurements for all returned values.

## Version History

Introduced in R2011a

### See Also

#### Functions

design | fdesign | normalizefreq

## mfilt

(To be removed) Multirate filter

### Compatibility

`mfilt` will be removed in a future release. See `dsp.CICDecimator`, `dsp.CICInterpolator`, `dsp.FIRDecimator`, `dsp.FIRInterpolator`, `dsp.FilterCascade`, `dsp.FarrowRateConverter`, `dsp.FIRRateConverter`, `dsp.IIRHalfbandDecimator`, or `dsp.IIRHalfbandInterpolator` instead.

### Syntax

```
hm = mfilt.structure(input1,input2,...)
```

### Description

`hm = mfilt.structure(input1,input2,...)` returns the object `hm` of type *structure*. As with `dfilt` objects, you must include the *structure* to construct a multirate filter object. You can, however, construct a default multirate filter object of a given structure by not including input arguments in your calling syntax.

Multirate filters include decimators and interpolators, and fractional decimators and fractional interpolators where the resulting interpolation or decimation factor is not an integer.

### Structures

Each of the following multirate filter structures has a reference page of its own.

| Filter Structure                | Description of Resulting Multirate Filter        | Coefficient Mapping Support in <code>realizemdl</code> |
|---------------------------------|--|--|
| <code>mfilt.cascade</code>      | Cascade multirate filters to form another filter | Supported  |
| <code>mfilt.cicdecim</code>     | Cascaded integrator-comb decimator               | Not supported  |
| <code>mfilt.cicinterp</code>    | Cascaded integrator-comb interpolator            | Not supported  |
| <code>mfilt.farrowsrc</code>    | Multirate Farrow filter                          | Supported.   |
| <code>mfilt.fftfirinterp</code> | Overlap-add FIR polyphase interpolator           | Not supported  |
| <code>mfilt.firdecim</code>     | Direct-form FIR polyphase decimator              | Supported  |
| <code>mfilt.firinterp</code>    | Direct-form FIR polyphase interpolator           | Supported  |
| <code>mfilt.firsrc</code>       | Direct-form FIR polyphase sample rate converter  | Supported  |

| Filter Structure                | Description of Resulting Multirate Filter      | Coefficient Mapping Support in <code>realizemdl</code> |
|---------------------------------|--|--|
| <code>mfilt.firtdecim</code>    | Direct-form transposed FIR polyphase decimator | Supported  |
| <code>mfilt.holdinterp</code>   | FIR hold interpolator                          | Not supported  |
| <code>mfilt.iirdecim</code>     | IIR decimator                                  | Supported  |
| <code>mfilt.iirinterp</code>    | IIR interpolator                               | Supported  |
| <code>mfilt.linearinterp</code> | FIR Linear interpolator                        | Supported  |
| <code>mfilt.iirwdfdecim</code>  | IIR wave digital filter decimator              | Supported  |
| <code>mfilt.iirwdfinterp</code> | IIR wave digital filter interpolator           | Supported  |

### Copying mfilt Objects

To create a copy of an `mfilt` object, use the `copy` method.

```
h2 = copy(hd)
```

---

**Note** The syntax `hd2 = hd` copies only the object handle. It does not create a new object. `hd2` and `hd` are not independent. If you change the property value for one of the two, such as `hd2`, you are changing the property for both.

---

### Examples

Decimation by a factor of two. Convert input sampled at 48 kHz to 24 kHz:

```
Fs = 4.8e4;
t = 0:1/Fs:1-(1/Fs);
x = cos(2*pi*4000*t);
Hm = mfilt.firdecim(2);
% Note cutoff frequency of 1/2 normalized frequency
fvtool(Hm);
% Note the group delay of 34 samples
fvtool(Hm, 'analysis', 'grpdelay');
y = filter(Hm, x);
% Note delay in output is consistent with 36/2
stem(y(1:48), 'markerfacecolor', [0 0 1]);
```

Using existing coefficients to decimate a signal by a factor of two:

```
M = 2; % Decimation factor
b = firhalfband('minorder', .45, 0.0001);
Hm = mfilt.firdecim(M, b);
% Decimate a signal which consists of the sum of 2 sinusoids.
N = 160;
x = sin(2*pi*.05*[0:N-1]+pi/3) + cos(2*pi*.03*[0:N-1]+pi/3);
y = filter(Hm, x);
```

---

**Note** Multirate filters can also have complex coefficients. For example, you can specify complex coefficients in the argument `num` passed to the filter structure. This works for all multirate filter structures.

```
m = 2;
num = [0.5 0.5+1j*0.2];
```

```
Hm = mfilt.firdecim(m, num);  
y = filter(Hm, [1:10]);
```

---

## **Version History**

**Introduced in R2011a**

# mfilt.cascade

Cascade filter objects

---

**Note** `mfilt.cascade` will be removed in a future release. Use `dsp.FilterCascade` instead.

---

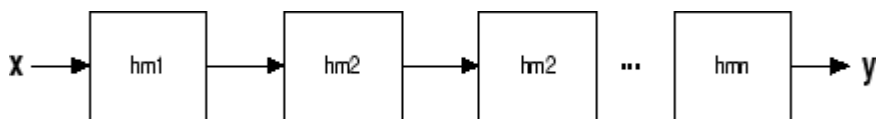
## Syntax

```
hm = cascade(hm1,hm2,...,hmn)
```

## Description

`hm = cascade(hm1,hm2,...,hmn)` creates filter object `hm` by cascading (connecting in series) the individual filter objects `hm1`, `hm2`, and so on to `hmn`.

In block diagram form, the cascade looks like this, with `x` as the input to the filter `hm` and `y` the output from the cascade filter `hm`:



`mfilt.cascade` accepts any combination of `mfilt` and `dfilt` objects (discrete time filters) to cascade, as well as Farrow filter objects.

## Examples

Create a variety of `mfilt` objects and cascade them together.

```

hm(1) = mfilt.firdecim(12);
hm(2) = mfilt.firdecim(4);
h1 = mfilt.cascade(hm(1),hm(2));
hm(3) = mfilt.firinterp(4);
hm(4) = mfilt.firinterp(12);
h2 = mfilt.cascade(hm(3),hm(4));
% Cascade h1 and h2 together
h3 = mfilt.cascade(h1,h2,9600);
  
```

## Version History

Introduced in R2011a

## See Also

`dfilt.cascade`

## mfilt.cicdecim

Fixed-point CIC decimator

---

**Note** `mfilt.cicdecim` will be removed in a future release. Use `dsp.CICDecimator` instead.

---

### Syntax

```
hm = mfilt.cicdecim(r,m,n,iwl,owl,wlps)
```

### Description

`hm = mfilt.cicdecim(r,m,n,iwl,owl,wlps)` returns a cascaded integrator-comb (CIC) decimation filter object.

All of the input arguments are optional. To enter any optional value, you must include all optional values to the left of your desired value.

When you omit one or more input options, the omitted option applies the default values shown in the table below.

The following table describes the input arguments for creating `hm`.

| Input Arguments  | Description   |
|------------------|---|
| <code>r</code>   | Decimation factor applied to the input signal. Sharpens the response curve to let you change the shape of the response. <code>r</code> must be an integer value greater than or equal to 1. The default value is 2.   |
| <code>m</code>   | Differential delay. Changes the shape, number, and location of nulls in the filter response. Increasing <code>m</code> increases the sharpness of the nulls and the response between nulls. In practice, differential delay values of 1 or 2 are the most common. <code>m</code> must be an integer value greater than or equal to 1. The default value is 1. |
| <code>n</code>   | Number of sections. Deepens the nulls in the response curve. Note that this is the number of either comb or integrator sections, not the total section count. 2 is the default value.   |
| <code>iwl</code> | Word length of the input signal. Use any integer number of bits. The default value is 16 bits.  |
| <code>owl</code> | Word length of the output signal. It can be any positive integer number of bits. By default, <code>owl</code> is 16 bits.   |

| Input Arguments | Description  |
|-----------------|--|
| wlps            | <p>Defines the number of bits per word in each filter section while accumulating the data in the integrator sections or while subtracting the data during the comb sections (using 'wrap' arithmetic). Enter wlps as a scalar or vector of length 2*n, where n is the number of sections. When wlps is a scalar, the scalar value is applied to each filter section. The default is 16 for each section in the decimator.</p> <p>When you elect to specify wlps as an input argument, the FilterInternals property automatically switches from the default value of 'FullPrecision' to 'SpecifyWordLengths'.</p> |

### Constraints and Word Length Considerations

CIC decimators have the following constraint — the word lengths of the filter section must be monotonically decreasing. The word length of each filter section must be the same size as, or smaller than, the word length of the previous filter section.

The formula for  $B_{max}$ , the most significant bit at the filter output, is given in the Hogenauer paper in the References on page 5-1176 below.

$$B_{max} = (N \log_2 RM + B_{in} - 1)$$

where  $B_{in}$  is the number of bits of the input.

The cast operations shown in the diagram in “Algorithms” on page 5-1176 perform the changes between the word lengths of each section. When you specify word lengths that do not follow the constraints above, the constructor returns an error.

When you specify the word lengths correctly, the most significant bit  $B_{max}$  stays the same throughout the filter, while the word length of each section either decreases or stays the same. This can cause the fraction length to change throughout the filter as least significant bits are truncated to decrease the word length, as shown in “Algorithms” on page 5-1176.

### Properties of the Object

Objects have properties that control the way the object behaves. This table lists all the properties for the filter, with a description of each.

| Name              | Values               | Default | Description  |
|-------------------|----------------------|---------|--|
| Arithmetic        | fixed                | fixed   | Reports the kind of arithmetic the filter uses. CIC decimators are always fixed-point filters. |
| DecimationFactor  | Any positive integer | 2       | Amount to reduce the input sampling rate.  |
| DifferentialDelay | Any positive integer | 1       | Sets the differential delay for the filter. Usually a value of one or two is appropriate.      |

| Name             | Values   | Default       | Description  |
|------------------|--|---------------|--|
| FilterStructure  | mfilt structure  | None          | Reports the type of filter object. You cannot set this property — it is always read only and results from your choice of mfilt objects.  |
| FilterInternals  | FullPrecision,<br>MinWordLengths,<br>SpecifyPrecision,<br>SpecifyWordLengths | FullPrecision | Set the usage mode for the filter. Refer to “Usage Modes” on page 5-1170 below for details.  |
| InputFracLength  | Any positive integer   | 15            | The number of bits applied to the fraction length to interpret the input data to the filter.   |
| InputOffset      | Integers in the range<br>$0 \leq \text{InputOffset} \leq r-1$                | 0             | Contains a value derived from the number of input samples and the decimation factor — $\text{InputOffset} = \text{mod}(\text{length}(nx), m)$ where $nx$ is the number of input samples that have been processed so far and $m$ is the decimation factor.<br><br>The InputOffset property applies only when you set the PersistentMemory property to true. See “InputOffset” on page 6-9 for more information. |
| InputWordLength  | Any positive integer   | 16            | The number of bits applied to the word length to interpret the input data to the filter.   |
| NumberOfSections | Any positive integer   | 2             | Number of sections used in the decimator. Generally called $n$ . Reflects either the number of decimator or comb sections, not the total number of sections in the filter.   |
| OutputFracLength | Any positive integer   | 15            | The number of bits applied to the fraction length to interpret the output data from the filter. Read-only.   |
| OutputWordLength | Any positive integer   | 16            | The number of bits applied to the word length to interpret the output data from the filter.  |



| Name             | Values        | Default | Description  |
|------------------|---------------|---------|--|
| PersistentMemory | false or true | false   | Determines whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. PersistentMemory returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. When PersistentMemory is false, you cannot access the filter states. Setting PersistentMemory to true reveals the States property so you can modify the filter states. |

| Name               | Values                                 | Default | Description   |
|--------------------|--|---------|---|
| SectionWordLengths | Any integer or a vector of length 2*n. | 16      | Defines the bits per section used while accumulating the data in the integrator sections or while subtracting the data during the comb sections (using 'wrap' arithmetic). Enter SectionWordLengths as a scalar or vector of length 2*n, where n is the number of sections. When SectionWordLengths is a scalar, the scalar value is applied to each filter section. When SectionWordLengths is a vector of values, the values apply to the sections in order. The default is 16 for each section in the decimator. Available when FilterInternals is 'SpecifyWordLengths'. |

| Name   | Values                             | Default  | Description  |
|--------|------------------------------------|--|--|
| States | <code>filtstates.cic</code> object | $m+1$ -by- $n$ matrix of zeros, after you call function <code>int</code> . | Stored conditions for the filter, including values for the integrator and comb sections before and after filtering. $m$ is the differential delay of the comb section and $n$ is the number of sections in the filter. The integrator states are stored in the first matrix row. States for the comb section fill the remaining rows in the matrix. Available for modification when <code>PersistentMemory</code> is true. Refer to the <code>filtstates</code> object in Signal Processing Toolbox documentation for more general information about the <code>filtstates</code> object. |

### Usage Modes

There are four modes of usage for this which are set using the `FilterInternals` property

- `FullPrecision` — All word and fraction lengths set to  $B_{max} + 1$ , called  $B_{accum}$  by Fred Harris in [3]. Full Precision is the default setting.
- `MinWordLengths` — Automatically set the sections for minimum word lengths.
- `SpecifyWordLengths` — Specify the word lengths for each section.
- `SpecifyPrecision` — Specify precision by providing values for the word and fraction lengths for each section.

Full Precision

In full precision mode, the word lengths of all sections and the output are set to  $B_{accum}$  as defined by

$$B_{accum} = \text{ceil}(N_{secs}(\text{Log}_2(D \times M)) + \text{InputWordLength})$$

where  $N_{secs}$  is the number of filter sections.

Section fraction lengths and the fraction length of the output are set to the input fraction length.

Here is the display for this mode:

```
FilterStructure: 'Cascaded Integrator-Comb Decimator'
Arithmetic: 'fixed'
DifferentialDelay: 1
NumberOfSections: 2
DecimationFactor: 4
PersistentMemory: false
```

```
InputWordLength: 16
InputFracLength: 15
```

```
FilterInternals: 'FullPrecision'
```

#### Minimum Wordlengths

In minimum word length mode, you control the output word length explicitly. When the output word length is less than  $B_{accum}$ , roundoff noise is introduced at the output of the filter. Hogenauer's bit pruning theory (refer to [1]) states that one valid design criterion is to make the word lengths of the different sections of the filter smaller than  $B_{accum}$  as well, so that the roundoff noise introduced by all sections does not exceed the roundoff noise introduced at the output.

In this mode, the design calculates the word lengths of each section to meet the Hogenauer criterion. The algorithm subtracts the number of bits computed using eq. 21 in Hogenauer's paper from  $B_{accum}$  to determine the word length each section.

To compute the fraction lengths of the different sections, the algorithm notes that the bits thrown out for this word length criterion are least significant bits (LSB), therefore each bit thrown out at a particular section decrements the fraction length of that section by one bit compared to the input fraction length. Setting the output wordlength for the filter automatically sets the output fraction length as well.

Here is the display for this mode:

```
FilterStructure: 'Cascaded Integrator-Comb Decimator'
Arithmetic: 'fixed'
DifferentialDelay: 1
NumberOfSections: 2
DecimationFactor: 4
PersistentMemory: false
```

```
InputWordLength: 16
InputFracLength: 15
```

```
FilterInternals: 'MinWordLengths'
```

```
OutputWordLength: 16
```

#### Specify word lengths

In this mode, the design algorithm discards the LSBs, adjusting the fraction length so that unrecoverable overflow does not occur, always producing a reasonable output.

You can specify the word lengths for all sections and the output, but you cannot control the fraction lengths for those quantities.

To specify the word lengths, you enter a vector of length  $2*(NumberOfSections)$ , where each vector element represents the word length for a section. If you specify a scalar, such as  $B_{accum}$ , the full-precision output word length, the algorithm expands that scalar to a vector of the appropriate size, applying the scalar value to each section.

The CIC design does not check that the specified word lengths are monotonically decreasing. There are some cases where the word lengths are not necessarily monotonically decreasing, for example

```
hcic=mfilt.cicdecim;
hcic.FilterInternals='minwordlengths';
hcic.Outputwordlength=14;
```

which are valid CIC filters but the word lengths do not decrease monotonically across the sections.

Here is the display for the SpecifyWordLengths mode.

```
FilterStructure: 'Cascaded Integrator-Comb Decimator'  
Arithmetic: 'fixed'  
DifferentialDelay: 1  
NumberOfSections: 2  
DecimationFactor: 4  
PersistentMemory: false
```

```
InputWordLength: 16  
InputFracLength: 15
```

```
FilterInternals: 'SpecifyWordLengths'
```

```
SectionWordLengths: [19 18 18 17]
```

```
OutputWordLength: 16
```

Specify precision

In this mode, you have full control over the word length and fraction lengths of all sections and the filter output.

When you elect the SpecifyPrecision mode, you must enter a vector of length  $2*(NumberOfSections)$  with elements that represent the word length for each section. When you enter a scalar such as  $B_{accum}$ , `mfilt.cicdecim` expands that scalar to a vector of the appropriate size and applies the scalar value to each section and the output. The design does not check that this vector is monotonically decreasing.

Also, you must enter a vector of length  $2*(NumberOfSections)$  with elements that represent the fraction length for each section as well. When you enter a scalar such as  $B_{accum}$ , `mfilt.cicdecim` applies scalar expansion as done for the word lengths.

Here is the SpecifyPrecision display.

```
FilterStructure: 'Cascaded Integrator-Comb Decimator'  
Arithmetic: 'fixed'  
DifferentialDelay: 1  
NumberOfSections: 2  
DecimationFactor: 4  
PersistentMemory: false
```

```
InputWordLength: 16  
InputFracLength: 15
```

```
FilterInternals: 'SpecifyPrecision'
```

```
SectionWordLengths: [19 18 18 17]  
SectionFracLengths: [14 13 13 12]
```

```
OutputWordLength: 16  
OutputFracLength: 11
```

## About the States of the Filter

In the `states` property you find the states for both the integrator and comb portions of the filter. `states` is a matrix of dimensions  $m + 1$ -by- $n$ , with the states apportioned as follows:

- States for the integrator portion of the filter are stored in the first row of the state matrix.
- States for the comb portion fill the remaining rows in the state matrix.

To review the states of a CIC filter, use `int` to assign the states to a variable in MATLAB. As an example, here are the states for a CIC decimator `hm` before and after filtering a data set.

```
x = fi(ones(1,10),true,16,0); % Fixed-point input data.
hm = mfilt.cicdecim(2,1,2,16,16,16);
sts=int(hm.states)
set(hm,'InputFracLength',0); % Integer input specified.
y=filter(hm,x);
sts=int(hm.states)
```

STS is an integer matrix that `int` returns from the contents of the `filtstates.cic` object in `hm`.

## Design Considerations

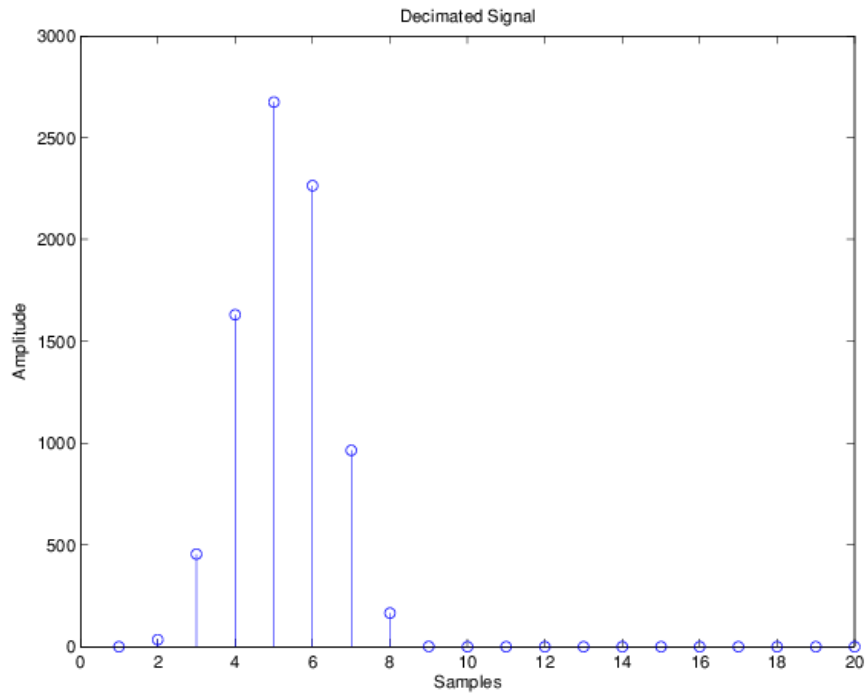
When you design your CIC decimation filter, remember the following general points:

- The filter output spectrum has nulls at  $\omega = k * 2\pi/rm$  radians,  $k = 1,2,3,\dots$
- Aliasing and imaging occur in the vicinity of the nulls.
- $n$ , the number of sections in the filter, determines the passband attenuation. Increasing  $n$  improves the filter ability to reject aliasing and imaging, but it also increases the droop (or rolloff) in the filter passband. Using an appropriate FIR filter in series after the CIC decimation filter can help you compensate for the induced droop.
- The DC gain for the filter is a function of the decimation factor. Raising the decimation factor increases the DC gain.

## Examples

This example applies a decimation factor  $r$  equal to 8 to a 160-point impulse signal. The signal output from the filter has  $160/r$ , or 20, points or samples. Choosing 10 bits for the word length represents a fairly common setting for analog to digital converters. The plot shown after the code presents the stem plot of the decimated signal, with 20 samples remaining after decimation:

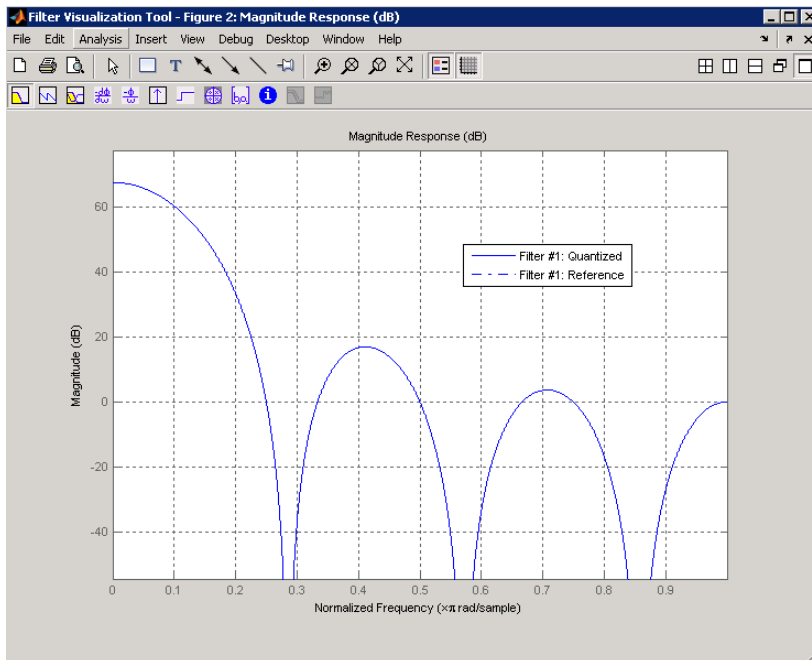
```
m = 2; % Differential delays in the filter.
n = 4; % Filter sections
r = 8; % Decimation factor
x = int16(zeros(160,1)); x(1) = 1; % Create a 160-point
% impulse signal.
hm = mfilt.cicdecim(r,m,n); % Expects 16-bit input
% by default.
y = filter(hm,x);
stem(double(y)); % Plot output as a stem plot.
xlabel('Samples'); ylabel('Amplitude');
title('Decimated Signal');
```



The next example demonstrates one way to compute the filter frequency response, using a 4-section decimation filter with the decimation factor set to 7:

```
hm = mfilt.cicdecim(7,1,4);  
fvtool(hm)
```

FVTool provides ways for you to change the title and x labels to match the figure shown. Here's the frequency response plot for the filter. For details about the transfer function used to produce the frequency response, refer to [1] in the References on page 5-1176 section.



This final example demonstrates the decimator for converting from 44.1 kHz audio to 22.05 kHz — decimation by two. To overlay the before and after signals, scale the output and plot the signals on a stem plot.

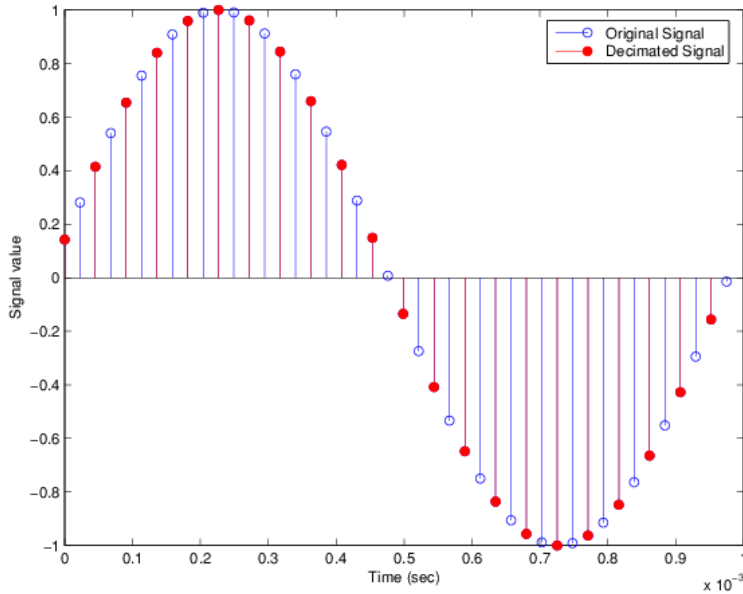
```

r = 2; % Decimation factor.
hm = mfilt.cicdecim(r); % Use default NumberOfSections &
% DifferentialDelay property values.
fs = 44.1e3; % Original sampling frequency: 44.1kHz.
n = 0:10239; % 10240 samples, 0.232 second long signal.
x = sin(2*pi*1e3/fs*n); % Original signal, sinusoid at 1kHz.

y_fi = filter(hm,x); % 5120 samples, still 0.232 seconds.

% Scale the output to overlay the stem plots.
x = double(x);
y = double(y_fi);
y = y/max(abs(y));
stem(n(1:44)/fs,x(2:45)); hold on; % Plot original signal
% sampled at 44.1kHz.
stem(n(1:22)/(fs/r),y(3:24),'r','filled'); % Plot decimated
% signal (22.05kHz)
% in red.
xlabel('Time (seconds)');ylabel('Signal Value');

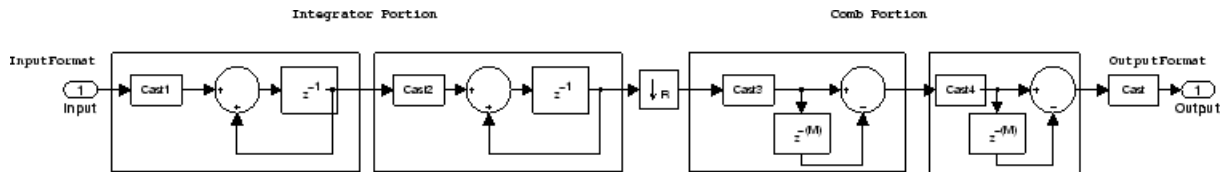
```



## Algorithms

To show how the CIC decimation filter is constructed, the following figure presents a block diagram of the filter structure for a two-section CIC decimation filter ( $n = 2$ ).  $fs$  is the high sampling rate, the input to the decimation process.

For details about the bits that are removed in the Comb section, refer to [1] in References.



`mfilt.cicdecim` calculates the fraction length at each section of the decimator to avoid overflows at the output of the filter.

## Version History

Introduced in R2011a

## References

- [1] Hogenauer, E. B., "An Economical Class of Digital Filters for Decimation and Interpolation," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-29(2): pp. 155-162, 1981
- [2] Meyer-Baese, Uwe, "Hogenauer CIC Filters," in *Digital Signal Processing with Field Programmable Gate Arrays*, Springer, 2001, pp. 155-172
- [3] Harris, Fredric J, *Multirate Signal Processing for Communication Systems*, Prentice-Hall PTR, 2004 , pp. 343



**See Also**

`mfilt` | `mfilt.cicinterp`

## mfilt.cicinterp

(Removed) Fixed-point CIC interpolator

---

**Note** `mfilt.cicinterp` has been removed. Use `dsp.CICInterpolator` instead. For more details, see “Compatibility Considerations”.

---

### Syntax

```
hm = mfilt.cicinterp(R,M,N,ILW,OWL,WLPS)
hm = mfilt.cicinterp
hm = mfilt.cicinterp(R,...)
```

### Description

`hm = mfilt.cicinterp(R,M,N,ILW,OWL,WLPS)` constructs a cascaded integrator-comb (CIC) interpolation filter object that uses fixed-point arithmetic.

All of the input arguments are optional. To enter any optional value, you must include all optional values to the left of your desired value.

When you omit one or more input options, the omitted option applies the default values shown in the table below.

The following table describes the input arguments for creating `hm`.

| Input Arguments | Description   |
|-----------------|---|
| R               | Interpolation factor applied to the input signal. Sharpens the response curve to let you change the shape of the response. R must be an integer value greater than or equal to 1. The default value is 2.   |
| M               | Differential delay. Changes the shape, number, and location of nulls in the filter response. Increasing M increases the sharpness of the nulls and the response between nulls. In practice, differential delay values of 1 or 2 are the most common. M must be an integer value greater than or equal to 1. The default value is 1. |
| N               | Number of sections. Deepens the nulls in the response curve. Note that this is the number of either comb or integrator sections, not the total section count. By default, the filter has two sections.  |
| ILW             | Word length of the input signal. Use any integer number of bits. The default value is 16 bits.  |
| OWL             | Word length of the output signal. It can be any positive integer number of bits. By default, OWL is 16 bits.  |

| Input Arguments | Description   |
|-----------------|---|
| WLPS            | <p>Defines the number of bits per word in each filter section while accumulating the data in the integrator sections or while subtracting the data during the comb sections (using 'wrap' arithmetic). Enter WLPS as a scalar or vector of length <math>2*N</math>, where <math>N</math> is the number of sections. When WLPS is a scalar, the scalar value is applied to each filter section. The default is 16 for each section in the integrator.</p> <p>When you elect to specify <code>wlps</code> as an input argument, the <code>FilterInternals</code> property automatically switches from the default value of 'FullPrecision' to 'SpecifyWordLengths'.</p> |

`hm = mfilt.cicinterp` constructs the CIC interpolator using the default values for the optional input arguments.

`hm = mfilt.cicinterp(R, ...)` constructs the CIC interpolator applying the values you provide for `R` and any other values you specify as input arguments.

### Constraints and Conversions

In Hogenauer [1], the author describes the constraints on CIC interpolator filters. `mfilt.cicinterp` enforces a constraint—the word lengths of the filter sections must be non-decreasing. That is, the word length of each filter section must be the same size as, or greater than, the word length of the previous filter section.

The formula for  $W_j$ , the minimum register width, is derived in [1]. The formula for  $W_j$  is given by

$$W_j = \text{ceil}(B_{in} + \log_2 G_j)$$

where  $G_j$ , the maximum register growth up to the  $j$ th section, is given by

$$G_j = \begin{cases} 2^j, & j = 1, 2, \dots, N \\ \frac{2^{2N-j}(RM)^{j-N}}{R}, & j = N + 1, \dots, 2N \end{cases}$$

When the differential delay,  $M$ , is 1, there is also a special condition for the register width of the last comb,  $W_N$ , that is given by

$$W_N = B_{in} + N - 1 \text{ if } M = 1$$

The conversions denoted by the cast blocks in the integrator diagrams in “Algorithms” on page 5-1184 perform the changes between the word lengths of each section. When you specify word lengths that do not follow the constraints described in this section, `mfilt.cicinterp` returns an error.

The fraction lengths and scalings of the filter sections do not change. At each section the word length is either staying the same or increasing. The signal scaling can change at the output after the final filter section if you choose the output word length to be less than the word length of the final filter section.

### Properties of the Object

The following table lists the properties for the filter with a description of each.

| Name                | Values   | Default       | Description   |
|---------------------|--|---------------|---|
| Arithmetic          | fixed  | fixed         | Reports the kind of arithmetic the filter uses. CIC interpolators are always fixed-point filters.   |
| InterpolationFactor | Any positive integer   | 2             | Amount to increase the input sampling rate.   |
| DifferentialDelay   | Any positive integer   | 1             | Sets the differential delay for the filter. Usually a value of one or two is appropriate.   |
| FilterStructure     | mfilt structure  | None          | Reports the type of filter object, such as an interpolator or fractional integrator. You cannot set this property — it is always read only and results from your choice of mfilt objects. |
| FilterInternals     | FullPrecision,<br>MinWordLengths,<br>SpecifyWordLengths,<br>SpecifyPrecision | FullPrecision | Set the usage mode for the filter. Refer to “Usage Modes” on page 5-1182 below for details.   |
| InputFracLength     | Any positive integer   | 16            | The number of bits applied as the fraction length to interpret the input data to the filter.  |
| InputWordLength     | Any positive integer   | 16            | The number of bits applied to the word length to interpret the input data to the filter.  |
| NumberOfSections    | Any positive integer   | 2             | Number of sections used in the interpolator. Generally called <i>n</i> . Reflects either the number of interpolator or comb sections, not the total number of sections in the filter.     |
| OutputFracLength    | Any positive integer   | 15            | The number of bits applied to the fraction length to interpret the output data from the filter. Read-only.  |
| OutputWordLength    | Any positive integer   | 16            | The number of bits applied to the word length to interpret the output data from the filter.   |

| Name               | Values   | Default | Description   |
|--------------------|--|---------|---|
| PersistentMemory   | false or true  | false   | <p>Determines whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it.</p> <p>PersistentMemory returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. When PersistentMemory is false, you cannot access the filter states. Setting PersistentMemory to true reveals the States property so you can modify the filter states.</p>                                |
| SectionWordLengths | <p>Any integer or a vector of length <math>2^N</math>, where <math>N</math> is a positive integer.</p> <p>This property only applies when the FilterInternals is SpecifyWordLengths.</p> | 16      | <p>Defines the bits per section used while accumulating the data in the integrator sections or while subtracting the data during the comb sections (using 'wrap' arithmetic). Enter SectionWordLengths as a scalar or vector of length <math>2*n</math>, where <math>n</math> is the number of sections. When SectionWordLengths is a scalar, the scalar value is applied to each filter section. When SectionWordLengths is a vector of values, the values apply to the sections in order. The default is 16 for each section in the interpolator. Available when FilterInternals is 'SpecifyWordLengths'.</p> |

| Name   | Values                             | Default  | Description  |
|--------|------------------------------------|--|--|
| States | <code>filtstates.cic</code> object | $m+1$ -by- $n$ matrix of zeros, after you call function <code>int</code> . | Stored conditions for the filter, including values for the integrator and comb sections before and after filtering. $m$ is the differential delay of the comb section and $n$ is the number of sections in the filter. The integrator states are stored in the first matrix row. States for the comb section fill the remaining rows in the matrix. Available for modification when <code>PersistentMemory</code> is true. Refer to the <code>filtstates</code> object in Signal Processing Toolbox documentation for more general information about the <code>filtstates</code> object. |

### Usage Modes

There are usage modes which are set using the `FilterInternals` property:

- **FullPrecision** — In this mode, the word and fraction lengths of the filter sections and outputs are automatically selected for you. The output and last section word lengths are set to:
 
$$\text{wordlength} = \text{ceil}(\log_2((RM)^N/R)) + I,$$
 where  $R$  is the interpolation factor,  $M$  is the differential delay,  $N$  is the number of filter sections, and  $I$  denotes the input word length.
- **MinWordLengths** — In this mode, you specify the word length of the filter output in the `OutputWordLength` property. The word lengths of the filter sections are automatically set in the same way as in the `FullPrecision` mode. The section fraction lengths are set to the input fraction length. The output fraction length is set to the input fraction length minus the difference between the last section and output word lengths.
- **SpecifyWordLengths** — In this mode, you specify the word lengths of the filter sections and output in the `SectionWordLengths` and `OutputWordLength` properties. The fraction lengths of the filter sections are set such that the spread between word length and fraction length is the same as in full-precision mode. The output fraction length is set to the input fraction length minus the difference between the last section and output word lengths.
- **SpecifyPrecision** — In this mode, you specify the word and fraction lengths of the filter sections and output in the `SectionWordLengths`, `SectionFracLengths`, `OutputWordLength`, and `OutputFracLength` properties.

### About the States of the Filter

In the `states` property you find the states for both the integrator and comb portions of the filter. `states` is a matrix of dimensions  $m+1$ -by- $n$ , with the states apportioned as follows:

- States for the integrator portion of the filter are stored in the first row of the state matrix.

- States for the comb portion fill the remaining rows in the state matrix.

To review the states of a CIC filter, use the `int` method to assign the states. As an example, here are the states for a CIC interpolator `hm` before and after filtering data:

```
x = fi(cos(pi/4*[0:99]),true,16,0); % Fixed-point input data
hm = mfilt.cicinterp(2,1,2,16,16,16);
% get initial states-all zero
sts=int(hm.states)
set(hm,'InputFracLength',0); % Integer input specified
y=filter(hm,x);
sts=int(hm.states)
%sts =
%
%      -1      -1
%      -1      -1
```

## Design Considerations

When you design your CIC interpolation filter, remember the following general points:

- The filter output spectrum has nulls at  $\omega = k * 2\pi/rm$  radians,  $k = 1,2,3,\dots$
- Aliasing and imaging occur in the vicinity of the nulls.
- $n$ , the number of sections in the filter, determines the passband attenuation. Increasing  $n$  improves the filter ability to reject aliasing and imaging, but it also increases the droop or rolloff in the filter passband. Using an appropriate FIR filter in series after the CIC interpolation filter can help you compensate for the induced droop.
- The DC gain for the filter is a function of the interpolation factor. Raising the interpolation factor increases the DC gain.

## Examples

Demonstrate interpolation by a factor of two, in this case from 22.05 kHz to 44.1 kHz. Note the scaling required to see the results in the stem plot and to use the full range of the `int16` data type.

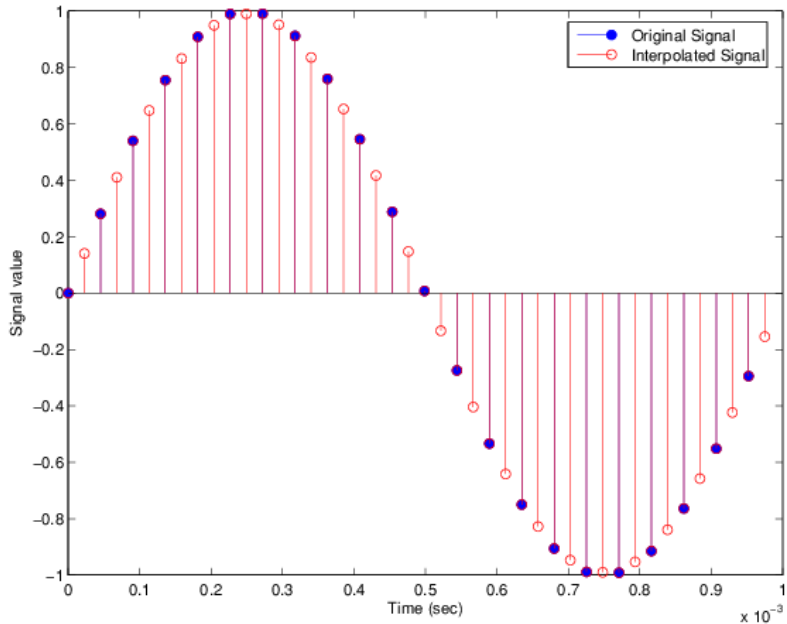
```
R = 2; % Interpolation factor.
hm = mfilt.cicinterp(R); % Use default NumberOfSections and
% DifferentialDelay property values.
fs = 22.05e3; % Original sample frequency:22.05 kHz.
n = 0:5119; % 5120 samples, .232 second long signal.
x = sin(2*pi*1e3/fs*n); % Original signal, sinusoid at 1 kHz.

y_fi = filter(hm,x); % 5120 samples, still 0.232 seconds.

% Scale the output to overlay stem plots correctly.
x = double(x);
y = double(y_fi);
y = y/max(abs(y));
stem(n(1:22)/fs,x(1:22),'filled'); % Plot original signal sampled
% at 22.05 kHz.

hold on;
stem(n(1:44)/(fs*R),y(4:47),'r'); % Plot interpolated signal
% (44.1 kHz) in red.
xlabel('Time (sec)');ylabel('Signal Value');
```

As you expect, the plot shows that the interpolated signal matches the input sine shape, with additional samples between each original sample.



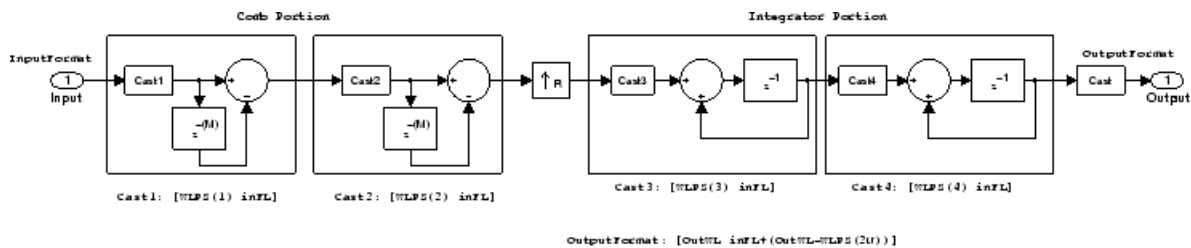
Use the filter visualization tool (FVTool) to plot the response of the interpolator object. For example, to plot the response of an interpolator with an interpolation factor of 7, 4 sections, and 1 differential delay, do something like the following:

```
hm = mfilt.cicinterp(7,1,4)
fvtool(hm)
```

## Algorithms

To show how the CIC interpolation filter is constructed, the following figure presents a block diagram of the filter structure for a two-section CIC interpolation filter ( $n = 2$ ).  $fs$  is the high sampling rate, the output from the interpolation process.

For details about the bits that are removed in the integrator section, refer to [1] in References.



When you select `MinWordLengths`, the filter section word lengths are automatically set to the minimum number of bits possible in a valid CIC interpolator. `mfilt.cicinterp` computes the wordlength for each section so the roundoff noise introduced by all sections is less than the roundoff noise introduced by the quantization at the output.

## Version History

### Introduced in R2011a



**R2022a: mfilt.cicinterp has been removed**

Errors starting in R2022a

mfilt.cicinterp has been removed. Use dsp.CICInterpolator instead.

**Update Code**

This table shows how the mfilt.cicinterp object is typically used and explains how to update existing code to use the dsp.CICInterpolator object.

| Discouraged Usage  | Recommended Replacement   |
|--|---|
| <p>Initialize the object. Set the interpolation factor to 2. Uses default NumSections = 2, DifferentialDelay = 1.</p> <pre>hm = mfilt.cicinterp(2);</pre> <p>To support frame-based processing, set PersistentMemory to true. This property retains the state of the filter.</p> <pre>hm.PersistentMemory = true;</pre> <p>Create fixed-point sinusoidal input signal. Original sample rate is 22.05 kHz. The signal has 512 samples and is 0.0113 seconds long.</p> <pre>Fs = 22.05e3; n = (0:511)'; x = fi(sin(2*pi*1e3/Fs*n), true, 16, 15); SigIn = dsp.SignalSource(x, 32);</pre> <p>Process the input signal. The interpolated output has 64 samples per frame.</p> <pre>K = 6; % Number of frames y_mfilt = zeros(64, K); for jj=1:K     u = SigIn();     y_mfilt(:, jj) = filter(hm, u); end</pre> | <p>Initialize the object. Set the interpolation factor to 2. Uses default NumSections = 2, DifferentialDelay = 1.</p> <pre>CICInterp = dsp.CICInterpolator(2);</pre> <p>Create fixed-point sinusoidal input signal. Original sample rate is 22.05 kHz. The signal has 512 samples and is 0.0113 seconds long.</p> <pre>Fs = 22.05e3; n = (0:511)'; x = fi(sin(2*pi*1e3/Fs*n), true, 16, 15); SigIn = dsp.SignalSource(x, 32);</pre> <p>Process the input signal. The interpolated output has 64 samples per frame.</p> <pre>K = 6; y = zeros(64, K); for jj=1:K     u = SigIn();     y(:, jj) = CICInterp(u); end</pre> |

**References**

[1] Hogenauer, E. B., "An Economical Class of Digital Filters for Decimation and Interpolation," IEEE Transactions on Acoustics, Speech, and Signal Processing, ASSP-29(2): pp. 155-162, 1981

[2] Meyer-Baese, Uwe, "Hogenauer CIC Filters," in Digital Signal Processing with Field Programmable Gate Arrays, Springer, 2001, pp. 155-172

[3] Harris, Fredric J., *Multirate Signal Processing for Communication Systems*, Prentice-Hall PTR, 2004, pp. 343

dsp.CICInterpolator

## mfilt.farrowsrc

(Removed) Sample rate converter with arbitrary conversion factor

### Compatibility

`mfilt.farrowsrc` has been removed. Use `dsp.FarrowRateConverter` instead. For more details, see “Compatibility Considerations” on page 5-1188.

### Syntax

```
hm = mfilt.farrowsrc(L,M,C)
hm = mfilt.farrowsrc
hm = mfilt.farrowsrc(l,...)
```

### Description

`hm = mfilt.farrowsrc(L,M,C)` returns a filter object that is a natural extension of `dfilt.farrowfd` with a time-varying fractional delay. It provides a economical implementation of a sample rate converter with an arbitrary conversion factor. This filter works well in the interpolation case, but may exhibit poor anti-aliasing properties in the decimation case.

---

**Note** You can use the `realizemdl` method to create a Simulink block of a filter created using `mfilt.farrowsrc`.

---

### Input Arguments

The following table describes the input arguments for creating `hm`.

| Input Argument | Description   |
|----------------|---|
| <code>l</code> | Interpolation factor for the filter. <code>l</code> specifies the amount to increase the input sampling rate. The default value of <code>l</code> is 3. |
| <code>m</code> | Decimation factor for the filter. <code>m</code> specifies the amount to decrease the input sampling rate. The default value for <code>m</code> is 2.   |
| <code>c</code> | Coefficients for the filter. When no input arguments are specified, the default coefficients are <code>[-1 1; 1, 0]</code>                              |

`hm = mfilt.farrowsrc` constructs the filter using the default values for `l`, `m`, and `c`.

`hm = mfilt.farrowsrc(l,...)` constructs the filter using the input arguments you provide and defaults for the argument you omit.

### mfilt.farrowsrc Object Properties

Every multirate filter object has properties that govern the way it behaves when you use it. Note that many of the properties are also input arguments for creating `mfilt.farrowsrc` objects. The next table describes each property for an `mfilt.farrowsrc` filter object.

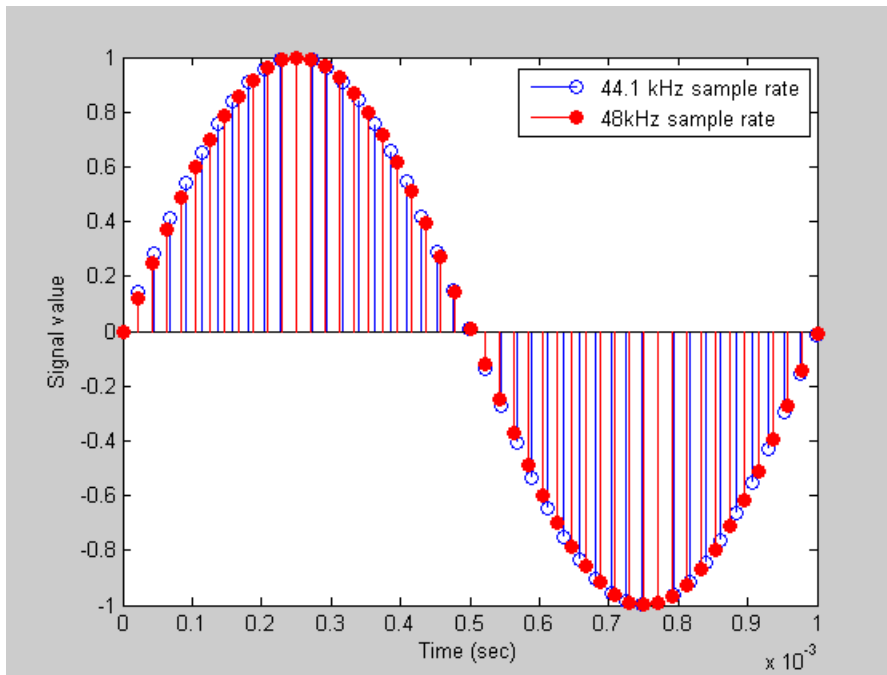
| Name                | Values           | Description  |
|---------------------|------------------|--|
| FilterStructure     | Character vector | Reports the type of filter object. You cannot set this property — it is always read only and results from your choice of <code>mfilt</code> object.  |
| Arithmetic          | Character vector | Reports the arithmetic precision used by the filter.   |
| Coefficients        | Vector           | Vector containing the coefficients of the FIR lowpass filter   |
| InterpolationFactor | Integer          | Interpolation factor for the filter. It specifies the amount to increase the input sampling rate.  |
| DecimationFactor    | Integer          | Decimation factor for the filter. It specifies the amount to increase the input sampling rate.   |
| PersistentMemory    | false or true    | Determines whether the filter states are restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. <code>PersistentMemory</code> returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. |

## Examples

Interpolation by a factor of 8. This object removes the spectral replicas in the signal after interpolation.

```
[L,M] = rat(48/44.1);
Hm = mfilt.farrowsrc(L,M);           % We use the default filter
Fs = 44.1e3;                         % Original sampling frequency
n = 0:9407;                           % 9408 samples, 0.213 seconds long
x = sin(2*pi*1e3/Fs*n);               % Original signal, sinusoid at 1kHz
y = filter(Hm,x);                     % 10241 samples, still 0.213 seconds
stem(n(1:45)/Fs,x(1:45))              % Plot original sampled at 44.1kHz
hold on
% Plot fractionally interpolated signal (48kHz) in red
stem((n(2:50)-1)/(Fs*L/M),y(2:50),'r','filled')
xlabel('Time (sec)');ylabel('Signal value')
legend('44.1 kHz sample rate','48kHz sample rate')
```

The results of the example are shown in the following figure:



## Version History

**Introduced in R2011a**

**R2023a: `mfilt.farrowsrc` has been removed**

*Errors starting in R2023a*

`mfilt.farrowsrc` has been removed. Use `dsp.FarrowRateConverter` instead.

### Update Code

This table shows how to update the existing code to use the `dsp.FarrowRateConverter` object.

| Discouraged Usage   | Recommended Replacement   |
|---|---|
| <p>Initialize the <code>mfilt.farrowsrc</code> object. Set the input and output sample rate to 44.1 kHz and 48.1 kHz, respectively.</p> <pre>[L,M] = rat(48/44.1); hm = mfilt.farrowsrc(L,M)</pre> <p>Pass the data to the object.</p> <pre>% Input sample rate fs = 44.1e3; % 5120 samples, 0.232 second long signal n = (0:5119)'; % Original signal, sinusoid at 1 kHz x = sin(2*pi*1e3/fs*n); y_mfilt = filter(hm,x);</pre> | <p>Initialize the <code>dsp.FarrowRateConverter</code> object. Set the <code>PolynomialOrder</code> property to 1 to have the same configuration as the default <code>mfilt.farrowsrc</code> object.</p> <pre>farrowSRCObj = dsp.FarrowRateConverter(... PolynomialOrder=1);</pre> <p>Pass the data through the object.</p> <p>Note that System objects operate column-wise on data while <code>mfilt</code> objects are agnostic to the dimension. Therefore, pass the data as a column vector to the object.</p> <pre>% Input sample rate fs = 44.1e3; % 5120 samples, 0.232 second long signal n = (0:5119)'; % Original signal, sinusoid at 1 kHz x = sin(2*pi*1e3/fs*n); yObj = farrowSRCObj(x);</pre> <p>Verify the results by comparing the first 50 samples.</p> <pre>plot(y_mfilt,'ro') hold on plot(yObj,'kx') xlim([1 50])</pre> |

## See Also

`dsp.FarrowRateConverter`

## mfilt.fftfirinterp

(Removed) Overlap-add FIR polyphase interpolator

---

**Note** `mfilt.fftfirinterp` has been removed. Use `dsp.FIRInterpolator` instead. For more details, see “Compatibility Considerations”.

---

### Syntax

```
hm = mfilt.fftfirinterp(l,num,bl)
hm = mfilt.fftfirinterp
hm = mfilt.fftfirinterp(l,...)
```

### Description

`hm = mfilt.fftfirinterp(l,num,bl)` returns a discrete-time FIR filter object that uses the overlap-add method for filtering input data.

The input arguments are optional. To enter any optional value, you must include all optional values to the left of your desired value.

When you omit one or more input options, the omitted option applies the default values shown in the table below.

The number of FFT points is given by  $[bl + \text{ceil}(\text{length}(\text{num})/l) - 1]$ . It is to your advantage to choose `bl` such that the number of FFT points is a power of two—using powers of two can improve the efficiency of the FFT and the associated interpolation process.

### Input Arguments

The following table describes the input arguments for creating `hm`.

| Input Argument   | Description   |
|------------------|---|
| <code>l</code>   | Interpolation factor for the filter. <code>l</code> specifies the amount to increase the input sampling rate. It must be an integer. When you do not specify a value for <code>l</code> it defaults to 2.   |
| <code>num</code> | Vector containing the coefficients of the FIR lowpass filter used for interpolation. When <code>num</code> is not provided as an input, <code>fftfirinterp</code> uses a lowpass Nyquist filter with gain equal to <code>l</code> and cutoff frequency equal to $\pi/l$ by default. |
| <code>bl</code>  | Length of each block of input data used in the filtering. <code>bl</code> must be an integer. When you omit input <code>bl</code> , it defaults to 100.   |

`hm = mfilt.fftfirinterp` constructs the filter using the default values for `l`, `num`, and `bl`.

`hm = mfilt.fftfirinterp(l,...)` constructs the filter using the input arguments you provide and defaults for the argument you omit.

## mfilt.fffirinterp Object Properties

Every multirate filter object has properties that govern the way it behaves when you use it. Note that many of the properties are also input arguments for creating `mfilt.fffirinterp` objects. The next table describes each property for an `mfilt.fffirinterp` filter object.

| Name                | Values        | Description  |
|---------------------|---------------|--|
| FilterStructure     |               | Reports the type of filter object. You cannot set this property — it is always read only and results from your choice of <code>mfilt</code> object.  |
| Numerator           |               | Vector containing the coefficients of the FIR lowpass filter used for interpolation.   |
| InterpolationFactor |               | Interpolation factor for the filter. It specifies the amount to increase the input sampling rate. It must be an integer.   |
| BlockLength         |               | Length of each block of input data used in the filtering.  |
| PersistentMemory    | false or true | Determines whether the filter states are restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. <code>PersistentMemory</code> returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. |
| States              |               | Stored conditions for the filter, including values for the interpolator states.  |

## Examples

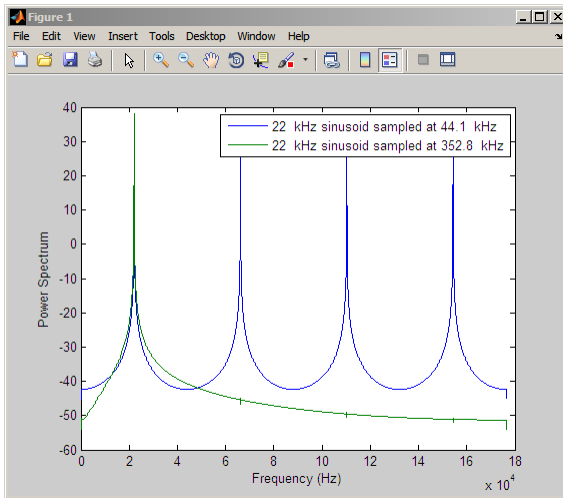
Interpolation by a factor of 8. This object removes the spectral replicas in the signal after interpolation.

```

l = 8; % Interpolation factor
hm = mfilt.fffirinterp(l); % We use the default filter
n = 8192; % Number of points
hm.blocklength = n; % Set block length to number of points
fs = 44.1e3; % Original sample freq: 44.1 kHz.
n = 0:n-1; % 0.1858 secs of data
x = sin(2*pi*n*22e3/fs); % Original signal, sinusoid at 22 kHz
y = filter(hm,x); % Interpolated sinusoid
xu = l*upsample(x,8); % Upsample to compare--the spectrum
% does not change
[px,f]=periodogram(xu,[],65536,l*fs); % Power spectrum of original
% signal
[py,f]=periodogram(y,[],65536,l*fs); % Power spectrum of
% interpolated signal
plot(f,10*log10([fs*px,l*fs*py]))
legend('22 kHz sinusoid sampled at 44.1 kHz',...
'22 kHz sinusoid sampled at 352.8 kHz')
xlabel('Frequency (Hz)'); ylabel('Power Spectrum');

```

To see the results of the example, look at this figure.



## Version History

Introduced in R2011a

**R2022a: `mfilt.fftfirminterp` has been removed**

*Errors starting in R2022a*

`mfilt.fftfirminterp` has been removed. Use `dsp.FIRInterpolator` instead.

### Update Code

This table shows how the `mfilt.fftfirminterp` object is typically used and explains how to update existing code to use the `dsp.FIRInterpolator` object.

| Discouraged Usage   | Recommended Replacement  |
|---|--|
| <p>Initialize the object. Set the interpolation factor to 8.</p> <pre>L = 8; hm = mfilter.fftfirminterp(L);</pre> <p>Create sinusoidal data and pass the data through the object.</p> <pre>% Number of points n = 8192; % Set block length to number of points hm.blocklength = n; % Original sample rate fs = 44.1e3; % 0.1858 secs of data n = (0:n-1)'; % Original signal, sinusoid at 22 kHz x = sin(2*pi*n*22e3/fs); y_mfilter = filter(hm,x);</pre> | <p>Initialize the object. Set the interpolation factor to 8.</p> <pre>L = 8; firI = dsp.FIRInterpolator(L)</pre> <p>Create sinusoidal data and pass the data through the object.</p> <pre>% Number of points n = 8192; % Original sample rate fs = 44.1e3; % 0.1858 secs of data n = (0:n-1)'; % Original signal, sinusoid at 22 kHz x = sin(2*pi*n*22e3/fs); y = firI(x);</pre> |



**See Also**

dsp.FIRInterpolator

## mfilt.firdecim

Direct-form FIR polyphase decimator

---

**Note** `mfilt.firdecim` will be removed in a future release. Use `dsp.FIRDecimator` instead.

---

### Syntax

```
hm = mfilt.firdecim(m)
hm = mfilt.firdecim(m,num)
```

### Description

`hm = mfilt.firdecim(m)` returns a direct-form FIR polyphase decimator object `hm` with a decimation factor of  $m$ . A lowpass Nyquist filter of gain 1 and cutoff frequency of  $\pi/m$  is designed by default. This filter allows some aliasing in the transition band but it very efficient because the first polyphase component is a pure delay.

`hm = mfilt.firdecim(m,num)` uses the coefficients specified by `num` for the decimation filter. This lets you specify more completely the FIR filter to use for the decimator.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hm` as follows:

- To change to single-precision filtering, enter
 

```
set(hm,'arithmetic','single');
```
- To change to fixed-point filtering, enter
 

```
set(hm,'arithmetic','fixed');
```

### Input Arguments

The following table describes the input arguments for creating `hm`.

| Input Argument   | Description   |
|------------------|---|
| <code>m</code>   | Decimation factor for the filter. <code>m</code> specifies the amount to reduce the sampling rate of the input signal. It must be an integer. When you do not specify a value for <code>m</code> it defaults to 2.  |
| <code>num</code> | Vector containing the coefficients of the FIR lowpass filter used for decimation. When <code>num</code> is not provided as an input, <code>mfilt.firdecim</code> constructs a lowpass Nyquist filter with gain of 1 and cutoff frequency equal to $\pi/m$ by default. The default length for the Nyquist filter is $24*m$ . Therefore, each polyphase filter component has length 24. |

## Object Properties

This section describes the properties for both floating-point filters (double-precision and single-precision) and fixed-point filters.

### Floating-Point Filter Properties

Every multirate filter object has properties that govern the way it behaves when you use it. Note that many of the properties are also input arguments for creating `mfilt.firdecim` objects. The next table describes each property for an `mfilt.firdecim` filter object.

| Name             | Values                | Description  |
|------------------|-----------------------|--|
| Arithmetic       | Double, single, fixed | Defines the arithmetic the filter uses. Gives you the options <code>double</code> , <code>single</code> , and <code>fixed</code> . In short, this property defines the operation mode for your filter.   |
| DecimationFactor | Integer               | Decimation factor for the filter. <code>m</code> specifies the amount to reduce the sampling rate of the input signal. It must be an integer.  |
| FilterStructure  | Character vector      | Reports the type of filter object. You cannot set this property — it is always read only and results from your choice of <code>mfilt</code> object. Describes the signal flow for the filter object.   |
| InputOffset      | Integers              | Contains a value derived from the number of input samples and the decimation factor — $\text{InputOffset} = \text{mod}(\text{length}(nx), m)$ where <code>nx</code> is the number of input samples that have been processed so far and <code>m</code> is the decimation factor.  |
| Numerator        | Vector                | Vector containing the coefficients of the FIR lowpass filter used for decimation.  |
| PersistentMemory | false, true           | Determines whether the filter states get restored to zeros for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. <code>PersistentMemory</code> set to <code>false</code> returns filter states to the default values after filtering. States that the filter does not change are not affected. Setting this to <code>true</code> allows you to modify the <code>States</code> , <code>InputOffset</code> , and <code>PolyphaseAccum</code> properties. |

| Name           | Values  | Description  |
|----------------|---|--|
| PolyphaseAccum | 0 in double, single, or fixed for the different filter arithmetic settings. | Differentiates between the adders in the filter that work in full precision at all times (PolyphaseAccum) and the adders in the filter that the user controls and that may introduce quantization effects when FilterInternals is set to SpecifyPrecision. |
| States         | Double, single, or fi matching the filter arithmetic setting.               | This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. Double is the default setting for floating-point filters in double arithmetic.                       |

### Fixed-Point Filter Properties

This table shows the properties associated with the fixed-point implementation of the filter. You see one or more of these properties when you set Arithmetic to fixed. Some of the properties have different default values when they refer fixed point filters. One example is the property PolyphaseAccum which stores data as doubles when you use your filter in double-precision mode, but stores a fi object in fixed-point mode.

**Note** The table lists all of the properties that a fixed-point filter can have. Many of the properties listed are dynamic, meaning they exist only in response to the settings of other properties. To view all of the characteristics for a filter at any time, use `info(hm)` where hm is a filter.

For further information about the properties of this filter or any `mfilt` object, refer to “Multirate Filter Properties” on page 6-2.

| Name            | Values   | Description  |
|-----------------|--|--|
| AccumFracLength | Any positive or negative integer number of bits [32] | Specifies the fraction length used to interpret data output by the accumulator. This is a property of FIR filters.   |
| AccumWordLength | Any integer number of bits [39]                      | Sets the word length used to store data in the accumulator.  |
| Arithmetic      | fixed for fixed-point filters                        | Setting this to fixed allows you to modify other filter properties to customize your fixed-point filter.   |
| CoeffAutoScale  | [true], false  | Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to false enables you to change the NumFracLength property value to specify the precision used. |
| CoeffWordLength | Any integer number of bits [16]                      | Specifies the word length to apply to filter coefficients.   |

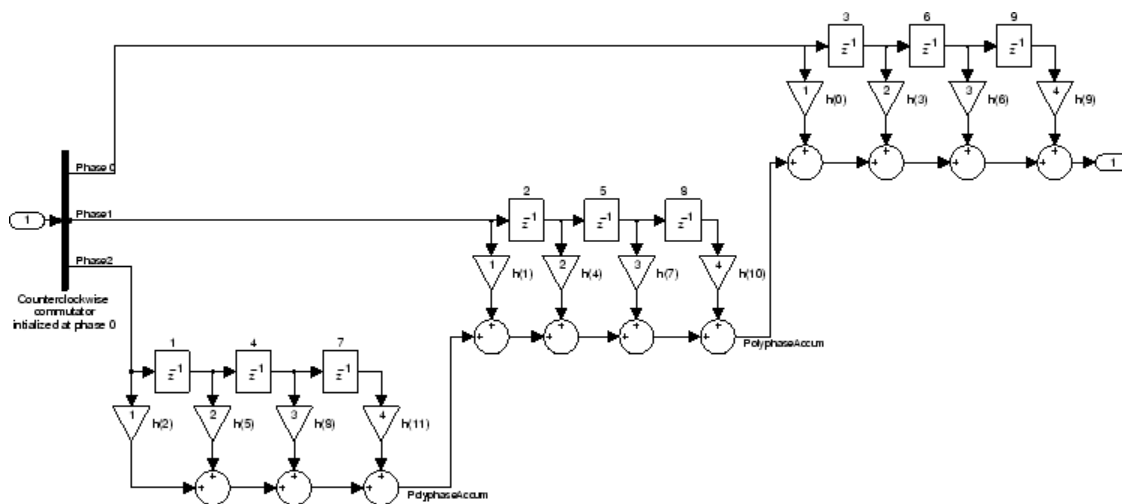
| Name             | Values   | Description  |
|------------------|--|--|
| FilterInternals  | [FullPrecision],<br>SpecifyPrecision                 | Controls whether the filter automatically sets the output word and fraction lengths, product word and fraction lengths, and the accumulator word and fraction lengths to maintain the best precision results during filtering. The default value, FullPrecision, sets automatic word and fraction length determination by the filter. SpecifyPrecision makes the output and accumulator-related properties available so you can set your own word and fraction lengths for them. |
| InputFracLength  | Any positive or negative integer number of bits [15] | Specifies the fraction length the filter uses to interpret input data.   |
| InputWordLength  | Any integer number of bits [16]                      | Specifies the word length applied to interpret input data.   |
| OutputFracLength | Any positive or negative integer number of bits [32] | Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set FilterInternals to SpecifyPrecision.  |
| OutputWordLength | Any integer number of bits [39]                      | Determines the word length used for the output data. You make this property editable by setting FilterInternals to SpecifyPrecision.   |
| OverflowMode     | saturate, [wrap]                                     | Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic.) The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision. |

| Name      | Values   | Description   |
|-----------|--|---|
| RoundMode | [convergent], ceil, fix, floor, nearest, round | <p>Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths).</p> <ul style="list-style-type: none"> <li>• <code>ceil</code> - Round toward positive infinity.</li> <li>• <code>convergent</code> - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.</li> <li>• <code>fix</code> - Round toward zero.</li> <li>• <code>floor</code> - Round toward negative infinity.</li> <li>• <code>nearest</code> - Round toward nearest. Ties round toward positive infinity.</li> <li>• <code>round</code> - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.</li> </ul> <p>The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.</p> |
| Signed    | [true], false                                  | Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.   |
| States    | fi object                                      | This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use <code>fi</code> objects, with the associated properties from those objects. For details, refer to fixed-point objects in Fixed-Point Designer documentation. For information about the ordering of the states, refer to the filter structure section.  |

## Filter Structure

To provide decimation, `mfilt.firdecim` uses the following structure. At the input you see a commutator that operates counterclockwise, moving from position 0 to position 2, position 1, and back to position 0 as input samples enter the filter.

The following figure details the signal flow for the direct form FIR filter implemented by `mfilt.firdecim`.



Notice the order of the states in the filter flow diagram. States 1 through 9 appear in the diagram above each delay element. State 1 applies to the first delay element in phase 2. State 2 applies to the first delay element in phase 1. State 3 applies to the first delay element in phase 0. State 4 applies to the second delay in phase 2, and so on. When you provide the states for the filter as a vector to the `States` property, the above description explains how the filter assigns the states you specify.

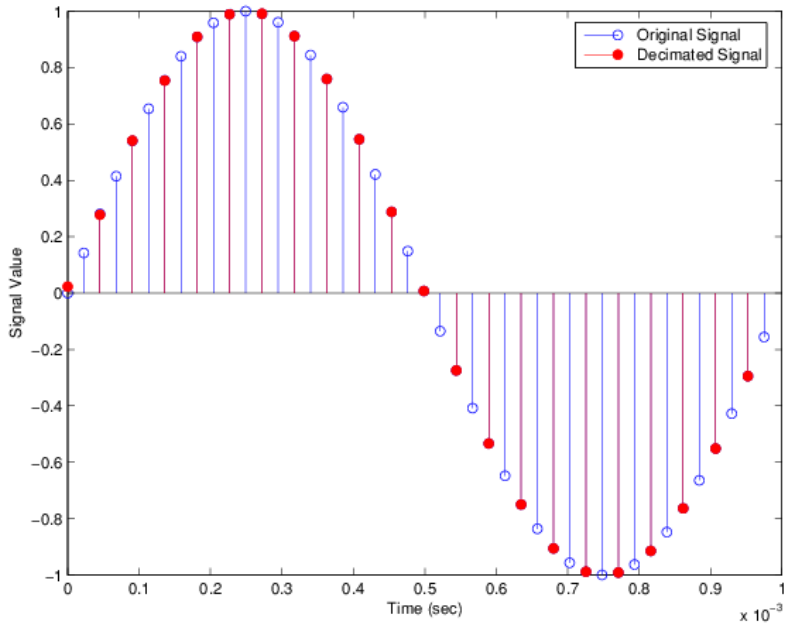
In property value form, the states for a filter `hm` are

```
hm.states=[1:9];
```

## Examples

Convert an input signal from 44.1 kHz to 22.05 kHz using decimation by a factor of 2. In the figure that appears after the example code, you see the results of the decimation.

```
m = 2; % Decimation factor.
hm = mfilter.firdecim(m); % Use the default filter.
fs = 44.1e3; % Original sample freq: 44.1kHz.
n = 0:10239; % 10240 samples, 0.232 second long
% signal.
x = sin(2*pi*1e3/fs*n); % Original signal--sinusoid at 1kHz.
y = filter(hm,x); % 5120 samples, 0.232 seconds.
stem(n(1:44)/fs,x(1:44)) % Plot original sampled at 44.1 kHz.
hold on % Plot decimated signal (22.05 kHz)
% in red.
stem(n(1:22)/(fs/m),y(13:34),'r','filled')
xlabel('Time (sec)');ylabel('Signal Value')
```



## Version History

Introduced in R2011a

### See Also

`mfilt.firtdecim` | `mfilt.firsrc` | `mfilt.cicdecim`



# mfilt.firinterp

(Removed) FIR filter-based interpolator

## Compatibility

`mfilt.firinterp` has been removed. Use `dsp.FIRInterpolator` instead. For more details, see “Compatibility Considerations” on page 5-1206.

## Syntax

```
Hm = mfilt.firinterp(L)
Hm = mfilt.firinterp(L,num)
```

## Description

`Hm = mfilt.firinterp(L)` returns a FIR polyphase interpolator object `Hm` with an interpolation factor of `L` and gain equal to `L`. `L` defaults to 2 if unspecified.

`Hm = mfilt.firinterp(L,num)` uses the values in the vector `num` as the coefficients of the interpolation filter.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `Hm` as follows:

- To change to single-precision filtering, enter
 

```
set(hm,'arithmetic','single');
```
- To change to fixed-point filtering, enter
 

```
set(hm,'arithmetic','fixed');
```

## Input Arguments

The following table describes the input arguments for creating `hm`.

| Input Argument   | Description  |
|------------------|--|
| <code>l</code>   | Interpolation factor for the filter. <code>l</code> specifies the amount to increase the input sampling rate. It must be an integer. When you do not specify a value for <code>l</code> it defaults to 2.  |
| <code>num</code> | Vector containing the coefficients of the FIR lowpass filter used for interpolation. When <code>num</code> is not provided as an input, <code>firinterp</code> uses a lowpass Nyquist filter with gain equal to <code>l</code> and cutoff frequency equal to $\pi/l$ by default. The default length for the Nyquist filter is $24 \cdot l$ . Therefore, each polyphase filter component has length 24. |

## Object Properties

This section describes the properties for both floating-point filters (double-precision and single-precision) and fixed-point filters.

### Floating-Point Filter Properties

Every multirate filter object has properties that govern the way it behaves when you use it. Note that many of the properties are also input arguments for creating `mfilt.firinterp` objects. The next table describes each property for an `mfilt.firinterp` filter object.

| Name                | Values  | Description   |
|---------------------|---|---|
| Arithmetic          | Double, single, fixed                                   | Defines the arithmetic the filter uses. Gives you the options <code>double</code> , <code>single</code> , and <code>fixed</code> . In short, this property defines the operation mode for your filter.  |
| FilterStructure     | Character vector  | Reports the type of filter object. You cannot set this property — it is always read only and results from your choice of <code>mfilt</code> object. Describes the signal flow for the filter object.  |
| InterpolationFactor | Integer   | Interpolation factor for the filter. <code>1</code> specifies the amount to increase the sampling rate of the input signal. It must be an integer.  |
| Numerator           | Vector  | Vector containing the coefficients of the FIR lowpass filter used for decimation.   |
| PersistentMemory    | [false], true   | Determines whether the filter states get restored to zeros for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. <code>PersistentMemory</code> set to <code>false</code> returns filter states to the default values after filtering. States that the filter does not change are not affected. Setting this to <code>true</code> allows you to modify the <code>States</code> property. |
| States              | Double, single, matching the filter arithmetic setting. | Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions.   |

### Fixed-Point Filter Properties

This table shows the properties associated with the fixed-point implementation of the `mfilt.firinterp` filter.

**Note** The table lists all of the properties that a fixed-point filter can have. Many of the properties listed are dynamic, meaning they exist only in response to the settings of other properties. To view all of the characteristics for a filter at any time, use

```
info(hm)
```

where `hm` is a filter.

For further information about the properties of this filter or any `mfilt` object, refer to “Multirate Filter Properties” on page 6-2.

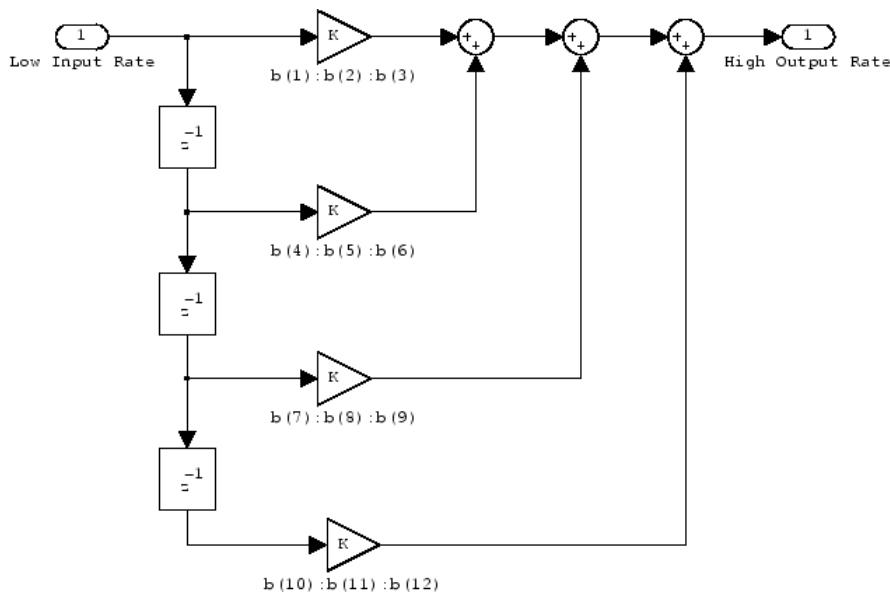
| Name             | Values  | Description   |
|------------------|---|---|
| AccumFracLength  | Any positive or negative integer number of bits. [32] | Specifies the fraction length used to interpret data output by the accumulator. This is a property of FIR filters and lattice filters. IIR filters have two similar properties — <code>DenAccumFracLength</code> and <code>NumAccumFracLength</code> — that let you set the precision for numerator and denominator operations separately.  |
| AccumWordLength  | Any integer number of bits [39]                       | Sets the word length used to store data in the accumulator.   |
| Arithmetic       | fixed for fixed-point filters                         | Setting this to <code>fixed</code> allows you to modify other filter properties to customize your fixed-point filter.   |
| CoeffAutoScale   | [true], false   | Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to <code>false</code> enables you to change the <code>NumFracLength</code> property value to specify the precision used.  |
| CoeffWordLength  | Any integer number of bits [16]                       | Specifies the word length to apply to filter coefficients.  |
| FilterInternals  | [FullPrecision], SpecifyPrecision                     | Controls whether the filter automatically sets the output word and fraction lengths, product word and fraction lengths, and the accumulator word and fraction lengths to maintain the best precision results during filtering. The default value, <code>FullPrecision</code> , sets automatic word and fraction length determination by the filter. <code>SpecifyPrecision</code> makes the output and accumulator-related properties available so you can set your own word and fraction lengths for them. |
| InputFracLength  | Any positive or negative integer number of bits [15]  | Specifies the fraction length the filter uses to interpret input data.  |
| InputWordLength  | Any integer number of bits [16]                       | Specifies the word length applied to interpret input data.  |
| NumFracLength    | Any positive or negative integer number of bits [14]  | Sets the fraction length used to interpret the numerator coefficients.  |
| OutputFracLength | Any positive or negative integer number of bits [32]  | Determines how the filter interprets the filter output data. You can change the value of <code>OutputFracLength</code> when you set <code>FilterInternals</code> to <code>SpecifyPrecision</code> .   |
| OutputWordLength | Any integer number of bits [39]                       | Determines the word length used for the output data. You make this property editable by setting <code>FilterInternals</code> to <code>SpecifyPrecision</code> .   |

| Name         | Values  | Description   |
|--------------|---|---|
| OverflowMode | saturate, [wrap]                                  | Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either <b>saturate</b> (limit the output to the largest positive or negative representable value) or <b>wrap</b> (set overflowing values to the nearest representable value using modular arithmetic.) The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.  |
| RoundMode    | [convergent], ceil, fix, floor, nearest, round    | <p>Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths).</p> <ul style="list-style-type: none"> <li>• <b>ceil</b> - Round toward positive infinity.</li> <li>• <b>convergent</b> - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.</li> <li>• <b>fix</b> - Round toward zero.</li> <li>• <b>floor</b> - Round toward negative infinity.</li> <li>• <b>nearest</b> - Round toward nearest. Ties round toward positive infinity.</li> <li>• <b>round</b> - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.</li> </ul> <p>The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.</p> |
| Signed       | [true], false                                     | Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.   |
| States       | fi object to match the filter arithmetic setting. | Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use <b>fi</b> objects, with the associated properties from those objects. For details, refer to fixed-point objects in Fixed-Point Designer documentation.   |

## Filter Structure

To provide interpolation, `mfilt.firinterp` uses the following structure.

The following figure details the signal flow for the direct form FIR filter implemented by `mfilt.firinterp`. In the figure, the delay line updates happen at the lower input rate. The remainder of the filter — the sums and coefficients — operate at the higher output rate.



## Examples

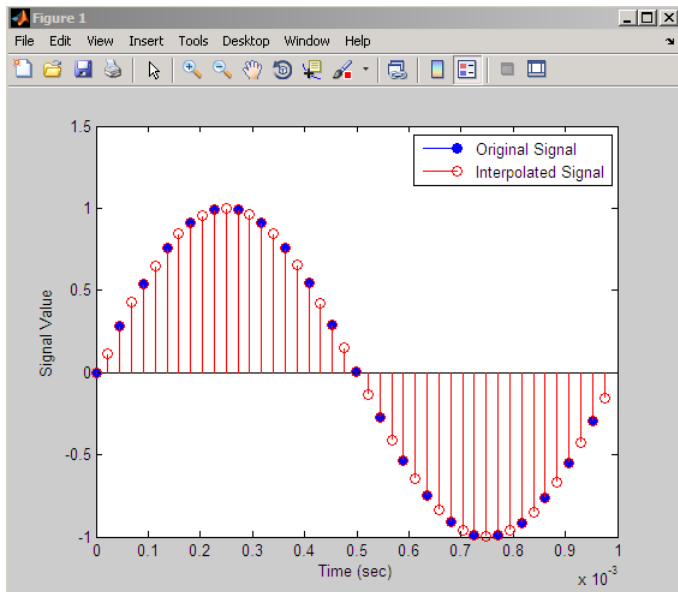
This example uses `mfilt.firinterp` to double the sample rate of a 22.05 kHz input signal. The output signal ends up at 44.1 kHz. Although `l` is set explicitly to 2, this represents the default interpolation value for `mfilt.firinterp` objects.

```
L = 2; % Interpolation factor.
Hm = mfilt.firinterp(L); % Use the default filter.
fs = 22.05e3; % Original sample freq: 22.05 kHz.
n = 0:5119; % 5120 samples, 0.232s long signal.
x = sin(2*pi*1e3/fs*n); % Original signal, sinusoid at 1 kHz.
y = filter(Hm,x); % 10240 samples, still 0.232s.
stem(n(1:22)/fs,x(1:22),'filled') % Plot original sampled at
% 22.05 kHz.

hold on;

% Plot interpolated signal (44.1 kHz) in red
stem(n(1:44)/(fs*L),y(25:68),'r')
xlabel('Time (sec)');ylabel('Signal Value')
legend('Original Signal','Interpolated Signal');
```

With interpolation by 2, the resulting signal perfectly matches the original, but with twice as many samples — one between each original sample, as shown in the following figure.



## Version History

Introduced in R2011a

**R2022a: `mfilt.firinterp` has been removed**

*Errors starting in R2022a*

`mfilt.firinterp` has been removed. Use `dsp.FIRInterpolator` instead.

### Update Code

This table shows how the `mfilt.firinterp` object is typically used and explains how to update existing code to use the `dsp.FIRInterpolator` object.

| Discouraged Usage   | Recommended Replacement  |
|---|--|
| Initialize the object. Set the interpolation factor to 8. <pre>L = 8; hm = mfilt.firinterp(L);</pre>  | Initialize the object. Set the interpolation factor to 8. <pre>L = 8; firI = dsp.FIRInterpolator(L)</pre>  |
| Create sinusoidal data and pass the data through the object. <pre>% Original sample rate is 22.05 kHz. fs = 22.05e3; % 5120 samples, 0.232s long signal. n = (0:5119)'; x = sin(2*pi*1e3/fs*n); y_mfilt = filter(hm,x);</pre> | Create sinusoidal data and pass the data through the object. <pre>% Original sample rate is 22.05 kHz. fs = 22.05e3; % 5120 samples, 0.232s long signal. n = (0:5119)'; x = sin(2*pi*1e3/fs*n); y = firI(x);</pre> |

**See Also**

`dsp.FIRInterpolator`

## mfilt.firsrc

Direct-form FIR polyphase sample rate converter

---

**Note** `mfilt.firsrc` will be removed in a future release. Use `dsp.FIRRateConverter` instead.

---

### Syntax

```
hm = mfilt.firsrc(l,m,num)
```

### Description

`hm = mfilt.firsrc(l,m,num)` returns a direct-form FIR polyphase sample rate converter. `l` specifies the interpolation factor. It must be an integer and when omitted in the calling syntax, it defaults to 2.

`m` is the decimation factor. It must be an integer. If not specified, `m` defaults to 1. If `l` is also not specified, `m` defaults to 3 and the overall rate change factor is 2/3.

You specify the coefficients of the FIR lowpass filter used for sample rate conversion in `num`. If omitted, a lowpass Nyquist filter with gain `l` and cutoff frequency of  $\pi/\max(l,m)$  is the default.

Combining an interpolation factor and a decimation factor lets you use `mfilt.firsrc` to perform fractional interpolation or decimation on an input signal. Using an `mfilt.firsrc` object applies a rate change factor defined by  $l/m$  to the input signal. For proper rate changing to occur, `l` and `m` must be relatively prime — meaning the ratio  $l/m$  cannot be reduced to a ratio of smaller integers.

When you are doing sample-rate conversion with large values of `l` or `m`, such as `l` or `m` greater than 20, using the `mfilt.firsrc` structure is the most effective approach.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hm` as follows:

- To change to single-precision filtering, enter

```
set(hm,'arithmetic','single');
```
- To change to fixed-point filtering, enter

```
set(hm,'arithmetic','fixed');
```

---

**Note** You can use the `realizemdl` method to create a Simulink block of a filter created using `mfilt.firsrc`.

---

### Input Arguments

The following table describes the input arguments for creating `hm`.



| Input Argument   | Description   |
|------------------|---|
| <code>l</code>   | Interpolation factor for the filter. <code>l</code> specifies the amount to increase the input sampling rate. It must be an integer. When you do not specify a value for <code>l</code> , it defaults to 2.   |
| <code>num</code> | Vector containing the coefficients of the FIR lowpass filter used for interpolation. When <code>num</code> is not provided as an input, <code>mfilt.firsrc</code> uses a lowpass Nyquist filter with gain equal to <code>l</code> and cutoff frequency equal to $\pi/\max(l, m)$ by default. The default length for the Nyquist filter is $24*\max(1, m)$ . Therefore, each polyphase filter component has length 24. |
| <code>m</code>   | Decimation factor for the filter. <code>m</code> specifies the amount to reduce the sampling rate of the input signal. It must be an integer. When you do not specify a value for <code>m</code> , it defaults to 1. When <code>l</code> is unspecified as well, <code>m</code> defaults to 3.  |

## Object Properties

This section describes the properties for both floating-point filters (double-precision and single-precision) and fixed-point filters.

### Floating-Point Filter Properties

Every multirate filter object has properties that govern the way it behaves when you use it. Note that many of the properties are also input arguments for creating `mfilt.firsrc` objects. The next table describes each property for an `mfilt.firsrc` filter object.

| Name             | Values                  | Description  |
|------------------|-------------------------|--|
| Arithmetic       | [Double], single, fixed | Defines the arithmetic the filter uses. Gives you the options <code>double</code> , <code>single</code> , and <code>fixed</code> . In short, this property defines the operation mode for your filter.   |
| FilterStructure  | Character vector        | Reports the type of filter object. You cannot set this property — it is always read only and results from your choice of <code>mfilt</code> object. Describes the signal flow for the filter object.   |
| InputOffset      | Integers                | Contains a value derived from the number of input samples and the decimation factor — $\text{InputOffset} = \text{mod}(\text{length}(nx), m)$ where <code>nx</code> is the number of input samples and <code>m</code> is the decimation factor.  |
| Numerator        | Vector                  | Vector containing the coefficients of the FIR lowpass filter used for decimation.  |
| PersistentMemory | false, true             | Determines whether the filter states get restored to zeros for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. <code>PersistentMemory</code> set to <code>false</code> returns filter states to the default values after filtering. States that the filter does not change are not affected. Setting this to <code>true</code> allows you to modify the <code>States</code> , <code>InputOffset</code> , and <code>PolyphaseAccum</code> properties. |

| Name              | Values  | Description  |
|-------------------|---|--|
| RateChangeFactors | Positive integers. [2 3]                                | Specifies the interpolation and decimation factors [l m] (the rate change factors ) for changing the input sample rate by nonintegral amounts. |
| States            | Double, single, matching the filter arithmetic setting. | Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions.        |

### Fixed-Point Filter Properties

This table shows the properties associated with the fixed-point implementation of the `mfilt.firsrc` filter.

**Note** The table lists all of the properties that a fixed-point filter can have. Many of the properties listed are dynamic, meaning they exist only in response to the settings of other properties. To view all of the characteristics for a filter at any time, use

`info(hm)`

where `hm` is a filter.

For further information about the properties of this filter or any `mfilt` object, refer to “Multirate Filter Properties” on page 6-2.

| Name            | Values  | Description  |
|-----------------|---|--|
| AccumFracLength | Any positive or negative integer number of bits. [32] | Specifies the fraction length used to interpret data output by the accumulator. This is a property of FIR filters.   |
| AccumWordLength | Any integer number of bits [39]                       | Sets the word length used to store data in the accumulator.  |
| Arithmetic      | fixed for fixed-point filters                         | Setting this to <code>fixed</code> allows you to modify other filter properties to customize your fixed-point filter.  |
| CoeffAutoScale  | [true], false   | Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to <code>false</code> enables you to change the <code>NumFracLength</code> property value to specify the precision used. |
| CoeffWordLength | Any integer number of bits [16]                       | Specifies the word length to apply to filter coefficients.   |

| Name              | Values   | Description   |
|-------------------|--|---|
| FilterInternals   | [FullPrecision],<br>SpecifyPrecision                 | Controls whether the filter automatically sets the output word and fraction lengths, product word and fraction lengths, and the accumulator word and fraction lengths to maintain the best precision results during filtering. The default value, <code>FullPrecision</code> , sets automatic word and fraction length determination by the filter. <code>SpecifyPrecision</code> makes the output and accumulator-related properties available so you can set your own word and fraction lengths for them. |
| InputFracLength   | Any positive or negative integer number of bits [15] | Specifies the fraction length the filter uses to interpret input data.  |
| InputWordLength   | Any integer number of bits [16]                      | Specifies the word length applied to interpret input data.  |
| NumFracLength     | Any positive or negative integer number of bits [14] | Sets the fraction length used to interpret the numerator coefficients.  |
| OutputFracLength  | Any positive or negative integer number of bits [32] | Determines how the filter interprets the filter output data. You can change the value of <code>OutputFracLength</code> when you set <code>FilterInternals</code> to <code>SpecifyPrecision</code> .   |
| OutputWordLength  | Any integer number of bits [39]                      | Determines the word length used for the output data. You make this property editable by setting <code>FilterInternals</code> to <code>SpecifyPrecision</code> .   |
| OverflowMode      | saturate, [wrap]                                     | Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either <code>saturate</code> (limit the output to the largest positive or negative representable value) or <code>wrap</code> (set overflowing values to the nearest representable value using modular arithmetic.) The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.  |
| RateChangeFactors | Positive integers [2 3]                              | Specifies the interpolation and decimation factors [l m] (the rate change factors) for changing the input sample rate by nonintegral amounts.   |

| Name      | Values   | Description   |
|-----------|--|---|
| RoundMode | [convergent], ceil, fix, floor, nearest, round | <p>Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths).</p> <ul style="list-style-type: none"> <li>• <b>ceil</b> - Round toward positive infinity.</li> <li>• <b>convergent</b> - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.</li> <li>• <b>fix</b> - Round toward zero.</li> <li>• <b>floor</b> - Round toward negative infinity.</li> <li>• <b>nearest</b> - Round toward nearest. Ties round toward positive infinity.</li> <li>• <b>round</b> - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.</li> </ul> <p>The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.</p> |
| Signed    | [true], false                                  | Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.   |
| States    | fi object                                      | Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use <code>fi</code> objects, with the associated properties from those objects. For details, refer to fixed-point objects in Fixed-Point Designer documentation. For information about the ordering of the states, refer to the filter structure section.  |

## Examples

This is an example of a common audio rate change process — changing the sample rate of a high end audio (48 kHz) signal to the compact disc sample rate (44.1 kHz). This conversion requires a rate change factor of 0.91875, or  $l = 147$  and  $m = 160$ .

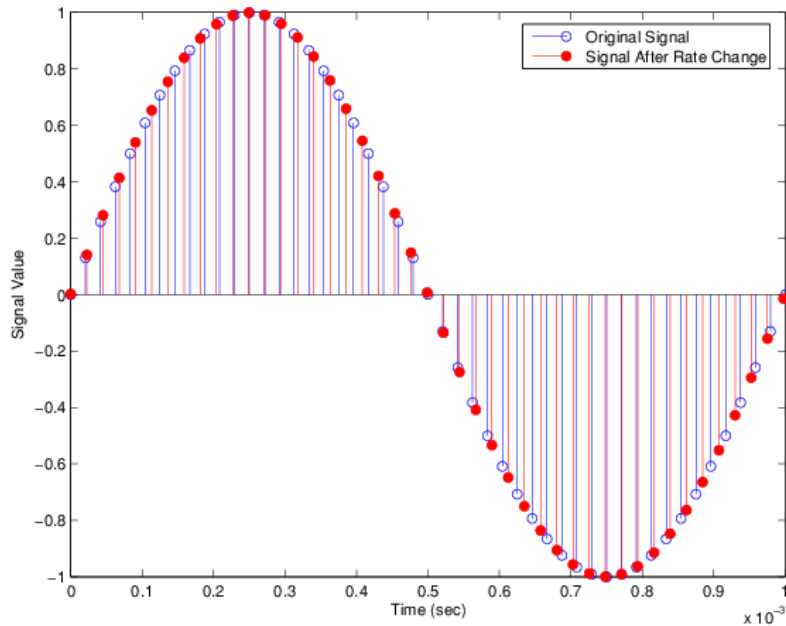
```

l = 147; m = 160;           % Interpolation/decimation factors.
hm = mfilt.firsrc(l,m);    % Use the default FIR filter.
fs = 48e3;                 % Original sample freq: 48 kHz.
n = 0:10239;               % 10240 samples, 0.213 seconds long.
x = sin(2*pi*1e3/fs*n);    % Original signal, sinusoid at 1 kHz.
y = filter(hm,x);          % 9408 samples, still 0.213 seconds.
stem(n(1:49)/fs,x(1:49))  % Plot original sampled at 48 kHz.
hold on

% Plot fractionally decimated signal (44.1 kHz) in red
stem(n(1:45)/(fs*l/m),y(13:57),'r','filled')
xlabel('Time (sec)');ylabel('Signal Value')

```

Fractional decimation provides you the flexibility to pick and choose the sample rates you want by carefully selecting  $l$  and  $m$ , the interpolation and decimation factors, that result in the final fractional decimation. The following figure shows the signal after applying the rate change filter `hm` to the original signal.



## Version History

Introduced in R2011a

## See Also

`mfilt.firinterp` | `mfilt.firdecim`

## mfilt.firtdecim

Direct-form transposed FIR filter

---

**Note** `mfilt.firtdecim` will be removed in a future release. Use `dsp.FIRDecimator` instead.

---

### Syntax

```
hm = mfilt.firtdecim(m)
hm = mfilt.firtdecim(m,num)
```

### Description

`hm = mfilt.firtdecim(m)` returns a polyphase decimator `mfilt` object `hm` based on a direct-form transposed FIR structure with a decimation factor of  $m$ . A lowpass Nyquist filter of gain 1 and cutoff frequency of  $\pi/m$  is the default.

`hm = mfilt.firtdecim(m,num)` uses the coefficients specified by `num` for the decimation filter. `num` is a vector containing the coefficients of the transposed FIR lowpass filter used for decimation. If omitted, a lowpass Nyquist filter with gain of 1 and cutoff frequency of  $\pi/m$  is the default.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hm` as follows:

- To change to single-precision filtering, enter
 

```
set(hm,'arithmetic','single');
```
- To change to fixed-point filtering, enter
 

```
set(hm,'arithmetic','fixed');
```

### Input Arguments

The following table describes the input arguments for creating `hm`.

| Input Argument   | Description   |
|------------------|---|
| <code>num</code> | Vector containing the coefficients of the FIR lowpass filter used for interpolation. When <code>num</code> is not provided as an input, <code>firtdecim</code> uses a lowpass Nyquist filter with gain equal to 1 and cutoff frequency equal to $\pi/m$ by default. The default length for the Nyquist filter is $24*m$ . Therefore, each polyphase filter component has length 24. |
| <code>m</code>   | Decimation factor for the filter. <code>m</code> specifies the amount to reduce the sampling rate of the input signal. It must be an integer. When you do not specify a value for <code>m</code> it defaults to 2.  |

## Object Properties

This section describes the properties for both floating-point filters (double-precision and single-precision) and fixed-point filters.

### Floating-Point Filter Properties

Every multirate filter object has properties that govern the way it behaves when you use it. Note that many of the properties are also input arguments for creating `mfilt.firtdecim` objects. The next table describes each property for an `mfilt.firtdecim` filter object.

| Name             | Values   | Description  |
|------------------|--|--|
| Arithmetic       | Double, single, fixed                                  | Specifies the arithmetic the filter uses to process data while filtering.  |
| DecimationFactor | Integer  | Decimation factor for the filter. <code>m</code> specifies the amount to reduce the sampling rate of the input signal. It must be an integer.  |
| FilterStructure  | Character vector                                       | Reports the type of filter object. You cannot set this property — it is always read only and results from your choice of <code>mfilt</code> object. Also describes the signal flow for the filter object.  |
| InputOffset      | Integers   | Contains a value derived from the number of input samples and the decimation factor — $\text{InputOffset} = \text{mod}(\text{length}(nx), m)$ where <code>nx</code> is the number of input samples that have been processed so far and <code>m</code> is the decimation factor.  |
| Numerator        | Vector   | Vector containing the coefficients of the FIR lowpass filter used for decimation.  |
| PersistentMemory | [false], true  | Determines whether the filter states get restored to zeros for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. <code>PersistentMemory</code> set to <code>false</code> returns filter states to the default values after filtering. States that the filter does not change are not affected. Setting this to <code>true</code> allows you to modify the <code>States</code> , <code>InputOffset</code> , and <code>PolyphaseAccum</code> properties. |
| PolyphaseAccum   | Double, single [0]                                     | The idea behind having both <code>PolyphaseAccum</code> and <code>Accum</code> is to differentiate between the adders in the filter that work in full precision at all times ( <code>PolyphaseAccum</code> ) from the adders in the filter that the user controls and that may introduce quantization effects when <code>FilterInternals</code> is set to <code>SpecifyPrecision</code> .  |
| States           | Double, single matching the filter arithmetic setting. | Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions.  |

### Fixed-Point Filter Properties

This table shows the properties associated with the fixed-point implementation of the `mfilt.firtdecim` filter.

**Note** The table lists all of the properties that a fixed-point filter can have. Many of the properties listed are dynamic, meaning they exist only in response to the settings of other properties. To view all of the characteristics for a filter at any time, use

```
info(hm)
```

where `hm` is a filter.

For further information about the properties of this filter or any `mfilt` object, refer to “Multirate Filter Properties” on page 6-2.

| Name            | Values  | Description   |
|-----------------|---|---|
| AccumFracLength | Any positive or negative integer number of bits. [32] | Specifies the fraction length used to interpret data output by the accumulator. This is a property of FIR filters and lattice filters. IIR filters have two similar properties — <code>DenAccumFracLength</code> and <code>NumAccumFracLength</code> — that let you set the precision for numerator and denominator operations separately.  |
| AccumWordLength | Any integer number of bits [39]                       | Sets the word length used to store data in the accumulator.   |
| Arithmetic      | fixed for fixed-point filters                         | Setting this to <code>fixed</code> allows you to modify other filter properties to customize your fixed-point filter.   |
| CoeffAutoScale  | [true], false   | Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to <code>false</code> enables you to change the <code>NumFracLength</code> property value to specify the precision used.  |
| CoeffWordLength | Any integer number of bits [16]                       | Specifies the word length to apply to filter coefficients.  |
| FilterInternals | [FullPrecision], SpecifyPrecision                     | Controls whether the filter automatically sets the output word and fraction lengths, product word and fraction lengths, and the accumulator word and fraction lengths to maintain the best precision results during filtering. The default value, <code>FullPrecision</code> , sets automatic word and fraction length determination by the filter. <code>SpecifyPrecision</code> makes the output and accumulator-related properties available so you can set your own word and fraction lengths for them. |
| InputFracLength | Any positive or negative integer number of bits [15]  | Specifies the fraction length the filter uses to interpret input data.  |
| InputWordLength | Any integer number of bits [16]                       | Specifies the word length applied to interpret input data.  |



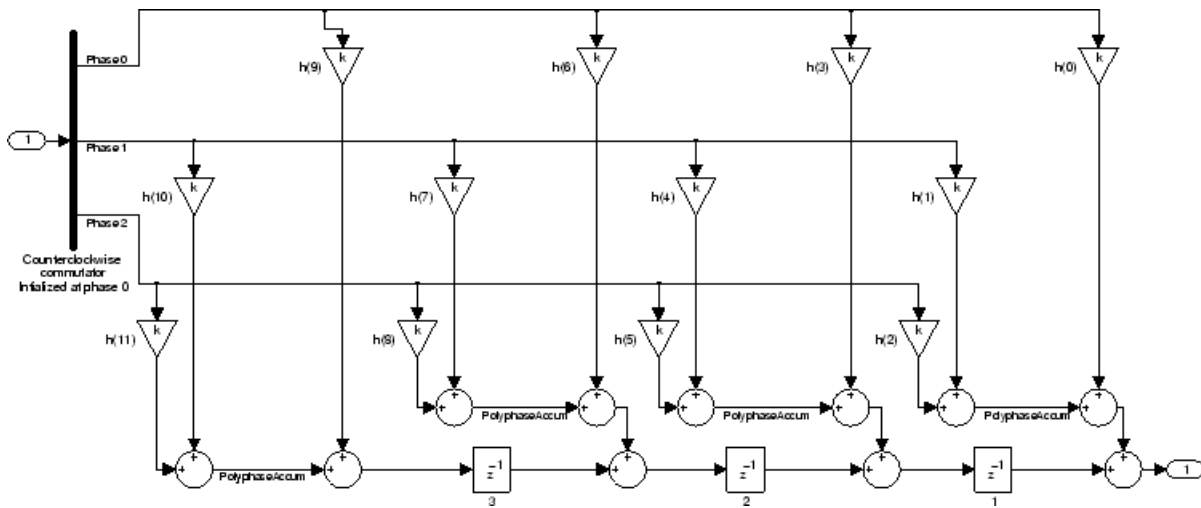
| Name             | Values   | Description   |
|------------------|--|---|
| NumFracLength    | Any positive or negative integer number of bits [14] | Sets the fraction length used to interpret the numerator coefficients.  |
| OutputFracLength | Any positive or negative integer number of bits [32] | Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set FilterInternals to SpecifyPrecision.   |
| OutputWordLength | Any integer number of bits [39]                      | Determines the word length used for the output data. You make this property editable by setting FilterInternals to SpecifyPrecision.  |
| OverflowMode     | saturate, [wrap]                                     | Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic.) The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.  |
| PolyphaseAccum   | fi object with zeros to start                        | Differentiates between the adders in the filter that work in full precision at all times (PolyphaseAccum) and the adders in the filter that the user controls and that may introduce quantization effects when FilterInternals is set to SpecifyPrecision.  |
| RoundMode        | [convergent],<br>ceil, fix, floor,<br>nearest, round | <p>Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths).</p> <ul style="list-style-type: none"> <li>• <code>ceil</code> - Round toward positive infinity.</li> <li>• <code>convergent</code> - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.</li> <li>• <code>fix</code> - Round toward zero.</li> <li>• <code>floor</code> - Round toward negative infinity.</li> <li>• <code>nearest</code> - Round toward nearest. Ties round toward positive infinity.</li> <li>• <code>round</code> - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.</li> </ul> <p>The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.</p> |

| Name   | Values        | Description   |
|--------|---------------|---|
| Signed | [true], false | Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.   |
| States | fi object     | Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use fi objects, with the associated properties from those objects. For details, refer to fixed-point objects in Fixed-Point Designer documentation. For information about the ordering of the states, refer to the filter structure section. |

### Filter Structure

To provide sample rate changes, `mfilt.firtddecim` uses the following structure. At the input you see a commutator that operates counterclockwise, moving from position 0 to position 2, position 1, and back to position 0 as input samples enter the filter. To keep track of the position of the commutator, the `mfilt` object uses the property `InputOffset` which reports the current position of the commutator in the filter.

The following figure details the signal flow for the direct form FIR filter implemented by `mfilt.firtddecim`.



Notice the order of the states in the filter flow diagram. States 1 through 3 appear in the following diagram at each delay element. State 1 applies to the third delay element in phase 2. State 2 applies to the second delay element in phase 2. State 3 applies to the first delay element in phase 2. When you provide the states for the filter as a vector to the `States` property, the above description explains how the filter assigns the states you specify.

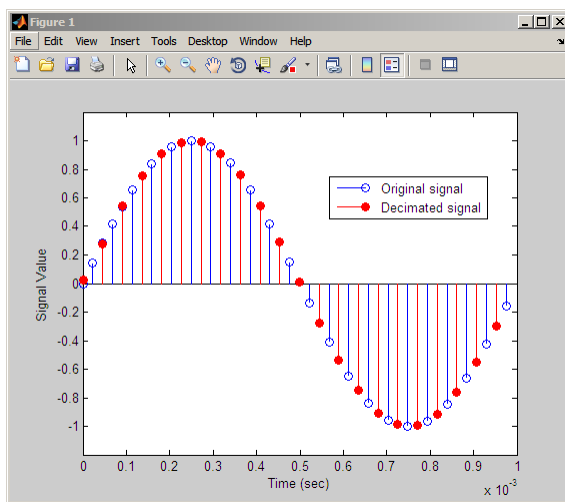
In property value form, the states for a filter `hm` are

```
hm.states=[1:3];
```

## Examples

Demonstrate decimating an input signal by a factor of 2, in this case converting from 44.1 kHz down to 22.05 kHz. In the figure shown following the code, you see the results of decimating the signal.

```
m = 2; % Decimation factor.
hm = mfilter.firtdecim(m); % Use the default filter coeffs.
fs = 44.1e3; % Original sample freq: 44.1 kHz.
n = 0:10239; % 10240 samples, 0.232 second long signal
x = sin(2*pi*1e3/fs*n); % Original signal--sinusoid at 1 kHz.
y = filter(hm,x); % 5120 samples, 0.232 seconds.
stem(n(1:44)/fs,x(1:44)) % Plot original sampled at 44.1 kHz.
axis([0 0.001 -1.2 1.2]);
hold on % Plot decimated signal (22.05 kHz) in red
stem(n(1:22)/(fs/m),y(13:34),'r','filled')
xlabel('Time (sec)');ylabel('Signal Value');
legend('Original signal','Decimated signal','location','best');
```



## Version History

Introduced in R2011a

## See Also

[mfilt.firdecim](#) | [mfilt.firsrc](#) | [mfilt.cicdecim](#)

## mfilt.holdinterp

(Removed) FIR hold interpolator

---

**Note** `mfilt.holdinterp` has been removed. Use `dsp.FIRInterpolator(L,'ZOH')` instead. For more details, see “Compatibility Considerations”.

---

### Syntax

```
hm = mfilt.holdinterp(l)
```

### Description

`hm = mfilt.holdinterp(l)` returns the object `hm` that represents a hold interpolator with the interpolation factor `l`. To work, `l` must be an integer. When you do not include `l` in the calling syntax, it defaults to 2. To perform interpolation by noninteger amounts, use one of the fractional interpolator objects, such as `mfilt.firsrc`.

When you use this hold interpolator, each sample added to the input signal between existing samples has the value of the most recent sample from the original signal. Thus you see something like a staircase profile where the interpolated samples form a plateau between the previous and next original samples. The example demonstrates this profile clearly. Compare this to the interpolation process for other interpolators in the toolbox, such as `mfilt.linearinterp`.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hm` as follows:

- To change to single-precision filtering, enter
 

```
set(hm,'arithmetic','single');
```
- To change to fixed-point filtering, enter
 

```
set(hm,'arithmetic','fixed');
```

### Input Arguments

The following table describes the input arguments for creating `hm`.

| Input Argument | Description   |
|----------------|---|
| <code>l</code> | Interpolation factor for the filter. <code>l</code> specifies the amount to increase the input sampling rate. It must be an integer. When you do not specify a value for <code>l</code> it defaults to 2. |

### Object Properties

This section describes the properties for both floating-point filters (double-precision and single-precision) and fixed-point filters.

### Floating-Point Filter Properties

Every multirate filter object has properties that govern the way it behaves when you use it. Note that many of the properties are also input arguments for creating `mfilt.holdinterp` objects. The next table describes each property for an `mfilt.interp` filter object.

| Name                | Values                 | Description   |
|---------------------|------------------------|---|
| Arithmetic          | Double, single, fixed  | Specifies the arithmetic the filter uses to process data while filtering.   |
| FilterStructure     | Character vector       | Reports the type of filter object. You cannot set this property — it is always read only and results from your choice of <code>mfilt</code> object.   |
| InterpolationFactor | Integer                | Interpolation factor for the filter. <code>l</code> specifies the amount to increase the input sampling rate. It must be an integer.  |
| PersistentMemory    | 'false' or 'true'      | Determines whether the filter states are restored to zero for each filtering operation.   |
| States              | Double or single array | Filter states. <code>states</code> defaults to a vector of zeros that has length equal to <code>nstates</code> ( <code>hm</code> ). Always available, but visible in the display only when <code>PersistentMemory</code> is true. |

### Fixed-Point Filter Properties

This table shows the properties associated with the fixed-point implementation of the `mfilt.holdinterp` filter.

---

**Note** The table lists all of the properties that a fixed-point filter can have. Many of the properties listed are dynamic, meaning they exist only in response to the settings of other properties. To view all of the characteristics for a filter at any time, use

```
info(hm)
```

where `hm` is a filter.

---

For further information about the properties of this filter or any `mfilt` object, refer to “Multirate Filter Properties” on page 6-2.

| Name       | Values                | Description   |
|------------|-----------------------|---|
| Arithmetic | Double, single, fixed | Specifies the arithmetic the filter uses to process data while filtering. |

| Name                | Values   | Description  |
|---------------------|--|--|
| FilterStructure     | Character vector                                     | Reports the type of filter object. You cannot set this property — it is always read only and results from your choice of <code>mfilt</code> object.  |
| InputFracLength     | Any positive or negative integer number of bits [15] | Specifies the fraction length the filter uses to interpret input data.   |
| InputWordLength     | Any integer number of bits [16]                      | Specifies the word length applied to interpret input data.   |
| InterpolationFactor | Integer  | Interpolation factor for the filter. <code>l</code> specifies the amount to increase the input sampling rate. It must be an integer.   |
| PersistentMemory    | 'false' or 'true'                                    | Determine whether the filter states get restored to zero for each filtering operation  |
| States              | <code>fi</code> object                               | Contains the filter states before, during, and after filter operations. For hold interpolators, the states are always empty — hold interpolators do not have states. The states use <code>fi</code> objects, with the associated properties from those objects. For details, refer to fixed-point objects in Fixed-Point Designer documentation. |

## Filter Structure

Hold interpolators do not have filter coefficients and their filter structure is trivial.

## Examples

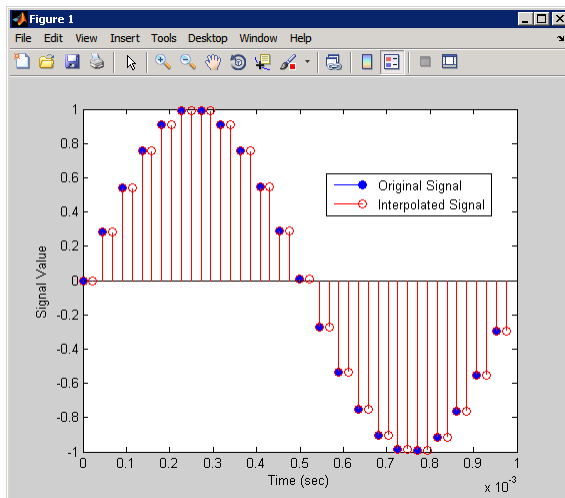
To see the effects of hold-based interpolation, interpolate an input sine wave from 22.05 to 44.1 kHz. Note that each added sample retains the value of the most recent original sample.

```

l = 2; % Interpolation factor
hm = mfilt.holdinterp(l);
fs = 22.05e3; % Original sample freq: 22.05 kHz.
n = 0:5119; % 5120 samples, 0.232 second long signal
x = sin(2*pi*1e3/fs*n); % Original signal, sinusoid at 1 kHz
y = filter(hm,x); % 10240 samples, still 0.232 seconds
stem(n(1:22)/fs,x(1:22),'filled') % Plot original sampled at
% 22.05 kHz
hold on % Plot interpolated signal (44.1 kHz)
stem(n(1:44)/(fs*l),y(1:44),'r')
legend('Original Signal','Interpolated Signal','Location','best');
xlabel('Time (sec)');ylabel('Signal Value')

```

The following figure shows clearly the step nature of the signal that comes from interpolating the signal using the hold algorithm approach. Compare the output to the linear interpolation used in `mfilt.linearinterp`.



## Version History

### Introduced in R2011a

### R2022a: `mfilt.holdinterp` has been removed

Errors starting in R2022a

`mfilt.holdinterp` has been removed. Use `dsp.FIRInterpolator(L, 'ZOH')` instead.

### Update Code

This table shows how the `mfilt.holdinterp` object is typically used and explains how to update existing code to use the `dsp.FIRInterpolator` object.

| Discouraged Usage   | Recommended Replacement   |
|---|---|
| <p>Initialize the object. Set the interpolation factor to 2.</p> <pre>L = 2; hm = mfilt.holdinterp(L);</pre> <p>Create a sinusoidal data and pass the data through the object.</p> <pre>% Original sample rate is 22.05 kHz fs = 22.05e3; % 5120 samples, 0.232 second long signal n = (0:5119)'; % Original signal, sinusoid at 1 kHz x = sin(2*pi*1e3/fs*n); % 10240 samples, still 0.232 seconds y_mfilt = filter(hm,x);</pre> | <p>Initialize the object. Set the interpolation factor to 2.</p> <pre>L = 2; firI = dsp.FIRInterpolator(L, 'ZOH')</pre> <p>Create a sinusoidal data and pass the data through the object.</p> <pre>% Original sample rate is 22.05 kHz fs = 22.05e3; % 5120 samples, 0.232 second long signal n = (0:5119)'; % Original signal, sinusoid at 1 kHz x = sin(2*pi*1e3/fs*n); % 10240 samples, still 0.232 seconds y = firI(x);</pre> |

## See Also

`dsp.FIRInterpolator`

## mfilt.iirdecim

IIR decimator

### Compatibility

---

**Note** `mfilt.iirdecim` will be removed in a future release. Use `dsp.IIRHalfbandDecimator` instead.

---

### Syntax

```
hm = mfilt.iirdecim(c1,c2,...)
```

### Description

`hm = mfilt.iirdecim(c1,c2,...)` constructs an IIR decimator filter given the coefficients specified in the cell arrays `c1`, `c2`, and so on. The resulting IIR decimator is a polyphase IIR filter where each phase is a cascade allpass IIR filter.

Each cell array `ci` contains a set of vectors representing a cascade of allpass sections. Each element in one cell array is one section. For more information about the contents of each cell array, refer to `dfilt.cascadeallpass`. The contents of the cell arrays are the same for both filter constructors and `mfilt.iirdecim` interprets them same way as `mfilt.cascadeallpass`.

The following exception applies to interpreting the contents of a cell array — if one of the cell arrays `ci` contains only one vector, and that vector comprises a series of 0s and one element equal to 1, that cell array represents a `dfilt.delay` section with latency equal to the number of zeros, rather than a `dfilt.cascadeallpass` section. This exception case occurs with quasi-linear phase IIR decimators.

Although the first example shows how to construct an IIR decimators explicitly, one usually constructs an IIR decimators filter as a result of designing an decimators, as shown in the subsequent examples.

### Examples

When the coefficients are known, you can construct the IIR decimator directly using `mfilt.iirdecim`. For example, if the filter's coefficients are [0.6 0.5] for the first phase in the first stage, 0.7 for the second phase in the first stage and 0.8 for the third phase in the first stage; as well as 0.5 for the first phase in the second stage and 0.4 for the second phase in the second stage, construct the filter as shown here.

```
Hm = mfilt.iirdecim({[0.6 0.5] 0.7 0.8},{0.5 0.4})
```

Also refer to the “Quasi-Linear Phase Halfband and Dyadic Halfband Designs” section of the “IIR Polyphase Filter Design” example, “IIR Halfband Stages in Multistage Filter Design” example.

When the coefficients are not known, use the approach given by the following set of examples. Start by designing an elliptic halfband decimator with a decimation factor of 2. The example specifies the optional sampling frequency argument.



```

tw = 100; % Transition width of filter.
ast = 80; % Stopband attenuation of filter.
fs = 2000; % Sampling frequency of signal to filter.
m = 2; % Decimation factor.
hm = fdesign.decimator(m,'halfband','tw,ast',tw,ast,fs);

```

hm contains the specifications for a decimator defined by tw, ast, m, and fs.

Use the specification object hm to design a `mfilt.iirdecim` filter object.

```

d = design(hm,'ellip','filterstructure','iirdecim');
% Note that realizemdl requires Simulink
realizemdl(d) % Build model of the filter.

```

Designing a linear phase decimator is similar to the previous example. In this case, design a halfband linear phase decimator with decimation factor of 2.

```

tw = 100; % Transition width of filter.
ast = 60; % Stopband attenuation of filter.
fs = 2000; % Sampling frequency of signal to filter.
m = 2; % Decimation factor.

```

Create a specification object for the decimator.

```

hm = fdesign.decimator(m,'halfband','tw,ast',tw,ast,fs);

```

Finally, design the filter d.

```

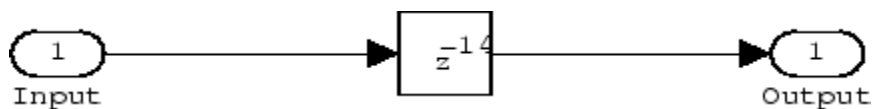
d = design(hm,'iirlinphase','filterstructure','iirdecim');
% Note that realizemdl requires Simulink
realizemdl(d) % Build model of the filter.

```

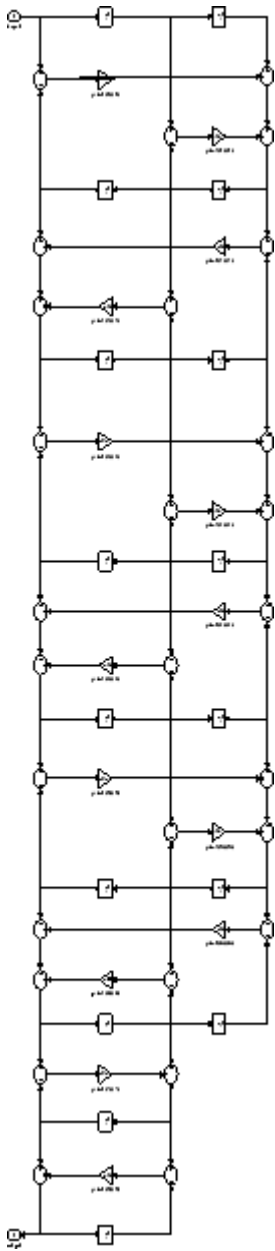
The filter implementation appears in this model, generated by `realizemdl` and Simulink.

Given the design specifications shown here

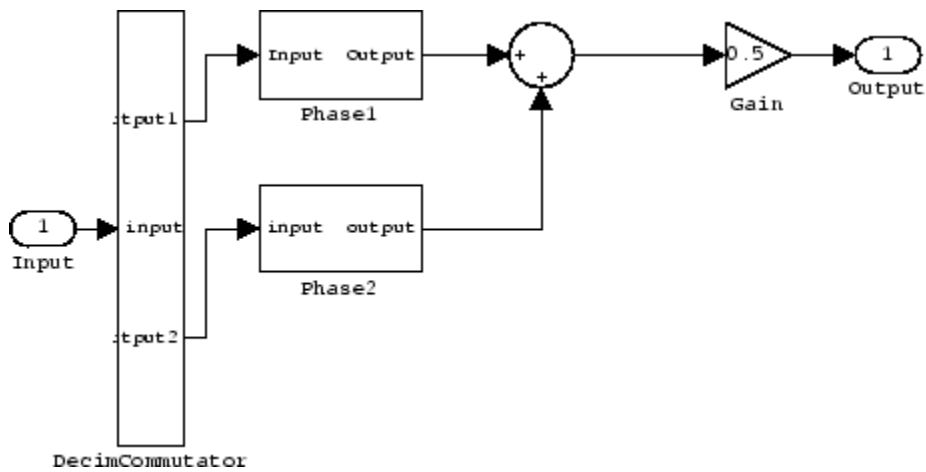
the first phase is a delay section with 0s and a 1 for coefficients and the second phase is a linear phase decimator, shown in the next models.



**Phase 1 model**



**Phase 2 model**



**Overall model**

## Version History

Introduced in R2011a

### See Also

`mfiltr` | `mfiltr.iirinterp` | `mfiltr.iirwdfdecim`

## mfilt.iirinterp

(Removed) IIR interpolator

### Compatibility

---

**Note** `mfilt.iirinterp` has been removed. Use `dsp.IIRHalfbandInterpolator` instead. For more details, see “Compatibility Considerations” on page 5-1230.

---

### Syntax

```
hm = mfilt.iirinterp(c1,c2,...)
```

### Description

`hm = mfilt.iirinterp(c1,c2,...)` constructs an IIR interpolator filter given the coefficients specified in the cell arrays `C1`, `C2`, etc.

The IIR interpolator is a polyphase IIR filter where each phase is a cascade allpass IIR filter.

Each cell array `ci` contains a set of vectors representing a cascade of allpass sections. Each element in one cell array is one section. For more information about the contents of each cell array, refer to `dfilt.cascadeallpass`. The contents of the cell arrays are the same for both filter constructors and `mfilt.iirdecim` interprets them same way as `mfilt.cascadeallpass`.

The following exception applies to interpreting the contents of a cell array—if one of the cell arrays `ci` contains only one vector, and that vector comprises a series of 0s and a unique element equal to 1, that cell array represents a `dfilt.delay` section with latency equal to the number of zeros, rather than a `dfilt.cascadeallpass` section. This exception case occurs with quasi-linear phase IIR interpolators.

Although the first example shows how to construct an IIR interpolator explicitly, one usually constructs an IIR interpolator filter as a result of designing an interpolator, as shown in the subsequent examples.

### Examples

When the coefficients are known, you can construct the IIR interpolator directly using `mfilt.iirinterp`. In the following example, a cascaded polyphase IIR interpolator filter is constructed using 2 phases for each of three stages. The coefficients are given below:

```
Phase1Sect1=0.0603;Phase1Sect2=0.4126; Phase1Sect3=0.7727;
Phase2Sect1=0.2160; Phase2Sect2=0.6044; Phase2Sect3=0.9239;
```

Next the filter is implemented by passing the above coefficients to `mfilt.iirinterp` as cell arrays, where each cell array represents a different phase.

```
Hm = mfilt.iirinterp({Phase1Sect1,Phase1Sect2,Phase1Sect3},...
{Phase2Sect1,Phase2Sect2,Phase2Sect3})
```

```

Hm =
    FilterStructure: 'IIR Polyphase Interpolator'
    Polyphase: Phase1: Section1: 0.0603
                Section2: 0.4126
                Section3: 0.7727
                Phase2: Section1: 0.216
                    Section2: 0.6044
                    Section3: 0.9239
    InterpolationFactor: 2
    PersistentMemory: false

```

Also refer to the “Quasi-Linear Phase Halfband and Dyadic Halfband Designs” section of “IIR Halfband Stages in Multistage Filter Design”.

When the coefficients are not known, use the approach given by the following set of examples. Start by designing an elliptic halfband interpolator with a interpolation factor of 2.

```

tw = 100; % Transition width of filter.
ast = 80; % Stopband attenuation of filter.
fs = 2000; % Sampling frequency of filter.
l = 2; % Interpolation factor.
d = fdesign.interpolator(l,'halfband','tw,ast',tw,ast,fs);

```

Specification object `d` stores the interpolator design specifics. With the details in `d`, design the filter, returning `hm`, an `mfilt.iirinterp` object. Use `hm` to realize the filter if you have Simulink installed.

```

hm = design(d,'ellip','filterstructure','iirinterp');
% Note that realizemdl requires Simulink
realizemdl(hm) % Build model of the filter.

```

Designing a linear phase halfband interpolator follows the same pattern.

```

tw = 100; % Transition width of filter.
ast= 60; % Stopband attenuation of filter.
fs = 2000; % Sampling frequency of filter.
l = 2; % Interpolation factor.
d = fdesign.interpolator(l,'halfband','tw,ast',tw,ast,fs);

```

`fdesign.interpolator` returns a specification object that stores the design features for an interpolator.

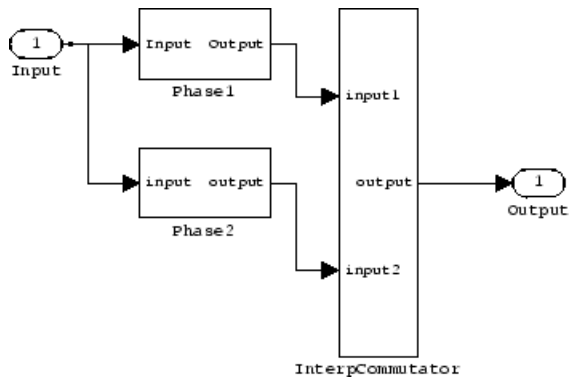
Now perform the actual design that results in an `mfilt.iirinterp` filter, `hm`.

```

hm = design(d,'iirlinphase','filterstructure','iirinterp');
% Note that realizemdl requires Simulink
realizemdl(hm) % Build model of the filter.

```

The toolbox creates a Simulink model for `hm`, shown here. `Phase1`, `Phase2`, and `InterpCommutator` are all subsystem blocks.



## Version History

Introduced in R2011a

**R2022a: `mfilt.iirinterp` has been removed**

*Errors starting in R2022a*

`mfilt.iirinterp` has been removed. Use `dsp.IIRHalfbandInterpolator` instead.

### Update Code

This table shows how the `mfilt.iirinterp` object is typically used and explains how to update existing code to use the `dsp.IIRHalfbandInterpolator` object.

| Discouraged Usage   | Recommended Replacement   |
|---|---|
| <p>Create an <code>mfilt.iirinterp</code> object using the <code>fdesign.interpolator</code> specification object followed by the <code>design</code> function.</p> <p>Note that the <code>fdesign.interpolator</code> object accepts the sample rate of the filter, while the <code>dsp.IIRHalfbandInterpolator</code> object accepts the sample rate of the input.</p> <p>The sample rate of the filter (output) is 2000 Hz.</p> <pre>% Sample rate of filter FsFilter = 2000; % Transition width of the filter (wrt filter sample rate) tw = 100; % Stopband attenuation of filter ast = 80;</pre> <p>Create an <code>mfilt.iirinterp</code> object using these specifications.</p> <pre>d = fdesign.interpolator(2,'halfband','tw,ast',tw,ast,FsFilter); Hm = design(d,'ellip','filterstructure','iirinterp')</pre> <p>Create sinusoidal data and pass the data through the object.</p> <pre>% Input sample rate FsIn = FsFilter/2; % 5120 samples, 0.232 second long signal n = (0:5119)'; % Original signal, sinusoid at 1 kHz x = sin(2*pi*1e3/FsIn*n); % 10240 samples, still 0.232 seconds y_mfilt = filter(Hm,x);</pre> | <p>Initialize the <code>dsp.IIRHalfbandInterpolator</code> object. This object accepts the sample rate of the input signal.</p> <pre>% Sample rate of filter FsFilter = 2000; % Input sample rate FsIn = FsFilter/2; % Transition width of the filter tw = 100; % Stopband attenuation of filter ast = 80;</pre> <p>Create a <code>dsp.IIRHalfbandInterpolator</code> object using these specifications.</p> <pre>iirI = dsp.IIRHalfbandInterpolator(SampleRate=FsIn,... TransitionWidth=tw,... StopbandAttenuation=ast);</pre> <p>Create sinusoidal data and pass the data through the object.</p> <pre>% 5120 samples, 0.232 second long signal n = (0:5119)'; % Original signal, sinusoid at 1 kHz x = sin(2*pi*1e3/FsIn*n); % 10240 samples, still 0.232 seconds y = iirI(x);</pre> |

## See Also

`dsp.IIRHalfbandInterpolator`

## mfilt.iirwdfdecim

IIR wave digital filter decimator

---

**Note** `mfilt.iirwdfdecim` will be removed in a future release. Use `dsp.IIRHalfbandDecimator` instead.

---

### Syntax

```
hm = mfilt.iirwdfdecim(c1,c2,...)
```

### Description

`hm = mfilt.iirwdfdecim(c1,c2,...)` constructs an IIR wave digital decimator given the coefficients specified in the cell arrays `c1`, `c2`, and so on. The IIR decimator `hm` is a polyphase IIR filter where each phase is a cascade wave digital allpass IIR filter.

Each cell array `ci` contains a set of vectors representing a cascade of allpass sections. Each element in one cell array is one section. For more information about the contents of each cell array, refer to `dfilt.cascadewdfallpass`. The contents of the cell arrays are the same for both filter constructors and `mfilt.iirwdfdecim` interprets them same way as `mfilt.cascadewdfallpass`.

The following exception applies to interpreting the contents of a cell array — if one of the cell arrays `ci` contains only one vector, and that vector comprises a series of 0s and one element equal to 1, that cell array represents a `dfilt.delay` section with latency equal to the number of zeros, rather than a `dfilt.cascadewdfallpass` section. This exception occurs with quasi-linear phase IIR decimators.

Usually you do not construct IIR wave digital filter decimators explicitly. Instead, you obtain an IIR wave digital filter decimator as a result of designing a halfband decimator. The first example in the following section illustrates this case.

### Examples

Design an elliptic halfband decimator with a decimation factor equal to 2. Both examples use the `iirwdfdecim` filter structure (an input argument to the `design` method) to design the final decimator.

The first portion of this example generates a filter specification object `d` that stores the specifications for the decimator.

```
tw = 100; % Transition width of filter to design, 100 Hz.  
ast = 80; % Stopband attenuation of filter 80 dB.  
fs = 2000; % Sampling frequency of the input signal.  
m = 2; % Decimation factor.  
d = fdesign.decimator(m,'halfband','tw,ast',tw,ast,fs);
```

Now perform the actual design using `d`. Filter object `hm` is an `mfilt.iirwdfdecim` filter.



```
Hm = design(d,'ellip','FilterStructure','iirwdfdecim');
% Note that realizemdl requires Simulink
realizemdl(hm) % Build model of the filter.
```

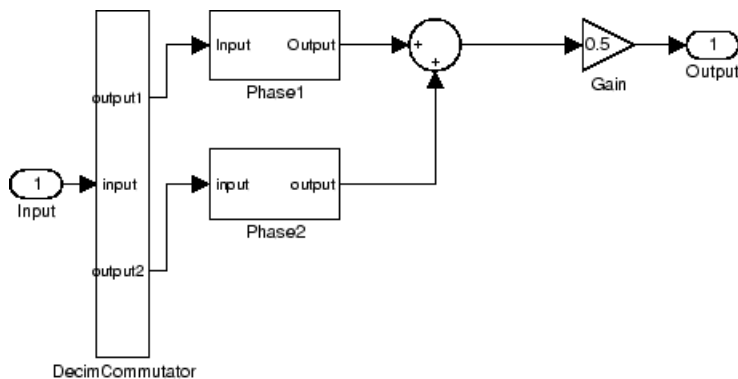
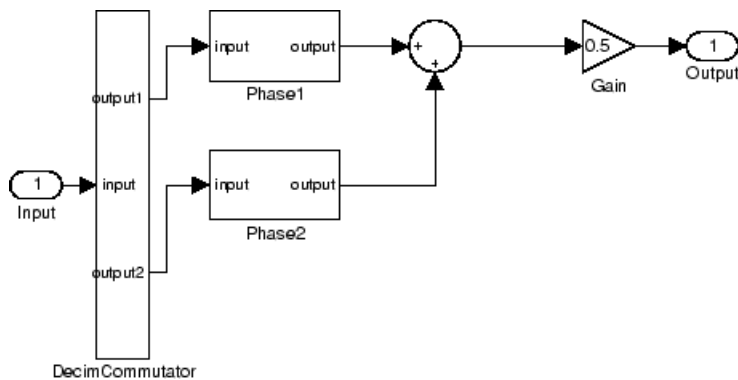
Design a linear phase halfband decimator for decimating a signal by a factor of 2.

```
tw = 100; % Transition width of filter, 100 Hz.
ast = 60; % Filter stopband attenuation = 80 dB
fs = 2000; % Input signal sampling frequency.
m = 2; % Decimation factor.
d = fdesign.decimator(m,'halfband','tw,ast',tw,ast,fs);
```

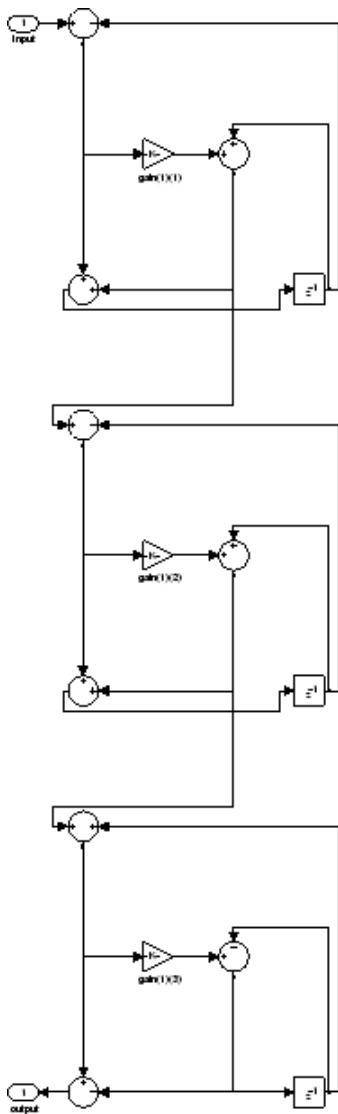
Use d to design the final filter hm, an `mfilt.iirwdfdecim` object.

```
hm = design(d,'iirlinphase','filterstructure',...
'iirwdfdecim');
% Note that realizemdl requires Simulink
realizemdl(hm) % Build model of the filter.
```

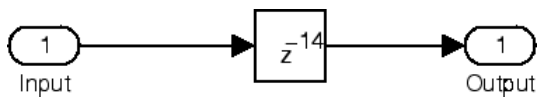
The models that `realizemdl` returns for each example appear below. At this level, the realizations of the filters are identical. The differences appear in the subsystem blocks Phase1 and Phase2.



This is the Phase1 subsystem from the halfband model.



Phase1 subsystem from the linear phase model is less revealing—an allpass filter.



## Version History

Introduced in R2011a

### See Also

`mfilt` | `mfilt.iirdecim` | `mfilt.iirwdfinterp`

# mfilt.iirwdfinterp

IIR wave digital filter interpolator

---

**Note** `mfilt.iirwdfinterp` will be removed in a future release. Use `dsp.IIRHalfbandInterpolator` instead.

---

## Syntax

```
hm = mfilt.iirwdfinterp(c1,c2,...)
```

## Description

`hm = mfilt.iirwdfinterp(c1,c2,...)` constructs an IIR wave digital interpolator given the coefficients specified in the cell arrays `c1`, `c2`, and so on. The IIR interpolator `hm` is a polyphase IIR filter where each phase is a cascade wave digital allpass IIR filter.

Each cell array `ci` contains a set of vectors representing a cascade of allpass sections. Each element in one cell array is one section. For more information about the contents of each cell array, refer to `dfilt.cascadewdfallpass`. The contents of the cell arrays are the same for both filter constructors and `mfilt.iirwdfinterp` interprets them same way as `mfilt.cascadewdfallpass`.

The following exception applies to interpreting the contents of a cell array — if one of the cell arrays `ci` contains only one vector, and that vector comprises a series of 0s and one element equal to 1, that cell array represents a `dfilt.delay` section with latency equal to the number of zeros, rather than a `dfilt.cascadewdfallpass` section. This exception occurs with quasi-linear phase IIR interpolators.

Usually you do not construct IIR wave digital filter interpolators explicitly. Rather, you obtain an IIR wave digital interpolator as a result of designing a halfband interpolator. The first example in the following section illustrates this case.

## Examples

Design an elliptic halfband interpolator with interpolation factor equal to 2. At the end of the design process, `hm` is an IIR wave digital filter interpolator.

```
tw = 100; % Transition width of filter, 100 Hz.
ast = 80; % Stopband attenuation of filter, 80 dB.
fs = 2000; % Sampling frequency after interpolation.
l = 2; % Interpolation factor.
d = fdesign.interpolator(l,'halfband','tw,ast',tw,ast,fs);
```

The specification object `d` stores the interpolator design requirements. Now use `d` to design the actual filter `hm`.

```
hm = design(d,'ellip','filterstructure','iirwdfinterp');
```

If you have Simulink installed, you can realize your filter as a model built from DSP System Toolbox blocks.

```
% Note that realizemdl requires Simulink
realizemdl(hm)      % Build model of the filter.
```

For variety, design a linear phase halfband interpolator with an interpolation factor of 2.

```
tw = 100; % Transition width of filter, 100 Hz.
ast = 80; % Stopband attenuation of filter, 80 dB.
fs = 2000; % Sampling frequency after interpolation.
l = 2; % Interpolation factor.
d = fdesign.interpolator(l,'halfband','tw,ast',tw,ast,fs);
```

Now perform the actual design process with `d`. Filter `hm` is an IIR wave digital filter interpolator. As in the previous example, `realizemdl` returns a Simulink model of the filter if you have Simulink installed.

```
hm = design(d,'iirlinphase','filterstructure',...
'iirwdfinterp');
% Note that realizemdl requires Simulink
realizemdl(hm)      % Build model of the filter.
```

## Version History

**Introduced in R2011a**

### See Also

`mfilt.iirinterp` | `mfilt.iirwdfdecim`

# mfilt.linearinterp

(Removed) Linear interpolator

---

**Note** `mfilt.linearinterp` has been removed. Use `dsp.FIRInterpolator(L,'Linear')` instead. For more details, see “Compatibility Considerations”.

---

## Syntax

```
hm = mfilt.linearinterp(l)
```

## Description

`hm = mfilt.linearinterp(l)` returns an FIR linear interpolator `hm` with an integer interpolation factor `l`. Provide `l` as a positive integer. The default value for the interpolation factor is 2 when you do not include the input argument `l`.

When you use this linear interpolator, the samples added to the input signal have values between the values of adjacent samples in the original signal. Thus you see something like a smooth profile where the interpolated samples continue a line between the previous and next original samples. The example demonstrates this smooth profile clearly. Compare this to the interpolation process for `mfilt.holdinterp`, which creates a staircase profile.

Make this filter a fixed-point or single-precision filter by changing the value of the **Arithmetic** property for the filter `hm` as follows:

- To change to single-precision filtering, enter
 

```
set(hm,'arithmetic','single');
```
- To change to fixed-point filtering, enter
 

```
set(hm,'arithmetic','fixed');
```

## Input Arguments

The following table describes the input argument for `mfilt.linearinterp`.

| Input Argument | Description   |
|----------------|---|
| <code>l</code> | Interpolation factor for the filter. <code>l</code> specifies the amount to increase the input sampling rate. It must be an integer. When you do not specify a value for <code>l</code> it defaults to 2. |

## Object Properties

This section describes the properties for both floating-point filters (double-precision and single-precision) and fixed-point filters.

### Floating-Point Filter Properties

Every multirate filter object has properties that govern the way it behaves when you use it. Note that many of the properties are also input arguments for creating `mfilt.linearinterp` objects. The next table describes each property for an `mfilt.linearinterp` filter object.

| Name                | Values                 | Description  |
|---------------------|------------------------|--|
| Arithmetic          | Double, single, fixed  | Specifies the arithmetic the filter uses to process data while filtering.  |
| FilterStructure     | Character vector       | Reports the type of filter object. You cannot set this property — it is always read only and results from your choice of <code>mfilt</code> object.  |
| InterpolationFactor | Integer                | Interpolation factor for the filter. <code>l</code> specifies the amount to increase the input sampling rate. It must be an integer.   |
| PersistentMemory    | 'false' or 'true'      | Determine whether the filter states get restored to zero for each filtering operation  |
| States              | Double or single array | Filter states. <code>states</code> defaults to a vector of zeros that has length equal to <code>nstates(hm)</code> . Always available, but visible in the display only when <code>PersistentMemory</code> is true. |

### Fixed-Point Filter Properties

This table shows the properties associated with the fixed-point implementation of the `mfilt.holdinterp` filter.

**Note** The table lists all of the properties that a fixed-point filter can have. Many of the properties listed are dynamic, meaning they exist only in response to the settings of other properties. To view all of the characteristics for a filter at any time, use

```
info(hm)
```

where `hm` is a filter.

For further information about the properties of this filter or any `mfilt` object, refer to “Multirate Filter Properties” on page 6-2.

| Name            | Values   | Description   |
|-----------------|--|---|
| AccumFracLength | Any positive or negative integer number of bits. Depends on <code>L</code> . [29 when <code>L=2</code> ] | Specifies the fraction length used to interpret data output by the accumulator. |
| AccumWordLength | Any integer number of bits [33]  | Sets the word length used to store data in the accumulator.                     |

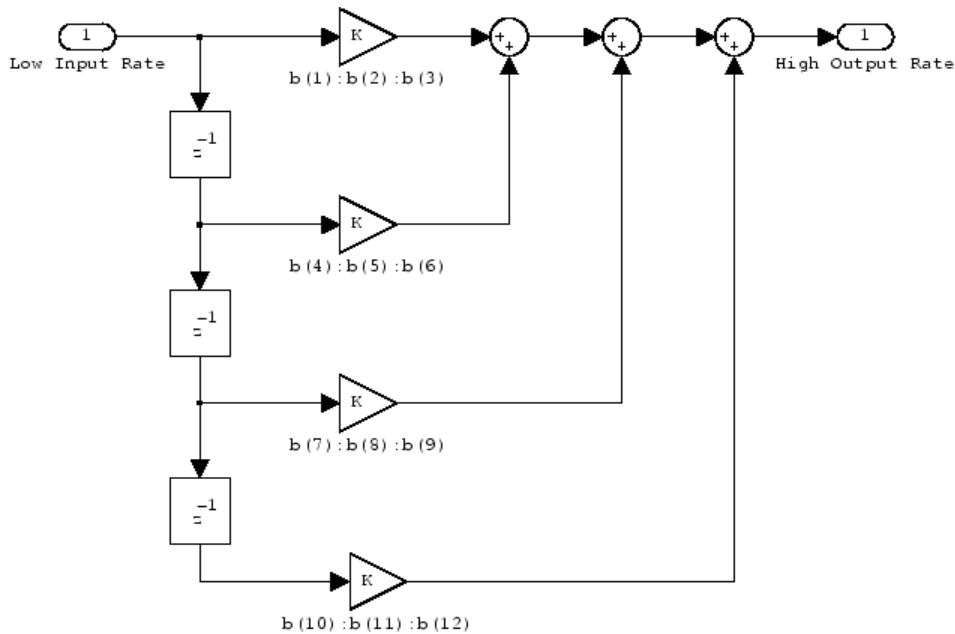
| Name             | Values   | Description   |
|------------------|--|---|
| Arithmetic       | fixed for fixed-point filters                        | Setting this to <code>fixed</code> allows you to modify other filter properties to customize your fixed-point filter.   |
| CoeffAutoScale   | [true], false  | Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to <code>false</code> enables you to change the <code>NumFracLength</code> property value to specify the precision used.  |
| CoeffWordLength  | Any integer number of bits [16]                      | Specifies the word length to apply to filter coefficients.  |
| FilterInternals  | [FullPrecision], SpecifyPrecision                    | Controls whether the filter automatically sets the output word and fraction lengths, product word and fraction lengths, and the accumulator word and fraction lengths to maintain the best precision results during filtering. The default value, <code>FullPrecision</code> , sets automatic word and fraction length determination by the filter. <code>SpecifyPrecision</code> makes the output and accumulator-related properties available so you can set your own word and fraction lengths for them. |
| InputFracLength  | Any positive or negative integer number of bits [15] | Specifies the fraction length the filter uses to interpret input data.  |
| InputWordLength  | Any integer number of bits [16]                      | Specifies the word length applied to interpret input data.  |
| NumFracLength    | Any positive or negative integer number of bits [14] | Sets the fraction length used to interpret the numerator coefficients.  |
| OutputFracLength | Any positive or negative integer number of bits [29] | Determines how the filter interprets the filter output data. You can change the value of <code>OutputFracLength</code> when you set <code>FilterInternals</code> to <code>SpecifyPrecision</code> .   |
| OutputWordLength | Any integer number of bits [33]                      | Determines the word length used for the output data. You make this property editable by setting <code>FilterInternals</code> to <code>SpecifyPrecision</code> .   |

| Name         | Values  | Description  |
|--------------|---|--|
| OverflowMode | saturate, [wrap]  | Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either <code>saturate</code> (limit the output to the largest positive or negative representable value) or <code>wrap</code> (set overflowing values to the nearest representable value using modular arithmetic.) The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.   |
| RoundMode    | [convergent], <code>ceil</code> , <code>fix</code> , <code>floor</code> , <code>nearest</code> , <code>round</code> | Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). <ul style="list-style-type: none"> <li>• <code>ceil</code> - Round toward positive infinity.</li> <li>• <code>convergent</code> - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.</li> <li>• <code>fix</code> - Round toward zero.</li> <li>• <code>floor</code> - Round toward negative infinity.</li> <li>• <code>nearest</code> - Round toward nearest. Ties round toward positive infinity.</li> <li>• <code>round</code> - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.</li> </ul> <p>The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.</p> |
| Signed       | [true], false   | Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.  |
| States       | <code>fi</code> object  | Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use <code>fi</code> objects, with the associated properties from those objects. For details, refer to fixed-point objects in Fixed-Point Designer documentation. For information about the ordering of the states, refer to the filter structure in the following section.  |



## Filter Structure

Linear interpolator structures depend on the FIR filter you use to implement the filter. By default, the structure is direct-form FIR.



## Examples

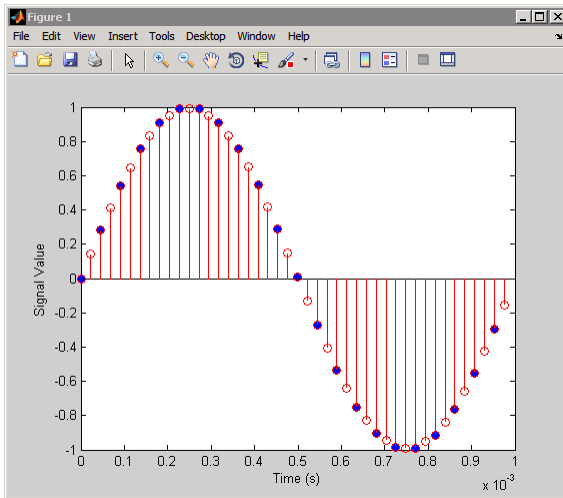
Interpolation by a factor of 2 (used to convert the input signal sampling rate from 22.05 kHz to 44.1 kHz).

```

l = 2; % Interpolation factor
hm = mfilt.linearinterp(l);
fs = 22.05e3; % Original sample freq: 22.05 kHz.
n = 0:5119; % 5120 samples, 0.232 second long signal
x = sin(2*pi*1e3/fs*n); % Original signal, sinusoid at 1 kHz
y = filter(hm,x); % 10240 samples, still 0.232 seconds
stem(n(1:22)/fs,x(1:22),'filled') % Plot original sampled at
% 22.05 kHz
hold on % Plot interpolated signal (44.1
% kHz) in red
stem(n(1:44)/(fs*l),y(2:45),'r')
xlabel('Time (s)');ylabel('Signal Value')

```

Using linear interpolation, as compared to the hold approach of `mfilt.holdinterp`, provides greater fidelity to the original signal.



## Version History

Introduced in R2011a

### R2022a: `mfilt.linearinterp` has been removed

Errors starting in R2022a

`mfilt.linearinterp` has been removed. Use `dsp.FIRInterpolator(L, 'Linear')` instead.

### Update Code

This table shows how the `mfilt.linearinterp` object is typically used and explains how to update existing code to use the `dsp.FIRInterpolator` object.

| Discouraged Usage  | Recommended Replacement  |
|--|--|
| <p>Initialize the object. Set the interpolation factor to 2.</p> <pre>L = 2; hm = mfilt.linearinterp(L)</pre> <p>Pass the data through the object.</p> <pre>% Input sample rate fs = 22.05e3; % 5120 samples, 0.232 second long signal n = (0:5119)'; % Original signal, sinusoid at 1 kHz x = sin(2*pi*1e3/fs*n); y_mfilt = filter(hm,x);</pre> | <p>Initialize the object. Set the interpolation factor to 2.</p> <pre>L = 2; firI = dsp.FIRInterpolator(L, 'linear')</pre> <p>Pass the data through the object.</p> <p>Note that System objects operate column-wise on data while <code>mfilt</code> objects are agnostic to the dimension. Therefore, pass the data vector as a column vector to the object algorithm.</p> <pre>% Input sample rate fs = 22.05e3; % 5120 samples, 0.232 second long signal n = (0:5119)'; % Original signal, sinusoid at 1 kHz x = sin(2*pi*1e3/fs*n); y = firI(x);</pre> |

**See Also**

`dsp.FIRInterpolator`

## minimizecoeffwl

Minimum word length fixed-point filter

### Syntax

```
Hq = minimizecoeffwl(Hd)
Hq = minimizecoeffwl(Hd,...,'NoiseShaping',NSFlag)
Hq = minimizecoeffwl(Hd,...,'NTrials',N)
Hq = minimizecoeffwl(Hd,...,'Apasstol',Apasstol)
Hq = minimizecoeffwl(Hd,...,'Astoptol',Astoptol)
Hq = minimizecoeffwl(Hd,...,'MatchrefFilter',RefFiltFlag)
```

### Description

`Hq = minimizecoeffwl(Hd)` returns the minimum word length fixed-point filter object `Hq` that meets the design specifications of the single-stage or multistage FIR filter object `Hd`. `Hd` must be generated using `fdesign` and `design`. If `Hd` is a multistage filter object, the procedure minimizes the word length for each stage separately. `minimizecoeffwl` uses a stochastic noise-shaping procedure by default to minimize the wordlength. To obtain repeatable results on successive function calls, initialize the uniform random number generator `rand`.

`Hq = minimizecoeffwl(Hd,...,'NoiseShaping',NSFlag)` enables or disables the stochastic noise-shaping procedure in the minimization of the word length. By default `NSFlag` is `true`. Setting `NSFlag` to `false` minimizes the word length without using noise-shaping.

`Hq = minimizecoeffwl(Hd,...,'NTrials',N)` specifies the number of Monte Carlo trials to use in the minimization. `Hq` is the filter with the minimum word length among the `N` trials that meets the specifications in `Hd`. `'NTrials'` defaults to one.

`Hq = minimizecoeffwl(Hd,...,'Apasstol',Apasstol)` specifies the passband ripple tolerance in dB. `'Apasstol'` defaults to `1e-4`.

`Hq = minimizecoeffwl(Hd,...,'Astoptol',Astoptol)` specifies the stopband tolerance in dB. `'Astoptol'` defaults to `1e-2`.

`Hq = minimizecoeffwl(Hd,...,'MatchrefFilter',RefFiltFlag)` determines whether the fixed-point filter matches the filter order and transition width of the floating-point design. Setting `'MatchRefFilter'` to `true` returns a fixed-point filter with the same order and transition width as `Hd`. The `'MatchRefFilter'` property defaults to `false` and the resulting fixed-point filter may have a different order and transition width than the floating-point design `Hd`.

You must have the Fixed-Point Designer software installed to use this function.

### Examples

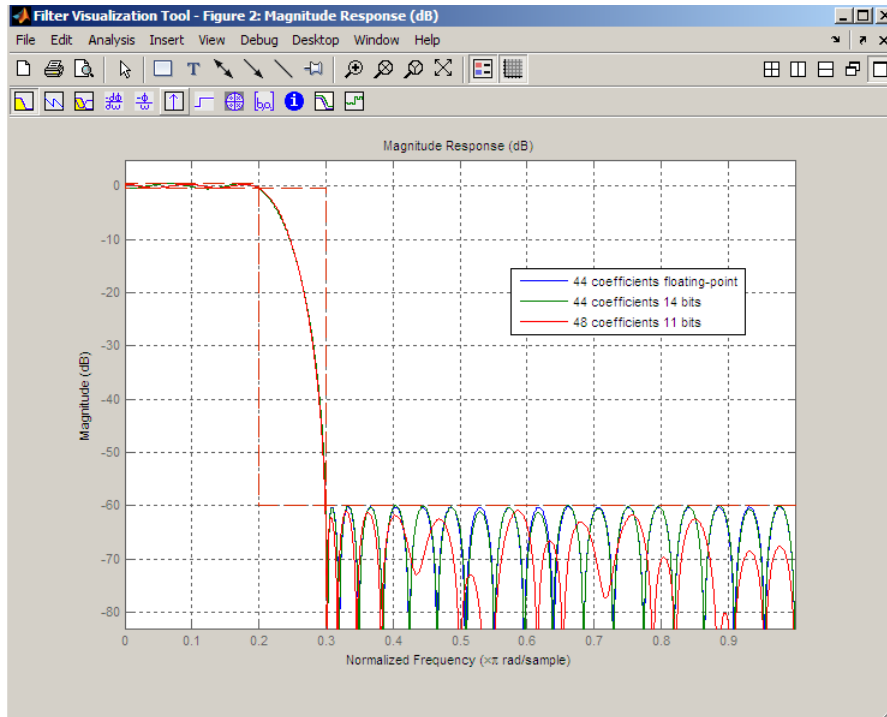
Minimize word length for lowpass FIR equiripple filter:

```
f=fdesign.lowpass('Fp,Fst,Ap,Ast',0.2,0.3,1,60);
% Design filter with double-precision floating point
Hd=design(f,'equiripple');
```

```

% Find minimum wordlength fixed-point filter
% with 0.15 dB stopband tolerance
Hq=minimizecoeffwl(Hd,'MatchRefFilter',true,'Astoptol',0.15);
Hq1=minimizecoeffwl(Hd,'Astoptol',0.15);
% Hq.coeffwordlength is 14 bits.
% Hq1.coeffwordlength is 11 bits
hfvf=fvtool(Hd,Hq,Hq1,'showreference','off');
legend(hfvf,'44 coefficients floating-point',...
'44 coefficients 14 bits','48 coefficients 11 bits');

```



## Version History

Introduced in R2011a

## See Also

[constraincoeffwl](#) | [design](#) | [fdesign](#) | [maximizestopband](#) | [measure](#) | [rand](#)

## Topics

“Fixed-Point Overview”

# modifyCursor

**Package:** dsp

Modify properties of Logic Analyzer cursor

## Syntax

```
modifyCursor(scope, tag)  
modifyCursor(scope, tag, Name, Value)
```

## Description

`modifyCursor(scope, tag)` modifies the properties of the Logic Analyzer cursor specified by the input tag.

`modifyCursor(scope, tag, Name, Value)` sets properties using one or more name-value pairs. Enclose each property name in single quotes.

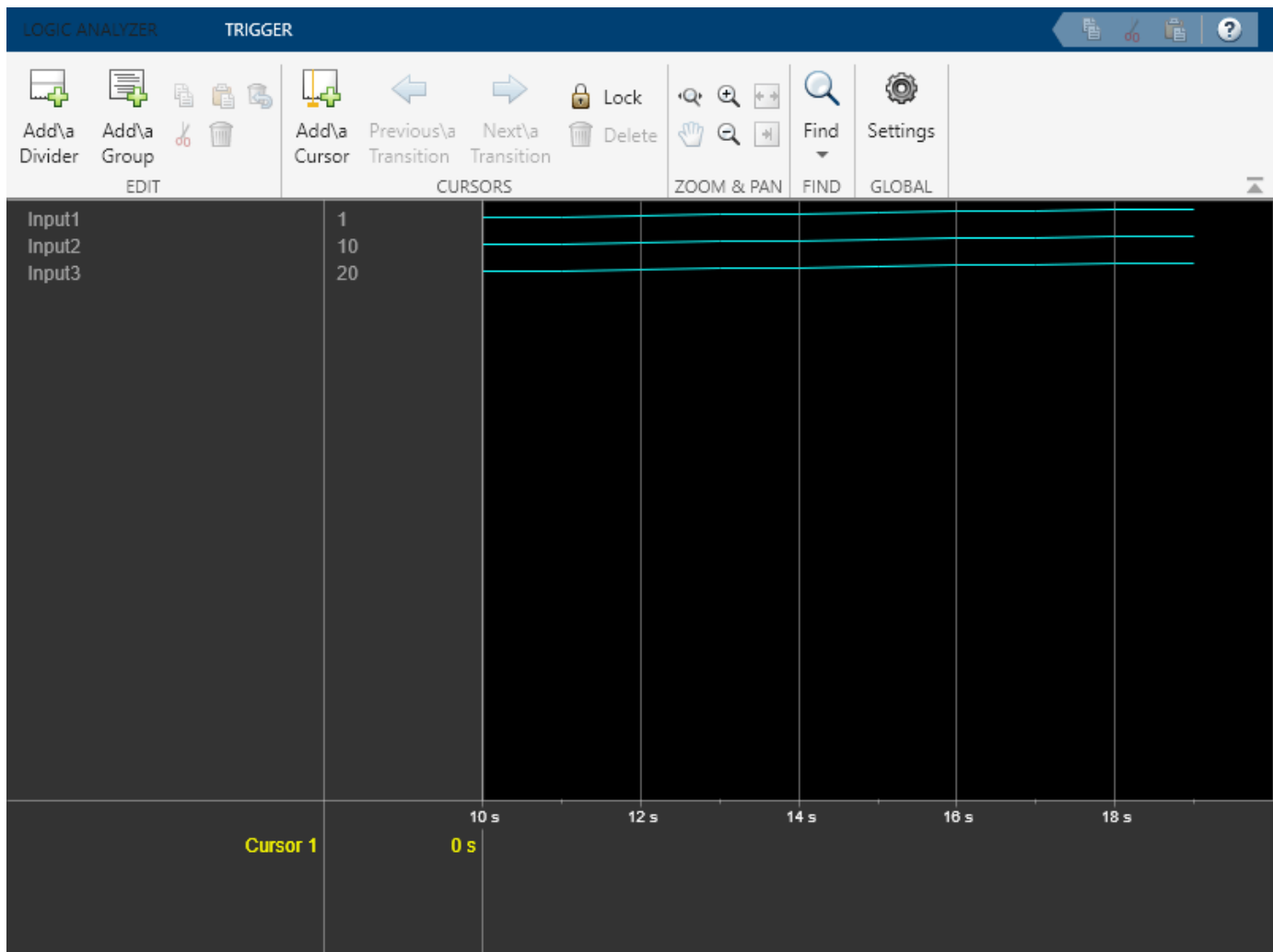
## Examples

### Modify Logic Analyzer Cursors Programmatically

This example shows how to use functions to create, manipulate, and delete cursors in a `dsp.LogicAnalyzer` object.

### Create Logic Analyzer and Signals

```
scope = dsp.LogicAnalyzer('NumInputPorts',3);  
for ii = 1:20  
    scope(ii,10*ii,20*ii);  
end
```



### Add Cursor

Add a cursor at 15 seconds and show the cursor information.

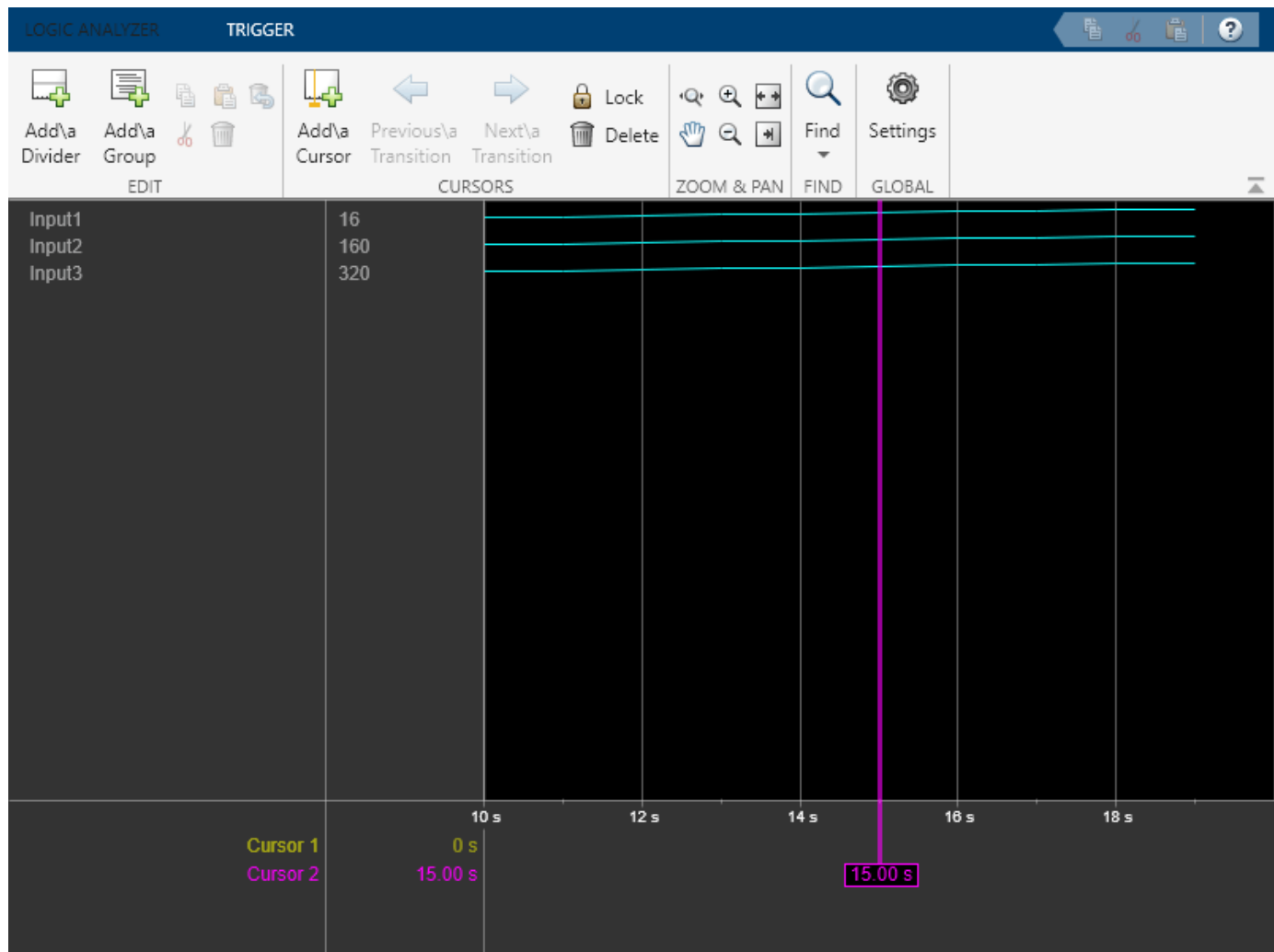
```
cursor = addCursor(scope, 'Location', 15, 'Color', 'Cyan');
getCursorInfo(scope, cursor)
```

```
ans = struct with fields:
    Location: 15
    Color: [0 1 1]
    Locked: 0
    Tag: 'C2'
```

### Modify Cursor

Change the cursor color to magenta.

```
hide(scope)
modifyCursor(scope, cursor, 'Color', 'Magenta')
show(scope)
```

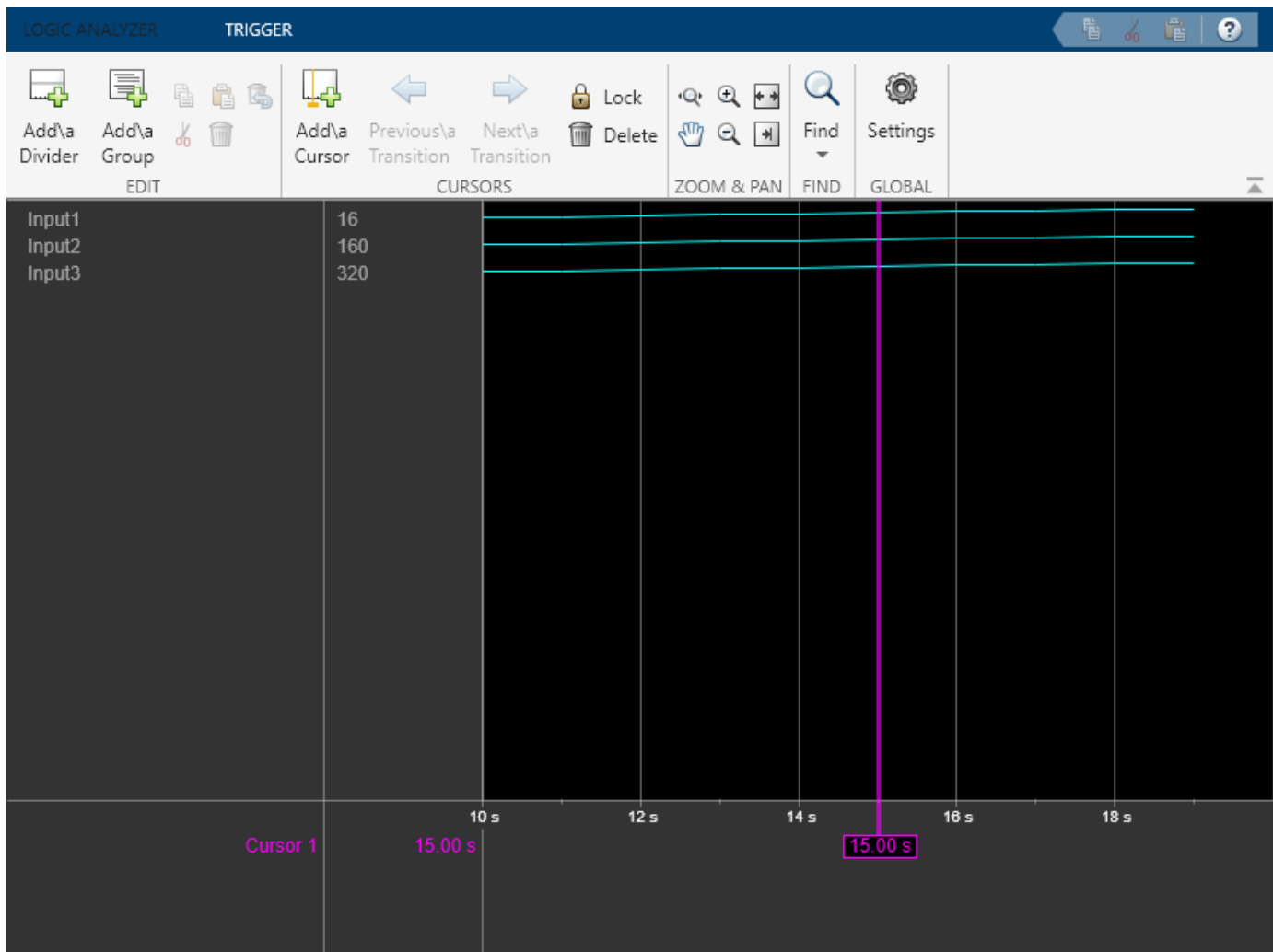


### Remove Cursor

Delete the yellow cursor at 0 seconds.

```
hide(scope)
tags = getCursorTags(scope);
deleteCursor(scope, tags{1});
show(scope)
```





## Input Arguments

**scope** — The Logic Analyzer object for which you want to modify a cursor

`dsp.LogicAnalyzer` object

The Logic Analyzer object for which you want to modify a cursor specified, as a handle to the `dsp.LogicAnalyzer` object.

**tag** — The tag identifying which cursor to modify

character vector | string scalar

The tag identifying which cursor to modify specified.

Example: `modifyCursor(scope, 'C4')` modifies a cursor in Logic Analyzer.

Example: `modifyCursor(scope, "C4")` modifies a cursor in Logic Analyzer.

Data Types: `char` | `string`

### Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, . . . , NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.





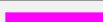



*Before R2021a, use commas to separate each name and value, and enclose `Name` in quotes.*

Example: `'Location',2,'Color','Blue'` specifies that a cursor should be moved to the 2-second mark and colored blue.

### Color — Color of the cursor

`'Yellow'` (default) | character vector | three element numeric vector | string scalar

Color of the cursor, specified as a `[R G B]` number value, color name, or color short name:

| Color Name | Short Name | RGB Triplet | Appearance  |
|------------|------------|-------------|---|
| "red"      | "r"        | [1 0 0]     |    |
| "green"    | "g"        | [0 1 0]     |    |
| "blue"     | "b"        | [0 0 1]     |    |
| "cyan"     | "c"        | [0 1 1]     |    |
| "magenta"  | "m"        | [1 0 1]     |    |
| "yellow"   | "y"        | [1 1 0]     |    |
| "black"    | "k"        | [0 0 0]     |  |
| "white"    | "w"        | [1 1 1]     |  |

Example: `'Color','blue'`

Example: `'Color',[0,0,1]`

Data Types: `char` | `string` | `double` | `single` | `uint8` | `uint16` | `uint32` | `uint64` | `int8` | `int16` | `int32` | `int64`

### Location — Location of the cursor

`0` (default) | numeric scalar

Specify as a numeric scalar value, in seconds, the cursor location.

Example: `'Location',1`

Data Types: `double`

### Locked — Locked status of the cursor

`false` (default) | `true`

Locked status of the cursor, specified as `false` or `true`.

- `true` — the cursor location cannot be changed. Logic Analyzer denotes the locked cursor by assigning a default color of gray. This color cannot be changed.
- `false` — the cursor location can be changed. Logic Analyzer denotes the unlocked cursor by assigning a default color of yellow.

Example: `'Locked',true`

## **Version History**

**Introduced in R2013a**

### **See Also**

`dsp.LogicAnalyzer` | `addCursor` | `deleteCursor` | `getCursorTags` | `getCursorInfo`

# modifyDisplayChannel

**Package:** dsp

Modify properties of Logic Analyzer display channel

## Syntax

```
modifyDisplayChannel(scope, tag, Name, Value)
```

## Description

`modifyDisplayChannel(scope, tag, Name, Value)` modifies the properties of `tag` using properties specified by one or more name-value pairs. Enclose each property name in single quotes.

## Examples

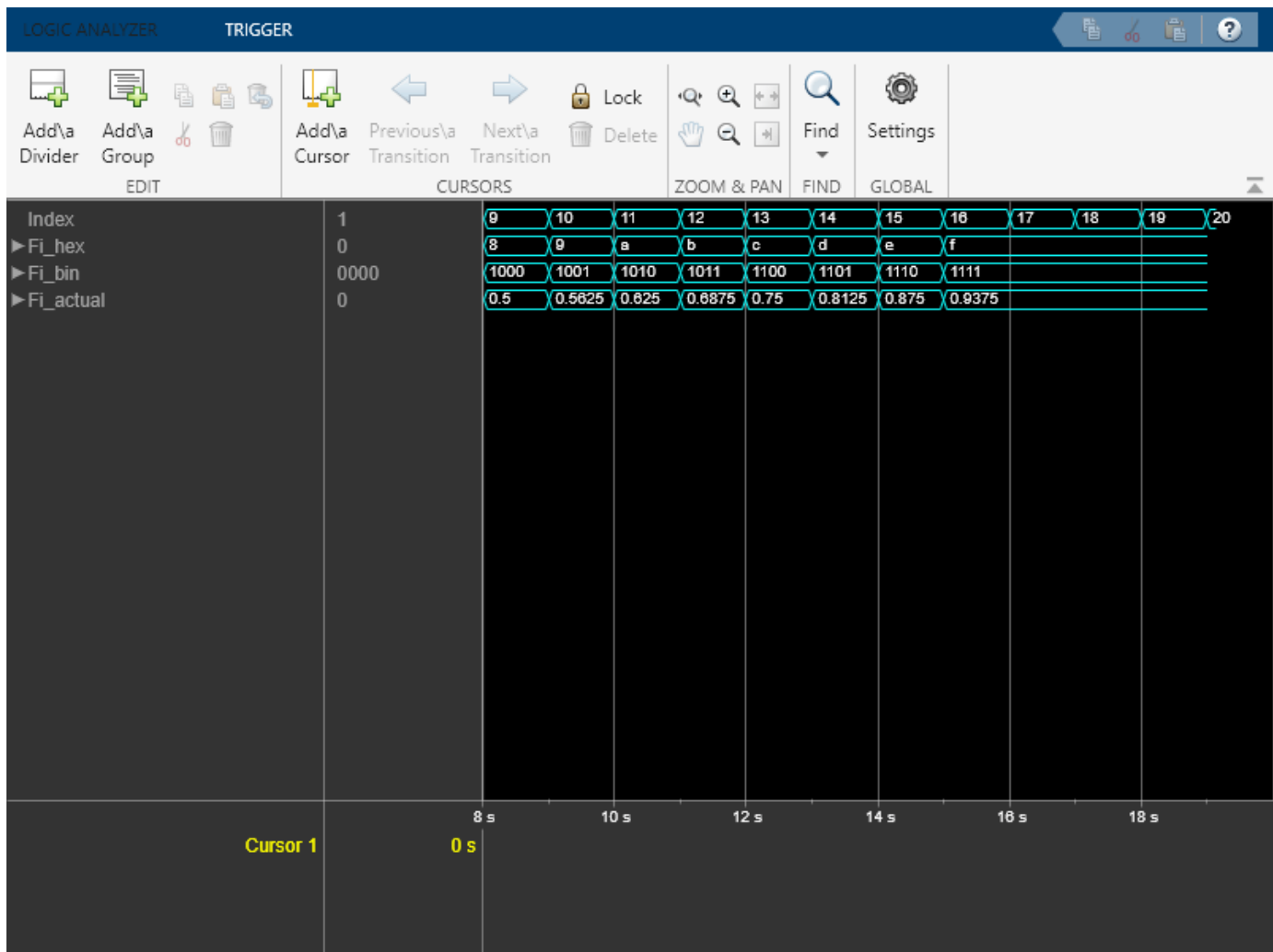
### Display Fixed-Point Signals

Create a `dsp.LogicAnalyzer` object with four channels. Call `modifyDisplayChannel` to set the radix of each of the channels. Run the scope in a loop to display the waves.

```
scope = dsp.LogicAnalyzer('NumInputPorts',4,'DisplayChannelFormat','Digital');
scope.TimeSpan = 12;

modifyDisplayChannel(scope,1,'Name','Index','Radix','Unsigned decimal');
modifyDisplayChannel(scope,2,'Name','Fi_hex','Radix','Hexadecimal');
modifyDisplayChannel(scope,3,'Name','Fi_bin','Radix','Binary');
modifyDisplayChannel(scope,4,'Name','Fi_actual','Radix','Signed decimal');

for ii = 1:20
    fival = fi((ii-1)/16,0,4,4);
    scope(ii,fival,fival,fival);
end
```



## Manipulate Logic Analyzer Programatically

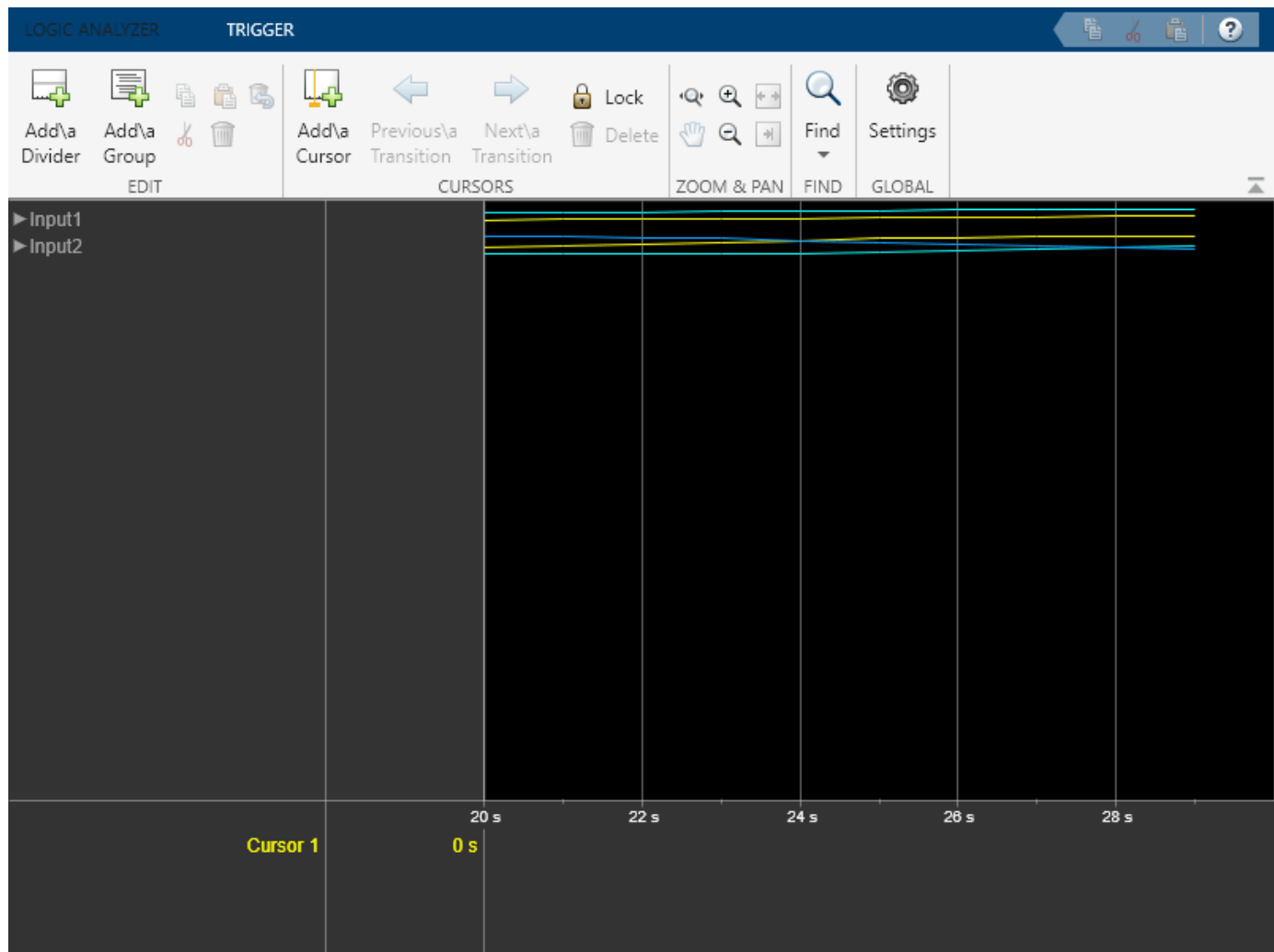
Use functions to construct and manipulate a dsp.LogicAnalyzer System object.

### Display Waves on Logic Analyzer scope.

```
scope = dsp.LogicAnalyzer('NumInputPorts',2);

stop = 30;
for count = 1:stop
    sinValVec = sin(count/stop*2*pi);
    cosValVec = cos(count/stop*2*pi);
    cosValVecOffset = cos((count+10)/stop*2*pi);

    scope([count (count-(stop/2))],[sinValVec cosValVec cosValVecOffset])
end
```



### Reorganize Display

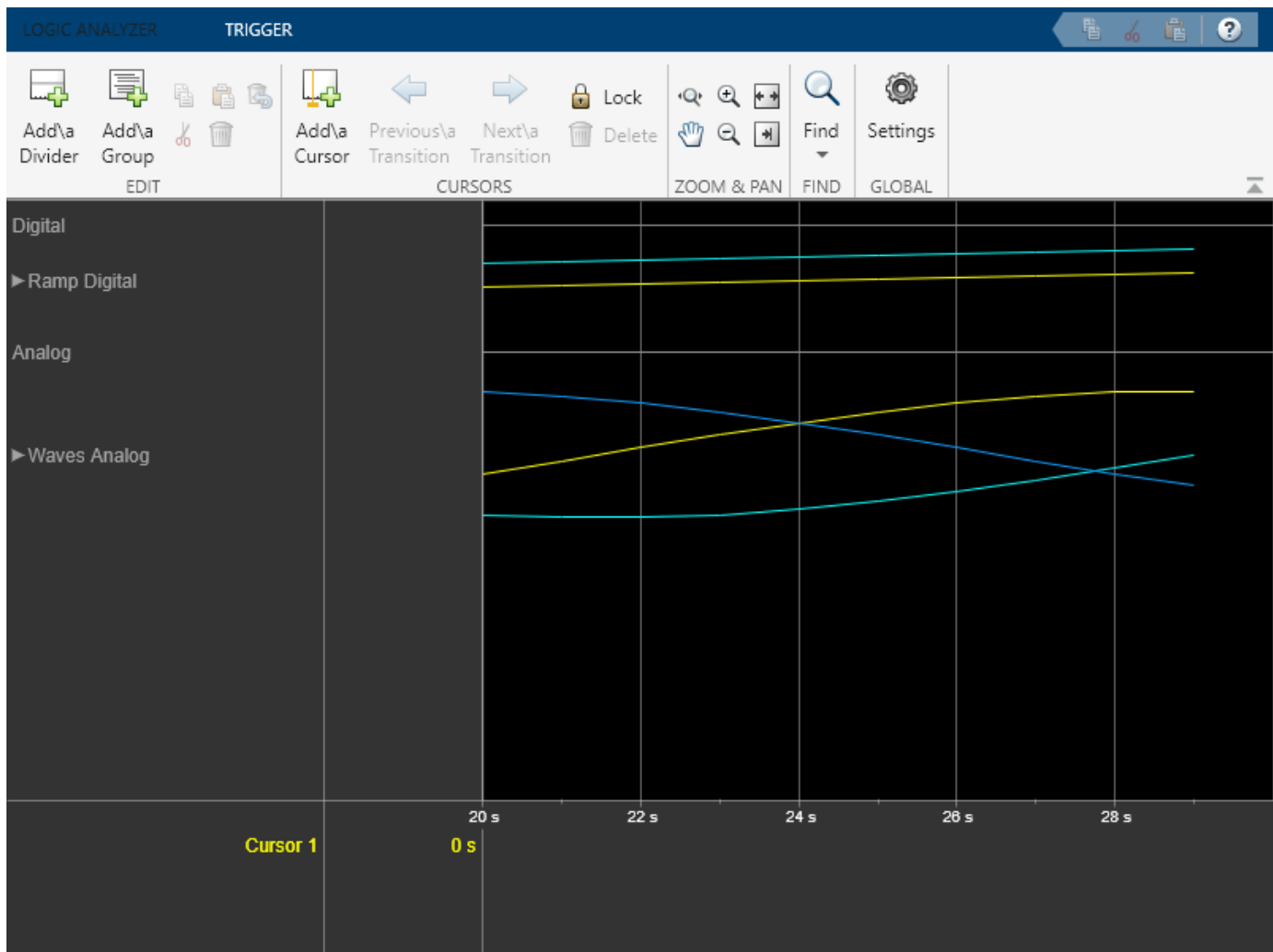
```
hide(scope)
digitalDividerTag = addDivider(scope, 'Name', 'Digital', 'Height', 20);
analogDividerTag = addDivider(scope, 'Name', 'Analog', 'Height', 40);

tags = getDisplayChannelTags(scope);

modifyDisplayChannel(scope, tags{1}, 'InputChannel', 1, ...
    'Name', 'Ramp Digital', 'Height', 40);
modifyDisplayChannel(scope, tags{2}, 'InputChannel', 2, ...
    'Name', 'Waves Analog', 'Format', 'Analog', 'Height', 80);

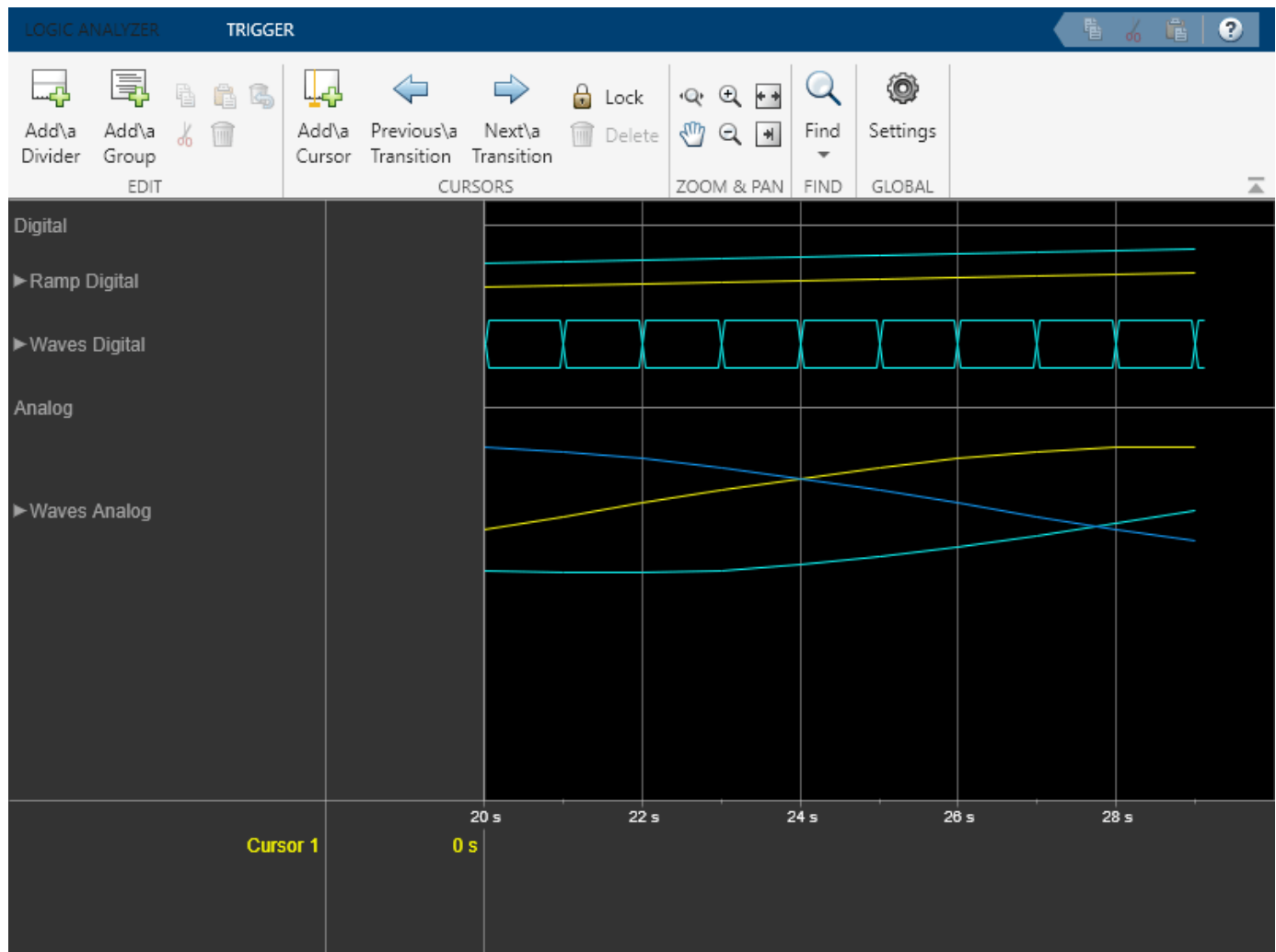
moveDisplayChannel(scope, digitalDividerTag, 'DisplayChannel', 1)
moveDisplayChannel(scope, tags{2}, 'DisplayChannel', length(tags))

show(scope)
```



### Duplicate Wave and Check Information

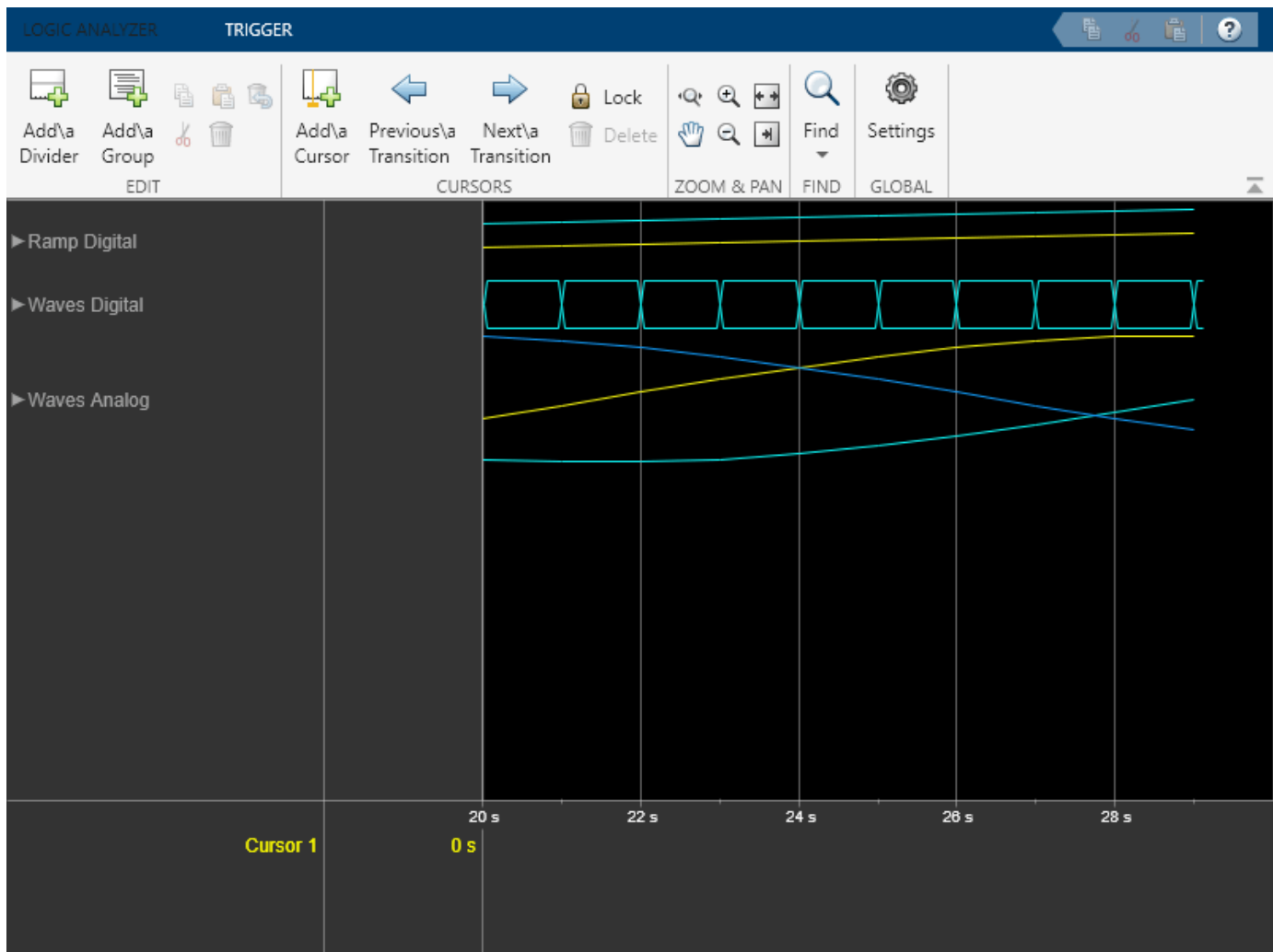
```
hide(scope)
addWave(scope, 'InputChannel', 2, 'Name', 'Waves Digital', 'Format', 'Digital', ...
'Height', 30, 'DisplayChannel', 3);
show(scope)
```



### Remove Dividers

```
hide(scope)
deleteDisplayChannel(scope,digitalDividerTag)
deleteDisplayChannel(scope,analogDividerTag)
show(scope)
```





### Clear variables

```
clear analogDividerTag cosValVec cosValVecOffset count digitalDividerTag duplicateWave scope sin
```

## Input Arguments

**scope** — Logic Analyzer object for which you want to modify a display channel

dsp.LogicAnalyzer object

The Logic Analyzer object for which you want to modify a display channel, specified as a handle to the dsp.LogicAnalyzer object.

**tag** — which display channel to modify

character vector | string scalar

The tag identifying which display channel to modify.

Example: `modifyDisplayChannel(scope,tag)`

Example: `modifyDisplayChannel(scope,'W4')`

Data Types: `char` | `string`

The first section on Name-Value Pair Arguments shows the properties you can set if the display channel contains a wave. The second section on Name-Value Pair Arguments shows the properties you can set if the display channel contains a divider.

### **Wave Name-Value Pair Arguments**

Specify optional pairs of arguments as `Name1=Value1, . . . ,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: `'InputChannel',2,'Color','Blue'` specifies that a wave should be added to input channel 1 and colored blue.

### **Color — Color of the wave**

`'Default'` (default) | character vector | three element numeric vector | string scalar

Color of the wave, specified as an [R G B] value or one of the following:

- `'Black'`
- `'Blue'`
- `'Cyan'`
- `'Default'`
- `'Green'`
- `'Magenta'`
- `'Red'`
- `'White'`
- `'Yellow'`

When you choose `'Default'`, the value of the `DisplayChannelColor` property in the Logic Analyzer is used.

Example: `'Color','Blue'`

Example: `'Color',[0,0,1]`

Data Types: `char` | `string` | `double` | `single` | `uint8` | `uint16` | `uint32` | `uint64` | `int8` | `int16` | `int32` | `int64`

### **DisplayChannel — Channel on the display that shows this wave**

`NumInputPorts` (default) | scalar numeric value in the range (1,NumInputPorts)

Specify as a scalar numeric value the display channel that shows this wave. By default, the wave is added to the end of the display.

Example: `'DisplayChannel',2`

Data Types: `double` | `single` | `uint8` | `uint16` | `uint32` | `uint64` | `int8` | `int16` | `int32` | `int64`

### **FontSize — Font size for values in the wave**

`0` (default) | scalar nonnegative integer

Specify as a scalar nonnegative integer the font size in points. When you choose 0, the value of the `DisplayChannelFontSize` property in the Logic Analyzer is used.

Example: `'FontSize',8`

Data Types: `double`

### **Format — Display format for the wave**

`'Default'` (default) | `'Analog'` | `'Digital'`

When you choose `'Default'`, the value of the `DisplayChannelFormat` property in the Logic Analyzer is used.

Example: `'Format','Digital'`

Data Types: `char` | `string`

### **Height — Height of the wave**

0 (default) | scalar integer

Specify as a scalar integer the height of the wave in the display in units of 16 pixels. When you choose 0, the value of the `DisplayChannelHeight` property in the Logic Analyzer is used.

Example: `'Height',2`

Data Types: `double`

### **InputChannel — Input channel that corresponds to this wave**

1 (default) | scalar integer in the range (1,NumInputPorts)

This property specifies the input channel whose data is used for this wave. By default, it will connect the first input to this wave.

Example: `'InputChannel',2`

Data Types: `double` | `single` | `uint8` | `uint16` | `uint32` | `uint64` | `int8` | `int16` | `int32` | `int64`

### **Name — Name or label for the wave**

`''` (default) | character vector | string scalar

Specify the name that you would like to set for the new wave.

Example: `'Name','MyWave'`

Data Types: `char` | `string`

### **Radix — Radix for the wave**

`'Default'` (default) | `'Binary'` | `'Hexadecimal'` | `'Octal'` | `'Signed decimal'` | `'Unsigned decimal'`

When the input signals are of class `double`, `single`, or logical, you should not set this property. When you choose `'Default'`, the value of the `DisplayChannelRadix` property in the Logic Analyzer is used.

Data Types: `char` | `string`

### **Divider Name-Value Pair Arguments**

Specify optional pairs of arguments as `Name1=Value1, . . . ,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose `Name` in quotes.*

Example: `'DisplayChannel',2,'Name','MyDivider'` specifies that a divider should be added to display channel 2 and named "MyDivider".

### **DisplayChannel — Channel on the display that shows this divider**

`NumInputPorts` (default) | scalar numeric value in the range (1,NumInputPorts)

Specify as a scalar numeric value the display channel that shows this divider. By default, the divider is added to the end of the display.

Example: `'DisplayChannel',2`

Data Types: `double` | `single` | `uint8` | `uint16` | `uint32` | `uint64` | `int8` | `int16` | `int32` | `int64`

### **Height — Height of the divider**

`0` (default) | scalar integer

Specify, in pixels, the height of the divider as a scalar integer in the range 8-200. If you choose 0, the value of the `DisplayChannelHeight` property in the Logic Analyzer is used.

Example: `'Height',2`

Data Types: `double`

### **Name — The name or label for the divider**

`''` (default) | character vector | string scalar

Specify the name that you would like to set for the new divider.

Example: `'Name','MyDivider'`

Data Types: `char` | `string`

## **Version History**

**Introduced in R2013a**

### **See Also**

`dsp.LogicAnalyzer` | `addDivider` | `addWave` | `deleteDisplayChannel` | `getDisplayChannelInfo` | `getDisplayChannelTags` | `moveDisplayChannel`

# moveDisplayChannel

**Package:** dsp

Move position of Logic Analyzer display channel

## Syntax

```
moveDisplayChannel(scope,tag,'DisplayChannel',displayChannelValue)
```

## Description

`moveDisplayChannel(scope,tag,'DisplayChannel',displayChannelValue)` moves the display channel, either a wave or a divider, specified by the input `tag`, to the new location specified by the input `displayChannelValue`.

## Examples

### Manipulate Logic Analyzer Programmatically

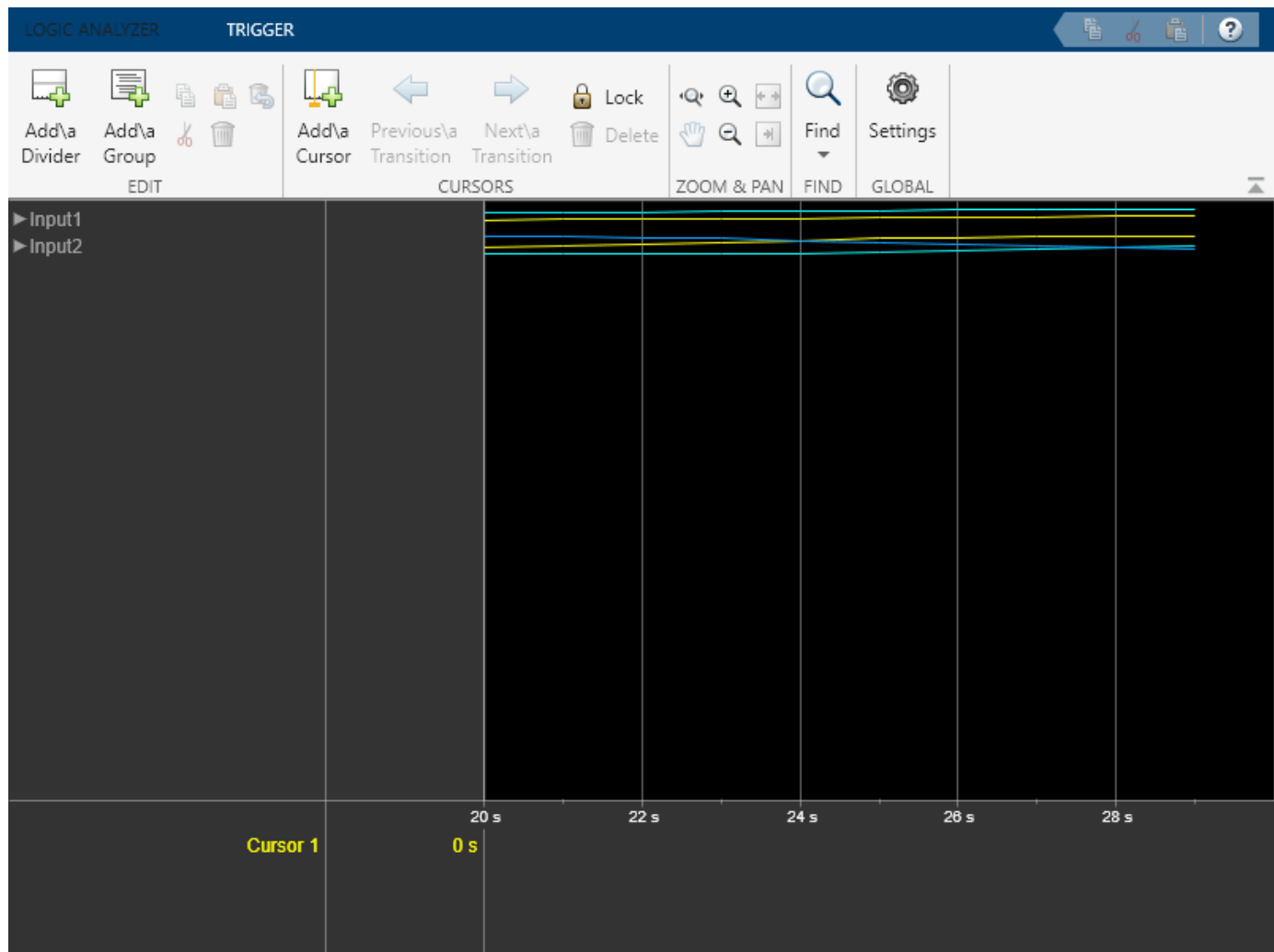
Use functions to construct and manipulate a `dsp.LogicAnalyzer` System object.

#### Display Waves on Logic Analyzer scope.

```
scope = dsp.LogicAnalyzer('NumInputPorts',2);

stop = 30;
for count = 1:stop
    sinValVec = sin(count/stop*2*pi);
    cosValVec = cos(count/stop*2*pi);
    cosValVecOffset = cos((count+10)/stop*2*pi);

    scope([count (count-(stop/2))],[sinValVec cosValVec cosValVecOffset])
end
```



### Reorganize Display

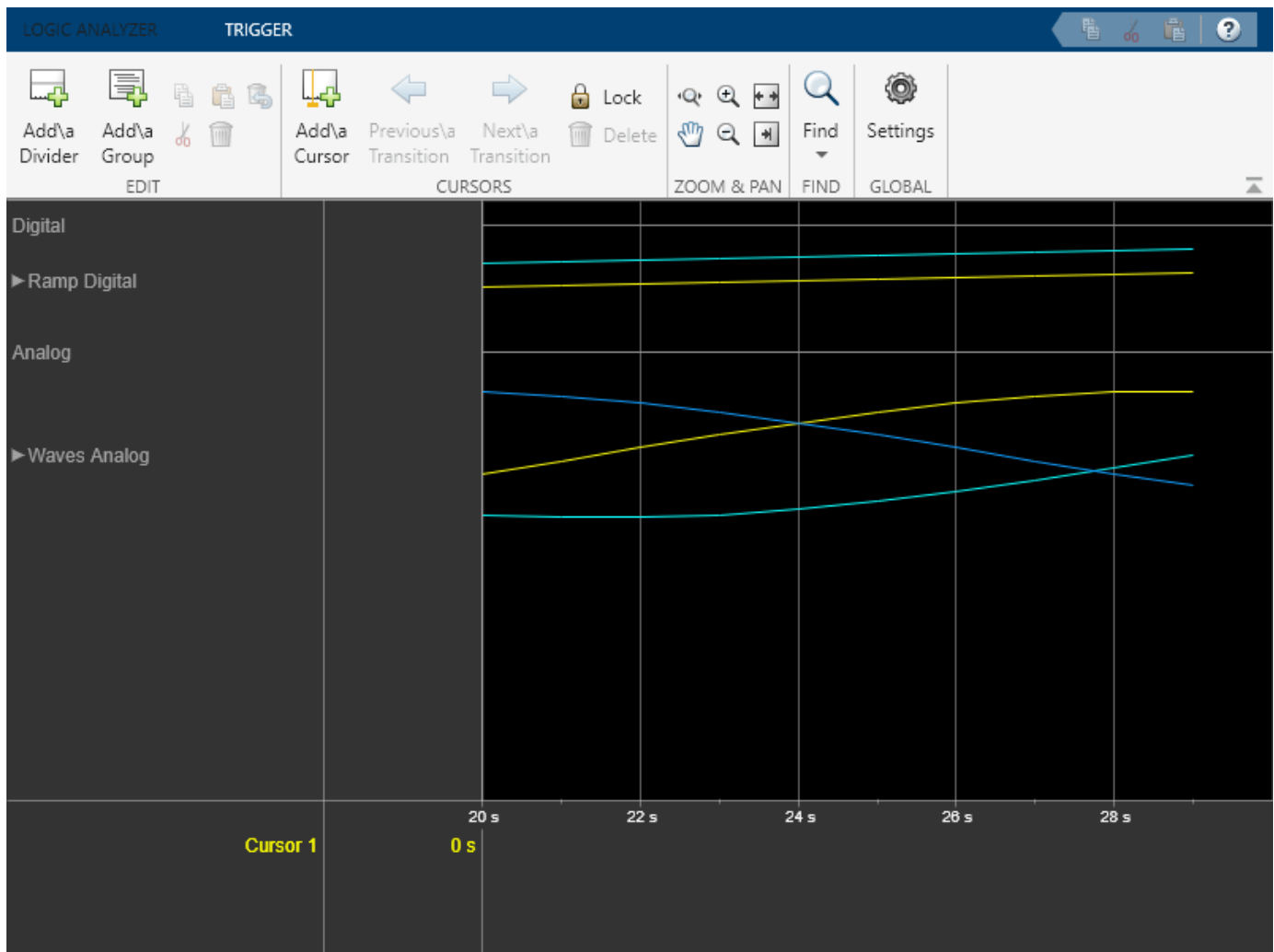
```
hide(scope)
digitalDividerTag = addDivider(scope, 'Name', 'Digital', 'Height', 20);
analogDividerTag = addDivider(scope, 'Name', 'Analog', 'Height', 40);

tags = getDisplayChannelTags(scope);

modifyDisplayChannel(scope, tags{1}, 'InputChannel', 1, ...
    'Name', 'Ramp Digital', 'Height', 40);
modifyDisplayChannel(scope, tags{2}, 'InputChannel', 2, ...
    'Name', 'Waves Analog', 'Format', 'Analog', 'Height', 80);

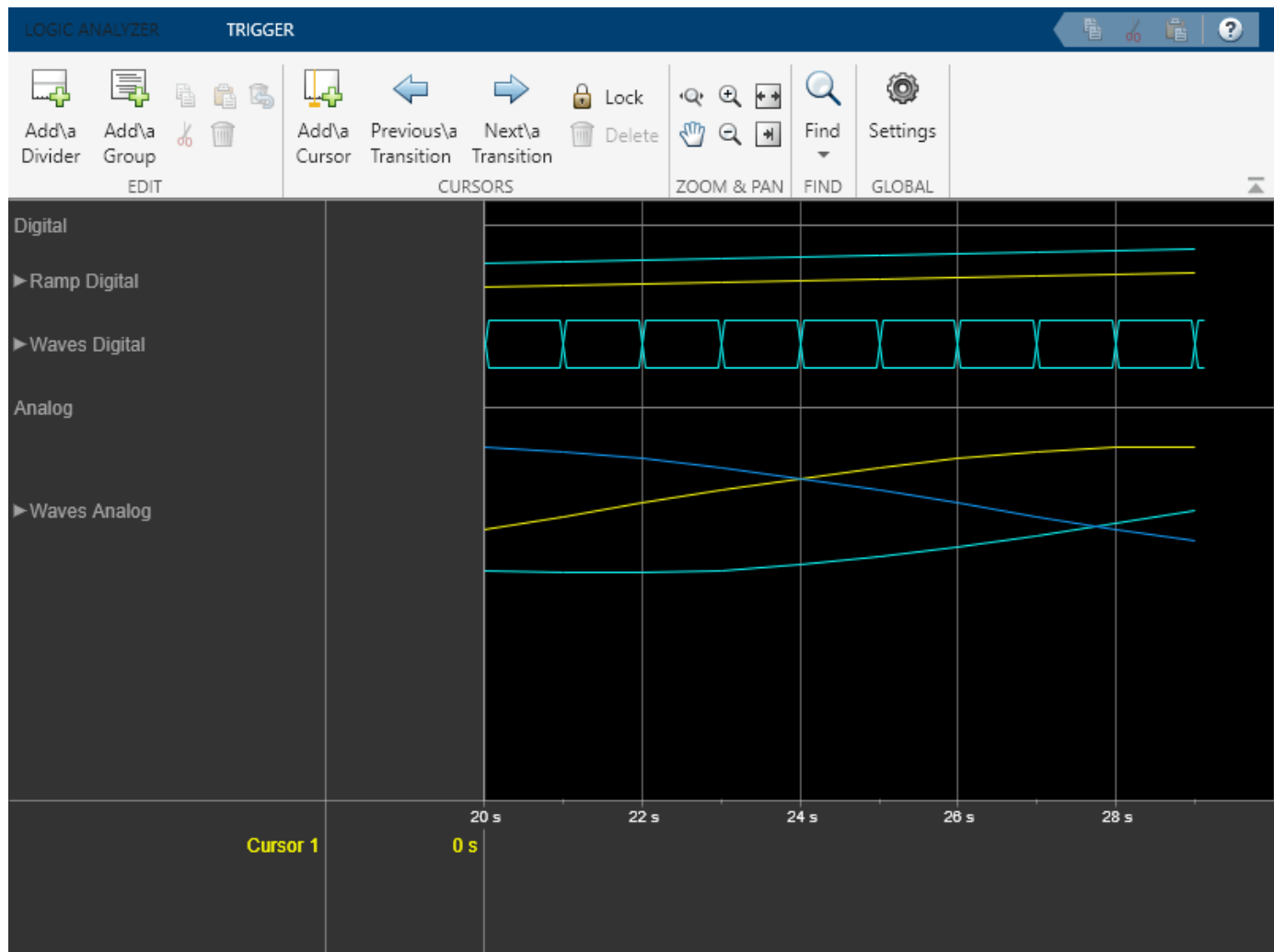
moveDisplayChannel(scope, digitalDividerTag, 'DisplayChannel', 1)
moveDisplayChannel(scope, tags{2}, 'DisplayChannel', length(tags))

show(scope)
```



### Duplicate Wave and Check Information

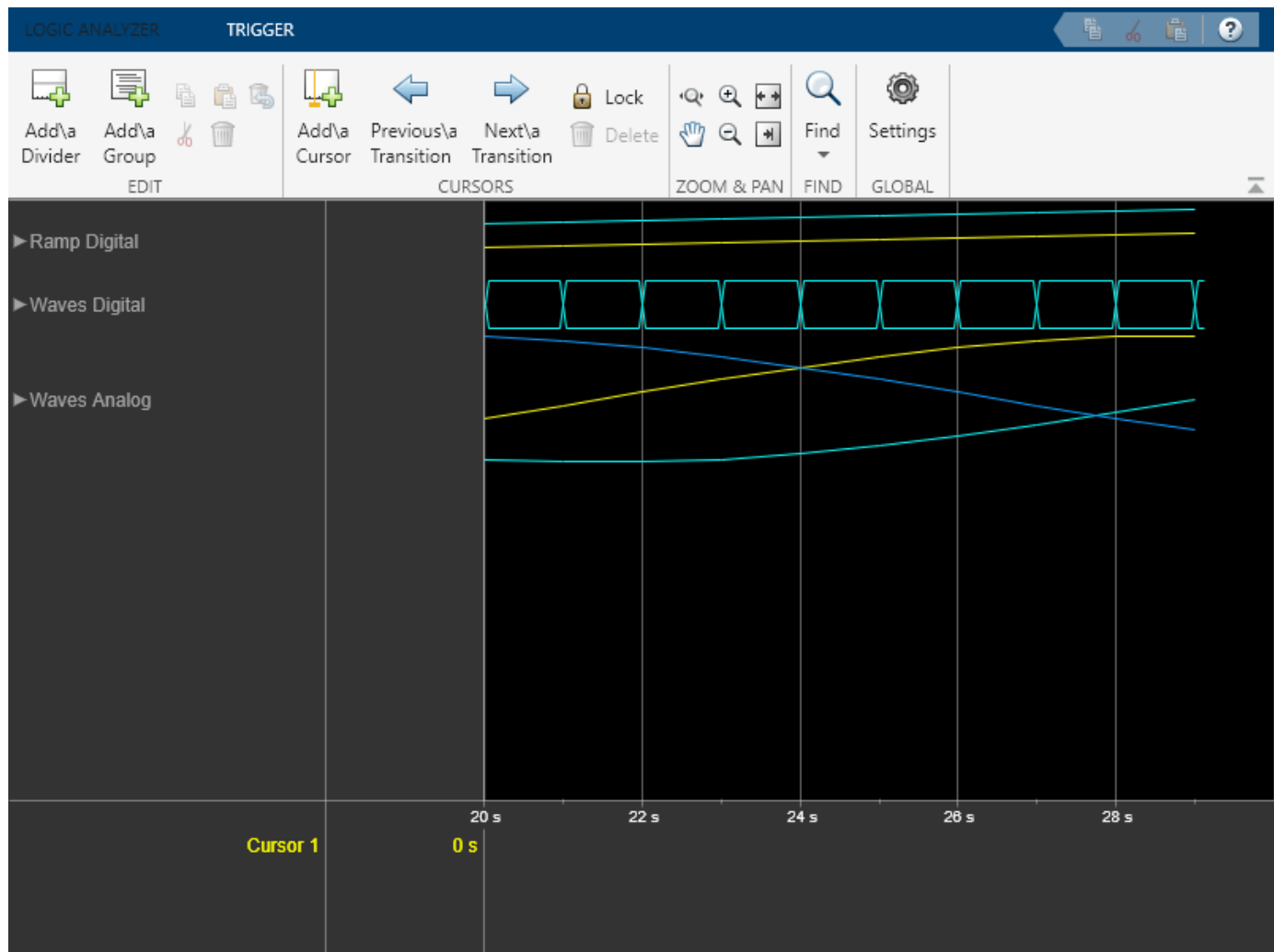
```
hide(scope)
addWave(scope, 'InputChannel', 2, 'Name', 'Waves Digital', 'Format', 'Digital', ...
'Height', 30, 'DisplayChannel', 3);
show(scope)
```



### Remove Dividers

```
hide(scope)
deleteDisplayChannel(scope,digitalDividerTag)
deleteDisplayChannel(scope,analogDividerTag)
show(scope)
```





### Clear variables

```
clear analogDividerTag cosValVec cosValVecOffset count digitalDividerTag duplicateWave scope sin
```

## Input Arguments

### scope — Logic Analyzer object

dsp.LogicAnalyzer object

The Logic Analyzer object in which you want to move a display channel, specified as a handle to the dsp.LogicAnalyzer object.

### tag — The tag identifying which display channel to move

character vector | string scalar

The tag identifying which display channel to move.

Example: 'W1'

Data Types: char | string

**displayChannelValue** – The location identifying where the display channel should be moved

scalar integer

The location identifying where the display channel should be moved, specified as a scalar integer.

Example: 'DisplayChannel',2

Data Types: double

## Version History

Introduced in R2013a

### See Also

`dsp.LogicAnalyzer` | `deleteDisplayChannel` | `getDisplayChannelInfo` | `getDisplayChannelTags` | `modifyDisplayChannel`

# msepred

**Package:** dsp

Predicted mean squared error for LMS adaptive filter

## Syntax

```
[mmseemse] = msepred(lmsFilt,x,d)
[mmseemse,meanw,mse,tracek] = msepred(lmsFilt,x,d)
[mmseemse,meanw,mse,tracek] = msepred(lmsFilt,x,d,m)
```

## Description

[mmseemse] = msepred(lmsFilt,x,d) predicts the steady-state values at convergence of the minimum mean squared error, mmse, and the excess mean squared error, emse, given the input and the desired response signal sequences in x and d and the quantities in the dsp.LMSFilter System object, lmsFilt.

[mmseemse,meanw,mse,tracek] = msepred(lmsFilt,x,d) also computes meanw, the sequence of means of the coefficient vectors, mse, sequence of mean-squared errors, and tracek, the sequence of total coefficient error powers.

[mmseemse,meanw,mse,tracek] = msepred(lmsFilt,x,d,m) specifies an optional decimation factor for computing meanw, mse, and tracek. If  $m > 1$ , every  $m^{\text{th}}$  predicted value of each of these sequences is saved. If omitted, the value of m is the default, which is one.

## Examples

### Predict Mean Squared Error for LMS Filter

The mean squared error (MSE) measures the average of the squares of the errors between the desired signal and the primary signal input to the adaptive filter. Reducing this error converges the primary input to the desired signal. Determine the predicted value of MSE and the simulated value of MSE at each time instant using the msepred and msesim functions. Compare these MSE values with each other and with respect to the minimum MSE and steady-state MSE values. In addition, compute the sum of the squares of the coefficient errors given by the trace of the coefficient covariance matrix.

### Initialization

Create a dsp.FIRFilter System object™ that represents the unknown system. Pass the signal, x, to the FIR filter. The output of the unknown system is the desired signal, d, which is the sum of the output of the unknown system (FIR filter) and an additive noise signal, n.

```
num = fir1(31,0.5);
fir = dsp.FIRFilter('Numerator',num);
iir = dsp.IIRFilter('Numerator',sqrt(0.75),...
    'Denominator',[1 -0.5]);
x = iir(sign(randn(2000,25)));
```

```
n = 0.1*randn(size(x));
d = fir(x) + n;
```

### LMS Filter

Create a `dsp.LMSFilter` System object to create a filter that adapts to output the desired signal. Set the length of the adaptive filter to 32 taps, step size to 0.008, and the decimation factor for analysis and simulation to 5. The variable `simmse` represents the simulated MSE between the output of the unknown system, `d`, and the output of the adaptive filter. The variable `mse` gives the corresponding predicted value.

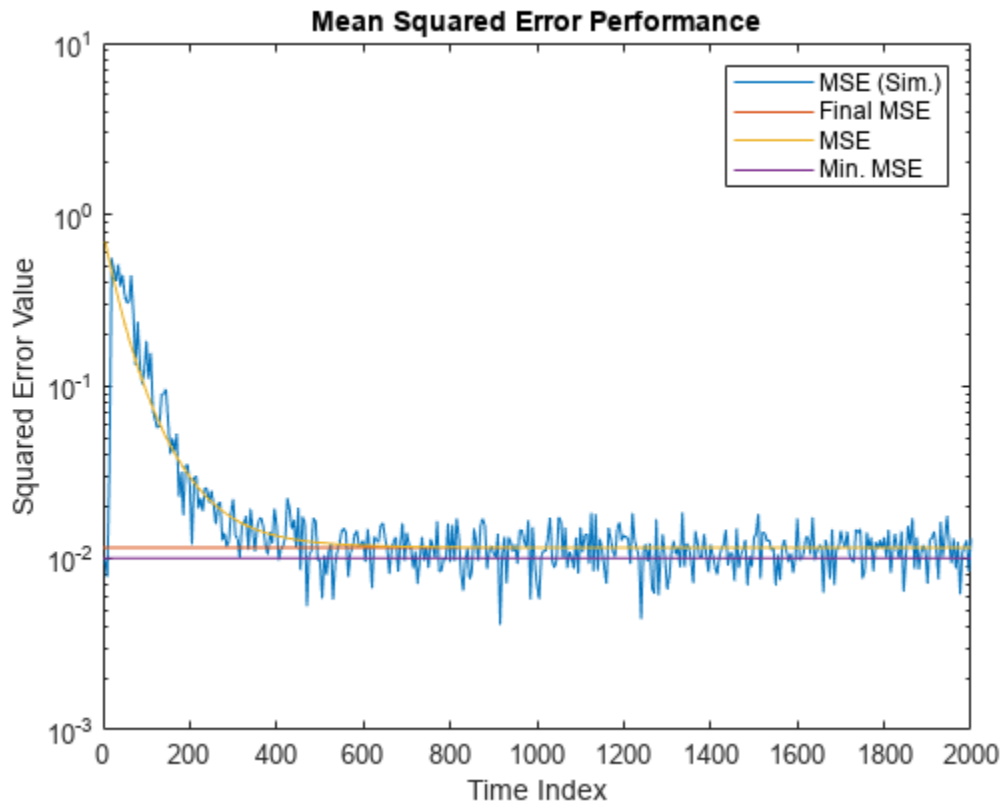
```
l = 32;
mu = 0.008;
m = 5;
```

```
lms = dsp.LMSFilter('Length',l,'StepSize',mu);
[mmse,emse,meanW,mse,traceK] = msepred(lms,x,d,m);
[simmse,meanWsim,Wsim,traceKsim] = msesim(lms,x,d,m);
```

### Plot the MSE Results

Compare the values of simulated MSE, predicted MSE, minimum MSE, and the final MSE. The final MSE value is given by the sum of minimum MSE and excess MSE.

```
nn = m:m:size(x,1);
semilogy(nn,simmse,[0 size(x,1)],[(emse+mmse)...
    (emse+mmse)],nn,mse,[0 size(x,1)],[mmse mmse])
title('Mean Squared Error Performance')
axis([0 size(x,1) 0.001 10])
legend('MSE (Sim.)','Final MSE','MSE','Min. MSE')
xlabel('Time Index')
ylabel('Squared Error Value')
```



The predicted MSE follows the same trajectory as the simulated MSE. Both these trajectories converge with the steady-state (final) MSE.

### Plot the Coefficient Trajectories

meanWsim is the mean value of the simulated coefficients given by msesim. meanW is the mean value of the predicted coefficients given by msepred.

Compare the simulated and predicted mean values of LMS filter coefficients 12,13,14, and 15.

```
plot(nn,meanWsim(:,12),'b',nn,meanW(:,12),'r',nn,...
meanWsim(:,13:15),'b',nn,meanW(:,13:15),'r')
PlotTitle ={'Average Coefficient Trajectories for';...
            'W(12), W(13), W(14), and W(15)'}

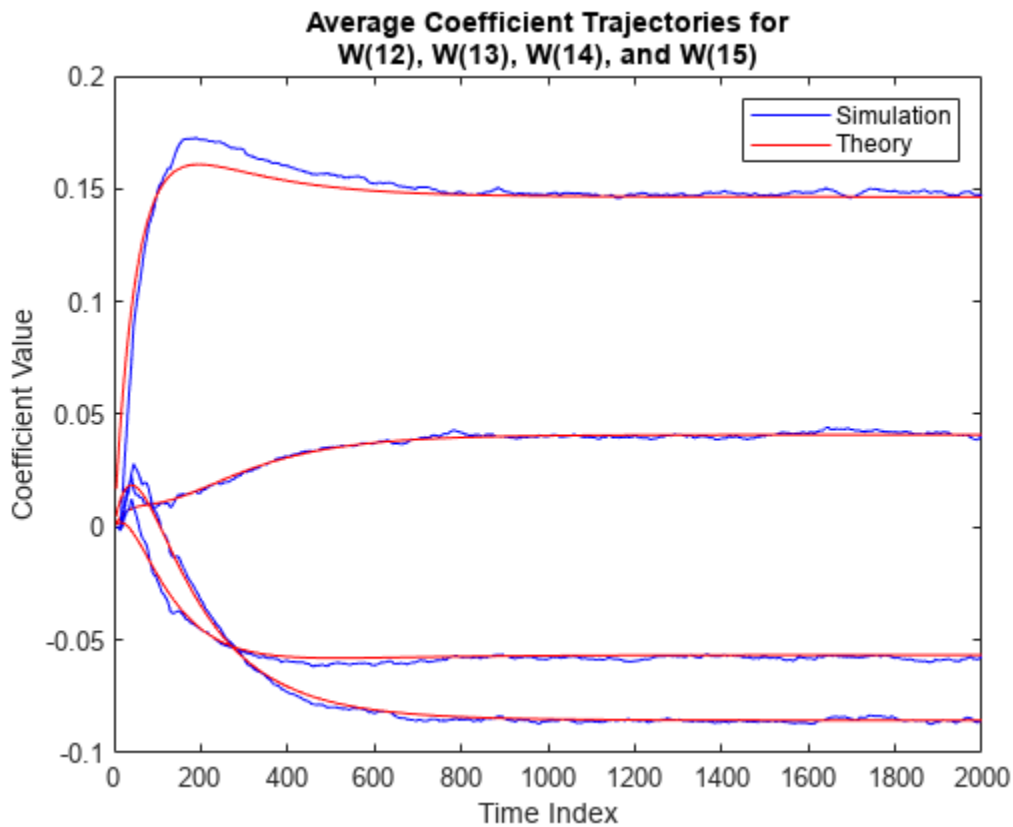
```

```
PlotTitle = 2x1 cell
            {'Average Coefficient Trajectories for'
            'W(12), W(13), W(14), and W(15)'}

```

```
title(PlotTitle)
legend('Simulation','Theory')
xlabel('Time Index')
ylabel('Coefficient Value')

```

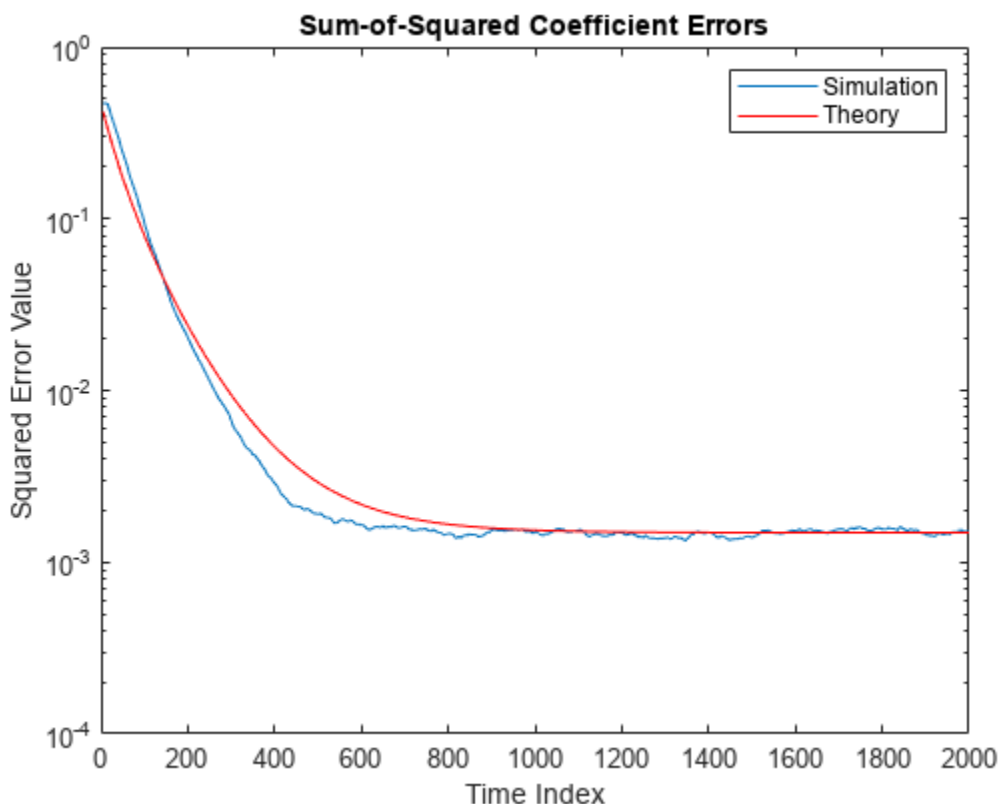


In steady state, both the trajectories converge.

### Sum of Squared Coefficient Errors

Compare the sum of the squared coefficient errors given by `msepred` and `msesim`. These values are given by the trace of the coefficient covariance matrix.

```
semilogy(nn,traceKsim,nn,traceK,'r')
title('Sum-of-Squared Coefficient Errors')
axis([0 size(x,1) 0.0001 1])
legend('Simulation','Theory')
xlabel('Time Index')
ylabel('Squared Error Value')
```



## Input Arguments

### **lmsFilt** — LMS adaptive filter System object

`dsp.LMSFilter`

LMS adaptive filter, specified as a `dsp.LMSFilter` System object.

### **x** — Input signal

scalar | column vector | matrix

Input signal, specified as a scalar, column vector, or matrix. Columns of the matrix `x` contain individual input signal sequences. The input, `x`, and the desired signal, `d`, must have the same size, data type, and complexity.

Data Types: `single` | `double`

### **d** — Desired signal

scalar | column vector | matrix

Desired response signal, specified as a scalar, column vector, or matrix. Columns of the matrix `d` contain individual desired signal sequences. The input, `x`, and the desired signal, `d`, must have the same size, data type, and complexity.

Data Types: `single` | `double`

### **m** — Decimation factor

1 (default) | positive scalar

Decimation factor, specified as a positive scalar. Every  $m$ th predicted value of the 3rd, 4th, and 5th predicted sequences output is saved into the corresponding output arguments, `meanw`, `mse`, and `tracek`. If  $m$  equals 1, every value of these sequences is saved.

$m$  must be a factor of the input frame size.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

## Output Arguments

### **mmse** — Minimum mean squared error

scalar

Minimum mean squared error (mmse), returned as a scalar. This parameter is estimated using a Wiener filter. The Wiener filter minimizes the mean squared error between the desired signal and the input signal filtered by the Wiener filter. A large value of the mean squared error indicates that the adaptive filter cannot accurately track the desired signal. The minimal value of the mean squared error ensures that the adaptive filter is optimal. The minimum mean squared error between a particular frame of the desired signal and the filtered signal is computed as the variance between the two frames of signals. The `msepred` function outputs the average of the mmse values for all the frames. For more details on how this parameter is calculated, see “Algorithms” on page 5-1273.

Data Types: `single` | `double`

### **emse** — Steady state excess mean squared error

scalar

Excess mean squared error, returned as a scalar. This error is the difference between the mean squared error introduced by adaptive filters and the minimum mean squared error produced by corresponding Wiener filter. For details on how this parameter is calculated, see “Algorithms” on page 5-1273.

Data Types: `single` | `double`

### **meanw** — Sequence of coefficient vector means

matrix

Sequence of coefficient vector means of the adaptive filter at each time instant, returned as a matrix. The columns of this matrix contain predictions of the mean values of the LMS adaptive filter coefficients at each time instant. If the decimation factor,  $m$  equals 1, the dimensions of `meanw` is  $M$ -by- $N$ .  $M$  is the frame size (number of rows) of the input signal,  $x$ .  $N$  is the length of the FIR filter weights vector, specified by the `Length` property of the `lmsFilt` System object. If  $m > 1$ , the dimensions of `meanw` is  $M/m$ -by- $N$ .

For details on how this parameter is calculated, see “Algorithms” on page 5-1273.

Data Types: `double`

### **mse** — Sequence of mean squared errors

column vector

Predictions of the mean squared error of the LMS adaptive filter at each time instant, returned as a column vector. If the decimation factor,  $m$  equals 1, the length of `mse` equals the frame size (number of rows) of the input signal,  $M$ . If  $m > 1$ , the length of `mse` equals  $M/m$ .



For details on how this parameter is calculated, see “Algorithms” on page 5-1273.

Data Types: `double`

### **tracek — Sequence of total coefficient error powers**

column vector

Predictions of the total coefficient error power of the LMS adaptive filter at each time instant, returned as a column vector. If the decimation factor,  $m$  equals 1, the length of `tracek` equals the frame size (number of rows) of the input signal, given by `size(x,1)`. If  $m > 1$ , the length of `tracek` equals the ratio of input frame size and the decimation factor,  $m$ .

For details on how this parameter is calculated, see “Algorithms” on page 5-1273.

Data Types: `double`

## **Algorithms**

### **Minimum Mean Squared Error (mmse)**

The `msepred` function computes the minimum mean-squared error (mmse) using the following equation:

$$mmse = \frac{\sum_{i=1}^N \text{var}(d_i - W(x_i))}{N}$$

where,

- $N$  -- Number of frames in the input signal,  $x$ .
- $d_i$  --  $i$ th frame (column) of the desired signal.
- $x_i$  --  $i$ th frame (column) of the input signal.
- $W(x_i)$  -- Output of the Wiener filter.
- `var` -- Variance

### **Excess Mean Squared Error (emse)**

The `msepred` function computes the steady-state excess mean squared error using the following equations:

$$\begin{aligned} emse &= K\lambda, \\ K &= \frac{B}{I(L) - A}, \\ B &= \mu^2 \cdot mmse \cdot \lambda', \\ A &= I(L) - 2\mu Lam + \mu^2(Lam^2(kurt + 2) + \lambda\lambda'). \end{aligned}$$

where,

- $K$  -- Final values of transformed coefficient variances.
- $\lambda$  -- Column vector containing the eigenvalues of the input autocorrelation matrix.
- $\lambda'$  -- Transpose of  $\lambda$ .

- $B$  -- MSE analysis driving term.
- $L$  -- Length of the FIR adaptive filter, given by `lmsFilt.Length`.
- $I(L)$  --  $L$ -by- $L$  identity matrix.
- $A$  -- MSE analysis transition matrix.
- $\mu$  -- Step size given by `lmsFilt.StepSize`.
- $Lam$  -- Diagonal matrix containing the eigenvalues.
- $kurt$  -- Average kurtosis value of eigenvector-filtered signals.

### Coefficient Vector Means (`meanw`)

The `msepred` function computes each element of the sequence of coefficient vector means using the following equations:

$$\begin{aligned} meanw &= meanw \cdot T \cdot D, \\ T &= I(L) - \mu R, \\ D &= \mu P. \end{aligned}$$

where,

- $meanw$  -- The initial value of  $meanw$  is given by `lmsFilt.InitialConditions`.
- $T$  -- Transition matrix for mean coefficient analysis.
- $I(L)$  --  $L$ -by- $L$  identity matrix.
- $\mu$  -- Step size given by `lmsFilt.StepSize`.
- $R$  -- Input autocorrelation matrix of size  $L$ -by- $L$ .
- $D$  -- Driving term for mean coefficient analysis.
- $P$  -- Cross correlation vector of size 1-by- $L$ .
- $kurt$  -- Average kurtosis value of eigenvector-filtered signals.

### Mean Squared Errors (`mse`)

The `msepred` function computes each element of the sequence of the mean squared errors using the following equations:

$$\begin{aligned} mse &= mmse + dk \cdot \lambda, \\ dk &= dk \circ diagA + mse \cdot D, \\ diagA &= (1 - 2\mu\lambda + \mu^2(kurt + 2)\lambda^2), \\ D &= \mu^2\lambda'. \end{aligned}$$

where,

- $mmse$  -- Minimum mean squared error.
- $dk$  -- Diagonal entries of coefficient covariance matrix. The initial value of  $dk$  is given by  $dk = ((meanw - W_{opt})Q)^2$ .
- $meanw$  -- Coefficient vector means given by `lmsFilt.InitialConditions`.
- $W_{opt}$  -- Optimal Wiener filter coefficients.
- $Q$  --  $L$ -by- $L$  matrix whose columns are the eigenvectors of the input autocorrelation matrix,  $R$ , so that  $RQ = QLam$ .  $Lam$  is the diagonal matrix containing the corresponding eigen values.

- *diagA* -- Portion of MSE analysis transition matrix.
- *dk* ◦ *diagA* -- Hadamard or entrywise product of *dk* and *diagA*.
- $\lambda$  -- Column vector containing the eigenvalues of the input autocorrelation matrix, *R*.
- $\mu$  -- Step size given by `lmsFilt.StepSize`.
- *kurt* -- Average kurtosis value of eigenvector filtered signals.
- $\lambda^2$  -- Hadamard or entrywise power of the column vector containing the eigenvalues.
- *D* -- Driving term for MSE analysis.

### Total Coefficient Error Powers (tracek)

The `msepred` function computes each element of the sequence of the total coefficient error powers. These values are given by the trace of the coefficient covariance matrix. The diagonal entries of the coefficient covariance matrix are given by *dk* in the following equations:

$$\begin{aligned}mse &= mmse + dk \cdot \lambda, \\dk &= dk \circ diagA + mse \cdot D, \\diagA &= (1 - 2\mu\lambda + \mu^2(kurt + 2)\lambda^2), \\D &= \mu^2\lambda' .\end{aligned}$$

The trace of the coefficient covariance matrix is given by the following equation:

$$tracek = sum(dk).$$

## Version History

Introduced in R2012a

## References

- [1] Hayes, M.H. *Statistical Digital Signal Processing and Modeling*. New York: John Wiley & Sons, 1996.

## See Also

### Functions

`maxstep` | `msesim`

### Objects

`dsp.LMSFilter`

### Topics

“Overview of Adaptive Filters and Applications”  
 “Signal Enhancement Using LMS and NLMS Algorithms”

## msesim

**Package:** dsp

Estimated mean squared error for adaptive filters

### Syntax

```
mse = msesim(adaptFilt,x,d)
[mse,meanw,w,tracek] = msesim(adaptFilt,x,d)
[ ___ ] = msesim(adaptFilt,x,m)
```

### Description

`mse = msesim(adaptFilt,x,d)` estimates the mean squared error of the adaptive filter at each time instant given the input and the desired response signal sequences in `x` and `d`.

`[mse,meanw,w,tracek] = msesim(adaptFilt,x,d)` also calculates the sequences of coefficient vector means, `meanw`, adaptive filter coefficients, `w`, and the total coefficient error powers, `tracek`, corresponding to the simulated behavior of the adaptive filter.

`[ ___ ] = msesim(adaptFilt,x,m)` specifies an optional decimation factor for computing `mse`, `meanw`, and `tracek`. If `m > 1`, every `m`th value of each of these sequences is saved. If omitted, the value of `m` defaults to 1.

### Examples

#### Predict Mean Squared Error for LMS Filter

The mean squared error (MSE) measures the average of the squares of the errors between the desired signal and the primary signal input to the adaptive filter. Reducing this error converges the primary input to the desired signal. Determine the predicted value of MSE and the simulated value of MSE at each time instant using the `msepred` and `msesim` functions. Compare these MSE values with each other and with respect to the minimum MSE and steady-state MSE values. In addition, compute the sum of the squares of the coefficient errors given by the trace of the coefficient covariance matrix.

#### Initialization

Create a `dsp.FIRFilter` System object™ that represents the unknown system. Pass the signal, `x`, to the FIR filter. The output of the unknown system is the desired signal, `d`, which is the sum of the output of the unknown system (FIR filter) and an additive noise signal, `n`.

```
num = fir1(31,0.5);
fir = dsp.FIRFilter('Numerator',num);
iir = dsp.IIRFilter('Numerator',sqrt(0.75),...
    'Denominator',[1 -0.5]);
x = iir(sign(randn(2000,25)));
n = 0.1*randn(size(x));
d = fir(x) + n;
```

## LMS Filter

Create a `dsp.LMSFilter` System object to create a filter that adapts to output the desired signal. Set the length of the adaptive filter to 32 taps, step size to 0.008, and the decimation factor for analysis and simulation to 5. The variable `simmse` represents the simulated MSE between the output of the unknown system, `d`, and the output of the adaptive filter. The variable `mse` gives the corresponding predicted value.

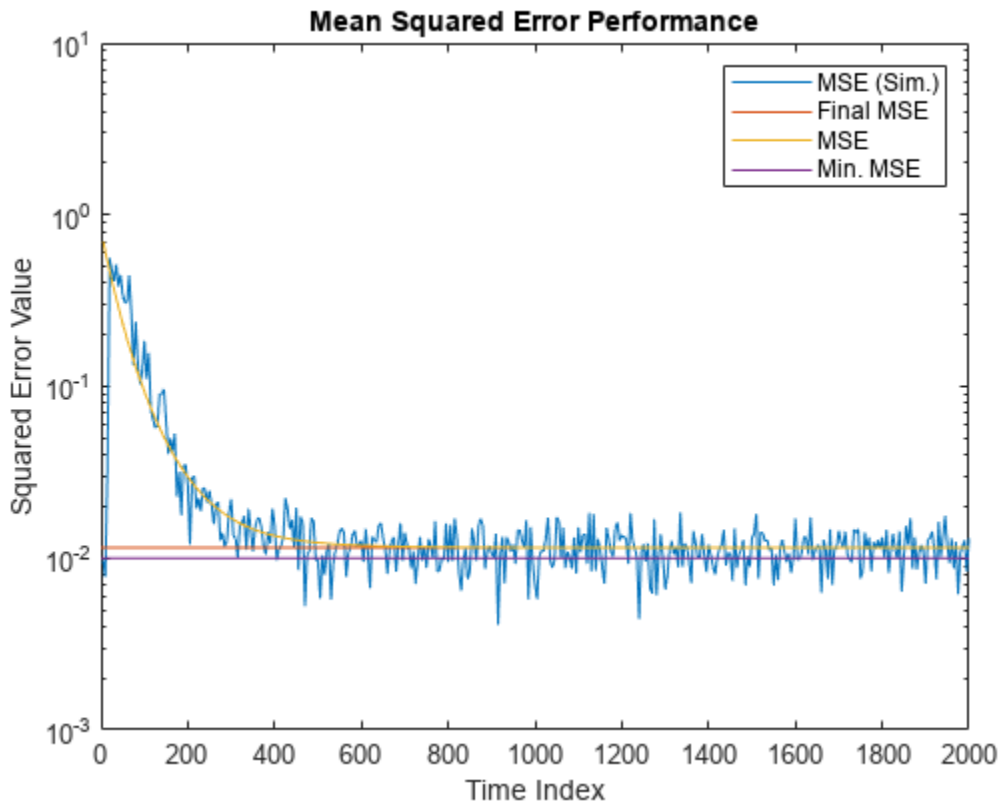
```
l = 32;
mu = 0.008;
m = 5;

lms = dsp.LMSFilter('Length',l,'StepSize',mu);
[mmse,emse,meanW,mse,traceK] = msepred(lms,x,d,m);
[simmse,meanWsim,Wsim,traceKsim] = msesim(lms,x,d,m);
```

## Plot the MSE Results

Compare the values of simulated MSE, predicted MSE, minimum MSE, and the final MSE. The final MSE value is given by the sum of minimum MSE and excess MSE.

```
nn = m:m:size(x,1);
semilogy(nn,simmse,[0 size(x,1)],[(emse+mmse)...
    (emse+mmse)],nn,mse,[0 size(x,1)],[mmse mmse])
title('Mean Squared Error Performance')
axis([0 size(x,1) 0.001 10])
legend('MSE (Sim.)','Final MSE','MSE','Min. MSE')
xlabel('Time Index')
ylabel('Squared Error Value')
```



The predicted MSE follows the same trajectory as the simulated MSE. Both these trajectories converge with the steady-state (final) MSE.

### Plot the Coefficient Trajectories

`meanWsim` is the mean value of the simulated coefficients given by `mseSim`. `meanW` is the mean value of the predicted coefficients given by `msePred`.

Compare the simulated and predicted mean values of LMS filter coefficients 12,13,14, and 15.

```
plot(nn,meanWsim(:,12),'b',nn,meanW(:,12),'r',nn,...
meanWsim(:,13:15),'b',nn,meanW(:,13:15),'r')
PlotTitle ={'Average Coefficient Trajectories for';...
            'W(12), W(13), W(14), and W(15)'}

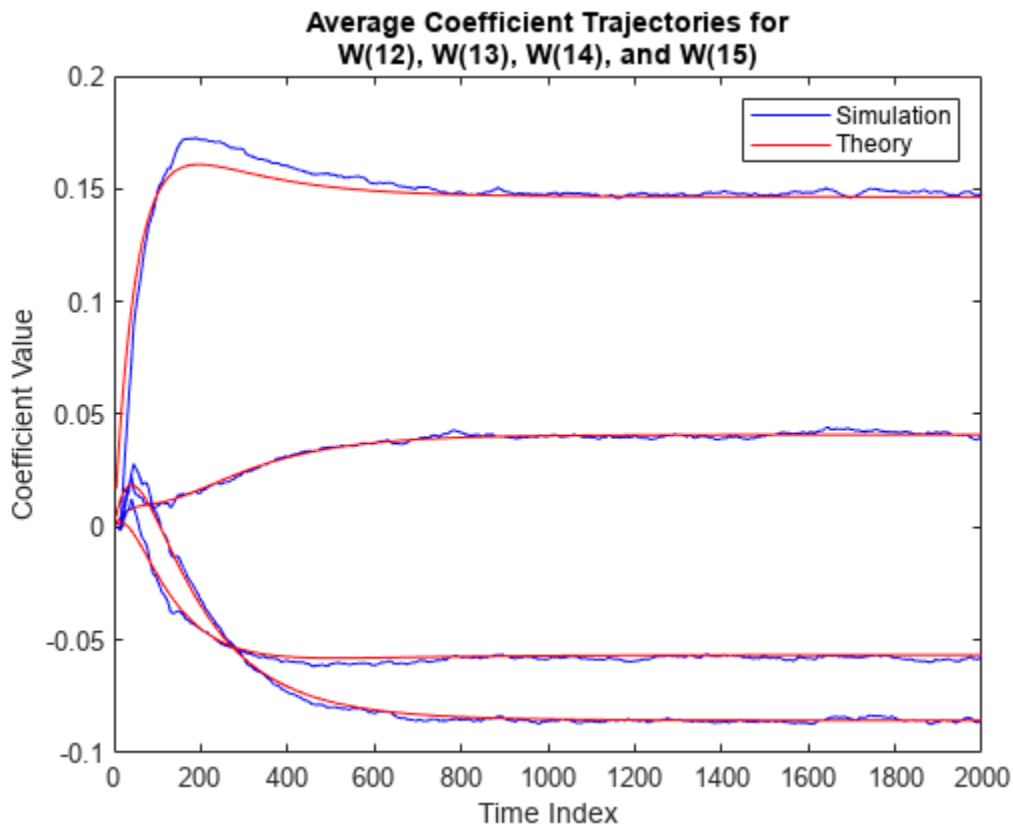
```

```
PlotTitle = 2x1 cell
            {'Average Coefficient Trajectories for'}
            {'W(12), W(13), W(14), and W(15)'}

```

```
title(PlotTitle)
legend('Simulation','Theory')
xlabel('Time Index')
ylabel('Coefficient Value')

```

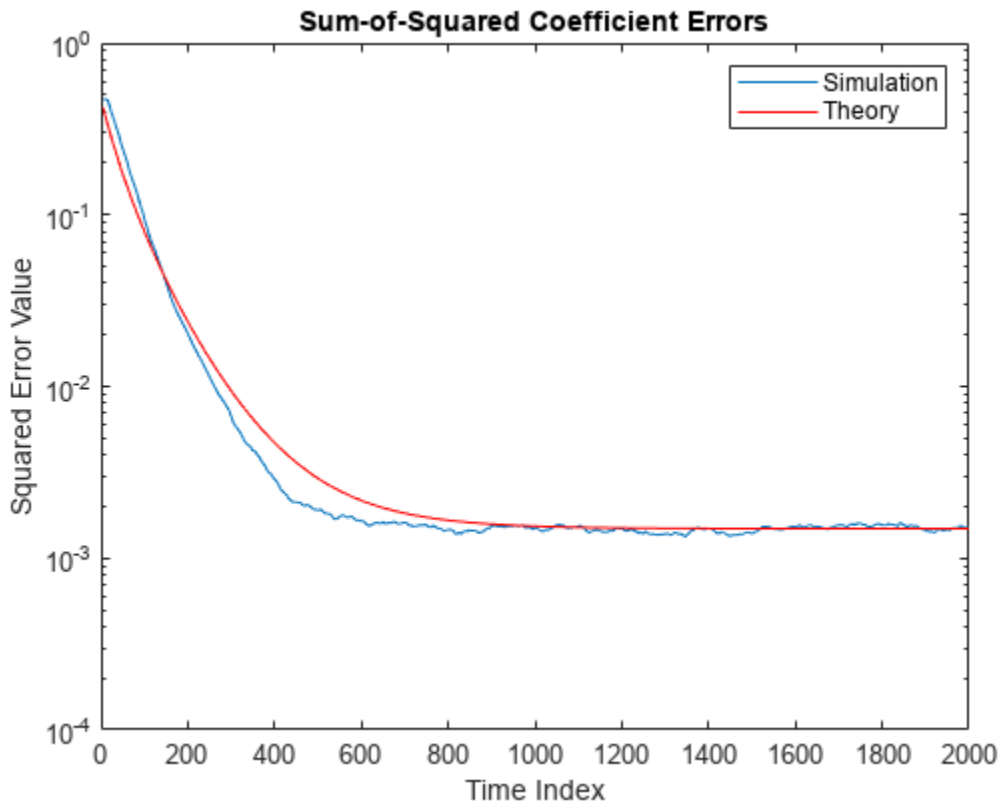


In steady state, both the trajectories converge.

### Sum of Squared Coefficient Errors

Compare the sum of the squared coefficient errors given by `msepred` and `mresim`. These values are given by the trace of the coefficient covariance matrix.

```
semilogy(nn,traceKsim,nn,traceK,'r')
title('Sum-of-Squared Coefficient Errors')
axis([0 size(x,1) 0.0001 1])
legend('Simulation','Theory')
xlabel('Time Index')
ylabel('Squared Error Value')
```



### System Identification of FIR Filter Using Filtered XLMS Filter

Identify an unknown system by performing active noise control using a filtered-x LMS algorithm. The objective of the adaptive filter is to minimize the error signal between the output of the adaptive filter and the output of the unknown system (or the system to be identified). Once the error signal is minimal, the unknown system converges to the adaptive filter.

#### Initialization

Create a `dsp.FIRFilter` System object that represents the system to be identified. Pass the signal, `x`, to the FIR filter. The output of the unknown system is the desired signal, `d`, which is the sum of the output of the unknown system (FIR filter) and an additive noise signal, `n`.

```
num = fir1(31,0.5);
fir = dsp.FIRFilter('Numerator',num);
iir = dsp.IIRFilter('Numerator',sqrt(0.75),...
    'Denominator',[1 -0.5]);
x = iir(sign(randn(2000,25)));
n = 0.1*randn(size(x));
d = fir(x) + n;
```

#### Adaptive Filter

Create a `dsp.FilteredXLMSFilter` System object to create an adaptive filter that uses the filtered-x LMS algorithm. Set the length of the adaptive filter to 32 taps, step size to 0.008, and the

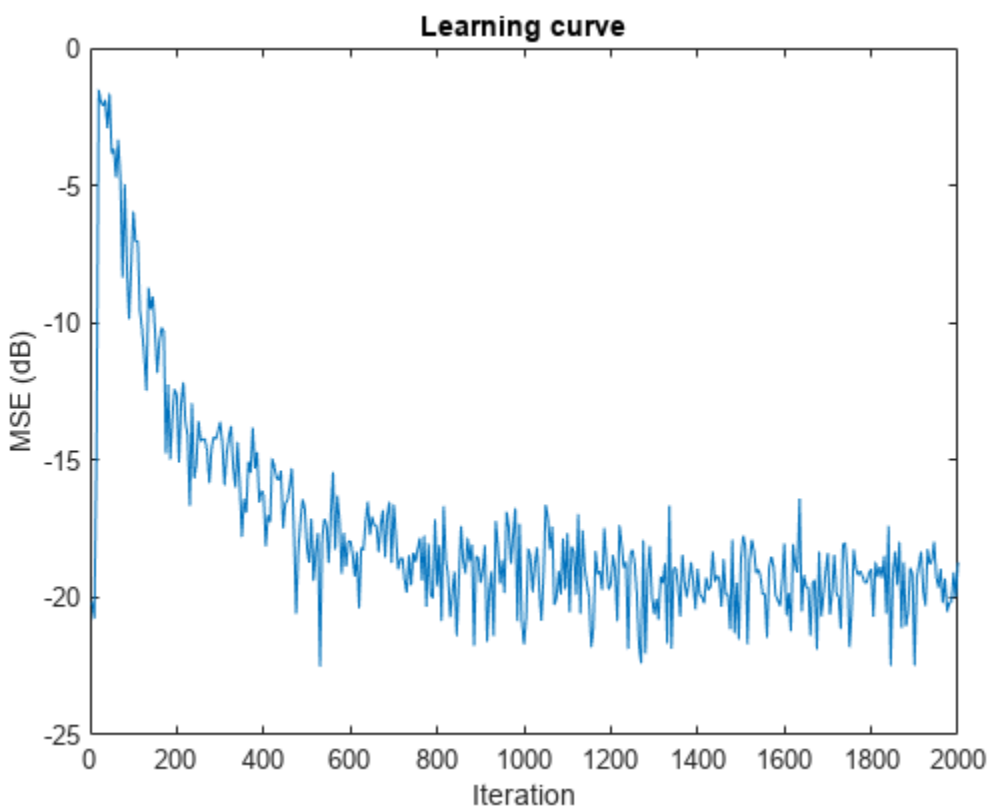


decimation factor for analysis and simulation to 5. The variable `simmse` represents the error between the output of the unknown system, `d`, and the output of the adaptive filter.

```

l = 32;
mu = 0.008;
m = 5;
fxlms = dsp.FilteredXLMSFilter(l,'StepSize',mu);
[simmse,meanWsim,Wsim,traceKsim] = mlesim(fxlms,x,d,m);
plot(m*(1:length(simmse)),10*log10(simmse))
xlabel('Iteration')
ylabel('MSE (dB)')
% Plot the learning curve for filtered-x LMS filter
% used in system identification
title('Learning curve')

```



With each iteration of adaptation, the value of `simmse` decreases to a minimal value, indicating that the unknown system has converged to the adaptive filter.

### System Identification of FIR Filter Using Adaptive Lattice Filter

```

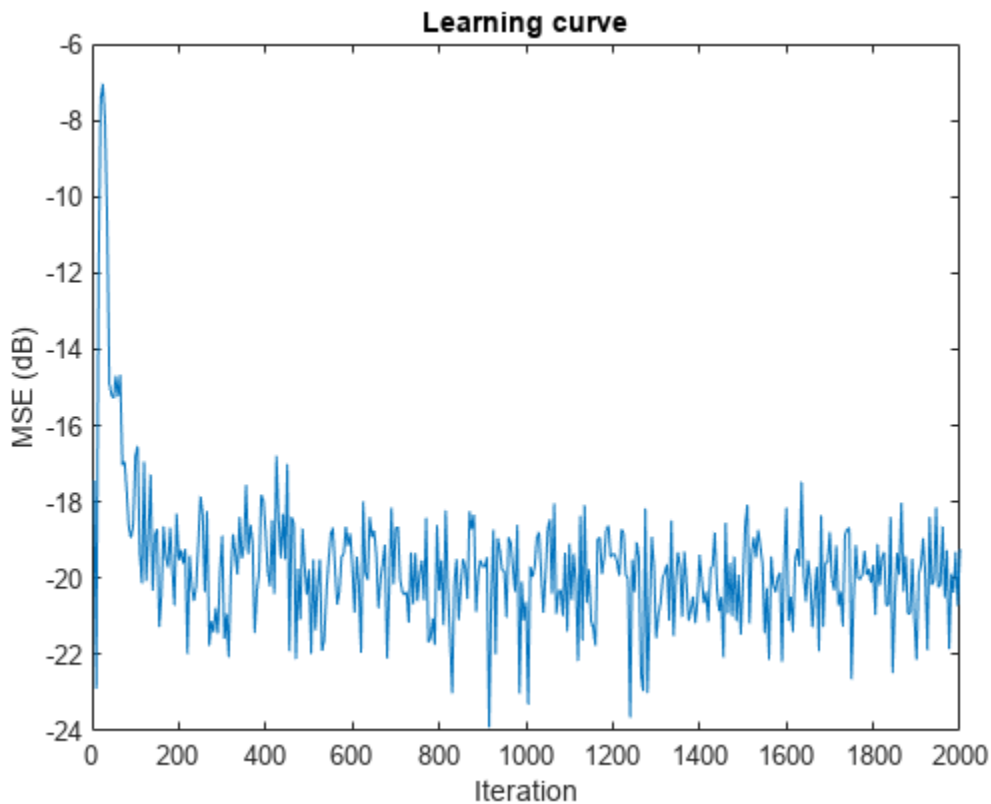
ha = fir1(31,0.5);
% FIR system to be identified
fir = dsp.FIRFilter('Numerator',ha);
iir = dsp.IIRFilter('Numerator',sqrt(0.75),...
'Denominator',[1 -0.5]);
x = iir(sign(randn(2000,25)));

```

```

% Observation noise signal
n = 0.1*randn(size(x));
% Desired signal
d = fir(x)+n;
% Filter length
l = 32;
% Decimation factor for analysis
% and simulation results
m = 5;
ha = dsp.AdaptiveLatticeFilter(l);
[simmse,meanWsim,Wsim,traceKsim] = msesim(ha,x,d,m);
plot(m*(1:length(simmse)),10*log10(simmse));
xlabel('Iteration');
ylabel('MSE (dB)');
% Plot the learning curve used for
% adaptive lattice filter used in system identification
title('Learning curve')

```



### System Identification of FIR Filter Using Block LMS Filter

```

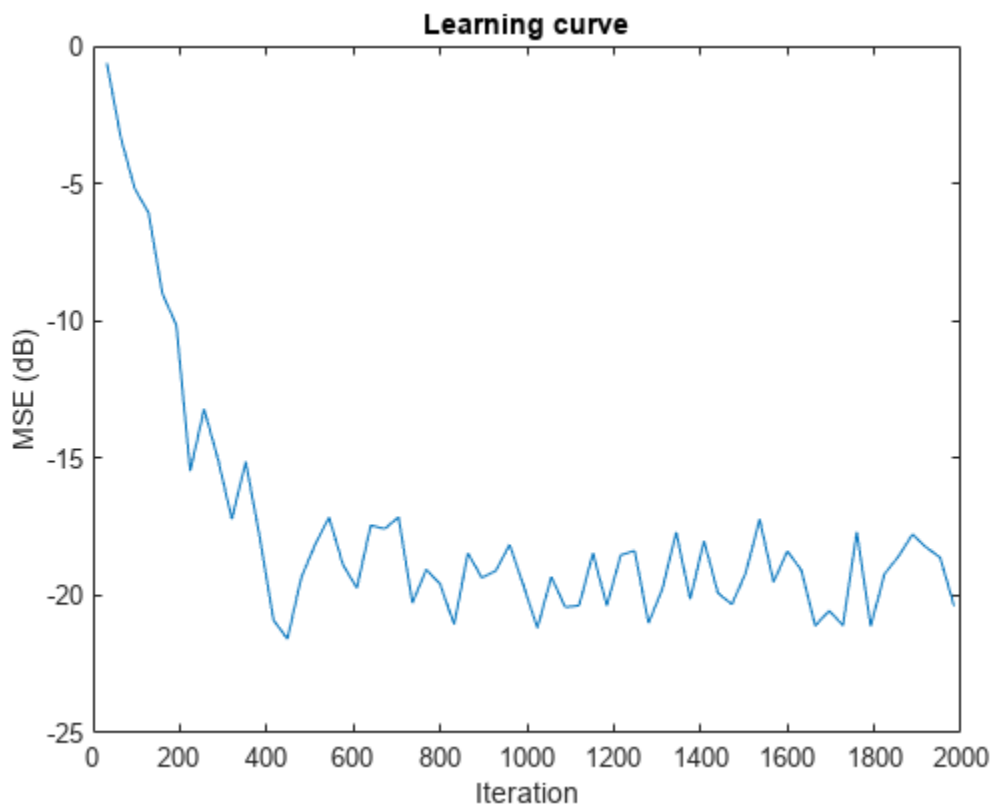
fir = fir1(31,0.5);
% FIR system to be identified
firFilter = dsp.FIRFilter('Numerator',fir);
iirFilter = dsp.IIRFilter('Numerator',sqrt(0.75),...
    'Denominator',[1 -0.5]);

```

```

x = iirFilter(sign(randn(2000,25)));
% Observation noise signal
n = 0.1*randn(size(x));
% Desired signal
d = firFilter(x)+n;
% Filter length
l = 32;
% Block LMS Step size
mu = 0.008;
% Decimation factor for analysis
% and simulation results
m = 32;
fir = dsp.BlockLMSFilter(l,'StepSize',mu);
[simmse,meanWsim,Wsim,traceKsim] = mlesim(fir,x,d,m);
plot(m*(1:length(simmse)),10*log10(simmse));
xlabel('Iteration'); ylabel('MSE (dB)');
% Plot the learning curve for
% block LMS filter used in system identification
title('Learning curve')

```



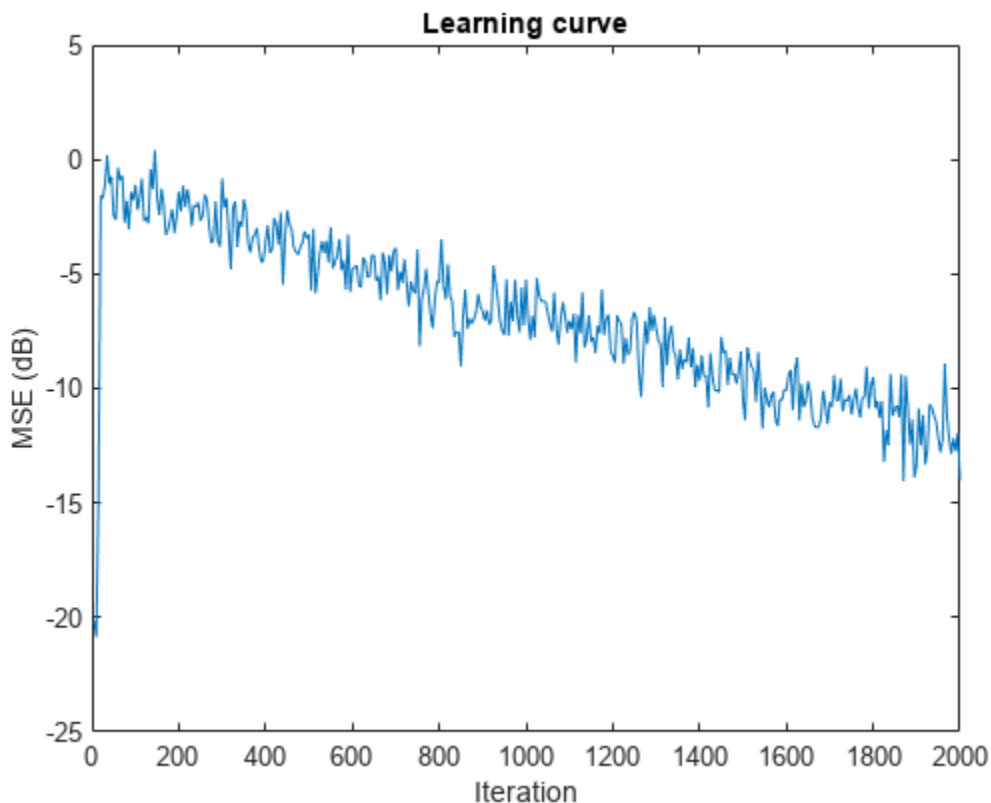
### System Identification of FIR Filter Using Affine Projection Filter

```

ha = fir1(31,0.5);
% FIR system to be identified
fir = dsp.FIRFilter('Numerator',ha);

```

```
iir = dsp.IIRFilter('Numerator',sqrt(0.75),...  
    'Denominator',[1 -0.5]);  
x = iir(sign(randn(2000,25)));  
% Observation noise signal  
n = 0.1*randn(size(x));  
% Desired signal  
d = fir(x)+n;  
% Filter length  
l = 32;  
% Affine Projection filter Step size.  
mu = 0.008;  
% Decimation factor for analysis  
% and simulation results  
m = 5;  
  
apf = dsp.AffineProjectionFilter(l, 'StepSize',mu);  
[simmse,meanWsim,Wsim,traceKsim] = msesim(apf,x,d,m);  
plot(m*(1:length(simmse)),10*log10(simmse));  
xlabel('Iteration'); ylabel('MSE (dB)');  
% Plot the learning curve for affine projection filter  
% used in system identification  
title('Learning curve')
```



## Input Arguments

### **adaptFilt** — Adaptive filter System object

adaptive filter System object

Adaptive filter, specified as one of the following System objects:

- `dsp.LMSFilter`
- `dsp.BlockLMSFilter`
- `dsp.AdaptiveLatticeFilter`
- `dsp.AffineProjectionFilter`
- `dsp.FastTransversalFilter`
- `dsp.FilteredXLMSFilter`
- `dsp.RLSFilter`

### **x** — Input signal

scalar | column vector | matrix

Input signal, specified as a scalar, column vector, or matrix. Columns of the matrix `x` contain individual input signal sequences. The input, `x`, and the desired signal, `d`, must have the same size and data type.

If `adaptFilt` is a `dsp.BlockLMSFilter` object, the input signal frame size must be greater than or equal to the value you specify in the `BlockSize` property of the object.

Data Types: `single` | `double`

### **d** — Desired signal

scalar | column vector | matrix

Desired response signal, specified as a scalar, column vector, or matrix. Columns of the matrix `d` contain individual desired signal sequences. The input, `x`, and the desired signal, `d`, must have the same size and the data type.

If `adaptFilt` is a `dsp.BlockLMSFilter` System object, the desired signal frame size must be greater than or equal to the value you specify in the `BlockSize` property of the object.

Data Types: `single` | `double`

### **m** — Decimation factor

1 (default) | positive scalar

Decimation factor, specified as a positive scalar. Every `m`th value of the estimated sequences is saved into the corresponding output arguments, `mse`, `meanw`, `w`, and `tracek`. If `m` equals 1, every value of these sequences is saved.

If `adaptFilt` is a `dsp.BlockLMSFilter` System object, the decimation factor must be a multiple of the value you specify in the `BlockSize` property of the object.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

## Output Arguments

### mse — Sequence of mean squared errors

column vector

Estimates of the mean squared error of the adaptive filter at each time instant, returned as a column vector.

If the adaptive filter is `dsp.BlockLMSFilter` and the decimation factor `m` is specified, length of `mse` equals  $\text{floor}(M/m)$ .  $M$  is the frame size (number of rows) of the input signal, `x`. If `m` is not specified, the length of `mse` equals  $\text{floor}(M/B)$ , where  $B$  is the value you specify in the `BlockSize` property of the object. The input signal frame size must be greater than or equal to the value you specify in the `BlockSize` property of the object. The decimation factor, if specified, must be a multiple of the `BlockSize` property.

For the other adaptive filters, if the decimation factor,  $m = 1$ , the length of `mse` equals the frame size of the input signal. If  $m > 1$ , the length of `mse` equals  $\text{floor}(M/m)$ .

Data Types: `double`

### meanw — Sequence of coefficient vector means

matrix

Sequence of coefficient vector means of the adaptive filter at each time instant, estimated as a matrix. The columns of this matrix contain estimates of the mean values of the adaptive filter coefficients at each time instant.

If the adaptive filter is `dsp.BlockLMSFilter` and the decimation factor `m` is specified, the dimensions of `meanw` is  $\text{floor}(M/m)\text{-by-}N$ .  $M$  is the frame size (number of rows) of the input signal, `x`.  $N$  is the length of the filter weights vector, specified by the `Length` property of the adaptive filter. If `m` is not specified, the dimensions of `meanw` is  $\text{floor}(M/B)\text{-by-}N$ , where  $B$  is the value you specify in the `BlockSize` property of the object. The input signal frame size must be greater than or equal to the value you specify in the `BlockSize` property of the object. The decimation factor, if specified, must be a multiple of the `BlockSize` property.

For the other adaptive filters, If the decimation factor,  $m = 1$ , the dimensions of `meanw` is  $M\text{-by-}N$ . If  $m > 1$ , the dimensions of `meanw` is  $\text{floor}(M/m)\text{-by-}N$ .

Data Types: `double`

### w — Final values of adaptive filter coefficients

row vector

Final values of the adaptive filter coefficients for the algorithm corresponding to `adaptFilt`, returned as a row vector. The length of the row vector equals the value you specify in the `Length` property of the object.

Data Types: `single` | `double`

### tracek — Sequence of total coefficient error powers

column vector

Sequence of total coefficient error powers, estimated as a column vector. This column vector contains estimates of the total coefficient error power of the adaptive filter at each time instant.

If the adaptive filter is `dsp.BlockLMSFilter` and the decimation factor `m` is specified, length of `tracek` equals  $\text{floor}(M/m)$ .  $M$  is the frame size (number of rows) of the input signal, `x`. If `m` is not

specified, the length of `tracek` equals  $\text{floor}(M/B)$ , where  $B$  is the value you specify in the `BlockSize` property of the object. The input signal frame size must be greater than or equal to the value you specify in the `BlockSize` property of the object. The decimation factor, if specified, must be a multiple of the `BlockSize` property.

For the other adaptive filters, if the decimation factor,  $m = 1$ , the length of `tracek` equals the frame size of the input signal. If  $m > 1$ , the length of `tracek` equals  $\text{floor}(M/m)$ .

Data Types: `double`

## Version History

Introduced in R2012a

## References

[1] Hayes, M.H. *Statistical Digital Signal Processing and Modeling*. New York: John Wiley & Sons, 1996.

## See Also

### Functions

`msepred` | `maxstep`

### Objects

`dsp.LMSFilter` | `dsp.BlockLMSFilter` | `dsp.AdaptiveLatticeFilter` |  
`dsp.AffineProjectionFilter` | `dsp.FastTransversalFilter` | `dsp.FilteredXLMSFilter` |  
`dsp.RLSFilter`

### Topics

“Overview of Adaptive Filters and Applications”

“Signal Enhancement Using LMS and NLMS Algorithms”

## multistage

Multistage filter from specification object

---

**Note** Support for multistage filter design using the `fdesign.interpolator` and `fdesign.decimator` objects will be removed in a future release. Use the `designMultistageInterpolator` and the `designMultistageDecimator` functions, respectively.

Multistage filter design using the `fdesign.lowpass` object and the `design` function now requires you to set the `SystemObject` flag to `true` unless the resulting design is a single-stage filter and not a cascade.

For more information on how to replace your existing code, see “Compatibility Considerations”.

---

### Syntax

```
msFilter = design(d, 'multistage', SystemObject=true)
msFilter = design(..., filterstructure=structure, SystemObject=true)
msFilter = design(..., nstages=nstages, SystemObject=true)
msFilter = design(..., usehalfbands=hb, SystemObject=true)
```

### Description

`msFilter = design(d, 'multistage', SystemObject=true)` designs a multistage filter whose response you specified by the filter specification object `d`.

`msFilter = design(..., filterstructure=structure, SystemObject=true)` returns a filter with the structure specified by `structure`. Input argument `structure` is `dffir` by default and can also be one of the following options.

| structure              | Valid with These Responses  |
|------------------------|-----------------------------|
| <code>firdecim</code>  | Lowpass or Nyquist response |
| <code>firtdecim</code> | Lowpass or Nyquist response |
| <code>firinterp</code> | Lowpass or Nyquist response |
| <code>lowpass</code>   | Default lowpass only        |

Multistage design applies to the default lowpass filter specification object and to decimators and interpolators that use either lowpass or Nyquist responses.

`msFilter = design(..., nstages=nstages, SystemObject=true)` specifies `nstages`, the number of stages to be used in the design. `nstages` must be an integer or `auto`. To allow the design algorithm to use the optimal number of stages while minimizing the cost of using the resulting filter, `nstages` is `auto` by default. When you specify an integer for `nstages`, the design algorithm minimizes the cost for the number of stages you specify.

`msFilter = design(..., usehalfbands=hb, SystemObject=true)` uses halfband filters when you set `hb` to `true`. The default value for `hb` is `false`.



---

**Note** To see a list of the design methods available for your filter, use `designmethods(hd)`.

---

## Examples

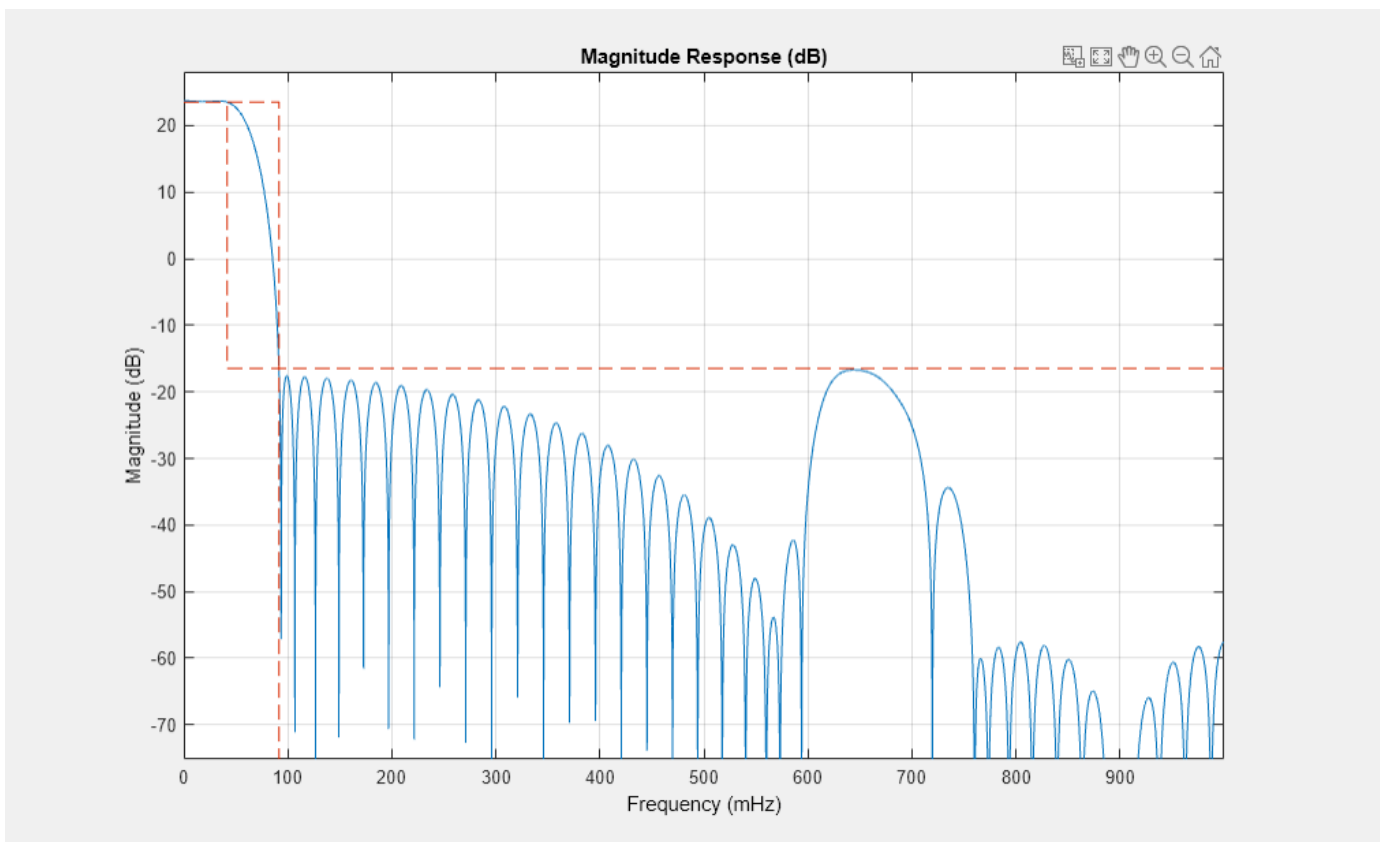
### Design a Multistage Interpolator

This example designs a minimum order, multistage Nyquist interpolator.

```
l = 15; % Interpolation factor. Also the Nyquist band.
tw = 0.05; % Normalized transition width
ast = 40; % Minimum stopband attenuation in dB
msMinInterp = designMultistageInterpolator(l,2,tw,ast)
```

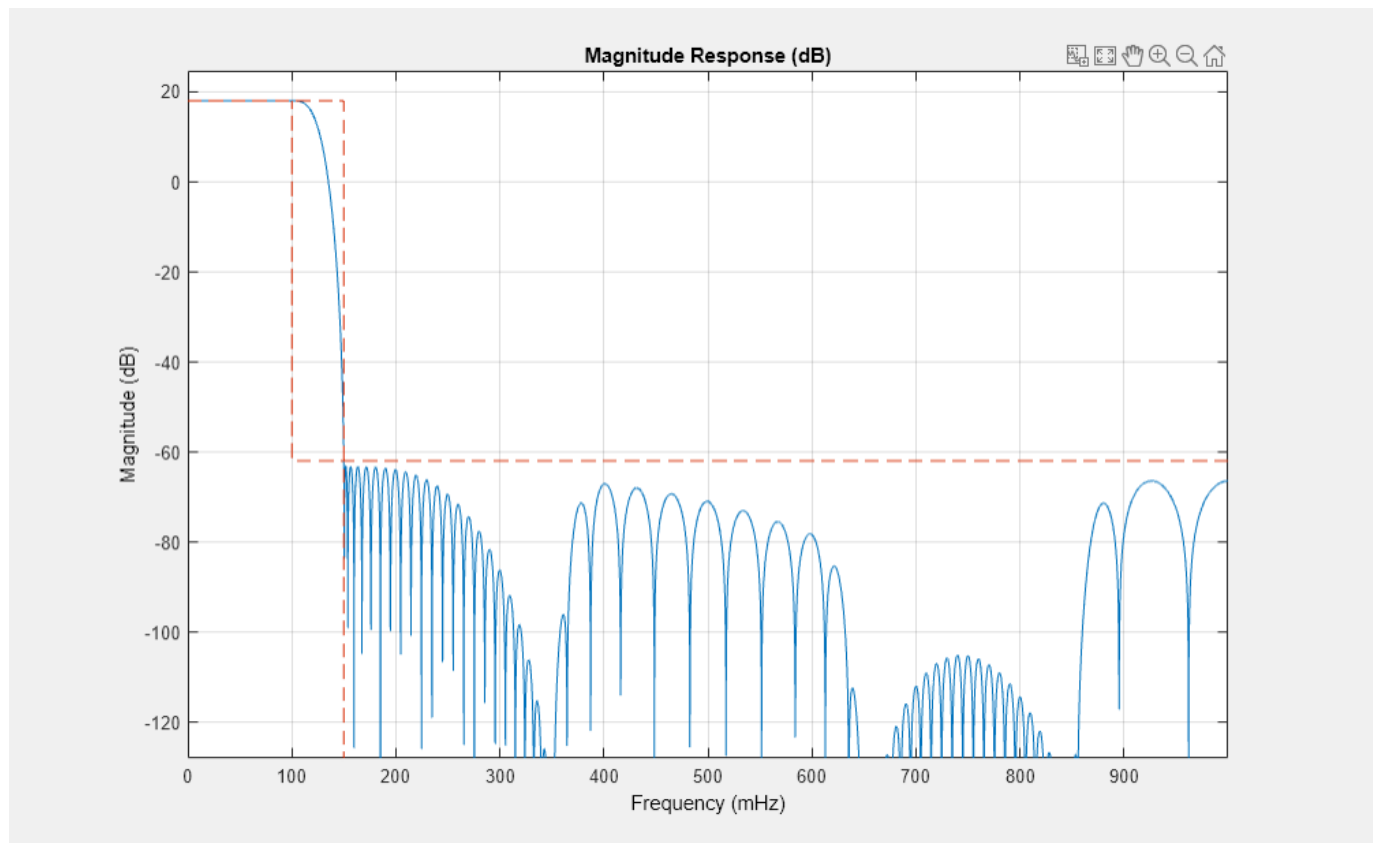
```
msMinInterp =
  dsp.FilterCascade with properties:
    Stage1: [1x1 dsp.FIRInterpolator]
    Stage2: [1x1 dsp.FIRInterpolator]
    CloneStages: false
```

```
fvtool(msMinInterp);
```



Design a multistage lowpass interpolator with an interpolation factor of 8 using the `designMultistageInterpolator` function.

```
m = 8;
msInterp = designMultistageInterpolator(m,2);
fvtool(msInterp);
```



## Version History

Introduced in R2011a

**R2022b: Support for multistage design in `fdesign.interpolator` and `fdesign.decimator` will be removed**

*Warns starting in R2022b*

Support for multistage filter design using the `fdesign.interpolator` and `fdesign.decimator` objects will be removed in a future release. Use the `designMultistageInterpolator` and the `designMultistageDecimator` functions, respectively.

### Update Code

This table shows how to update your code to use the equivalent functions.

| Discouraged Usage   | Recommended Replacement  |
|---|--|
| Using the design function and passing the 'multistage' flag as an input.<br><br><pre>L = 12; % Interpolation factor TW = 2205; % Transition width Astop = 80; % Stopband attenuation Fs = 44100; % Output sample rate filtSpec = fdesign.interpolator(L, 'nyquist', L, TW, Astop, Fs) filtObj = design(filtSpec, 'multistage', SystemObject=true)</pre> | When you set the cost method to 'design', the function designs an optimal number of stages so that the overall implementation cost is low.<br><br><pre>filtObj = designMultistageInterpolator(L, Fs, TW, Astop, CostMethod='design') filtObj = design(filtSpec, 'multistage', SystemObject=true)</pre> |
| Using the multistage function directly.<br><br><pre>filtObj = multistage(filtSpec, SystemObject=true)</pre>   |  |
| Using the design function and passing the 'multistage' flag as an input.<br><br><pre>D = 9; % Decimation factor TW = 2205; % Transition width Astop = 80; % Stopband attenuation Fs = 44100; % Input sample rate filtSpec = fdesign.decimator(D, 'nyquist', D, TW, Astop, Fs) filtObj = design(filtSpec, 'multistage', SystemObject=true)</pre>         | When you set the cost method to 'design', the function designs an optimal number of stages so that the overall implementation cost is low.<br><br><pre>filtObj = designMultistageDecimator(D, Fs, TW, Astop, CostMethod='design') filtObj = design(filtSpec, 'multistage', SystemObject=true)</pre>    |
| Using the multistage function directly.<br><br><pre>filtObj = multistage(filtSpec, SystemObject=true)</pre>   |  |

## R2022b: Support for designing `mfilt.cascade` object using the `fdesign.lowpass` object has been removed

Errors starting in R2022b

Multistage filter design using the `fdesign.lowpass` object and the `design` function now requires you to set the `SystemObject` flag to `true` unless the resulting design is single-stage filter and not a cascade. This setting generates a `dsp.FilterCascade` object instead of the `mfilt.cascade` object.

```
fdesignObj = fdesign.lowpass('Fp,Fst,Ap,Ast', ...
0.1,0.12,0.1,1e-4, 'linear');
FIRInterpObj = multistage(fdesignObj, SystemObject=true)
```

FIRInterpObj =

dsp.FilterCascade with properties:

```
Stage1: [1x1 dsp.FIRDecimator]
Stage2: [1x1 dsp.FIRDecimator]
Stage3: [1x1 dsp.FIRDecimator]
Stage4: [1x1 dsp.FIRInterpolator]
Stage5: [1x1 dsp.FIRInterpolator]
Stage6: [1x1 dsp.FIRInterpolator]
CloneStages: true
```

## See Also

[design](#) | [designopts](#)

## noisepsd

**Package:** dsp

Power spectral density of filter output due to roundoff noise

### Syntax

```
noisepsdOut = noisepsd(sysobj,L)
noisepsdOut = noisepsd( ____,Name,Value)
noisepsdOut = noisepsd(sysobj,L,opts)
noisepsdOut = noisepsd(sysobj,'Arithmetic',arithType)
noisepsd(sysobj, ____)
```

### Description

`noisepsdOut = noisepsd(sysobj,L)` computes the power spectral density (PSD) of filter output occurring because of roundoff noise. The roundoff noise is produced by quantization errors within the filter. The PSD is computed as an average over the `L` trials.

`noisepsdOut = noisepsd( ____,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

`noisepsdOut = noisepsd(sysobj,L,opts)` uses an options object to specify the optional input parameters in lieu of specifying name-value pairs.

`noisepsdOut = noisepsd(sysobj,'Arithmetic',arithType)` analyzes the filter System object based on the arithmetic specified in the `arithType` input.

`noisepsd(sysobj, ____)` with no output argument launches FVTool and shows the noise PSD estimate of the filter System object.

### Examples

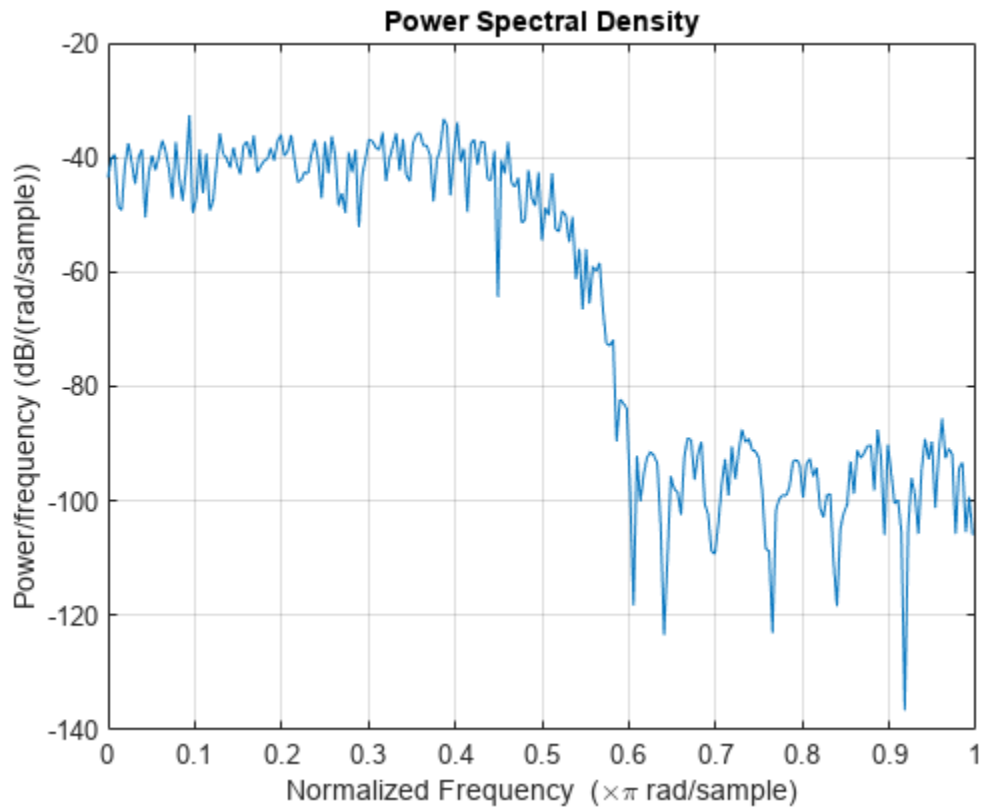
#### Compute the PSD of Output Noise

Compute the PSD of the output noise caused by the quantization processes in a fixed-point, direct-form FIR filter. The input signal has fixed-point data type. The `noisepsd` function performs the analysis based on the data type of the locked input.

```
b = firgr(27,[0 .4 .6 1],[1 1 0 0]);
firfilt = dsp.FIRFilter('Numerator',b); % Create the filter object.
data = fi(randn(15,16),1,16,3);
output = firfilt(data);
```

Quantize the filter to fixed-point. Plot the `hpsd` data. This is the data resulting from the noise PSD calculation. You can review the data in `hpsd.data`.

```
hpsd = noisepsd(firfilt,'Arithmetic','fixed');
plot(hpsd)
```



## Input Arguments

### **sysobj** — Input filter

filter System object

Input filter, specified as one of the following filter System objects:

- `dsp.AllpassFilter`
- `dsp.AllpoleFilter`
- `dsp.BiquadFilter`
- `dsp.CoupledAllpassFilter`
- `dsp.Differentiator`
- `dsp.FIRFilter`
- `dsp.FourthOrderSectionFilter`
- `dsp.HighpassFilter`
- `dsp.IIRFilter`
- `dsp.LowpassFilter`
- `dsp.NotchPeakFilter`
- `dsp.SOSFilter`

- `dsp.VariableBandwidthFIRFilter`
- `dsp.VariableBandwidthIIRFilter`

### **L — Number of trials**

10 | positive integer

Number of trials used to compute the average, specified as a positive integer. The larger you specify *L*, the more accurate is the estimate of power spectral density but at the expense of longer computation time. When you do not specify *L*, the function sets *L* to the default value of 10 trials.

Data Types: `single` | `double`

### **opts — Options object**

options object

Options object to specify the optional input parameters in lieu of specifying name-value pairs.

Create an `opts` object with

```
opts = noisepsdopts(sysobj);
```

`opts` then takes the `noisepsd` settings from `sysobj`.

Because `opts` is an object, use the `set` function to change the parameter values in `opts` before using it with the `noisepsd` function. For example, you could specify a new sample rate with

```
set(opts, 'NormalizedFrequency', false, 'Fs', 48e3);
```

### **arithType — Arithmetic type**

'double' | 'single' | 'fixed'

Arithmetic used in the analysis, specified as 'double', 'single', or 'Fixed'. You cannot use the 'Arithmetic' argument unless you know the data type of the input. If you do not specify the 'Arithmetic' argument, that is, you use the syntax `noisepsdOut = noisepsd(sysobj)`, then the `noisepsd` function applies these rules:

- If the System object is Unlocked — The `noisepsd` function performs double-precision analysis.
- If the System object is Locked — The `noisepsd` function performs the analysis based on the data type of the locked input.

If you do specify the 'Arithmetic' argument, that is, you use the syntax `noisepsdOut = noisepsd(sysobj, 'Arithmetic', arithType)`, the `noisepsd` function applies these rules depending on the value you set for `arithType`.

| Value                | System Object State | Rule   |
|----------------------|---------------------|--|
| arithType = 'double' | Unlocked            | noisepsd performs double-precision analysis. |
|                      | Locked              | noisepsd performs double-precision analysis. |
| arithType = 'single' | Unlocked            | noisepsd performs single-precision analysis. |
|                      | Locked              | noisepsd performs single-precision analysis. |

| Value               | System Object State | Rule   |
|---------------------|---------------------|--|
| arithType = 'fixed' | Unlocked            | noisepsd produces an error because data type of the fixed-point input is unknown.  |
|                     | Locked              | When the input data type is double or single, then noisepsd produces an error because data type of the fixed-point input is unknown. |
|                     |                     | When the input data is of fixed-point type, noisepsd performs analysis based on the data type of the locked input.                   |

### Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1, . . . , NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

```
Example: b = firgr(27,[0 .4 .6 1],[1 1 0 0]); firfilt =
dsp.FIRFilter('Numerator',b); noisepsdOut = noisepsd(firfilt);
```

### NFFT — Number of FFT points

512 (default) | positive integer

Number of FFT points used in calculating the PSD, specified as a positive integer. This value determines the length of the PSD data vector shown in the Data field of the noisepsdOut structure. When 'NFFT' is set to an:

- Odd integer -- The length of the PSD data is given by  $(NFFT+1)/2$ .
- Even integer -- The length of the PSD data is given by  $(NFFT/2)+1$ .

When not specified, NFFT defaults to 512.

---

**Note** If the spectrum data you specify is calculated over half the Nyquist interval and you do not specify a corresponding frequency vector, the default frequency vector assumes that the number of points in the whole FFT is even. Also, the plot option to convert to a whole or two-sided spectrum assumes the original whole FFT length is even.

---

Data Types: single | double

### NormalizedFrequency — Use normalized frequency

true (default) | false

Indicate whether to use normalized frequency or linear frequency, specified as either:

- true -- Use normalized frequency. When not specified, the function defaults to true.

- `false` -- Use linear frequency. When you specify `false`, you must supply the sampling frequency `Fs`.

### **Fs — Sampling frequency**

'Normalized' | positive scalar

Sampling frequency to be specified in Hz when 'NormalizedFrequency' is set to `false`. When 'NormalizedFrequency' is set to `true`, 'Fs' is set to 'Normalized'.

Data Types: `single` | `double`

### **SpectrumType — Spectrum type**

'Onesided' (default) | 'Twosided'

Spectrum type of the generated PSD data, specified as:

- 'Onesided' -- Converts the type to a spectrum that is calculated over half the Nyquist interval. All properties affected by the new frequency range are adjusted automatically.
- 'Twosided' -- Converts the type to a spectrum that is calculated over the whole Nyquist interval. All properties affected by the new frequency range are adjusted automatically. If you choose a two-sided computation, you can also choose 'CenterDC' as `true`. Otherwise, 'CenterDC' must be `false`.

### **CenterDC — Set center of spectrum to DC**

`false` (default) | `true`

Shift the zero-frequency component to the center of a two-sided spectrum.

- When you set 'SpectrumType' to 'Onesided', it is changed to 'Twosided' and the data is converted to a two-sided spectrum.
- Setting 'CenterDC' to `false` shifts the data and the frequency values in the object so that DC is in the left edge of the spectrum. This operation does not affect the 'SpectrumType' property setting.

## **Output Arguments**

### **noisepsdOut — PSD data object**

psd data object

Data object containing the PSD data, returned as a `psd` object. To extract the PSD vector from `noisepsdOut`, enter the following in the MATLAB command prompt:

```
get(noisepsdOut, 'data')
```

Plot the PSD data with `plot(noisepsdOut)`. The average power of the output noise (the integral of the PSD) can be computed with `avgpower`, a method of `dspdata` objects:

```
avgpwr = avgpower(hpsd)
```

## **Version History**

**Introduced in R2011a**



## References

- [1] McClellan, James H., editor. *Computer-Based Exercises for Signal Processing Using MATLAB 5*. Prentice-Hall, 1998.

## See Also

### Functions

`filter` | `noisepsopts` | `reorder` | `scale`

## noisepsdopts

**Package:** dsp

Create an options object for output noise PSD computation

### Syntax

```
opts = noisepsdopts(sysobj)
```

### Description

`opts = noisepsdopts(sysobj)` uses the current settings in the filter System object to create an options object that contains specified options for computing the output noise power spectral density. You can pass the `opts` object as an input argument to the `noisepsd` function.

### Examples

#### Set the Noise PSD Options

Use the `noisepsdopts` function to set options to compute the output noise PSD. `filt1` and `filt2` are lowpass filters that use different design methods. The `opts` object makes it easier to set the same conditions for the noise PSD computation in the `noisepsd` function.

```
d = fdesign.lowpass
```

```
d =
```

```
lowpass with properties:
```

```

    Response: 'Lowpass'
  Specification: 'Fp,Fst,Ap,Ast'
  Description: {4x1 cell}
  NormalizedFrequency: 1
                Fpass: 0.4500
                Fstop: 0.5500
                Apass: 1
                Astop: 60
```

```
filt1 = design(d,'butter','Systemobject',true)
```

```
filt1 =
```

```
dsp.SOSFilter with properties:
```

```

    Structure: 'Direct form II'
  CoefficientSource: 'Property'
    Numerator: [13x3 double]
  Denominator: [13x3 double]
  HasScaleValues: true
    ScaleValues: [0.4151 0.3718 0.3374 0.3099 0.2878 0.2701 0.2558 0.2445 0.2358 0.2293 0.2245 0.2193 0.2145]
```

```
Show all properties
```

```
filt2 = design(d, 'cheby2', 'Systemobject', true)
filt2 =
    dsp.SOSFilter with properties:
        Structure: 'Direct form II'
        CoefficientSource: 'Property'
        Numerator: [5x3 double]
        Denominator: [5x3 double]
        HasScaleValues: true
        ScaleValues: [0.7003 0.5771 0.4734 0.3925 0.3466 1]
```

Show all properties

```
opts = noisepsdopts(filt1)
```

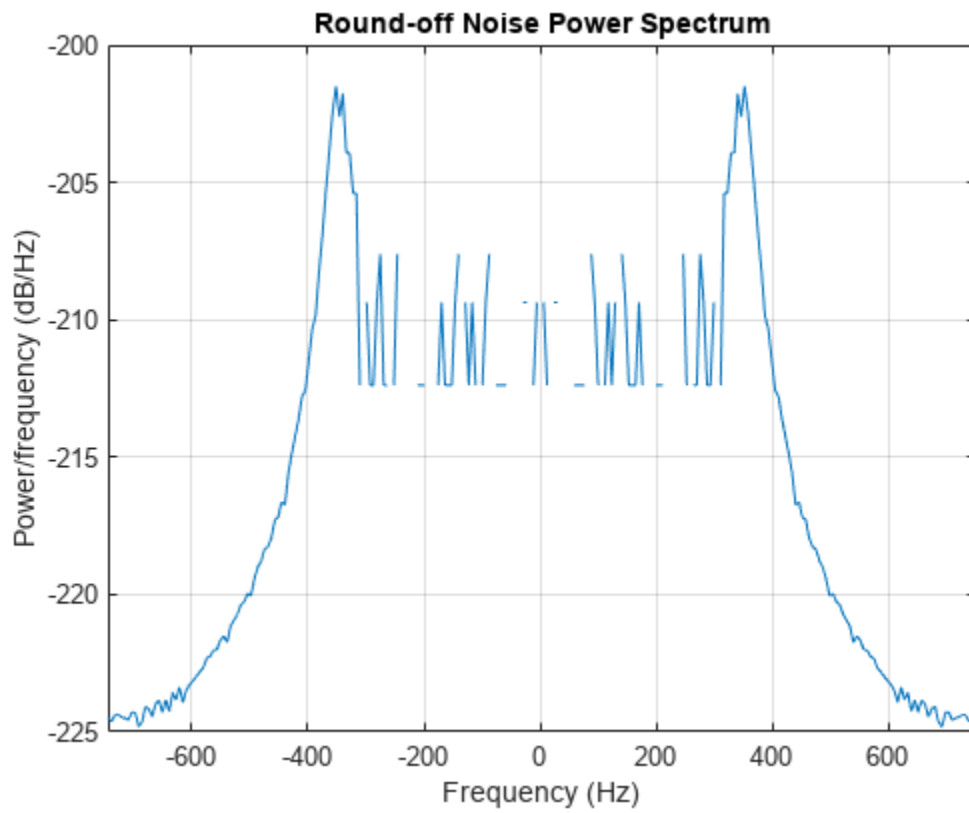
```
opts =
    struct with fields:
        FreqPoints: 'All'
        NFFT: 512
        NormalizedFrequency: true
        Fs: 'Normalized'
        SpectrumType: 'Onesided'
        CenterDC: false
        ConfLevel: 'Not Specified'
        ConfInterval: []
```

```
opts.NFFT = 256; % Same as set(opts, 'nfft', 256).
opts.NormalizedFrequency = false;
opts.Fs = 1.5e3;
opts.CenterDC = true
```

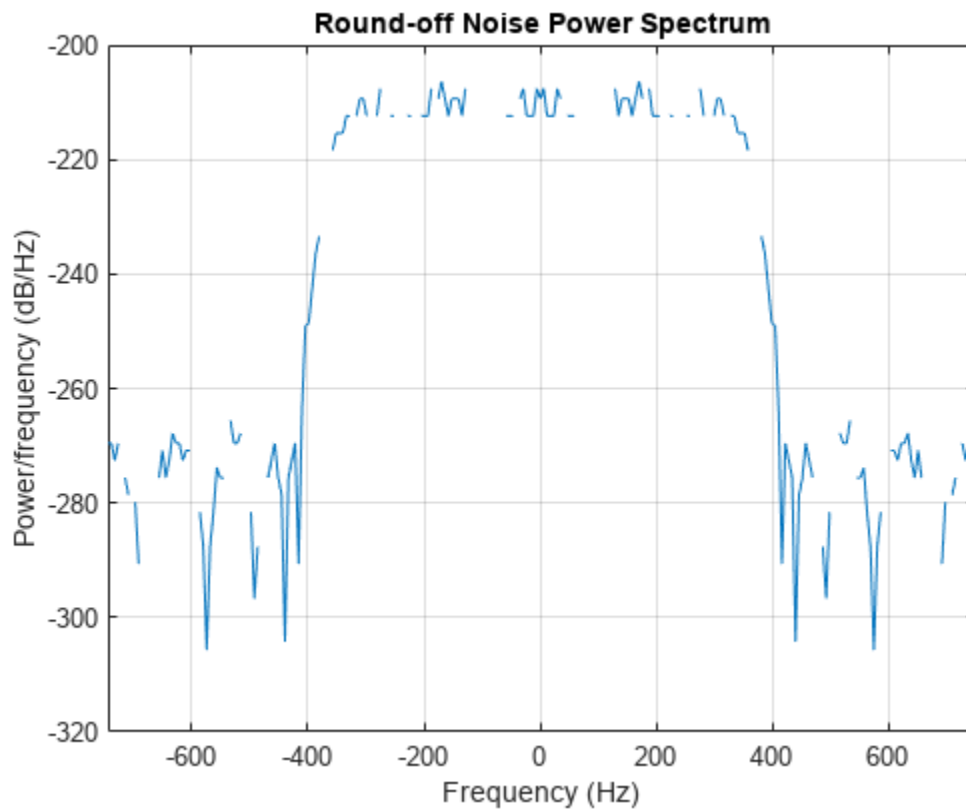
```
opts =
    struct with fields:
        FreqPoints: 'All'
        NFFT: 256
        NormalizedFrequency: false
        Fs: 1500
        SpectrumType: 'Twosided'
        CenterDC: true
        ConfLevel: 'Not Specified'
        ConfInterval: []
```

With `opts` configured as needed, use it as an input argument for the `noisepsd` function.

```
npsd = noisepsd(filt1, 20, opts);
plot(npsd)
title("Round-off Noise Power Spectrum")
```



```
npsd = noisepsd(filt2,20,opts);  
plot(npsd)  
title("Round-off Noise Power Spectrum")
```



## Input Arguments

### **sysobj** — Input filter

filter System object

Input filter, specified as one of the following filter System objects:

- `dsp.AllpassFilter`
- `dsp.AllpoleFilter`
- `dsp.BiquadFilter`
- `dsp.CoupledAllpassFilter`
- `dsp.Differentiator`
- `dsp.FIRFilter`
- `dsp.FourthOrderSectionFilter`
- `dsp.HighpassFilter`
- `dsp.IIRFilter`
- `dsp.LowpassFilter`
- `dsp.NotchPeakFilter`
- `dsp.SOSFilter`

- `dsp.VariableBandwidthFIRFilter`
- `dsp.VariableBandwidthIIRFilter`

## Output Arguments

### **opts** — Options object

spectrum object

Options object which contains the options for computing the output noise PSD, returned as a spectrum object. You pass the `opts` object as an input argument to the `noisepsd` function to specify values for the input parameters.

Using `opts`, you can set the following properties for `noisepsd`:

| Property Name       | Default Value | Description and Valid Entries  |
|---------------------|---------------|--|
| NFFT                | 512           | Specify the number of FFT points to use to calculate the PSD.  |
| NormalizedFrequency | true          | Determine whether to use normalized frequency. Enter a logical value of the logical <code>true</code> or <code>false</code> . Because this property is a logical value, do not enclose with single quotation marks.  |
| Fs                  | 'Normalized'  | Specify the sampling frequency to use when you set <code>NormalizedFrequency</code> to <code>false</code> . Use any integer value greater than 1. Enter the value in Hz.   |
| SpectrumType        | 'Onesided'    | Specify how <code>noisepsd</code> should generate the PSD. Options are 'Onesided' or 'Twosided'. If you choose a two-sided computation, you can also choose <code>CenterDC = true</code> . Otherwise, <code>CenterDC</code> must be <code>false</code> . <ul style="list-style-type: none"> <li>• 'Onesided' converts the type to a spectrum that is calculated over half the Nyquist interval. All properties affected by the new frequency range are adjusted automatically.</li> <li>• 'Twosided' converts the type to a spectrum that is calculated over the whole Nyquist interval. All properties affected by the new frequency range are adjusted automatically.</li> </ul> |

| Property Name | Default Value | Description and Valid Entries   |
|---------------|---------------|---|
| CenterDC      | false         | <p>Shift the zero-frequency component to the center of a two-sided spectrum.</p> <ul style="list-style-type: none"> <li>When you set <code>SpectrumType</code> to 'Onesided', it is changed to 'Twosided' and the data is converted to a two-sided spectrum.</li> <li>Setting <code>CenterDC</code> to false shifts the data and the frequency values in the object so that DC is in the left edge of the spectrum. This operation does not effect the <code>SpectrumType</code> property setting.</li> </ul> |
| Arithmetic    | arithType     | <p>Analyze the filter System object, based on the arithmetic specified in the <code>arithType</code> input. <code>arithType</code> can be set to 'double', 'single', or 'fixed'. The analysis tool assumes a double-precision filter when the arithmetic input is not specified and the filter System object is in an unlocked state.</p>   |

## Version History

Introduced in R2011a

## See Also

### Functions

noisepsd | freqrespest

## norm

P-norm of filter

### Syntax

```
l = norm(hd)
l = norm(hd, pnorm)
```

### Description

All of the variants of `norm` return the filter p-norm for the object in the syntax, a digital filter. When you omit the `pnorm` argument, `norm` returns the L2-norm for the object.

Note that by Parseval's theorem, the L2-norm of a filter is equal to the `l2` norm. This equality is not true for the other norm variants.

#### For `dfilt` Objects

`l = norm(hd)` returns the L2-norm of a discrete-time filter.

`l = norm(hd, pnorm)` includes input argument `pnorm` that lets you specify the norm returned. `pnorm` can be either

- Frequency-domain norms specified by one of `L1`, `L2`, or `Linf`
- Discrete-time domain norms specified by one of `l1`, `l2`, or `linf`

By Parseval's theorem, the L2-norm of a filter is equal to the `l2` norm. This equality is not true for the other norm variants.

IIR filters respond slightly differently to `norm`. When you compute the `l2`, `linf`, `L1`, and `L2` norms for an IIR filter, `norm(..., L2, tol)` lets you specify the tolerance for the accuracy in the computation. For `l1`, `l2`, `L2`, and `linf`, `norm` uses the tolerance to truncate the infinite impulse response that it uses to calculate the norm. For `L1`, `norm` passes the tolerance to the numerical integration algorithm. Refer to Examples on page 5-0 to see this in use. You cannot specify `Linf` for the norm and include the `tol` option.

## Examples

### Norm of an IIR Filter

This example shows how to compute the L2 norm of an IIR filter. A tolerance of 1e-10 is used.

```
spec = fdesign.lowpass('n,fc',5,0.4);
filter = butter(spec);
filternorm = norm(filter, 'l2', 1e-10)
```

```
filternorm = 0.6336
```



## **Version History**

**Introduced in R2011a**

### **See Also**

reorder | scale | scalecheck

## normalize

Normalize filter numerator or feed-forward coefficients

### Syntax

```
normalize(hq)  
g = normalize(hd)
```

### Description

`normalize(hq)` normalizes the filter numerator coefficients for a quantized filter to have values between -1 and 1. The coefficients of `hq` change — `normalize` does not copy `hq` and return the copy. To restore the coefficients of `hq` to the original values, use `denormalize`.

Note that for lattice filters, the feed-forward coefficients stored in the property `lattice` are normalized.

`g = normalize(hd)` normalizes the numerator coefficients for the filter `hq` to between -1 and 1 and returns the gain `g` due to the normalization operation. Calling `normalize` again does not change the coefficients. `g` always returns the gain returned by the first call to `normalize` the filter.

### Examples

#### Normalize the Coefficients of a Direct Form II Filter

Create a direct form II quantized filter that uses second-order sections. Then use `normalize` to maximize the use of the range of representable coefficients.

```
d = fdesign.lowpass('n,fp,ap,ast',8,.5,2,40);  
hd = design(d,'ellip');  
hd.arithmetic = 'fixed';
```

Check the filter coefficients. Note that `InitialSOSMatrix(3,2)>1`

```
InitialSOSMatrix = hd.sosMatrix;
```

Use `normalize` to modify the coefficients into the range between -1 and 1. The output `g` contains the gains applied to each section of the SOS filter.

```
g = normalize(hd);
```

None of the numerator coefficients exceed -1 or 1.

## Version History

Introduced in R2011a

# normalizefreq

Switch filter specification between normalized frequency and absolute frequency

## Syntax

```
normalizefreq(d)  
normalizefreq(d, flag)  
normalizefreq(d, false, fs)
```

## Description

`normalizefreq(d)` normalizes the frequency specifications in filter specifications object `d`. By default, the `NormalizedFrequency` property is set to `true` when you create a design object. You provide the design specifications in normalized frequency units. `normalizefreq` does not affect filters that already use normalized frequency.

If you use this syntax when `d` does not use normalized frequency specifications, all of the frequency specifications are normalized by  $fs/2$  so they lie between 0 and 1, where `fs` is specified in the object. Included in the normalization are the filter properties that define the filter pass and stopband edge locations by frequency:

- `F3 dB` — Used by IIR filter specifications objects to describe the passband cutoff frequency
- `Fcutoff` — Used by FIR filter specifications objects to describe the passband cutoff frequency
- `Fpass` — Describes the passband edges
- `Fstop` — Describes the stopband edges

In this syntax, `normalizefreq(d)` assumes you specified `fs` when you created `d` or changed `d` to use absolute frequency specifications.

`normalizefreq(d, flag)` where `flag` is either **true** or **false**, specifies whether the `NormalizedFrequency` property value is `true` or `false` and therefore whether the filter normalizes the sampling frequency `fs` and other related frequency specifications. `fs` defaults to 1 for this syntax.

When you do not provide the input argument `flag`, it defaults to `true`. If you set `flag` to `false`, affected frequency specifications are multiplied by  $fs/2$  to remove the normalization. Use this syntax to switch your filter between using normalized frequency specifications and not using normalized frequency specifications.

`normalizefreq(d, false, fs)` lets you specify a new sampling frequency `fs` when you set the `NormalizedFrequency` property to `false`.

## Examples

### Normalize the Frequency Specifications of a Filter

These examples demonstrate using `normalizefreq` in both of the major syntax applications—setting the design object frequency specifications to use absolute frequency (`normalizefreq(hd,false,fs)`) and resetting a design object to using normalized frequencies (`normalizefreq(d)`).

Construct a highpass filter specifications object by specifying the passband and stopband edges and the desired attenuations in the bands. By default, provide the frequency specifications in normalized values between 0 and 1.

```
d = fdesign.highpass(0.35, 0.45, 60, 40);
```

`Fstop` and `Fpass` are in normalized form, and the property `NormalizedFrequency` is true.

Now use `normalizedfreq` to convert to absolute frequency specifications, with a sampling frequency of 1000 Hz.

```
normalizefreq(d,false,1e3);
```

Both of the attenuation specifications remain the same. The passband and stopband edge definitions now appear in Hz, where the new value represents the normalized values multiplied by  $F_s/2$ , or 500 Hz.

Converting to using normalized frequencies consists of using `normalizefreq` with the design object `d`.

```
normalizefreq(d)
```

For bandstop, bandpass, and multiple band filter specifications objects, `normalizefreq` works the same way for all band edge definitions. When you do not provide the sampling frequency  $F_s$  as an input argument and you are converting to absolute frequency specifications, `normalizefreq` sets  $F_s$  to 1, as shown in this example.

```
d=fdesign.bandstop(0.25,0.35,0.55,0.65,50,60);  
normalizefreq(d,false)
```

## Version History

Introduced in R2011a

### See Also

`fdesign.lowpass` | `fdesign.halfband` | `fdesign.highpass` | `fdesign.interpolator`

# nstates

Number of filter states

## Syntax

```
n = nstates(hd)
```

## Description

### Discrete-Time Filters

`n = nstates(hd)` returns the number of states `n` in the discrete-time filter `hd`. The number of states depends on the filter structure and the coefficients.

## Examples

### Number of States of a Filter

Determine the number of states of a direct form FIR filter.

```
FIRFilter = firls(30,[0 .1 .2 .5]*2,[1 1 0 0]);  
DiscFilter = dfilt.dffir(FIRFilter);  
NstateDF = nstates(DiscFilter)
```

```
NstateDF = 30
```

## Version History

Introduced in R2011a

## order

**Package:** dsp

Order of discrete-time filter System object

### Syntax

```
n = order(sysobj)
```

### Description

`n = order(sysobj)` returns the order `n` of the filter System object. The order depends on the filter structure and the reference double-precision floating-point coefficients.

### Examples

#### Determine Filter Order

Design a compensation decimator for a CIC decimator using the `dsp.FilterCascade` object. Determine the order of the overall filter.

```
cicdecim = dsp.CICDecimator('DecimationFactor', 6, ...  
    'NumSections', 6);  
  
fs = 16e3;    % Sampling frequency of input of compensation decimator  
fPass = 4e3; % Passband frequency  
fStop = 4.5e3; % Stopband frequency  
ciccomp = dsp.CICCompensationDecimator(cicdecim, ...  
    'DecimationFactor', 2, ...  
    'PassbandFrequency', fPass, ...  
    'StopbandFrequency', fStop, ...  
    'SampleRate', fs);  
  
filtchain = dsp.FilterCascade(cicdecim, ciccomp);  
  
order(filtchain)  
  
ans = 648
```

### Input Arguments

#### **sysobj** — Input filter

filter System object

Input filter, specified as one of the following filter System objects:

- `dsp.AllpassFilter`
- `dsp.AllpoleFilter`

- `dsp.BiquadFilter`
- `dsp.Channelizer`
- `dsp.CICCompensationDecimator`
- `dsp.CICCompensationInterpolator`
- `dsp.CICDecimator`
- `dsp.CICInterpolator`
- `dsp.CoupledAllpassFilter`
- `dsp.Differentiator`
- `dsp.FarrowRateConverter`
- `dsp.FilterCascade`
- `dsp.FIRDecimator`
- `dsp.FIRFilter`
- `dsp.FIRHalfbandDecimator`
- `dsp.FIRHalfbandInterpolator`
- `dsp.FIRInterpolator`
- `dsp.FIRRateConverter`
- `dsp.FourthOrderSectionFilter`
- `dsp.HighpassFilter`
- `dsp.IIRFilter`
- `dsp.IIRHalfbandDecimator`
- `dsp.IIRHalfbandInterpolator`
- `dsp.LowpassFilter`
- `dsp.NotchPeakFilter`
- `dsp.SOSFilter`
- `dsp.VariableBandwidthFIRFilter`
- `dsp.VariableBandwidthIIRFilter`

## Output Arguments

### **n** — Filter order

scalar

Filter order, returned as a scalar. The order depends on the filter structure and the reference double-precision floating-point coefficients.

Data Types: `double`

## Version History

Introduced in R2011a

## See Also

`info`

## outputDelay

**Package:** dsp

Determine output delay of single-rate or multirate filter

### Syntax

```
D = outputDelay(sysobj)
[D,FsOut] = outputDelay(sysobj)
[D,FsOut,B] = outputDelay(sysobj)
[ ___ ] = outputDelay(sysobj,Name=Value)
```

### Description

`D = outputDelay(sysobj)` returns the system output delay of the single-rate or multirate filter object. A multirate filter is any cascade combining upsampling, downsampling, and convolution filters (FIR or IIR). Single-rate filters are special cases of multirate filters with a rate conversion factor of 1. The delay  $D$  results from the group delay of the convolution stages within the filter.

Use this syntax when `sysobj` is a linear-phase single-rate filter. To determine if a filter object has linear phase, use the `islinphase` function.

`[D,FsOut] = outputDelay(sysobj)` also returns the output sample rate. Use this syntax when `sysobj` is a linear-phase multirate filter.

`[D,FsOut,B] = outputDelay(sysobj)` also returns the input frequency band  $B$  over which the delay value is within the default tolerance of 5% (in input sample units) of  $D$ . That is,  $|D(f) - D| \leq 0.05/FsIn$ , where  $f$  is a frequency in the band  $B$ .

Use this syntax when `sysobj` has filter stages that have a nonlinear phase.

`[ ___ ] = outputDelay(sysobj,Name=Value)` specifies options using one or more name-value arguments in addition to the input arguments in previous syntaxes. For example, `outputDelay(sysobj, Tol=0.01)` estimates the band of input frequencies over which the delay value is within the tolerance of 1%.

Use this syntax to specify `FsIn`, `Fc`, `Tol`, and `FFTLength`.

## Examples

### Compute Output Delay of Filter Cascade

Compute the output delay of a cascade of `dsp.FIRRateConverter` objects. Use this delay value to plot the input and resampled signals on the same plot in a time scope.

### Resample Input Signal

First, let us inspect the delay and scaling that occurs in filtering. To do that, create a sinusoidal input signal. Initialize a cascade of `dsp.FIRRateConverter` objects to resample the input signal.



```
n = (0:7*17-1)';
u = cos(6*pi*n/147);

src = cascade(dsp.FIRRateConverter(13,17), dsp.FIRRateConverter(18,7))

src =
  dsp.FilterCascade with properties:
    Stage1: [1x1 dsp.FIRRateConverter]
    Stage2: [1x1 dsp.FIRRateConverter]
    CloneStages: true
```

Resample the input signal by passing the signal through the cascade of `dsp.FIRRateConverter` objects.

```
y = src(u);
```

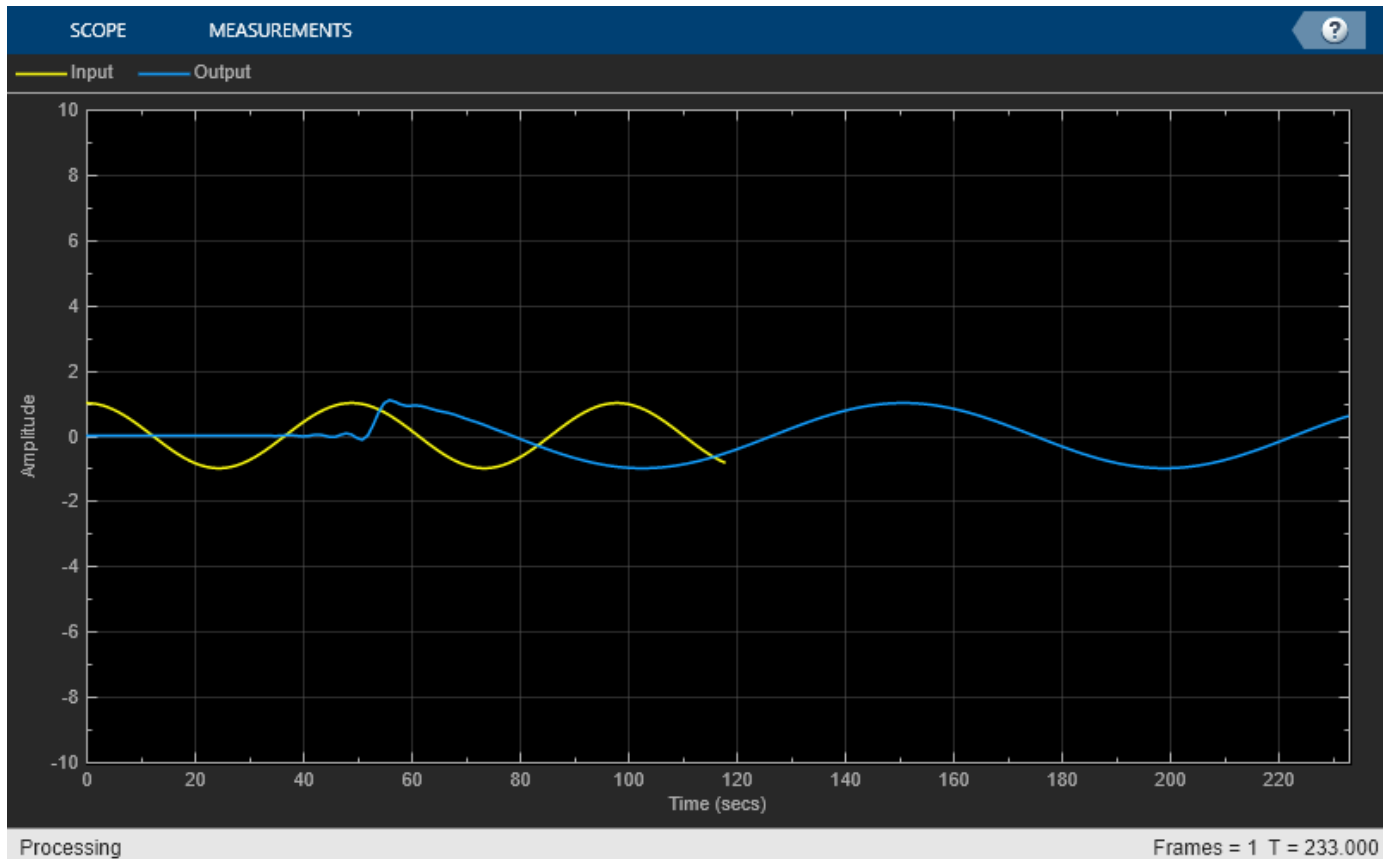
### Plot the Input and Output Signals

Initialize a time scope using the `timescope` object to visualize the input and the resampled signals.

```
tsnosync = timescope(NumInputPorts=2,...
    ChannelNames={'Input', 'Output'});
```

Plot the input and the resampled signals on the same plot in the time scope. You can see that the two signals are out of sync with respect to each other by a certain amount of delay and stretch over a different time scale

```
tsnosync(u,y);
```



### Using `outputDelay` to Synchronize Signals

To plot the input and resampled signals on the same plot, you need to account for the output delay of the filter. To compute the output delay, use the `outputDelay` function. The value of this delay depends on the filter structure and order, the rate conversion factor, and the signal input to the filter. In addition to the output delay, the rate conversion operation of the filter introduces scaling on the time domain. You must account for the output delay and the scaling.

Use the `outputDelay` function to compute the delay and the output sample rate introduced by the multirate filter. You can optionally specify the input sample rate to the function. Apply the delay value to the output signal through the `TimeDisplayOffset` property of the time scope.

To account for the scaling, specify the input and output sample rates through the `SampleRate` property of the time scope. The inverse of the input and output sample rates vector determines the x-axis (time axis) spacing between points in the input signal and the output signal, respectively.

Initialize the time scope to use the updated `TimeDisplayOffset` and `SampleRate` properties. Visualize the input and the resampled signals on the time scope. With the delay and scaling accounted for, you can see that signals are synchronized and have no delay with respect to each other.

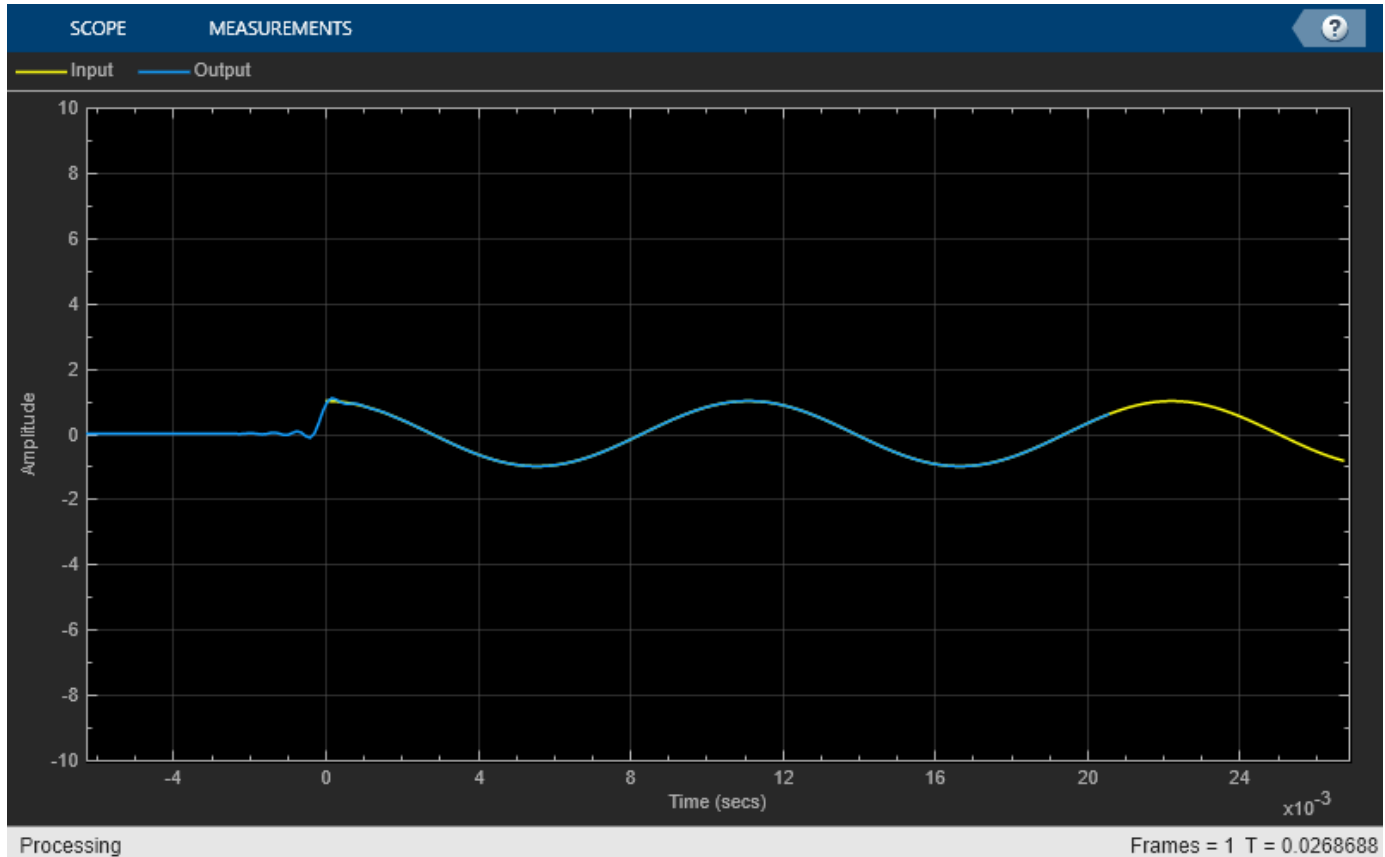
```
FsIn = 44.1e2;
[D,FsOut] = outputDelay(src,FsIn=FsIn);

tssync = timescope(NumInputPorts=2,...
    SampleRate=[FsIn FsOut], ...
```

```

TimeDisplayOffset=[0, -D],...
ChannelNames={'Input', 'Output'});
tssync(u,y);

```



### Compute Output Delay for Farrow Rate Converter

Compute the output delay for a farrow rate converter, and use this delay value to plot the input sinusoidal signal and the rate converted signal on the same plot.

Create an input sinusoidal signal using the `sin` function. Initialize the `dsp.FarrowRateConverter` object to model the farrow rate converter.

```

u = sin(6*pi*(1:50)'/200);
FsIn = 48000;
frc = dsp.FarrowRateConverter(FsIn, 44100);

```

Compute the output delay of the rate converter using the `outputDelay` function.

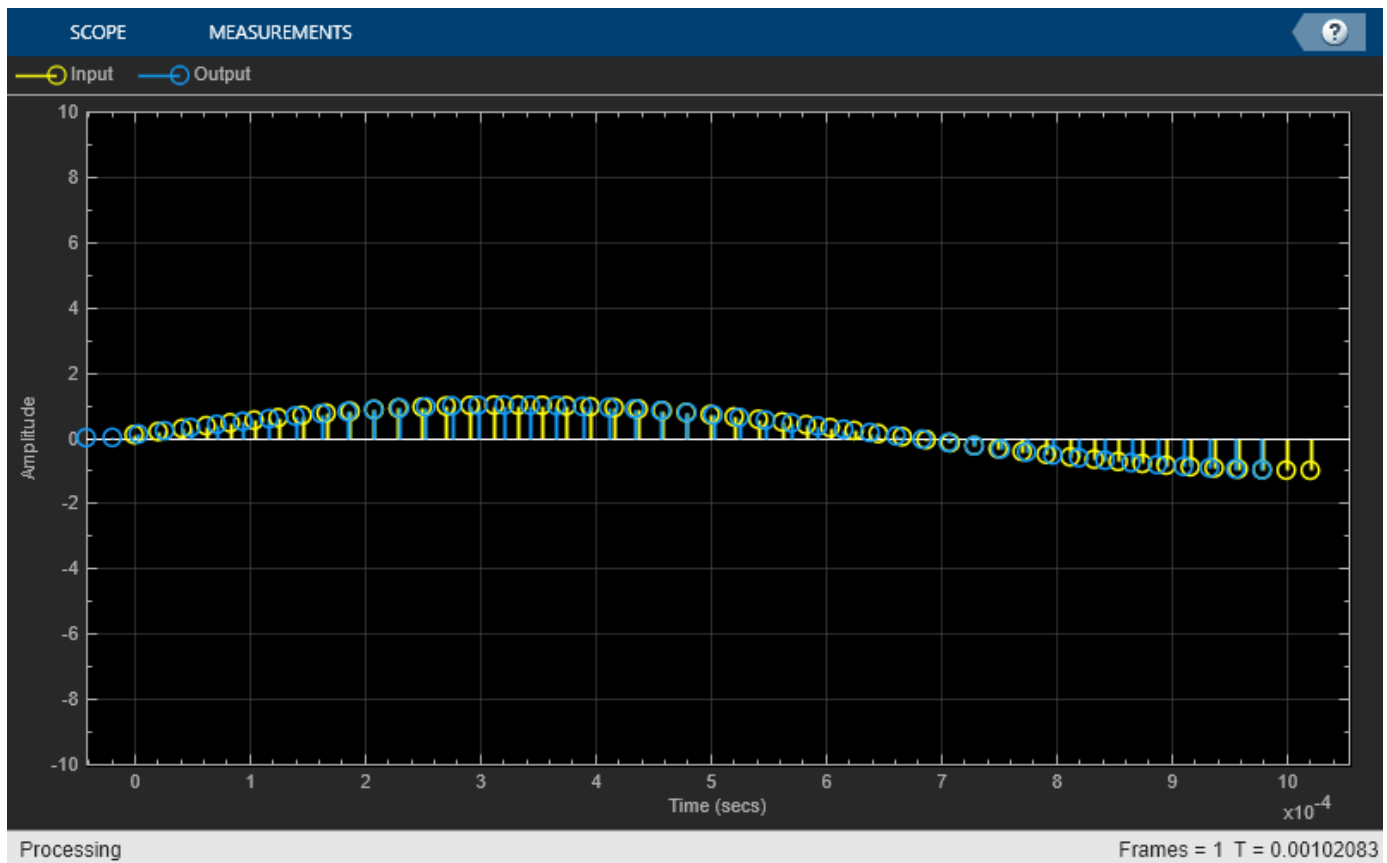
```
[D, FsOut] = outputDelay(frc)
```

```
D = 4.1667e-05
```

```
FsOut = 44100
```

Resample the input using the farrow rate converter. Visualize the input signal and the resampled signal on the time scope. To account for the time delay and the scaling, set the `TimeDisplayOffset` and `SampleRate` properties of the time scope to `[0 -D]` and `[FsIn FsOut]`, respectively.

```
y = frc(u);
ts = timescope(NumInputPorts = 2, SampleRate = [FsIn FsOut], ...
    TimeDisplayOffset = [0, -D], ...
    ChannelNames = {'Input', 'Output'}, ...
    TimeSpan = length(y)*1.1/FsIn, ...
    PlotType = 'stem');
ts(u, y);
```



### Calculate Frequency Band for Nonlinear Phase Filters

Nonlinear phase filters have a group delay that depends on the input frequency. Due to the nonlinear phase nature, such filters distort input signals. Therefore, the output of these filters cannot be obtained by shifting and scaling the input on the time domain. To use the `outputDelay` function to compute the output delay of such filters, the filters must have a relatively constant group delay over the input signal band.

Start by calculating the frequency band of two nonlinear fractional delay FIR filters with partially flat group delay. One filter has a higher bandwidth compared to the other filter. Note the effect of the higher bandwidth on the output delay and the input frequency band over which the function computes the delay.

Then, consider a multirate filter cascade that has a highly nonlinear group delay response over a given band but has a relatively flat group delay over other frequency bands. Increase the tolerance value that you specify to the `outputDelay` function and see the effect on the input frequency band that the function returns.

### Fractional Delay FIR Filter with Nonlinear Phase and Partially Flat Group Delay

Since the fractional delay FIR filter has a nonlinear phase response, specify a center frequency around which the `outputDelay` function should compute the delay. When you specify the center frequency, the function returns an interval of frequencies (as the third output argument) over which the delay value is within the specified tolerance.

Design a fractional delay filter using the `designFracDelayFIR` function. Specify the filter to have a fractional delay of  $1.25e-3$  seconds and a passband coverage of 50%.

```
FsIn = 20;
Fc = 1;
FD = 1.25e-3;
```

```
n1PhaseFilterObj1 = dsp.FIRFilter(designFracDelayFIR(FD*FsIn, 0.5));
```

Using the `outputDelay` function, calculate the output delay around the center frequency of 1 Hz. Specify the input sample rate to 20 Hz and the tolerance to 0.01 (1%). In addition to the output delay, the function returns the input band over which the deviation in delay is upto 1% of `D1`.

```
[D1,~,B1] = outputDelay(n1PhaseFilterObj1,FsIn=FsIn,Fc=Fc,Tol=0.01)
```

```
D1 = 0.1513
```

```
B1 = 1×2
```

```
    -6.3794    6.3794
```

Use the `grpdelay` function to compute the group delay  $G$  of the filter and the frequencies  $W$  (in Hz) at which the group delay is evaluated. Specify the FFT length to 8192, and the sample rate to be the same as the input sample rate.

Plot the overall filter group delay. On the same figure, plot the input frequency band and the output delay value at the center frequency. The input frequency band indeed contains the center frequency.

```
[G1,W1] = n1PhaseFilterObj1.grpdelay(8192, FsIn); % Use 8192 FFT points to calculate the group delay
I1 = W1>=B1(1) & W1<=B1(2); % Mask applicable frequencies
```

```
plot(W1,G1/FsIn);
hold on
plot(W1(I1), G1(I1)/FsIn, 'r', LineWidth=2);
plot(Fc, D1, 'ro');
yline(D1+0.01/FsIn*[-1 1], 'k:');
```

```
title(sprintf('Applicable Band (FsIn=%iHz, Fc=%1.1fHz)',FsIn,Fc))
```

```
legend('Output delay (in sample time units)',...
```

```
    'Applicable input frequencies band',...
```

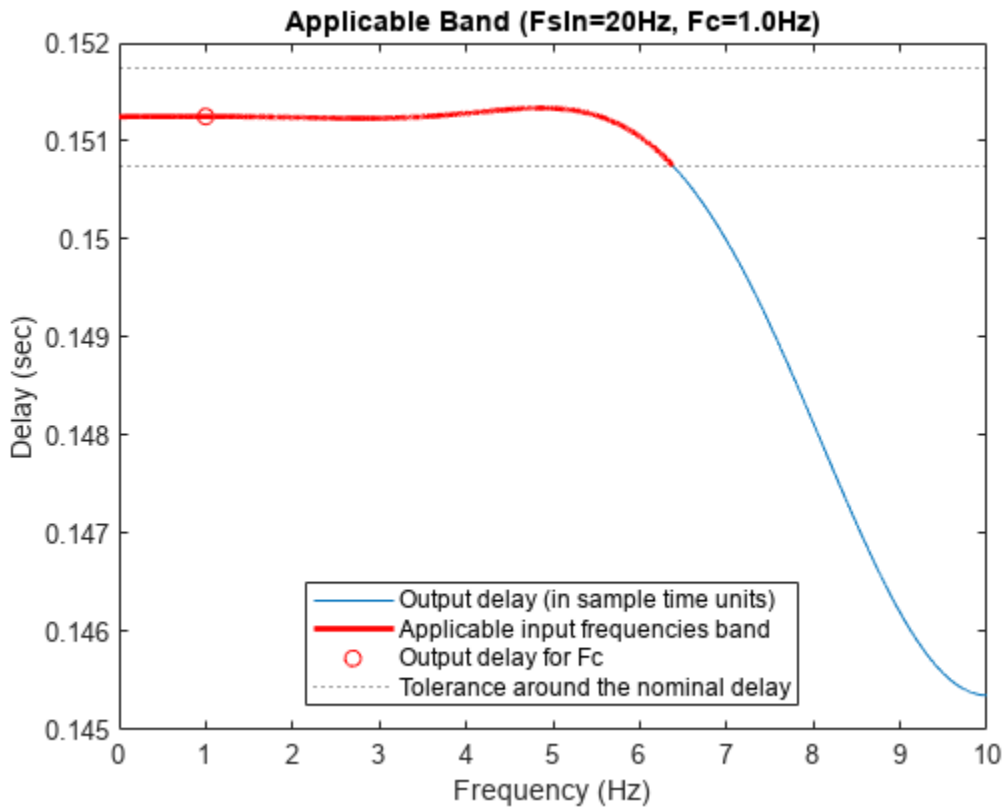
```
    'Output delay for Fc',...
```

```
    'Tolerance around the nominal delay', Location='best')
```

```
ylabel('Delay (sec)')
```

```
xlabel('Frequency (Hz)')
```

```
hold off
```



### Comparing with Higher Bandwidth Design

Compare the previous design with a higher bandwidth fractional delay design. For more information on the bandwidth of a fractional-delay FIR filter, see `designFracDelayFIR`.

Design a fractional-delay FIR filter with the same fractional delay ( $1.25e-3$  seconds) as the previous design but with a 70% bandwidth coverage.

```
n1PhaseFilterObj2 = dsp.FIRFilter(designFracDelayFIR(FD*FsIn, 0.7));
```

Measure the output delay of this alternative design. The output delay  $D2$  is larger due to the longer FIR length, but the input band is also larger, allowing signals up to 7.2 Hz (compared with only 6.4 Hz in the previous design).

```
[D2,~,B2] = outputDelay(n1PhaseFilterObj2,FsIn=FsIn,Fc=Fc,Tol=0.01)
```

```
D2 = 0.2512
```

```
B2 = 1×2
```

```
    -7.2217    7.2217
```

```
[G2,W2] = n1PhaseFilterObj2.grpdelay(8192, FsIn);
```

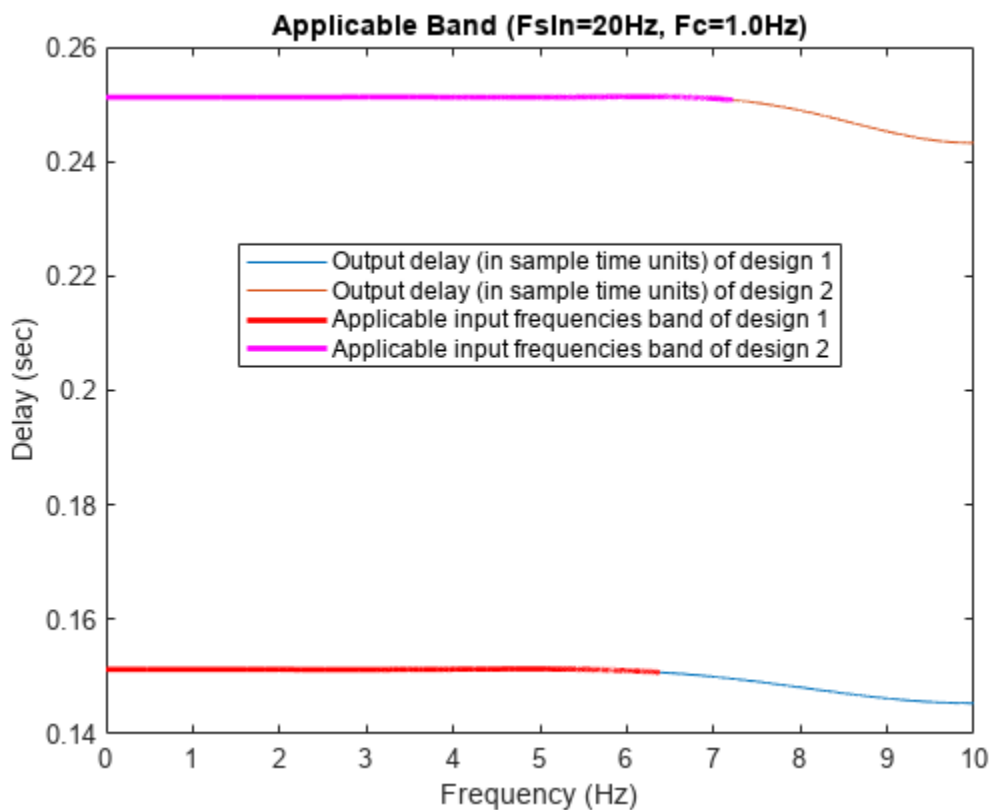
Plot the group delay response of the alternative design against the design with a 50% coverage.

```
I2 = W2>=B2(1) & W2<=B2(2); % Mask applicable frequencies
```

```

plot(W1,G1/FsIn);
hold on
plot(W2,G2/FsIn);
plot(W1(I1), G1(I1)/FsIn, 'r', LineWidth=2);
plot(W2(I2), G2(I2)/FsIn, 'm', LineWidth=2);
title(sprintf('Applicable Band (FsIn=%iHz, Fc=%1.1fHz)',FsIn,Fc))
legend('Output delay (in sample time units) of design 1',...
      'Output delay (in sample time units) of design 2',...
      'Applicable input frequencies band of design 1',...
      'Applicable input frequencies band of design 2', ...
      Location='best')
ylabel('Delay (sec)')
xlabel('Frequency (Hz)')
hold off

```



### Multirate Filter Cascade with Nonlinear Phase

Design a two-stage filter cascade that contains a stable IIR lowpass (which has a nonlinear phase) followed by an FIR decimator with a rate conversion factor of 2.

```

FsIn = 10;
Fc = 1.5;
LP = dsp.LowpassFilter(FilterType='IIR', SampleRate=FsIn, PassbandFrequency=2.3, StopbandFrequency=2.5);
nlPhaseCascadeObj = cascade(LP, dsp.FIRDecimator(2));

```

Measure the output delay, output sample rate, and the input frequency band for this cascade using a narrowband input signal with a carrier frequency of 1.5Hz. Set the tolerance to 0.001 samples. The

function returns an output sample rate of 5 Hz (which is half the input sample rate of 10 Hz, owing to the half-rate decimator).

```
[D,FsOut,B] = outputDelay(nlPhaseCascadeObj, Fc=Fc, Tol=0.001)
D = 2.9209
FsOut = 5
B = 1×2
    1.5000    1.5000
```

The band the function returns is a trivial band. That is, it contains a single point (the left and right boundaries are the same). This means that a tolerance value of 0.001 is too small.

To fix that, increase the tolerance to 0.05 seconds (0.5 of the input sample time). While the delay and the output sample rate do not change, the band is now nontrivial, and it contains the center frequency. The band is also relatively narrow due to the highly nonlinear phase nature of the IIR design.

```
[D1,FsOut1,B1] = outputDelay(nlPhaseCascadeObj, Fc=Fc, Tol=0.5)
D1 = 2.9209
FsOut1 = 5
B1 = 1×2
    1.3562    1.5967
```

Increase the tolerance value to 2. The band now increases even more as you have specified a more lax tolerance.

```
[D2,FsOut2,B2] = outputDelay(nlPhaseCascadeObj, Fc=Fc, Tol=2)
D2 = 2.9209
FsOut2 = 5
B2 = 1×2
    0.4565    1.7786
```

```
[G,W] = nlPhaseCascadeObj.grpdelay(8192, FsIn);
I1 = W>=B1(1) & W<=B1(2); % Mask applicable frequencies
I2 = W>=B2(1) & W<=B2(2); % Mask applicable frequencies

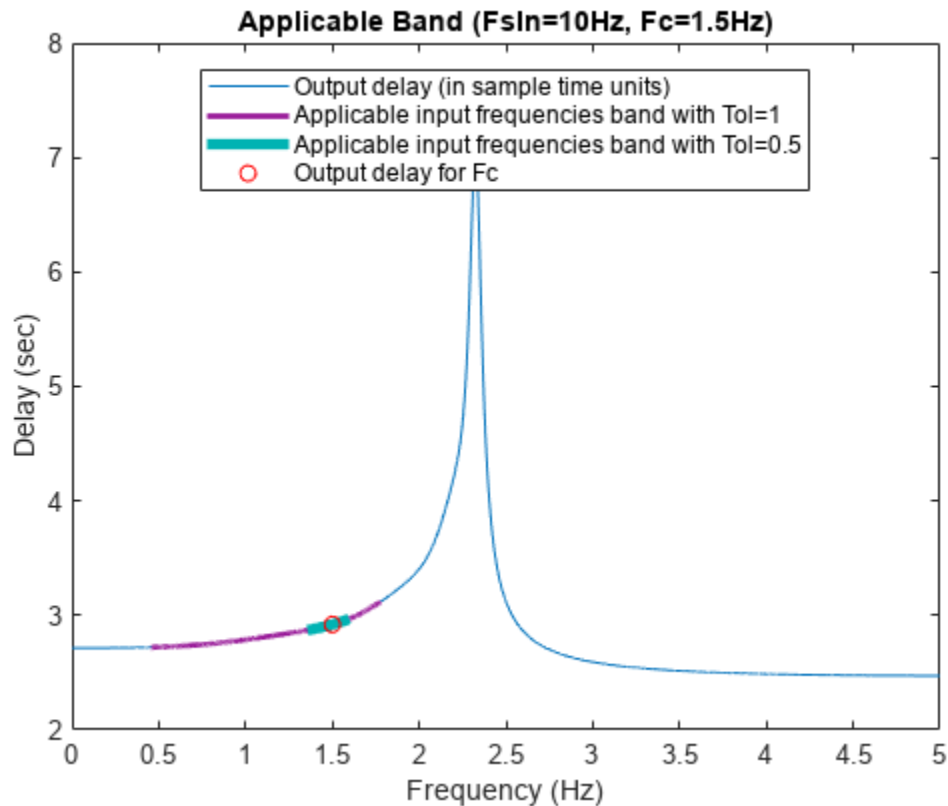
plot(W,G/FsIn);
hold on
plot(W(I2), G(I2)/FsIn, Color=[0.6 0.1 0.6], LineWidth=2);
plot(W(I1), G(I1)/FsIn, Color=[0 0.7 0.7], LineWidth=4);
plot(Fc, D1, 'ro');
title(sprintf('Applicable Band (FsIn=%iHz, Fc=%1.1fHz)',FsIn,Fc))
legend('Output delay (in sample time units)',...
       'Applicable input frequencies band with Tol=1',...)
```



```

    'Applicable input frequencies band with Tol=0.5',...
    'Output delay for Fc',...
    Location='best')
ylabel('Delay (sec)')
xlabel('Frequency (Hz)')
hold off

```



## Input Arguments

### sysobj — Input filter

filter System object

Input filter, specified as one of the following filter System objects:

- `dsp.AllpassFilter`
- `dsp.AllpoleFilter`
- `dsp.BiquadFilter`
- `dsp.CICCompensationDecimator`
- `dsp.CICCompensationInterpolator`
- `dsp.CICDecimator`
- `dsp.CICInterpolator`
- `dsp.Differentiator`

- `dsp.FarrowRateConverter`
- `dsp.FilterCascade`
- `dsp.FIRDecimator`
- `dsp.FIRFilter`
- `dsp.FIRHalfbandDecimator`
- `dsp.FIRHalfbandInterpolator`
- `dsp.FIRInterpolator`
- `dsp.FIRRateConverter`
- `dsp.FourthOrderSectionFilter`
- `dsp.HighpassFilter`
- `dsp.IIRFilter`
- `dsp.IIRHalfbandDecimator`
- `dsp.IIRHalfbandInterpolator`
- `dsp.LowpassFilter`
- `dsp.NotchPeakFilter`
- `dsp.SOSFilter`
- `dsp.VariableBandwidthFIRFilter`
- `dsp.VariableBandwidthIIRFilter`

### Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, . . . , NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `FsIn=48000, Fc=-10.3, Tol=0.01, FFTLength=100`

### **FsIn** — Input sample rate

1 (default) | positive scalar

Input sample rate of the filter object `sysobj`, specified as a positive scalar.

If you do not specify `FsIn`, the function uses the input sample rate specified in the built-in input sample rate property of the filter object if it exists, otherwise the function uses the default rate of 1.

Specify `FsIn` when you want to override the input sample rate specified in the filter object.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **Fc** — Carrier frequency

0 (default) | real scalar

Carrier frequency from which the function derives the group delay of the convolution stages, specified as a real scalar. Specify `Fc` when the filter object `sysobj` has stages with a nonlinear phase.

For filters with a nonlinear phase (a varying group delay), the delay value `D` the `outputDelay` function returns depends on the input frequency. For more details, see “Linear and Nonlinear Phase Filters” on page 5-1326. For an example, see “Calculate Frequency Band for Nonlinear Phase Filters” on page 5-1316.

If you do not specify  $F_c$ , the function uses the default center frequency of 0 for most filters except the bandpass and highpass filters. Make sure that the absolute value of the carrier frequency is not larger than the Nyquist frequency. That is,  $|f_c| \leq F_{sIn}/2$ , where  $F_{sIn}$  is the input sample rate of the filter object.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **Tol — Tolerance of estimated delay**

0.05 (default) | real scalar

Tolerance of the estimated delay value  $D$ , specified as a real scalar. Specify `Tol` when you want to calculate the input band  $B$ .

When you specify a tolerance `Tol`, the function can estimate a band of input frequencies  $B$  for which the delay is close to  $D$  up to a given tolerance `Tol`. That is,  $|D(f) - D| \leq Tol/F_{sIn}$ , where  $f$  is a frequency in the band given by the  $B$  vector.

When you use the following signature, the function uses the default 5% tolerance (in input sample units) of  $D$ .

```
[D, FsOut, B] = outputDelay(sysobj, Fc=fc)
```

Alternatively, you can specify the tolerance value using the `Tol` argument.

```
[D, FsOut, B] = outputDelay(sysobj, Fc=fc, Tol=T)
```

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **FFTLength — FFT length**

8192 (default) | nonnegative integer

FFT length, specified as a nonnegative integer. Specify `FFTLength` when you want to calculate the input band  $B$ .

This value determines the frequency resolution with which the function estimates the input frequency band  $B$ . Higher the FFT length, the higher is the frequency resolution and accuracy of  $B$ .

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

## **Output Arguments**

### **D — Output delay**

nonnegative scalar

Output or resampling delay of the multirate filter on page 5-1324, returned as a nonnegative scalar.

The delay value  $D$  is determined by the group delay of the convolution filter stages, and the value depends on the filter structure, filter order, and rate conversion factors. When the filter has a nonlinear phase, the delay value also depends on the input signal.

For filters with a linear phase (that is, a flat group delay), the value of  $D$  applies to all inputs. For filters with a nonlinear phase (that is, a varying group delay), the delay value  $D$  depends on the input frequencies.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**FsOut — Output sample rate**

positive scalar

Output sample rate of the multirate filter object for the given input sample rate `FsIn`, returned as a positive scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**B — Estimated input frequency band**

two-element row vector

Estimated input frequency band  $[f_1 \ f_2]$  over which the delay value is within tolerance `Tol` (in input sample units) of `D`, returned as a two-element row vector. If you do not specify `Tol`, the function uses a tolerance value of 0.05 (or 5%).

The frequency band stretches from  $f_1$  counterclockwise to  $f_2$ . That is, if  $f_2 > f_1$ , then the band is  $\{f_1 < f < f_2\}$ . If  $f_2 < f_1$ , then the band wraps around the Nyquist frequency, that is the union of  $\{-FsIn/2 < f < f_2\}$  and  $\{f_1 < f < FsIn/2\}$ . If  $f_1 = f_2$ , then the band has a single frequency due to the FFT resolution being too coarse. In such cases, increase the value of `FFTLength` or the tolerance `Tol`.

Band measurement is available only for systems that can be reduced to a rational rate converter. For a linear phase system, the band `B` is always full,  $B = [-FsIn/2, \ FsIn/2]$  and the delay `D` is applicable for all input frequencies (flat group delay).

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**More About****Multirate Filters**

A multirate filter is any cascade combining upsampling, downsampling, and convolution filters (FIR or IIR).



Most filter System objects in DSP System Toolbox are multirate filters. For example, `dsp.FIRDecimator(5)` is the multirate filter  $H(z)$  followed by a downsampler that has a downsampling ratio of 5.

Single-rate filters are special cases of multirate filters with a rate conversion factor of 1. Single-stage filters are special cases of cascades of multirate filters. For example, `dsp.LowpassFilter` is a single-rate, single-stage filter of the form  $H(z)$ .

**Multirate Filters as Resampling Systems**

The `outputDelay` function assumes that the multirate filter `sysobj` models a resampling system. The output of the resampling system is a filtered and resampled version of the input, and hence has a different rate compared to the input.

A resampling system interpolates the input  $u(n)$  to a continuous domain function  $f(t)$  that approximates the input.

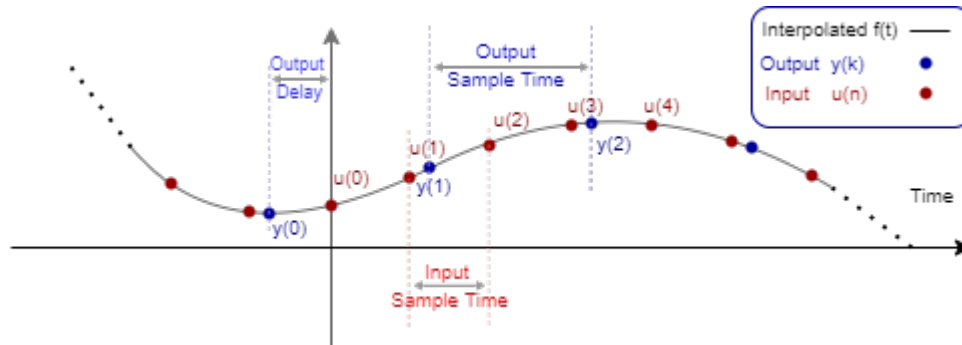
$$u(n) \approx f(n/FsIn)$$

The output is then sampled from  $f(t)$  on a different time scale.

$$y(k) = f(k/FsOut - D)$$

The values  $FsIn$  and  $FsOut$  are the input and output sample rates, respectively, and  $D$  is the resampling or output delay.

This graph overlays the input and output sequences on the same plot, shows the input and output sample times, and also shows the output delay that the function returns.



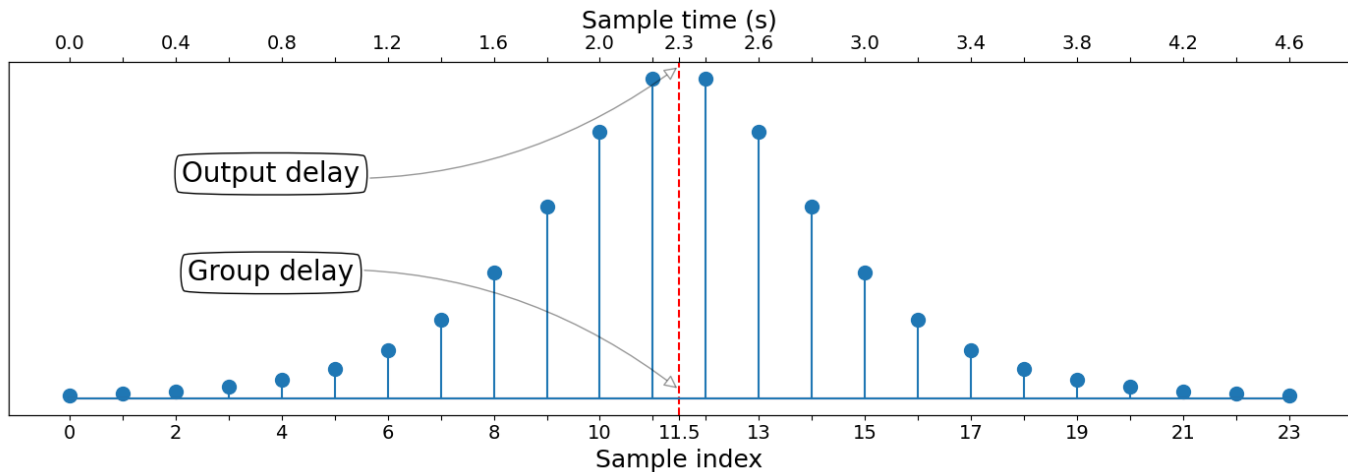
For examples that demonstrate the effect of the resampling delay and scaling in multirate filters, see “Compute Output Delay of Filter Cascade” on page 5-1312 and “Time Delay and Scaling in Multirate DSP Filters”.

### Output Delay and Group Delay

The output delay and the group delay are related to each other.

The output or resampling delay is the delay of the signal in time units as it goes through a convolution filter (FIR or IIR). For a single-rate filter, the output delay ( $D$ ) and the group delay ( $\tau_g$ ) are scaled by the sample rate,  $D = \tau_g/FsIn$ . If the filter has a linear phase, both the output delay and the group delay do not depend on the input frequency. However, if the filter has nonlinear phase, then the group delay and the output delay depend on the input frequency.

This figure shows the impulse response of a symmetric 24-tap FIR filter. You can see that the output delay and the group delay are equivalent quantities in different scales.



If the filter is a multirate filter, the output delay is the total delay that the filter accumulates on all stages of the cascade. The output sample rate  $F_{SOut}$  of each stage becomes the input sample rate  $F_{SIn}$  of the subsequent stage. The function scales the delay values accumulated over each stage by the rate conversion factors of the individual stages.

The group delay is defined only for single-rate filters while the output delay is defined for both single-rate and multirate filters.

The group delay refers to the average delay of the convolution filter (FIR or IIR) as a function of frequency. It is computed as the negative first derivative of the phase response of the filter. If the frequency response of a filter is  $H(e^{j\omega})$ , then the group delay is

$$\tau_g(\omega) = -\frac{d\theta(\omega)}{d\omega},$$

where  $\theta(\omega)$  is the phase, or argument, of  $H(e^{j\omega})$ .

If the filter is a linear phase filter, the group delay response is flat. The output delay  $D$  does not depend on input frequency. If the filter is a nonlinear phase filter, the group delay response varies with frequency and the delay value  $D$  is valid only for bandlimited inputs around a carrier frequency  $F_c$ . The function determines the edges of the frequency band  $B$  such that the group delay over this frequency region is approximately constant. For more information, see “Linear and Nonlinear Phase Filters” on page 5-1326. For an example, see “Time Delay and Scaling in Multirate DSP Filters”.

### Linear and Nonlinear Phase Filters

Depending on the group delay of the filter stages (whether flat or varies with frequency), the `outputDelay` function can operate in two modes.

If all the filter stages have a linear phase (flat group delay), then the `outputDelay` function operates in the full band mode on page 5-1327. If the individual filter stages have a nonlinear phase but can be reordered through noble identity operations such that the equivalent stage has a linear phase, then the `outputDelay` function operates in the full band mode and you do not need to specify the carrier frequency  $F_c$ .

For an example of nonlinear filter stages having a linear phase equivalent, see “A Nonlinear Phase Multirate Filter with Linear Phase Equivalent” on page 5-1328.

If one or more stages have a nonlinear phase, and the filter does not have a linear phase equivalent, then the `outputDelay` function operates in the bandlimited mode on page 5-1327. In the bandlimited mode, the function requires you to specify the input center frequency  $F_c$  in order to compute the delay.

### Full-Band Mode in Linear Phase Filters

In the full band mode, the delay value  $D$  applies to all inputs. All symmetric filter designs work in this mode.

The `outputDelay` function uses the group delay of the filter stages to calculate the overall output delay. When the filters have a linear phase, the group delay of the filter does not vary with frequency. The band  $B$  is always full (flat group delay).

For symmetric filters, the output delay ( $D$ ) can be thought of as the center of mass of the impulse response ( $h[k]$ ), which can be obtained by the weighted sum:

$$D = \frac{1}{FsIn} \cdot \tau_g,$$

$$D = \frac{1}{FsIn} \cdot \frac{\sum_{k=0}^{L-1} k \cdot h[k]}{\sum_{j=0}^{L-1} h[j]}$$

where  $\tau_g$  is the group delay of the filter, and  $L$  is the length of the filter.

The function obtains the output delay by scaling the center of mass with the sample time. For an example, see “Time Delay and Scaling in Multirate DSP Filters”.

This center-of-mass delay formulation can be generalized for asymmetric filters, where it takes the form of group delay. Symmetric and anti-symmetric filters have a linear phase response, hence their group delay response is constant and does not depend on frequency. Filters with a varying group delay (nonlinear phase filters) can exhibit distortion on the time domain.

### Bandlimited Mode for Nonlinear Phase Filters

In the bandlimited mode, the `outputDelay` function calculates the delay  $D$  for the input frequency  $F_c$ . The function uses this mode with filters that have nonlinear phase stages. Many causal and stable rational IIR filters have a nonlinear phase, and therefore have a varying group delay. Generally, those filters distort input signals, which breaks the resampling model described in “Multirate Filters” on page 5-1324. In such cases, the resampling model and the delay value  $D$  are valid only for bandlimited inputs around  $F_c$ . The function determines the width of the frequency band  $B$  such that the group delay over this frequency region is approximately constant. Use the `grpdelay` function to inspect the flatness of the group delay of the filter stages.

Assume that the input is a narrowband signal centered around some carrier frequency  $\omega_c = 2\pi F_c$ . The resampling model is slightly different in this case, where the input is

$$u(n) \approx f(n/FsIn) \cdot \cos(\omega_c n),$$

and the output is given by

$$y(k) \approx f(k/FsOut - D) \cdot \cos(\omega_c(k/FsOut - D_p)).$$

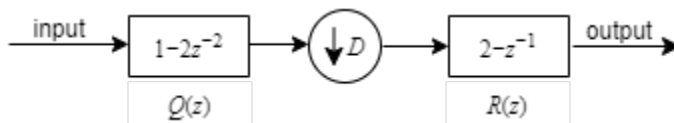
The baseband is delayed by output delay  $D$ , which is calculated using the group delay specific to the input frequency. The carrier, however, experiences a different delay,  $D_p$ , which is obtained by the phase delay of the filter rather than its group delay.

You can specify the input carrier frequency through the input argument  $F_c$ . The `outputDelay` function can also calculate the interval of input frequencies  $B$  that the value of  $D$  is applicable up to a certain tolerance. This band is returned as the third return value of the function (with the second one being the output sample rate  $F_sOut$ ). The tolerance (in sample time units) can be specified using the `Tol` argument.

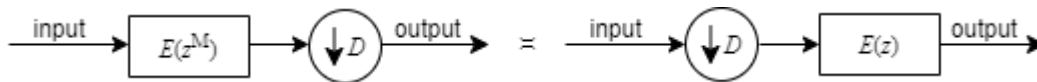
To determine if a filter object is linear, use the `islinphase` function.

### A Nonlinear Phase Multirate Filter with Linear Phase Equivalent

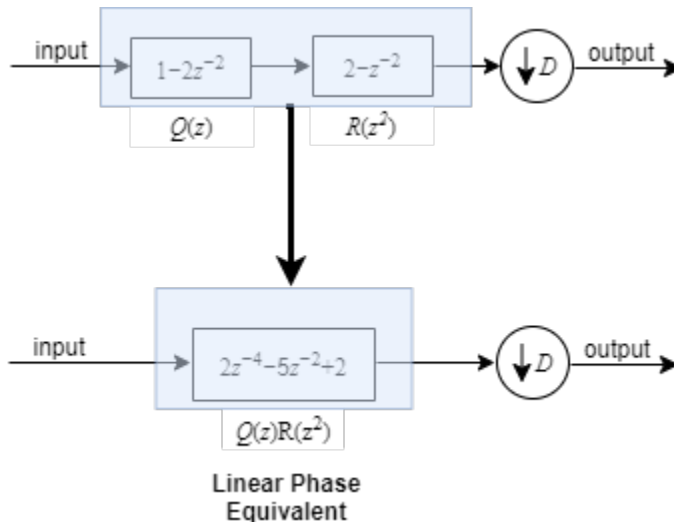
Consider the multirate filter cascade below with nonlinear phase filter stages  $Q(z)$  and  $R(z)$ .



The noble identity for decimation is given by:



Apply the noble identity for decimation on  $R(z)$ . The product  $Q(z)R(z^2)$  is a linear phase system.



Even though the individual filter stages have a nonlinear phase, the combined equivalent filter after applying the noble identity has a linear phase. The `outputDelay` function treats this filter as a linear phase filter.

All combinations of nonlinear phase stages cannot be reordered to form linear phase equivalents. The function treats such filters as nonlinear filters and operates in the bandlimited mode.



## **Version History**

**Introduced in R2022a**

### **See Also**

grpdelay | timescope

### **Topics**

*"Multirate and Multistage Filters"*

*"Analysis Methods for Filter System Objects" on page 3-2*

*"Time Delay and Scaling in Multirate DSP Filters"*

# phasedelay

**Package:** dsp

Phase delay response of discrete-time filter System object

## Syntax

```
[phi,w] = phasedelay(sysobj)
[phi,w] = phasedelay(sysobj,n)
[phi,w] = phasedelay(sysobj,n,fs)
[phi,w] = phasedelay( ____, 'Arithmetic', arithType)
phasedelay(sysobj)
```

## Description

`[phi,w] = phasedelay(sysobj)` returns the phase delay response vector, `phi`, and the corresponding frequency vector, `w`, in radians/sample of the filter System object based on the current filter coefficients.

`[phi,w] = phasedelay(sysobj,n)` returns the `n`-point phase delay response vector and the corresponding `n`-point frequency vector in radians/sample of the filter System object.

`[phi,w] = phasedelay(sysobj,n,fs)` uses `fs` to calculate the delay response. When you specify this syntax without any output arguments, the function launches `fvtool` and plots the response to `fs/2`.

`[phi,w] = phasedelay( ____, 'Arithmetic', arithType)` analyzes the filter System object, based on the arithmetic specified in `arithType` using anyone of the previous syntaxes.

`phasedelay(sysobj)` launches `fvtool` and shows the phase delay response of the filter System object.

For more input options, see `phasedelay` in Signal Processing Toolbox.

## Examples

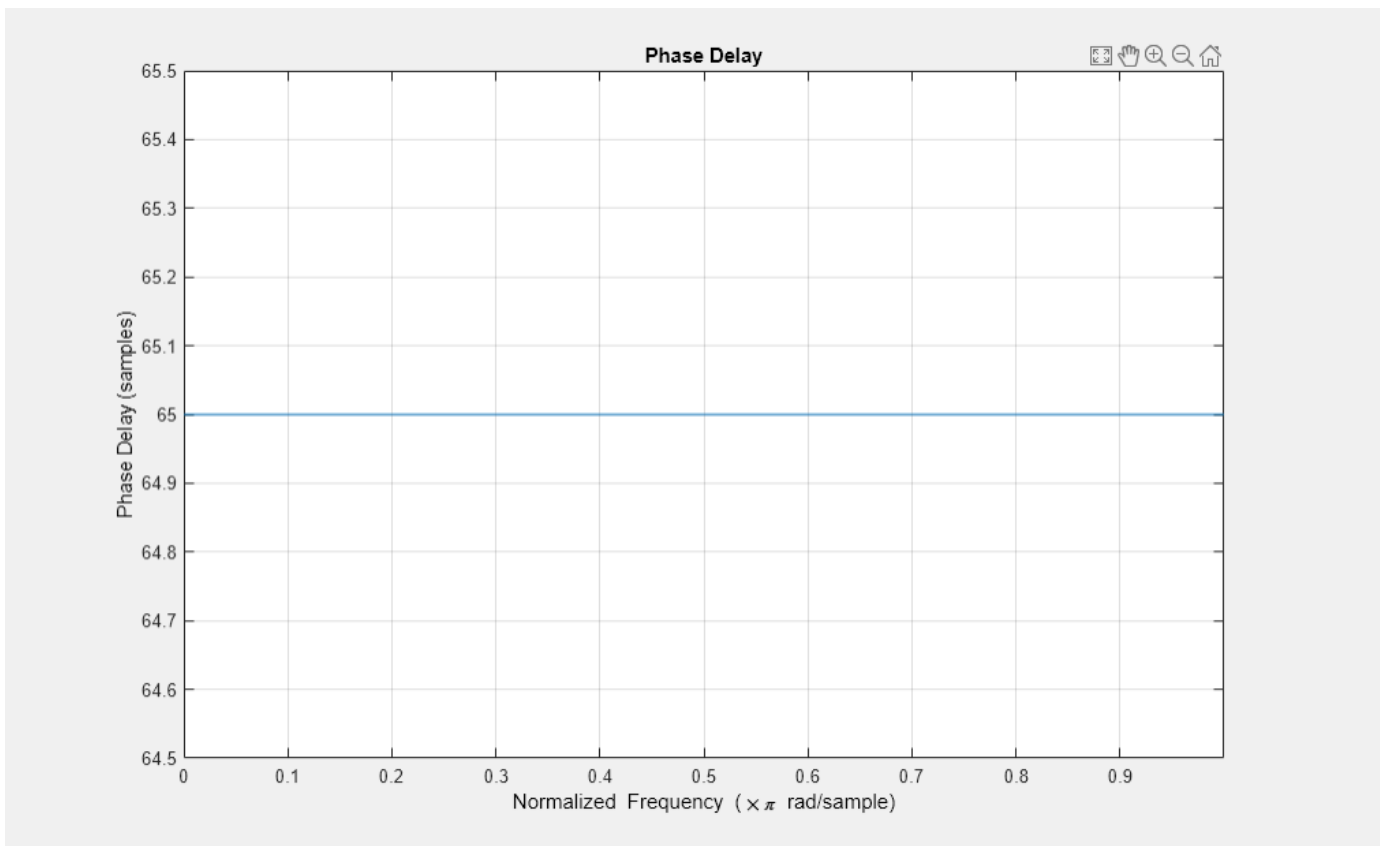
### Phase Delay of Discrete-Time Filter

Design an FIR filter.

```
Fs = 8000; Fcutoff = 2000;
FIRfilt = dsp.FIRFilter('Numerator', fir1(130,Fcutoff/(Fs/2)));
```

The `phasedelay` function computes the phase delay of the filter and displays it using `FVTool`.

```
phasedelay(FIRfilt)
```



## Input Arguments

### **sysobj** — Input filter

filter System object

Input filter, specified as one of the following filter System objects:

- `dsp.AllpassFilter`
- `dsp.AllpoleFilter`
- `dsp.BiquadFilter`
- `dsp.Channelizer`
- `dsp.CICCompensationDecimator`
- `dsp.CICCompensationInterpolator`
- `dsp.CICDecimator`
- `dsp.CICInterpolator`
- `dsp.CoupledAllpassFilter`
- `dsp.Differentiator`
- `dsp.FarrowRateConverter`
- `dsp.FilterCascade`

- `dsp.FIRDecimator`
- `dsp.FIRFilter`
- `dsp.FIRHalfbandDecimator`
- `dsp.FIRHalfbandInterpolator`
- `dsp.FIRInterpolator`
- `dsp.FIRRateConverter`
- `dsp.FourthOrderSectionFilter`
- `dsp.HighpassFilter`
- `dsp.IIRFilter`
- `dsp.IIRHalfbandDecimator`
- `dsp.IIRHalfbandInterpolator`
- `dsp.LowpassFilter`
- `dsp.NotchPeakFilter`
- `dsp.SOSFilter`
- `dsp.VariableBandwidthFIRFilter`
- `dsp.VariableBandwidthIIRFilter`

**n — Number of points over which phase response is evaluated**

8192 (default) | positive integer

Number of points over which the phase response is evaluated around the upper half of the unit circle. For an FIR filter where *n* is a power of two, the computation is done faster using FFTs.

Data Types: `double`

**fs — Sampling frequency**

1 (default) | positive scalar

Sampling frequency used in computing the phase delay response, specified as a positive scalar.

Data Types: `single` | `double`

**arithType — Arithmetic type**

'double' (default) | 'single' | 'Fixed'

Arithmetic used in the filter analysis, specified as 'double', 'single', or 'Fixed'. When the arithmetic input is not specified and the filter System object is unlocked, the analysis tool assumes a double-precision filter. When the arithmetic input is not specified and the System object is locked, the function performs the analysis based on the data type of the locked input.

The 'Fixed' value applies to filter System objects with fixed-point properties only.

When the 'Arithmetic' input argument is specified as 'Fixed' and the filter object has the data type of the coefficients set to 'Same word length as input', the arithmetic analysis depends on whether the System object is unlocked or locked.

- `unlocked` -- The analysis object function cannot determine the coefficients data type. The function assumes that the coefficients data type is signed, has a 16-bit word length, and is auto scaled. The function performs fixed-point analysis based on this assumption.

- **locked** -- When the input data type is 'double' or 'single', the analysis object function cannot determine the coefficients data type. The function assumes that the data type of the coefficients is signed, has a 16-bit word length, and is auto scaled. The function performs fixed-point analysis based on this assumption.

To check if the System object is locked or unlocked, use the `isLocked` function.

When the arithmetic input is specified as 'Fixed' and the filter object has the data type of the coefficients set to a custom numeric type, the object function performs fixed-point analysis based on the custom numeric data type.

## Output Arguments

### **phi** — Phase delay response vector

vector

Phase delay response vector, evaluated at  $n$  equally-spaced points around the upper half of the unit circle, returned as a vector. If  $n$  is not specified, the function uses a default value of 8192.

Data Types: double

### **w** — Frequency vector

vector

Frequency vector of length  $n$  in radians/sample.  $w$  consists of  $n$  equally-spaced points around the upper half of the unit circle (from 0 to  $\pi$  radians/sample). If  $n$  is not specified, the function uses a default value of 8192.

Data Types: double

## Version History

Introduced in R2011a

## See Also

### Functions

`freqz` | `grpdelay` | `phasez` | `zerophase` | `zplane` | `fvtool` | `phasedelay`

# phasez

**Package:** dsp

Phase response of discrete-time filter System object (unwrapped)

## Syntax

```
[phi,w] = phasez(sysobj)
[phi,w] = phasez(sysobj,n)
[phi,w] = phasez( ____, 'Arithmetic',arithType)
phasez(sysobj)
```

## Description

`[phi,w] = phasez(sysobj)` returns the unwrapped phase response `phi` of the filter System object, `sysobj`, based on the current filter coefficients. The vector `w` contains the frequencies (in radians) at which the function evaluates the phase response. The phase response is evaluated at 8192 points equally spaced around the upper half of the unit circle.

`[phi,w] = phasez(sysobj,n)` returns the phase response of the filter System object and the corresponding frequencies at `n` points equally spaced around the upper half of the unit circle.

`[phi,w] = phasez( ____, 'Arithmetic',arithType)` analyzes the filter System object, based on the arithmetic specified in `arithType`, using either of the previous syntaxes.

`phasez(sysobj)` displays the phase response of the filter System object `sysobj` in the `fvtool`.

For more input options, see `phasez` in Signal Processing Toolbox.

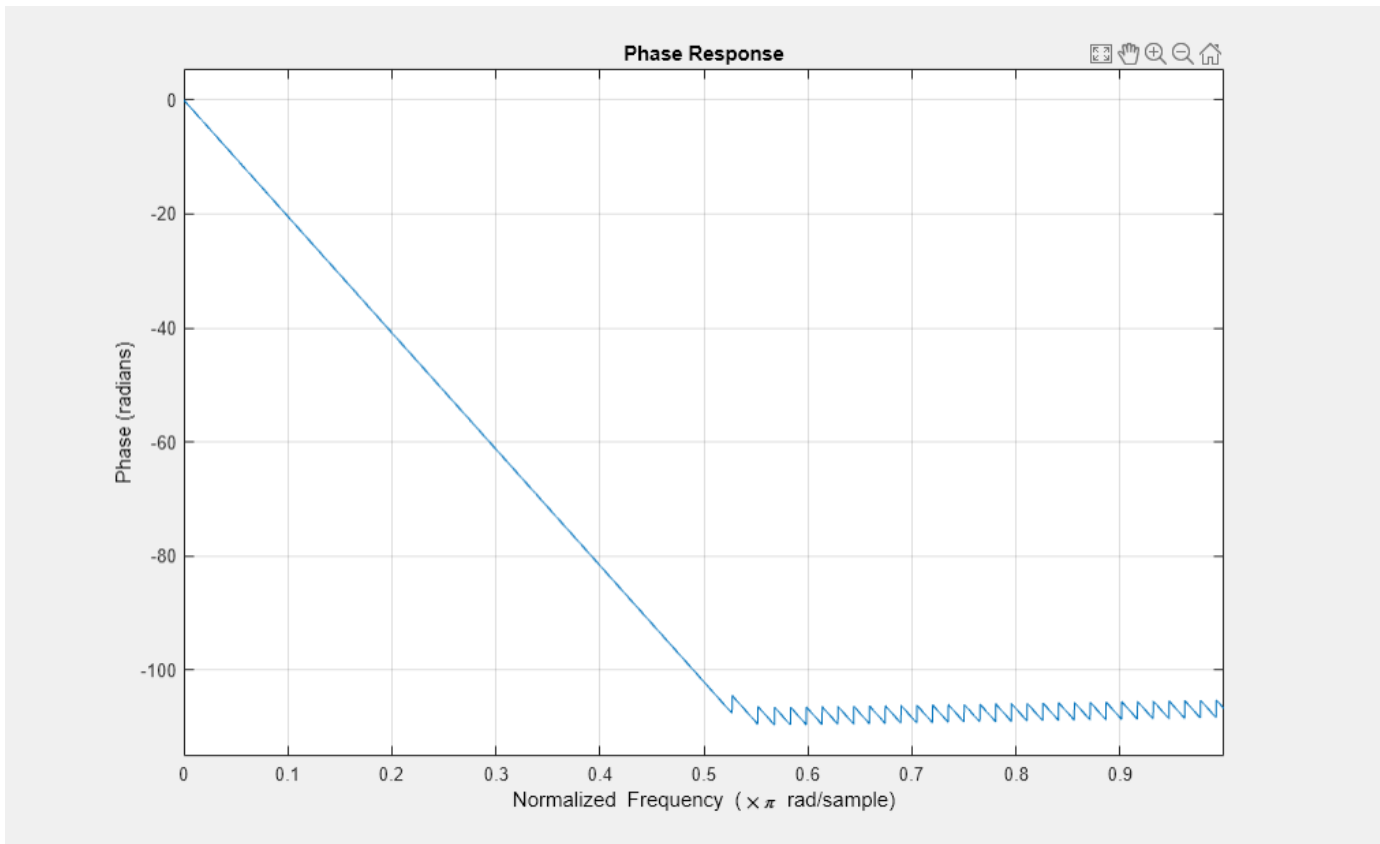
## Examples

### Unwrapped Phase Response of a Discrete-Time Filter

```
Fs = 8000; Fcutoff = 2000;
FIRfilt = dsp.FIRFilter('Numerator', fir1(130,Fcutoff/(Fs/2)));
```

`phasez` computes the unwrapped phase response of the filter and displays it using `fvtool`

```
phasez(FIRfilt);
```



## Input Arguments

### **sysobj** — Input filter

filter System object

Input filter, specified as one of the following filter System objects:

- `dsp.AllpassFilter`
- `dsp.AllpoleFilter`
- `dsp.BiquadFilter`
- `dsp.CICCompensationDecimator`
- `dsp.CICCompensationInterpolator`
- `dsp.CICDecimator`
- `dsp.CICInterpolator`
- `dsp.CoupledAllpassFilter`
- `dsp.Differentiator`
- `dsp.FarrowRateConverter`
- `dsp.FilterCascade`
- `dsp.FIRDecimator`

- `dsp.FIRFilter`
- `dsp.FIRHalfbandDecimator`
- `dsp.FIRHalfbandInterpolator`
- `dsp.FIRInterpolator`
- `dsp.FIRRateConverter`
- `dsp.FourthOrderSectionFilter`
- `dsp.HighpassFilter`
- `dsp.IIRFilter`
- `dsp.IIRHalfbandDecimator`
- `dsp.IIRHalfbandInterpolator`
- `dsp.LowpassFilter`
- `dsp.NotchPeakFilter`
- `dsp.SOSFilter`
- `dsp.VariableBandwidthFIRFilter`
- `dsp.VariableBandwidthIIRFilter`

**n — Number of points over which the frequency response is computed**

8192 (default) | positive integer

Number of points over which the frequency response is computed. For an FIR filter where *n* is a power of two, the computation is done faster using FFTs.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**arithType — Arithmetic type**

'double' (default) | 'single' | 'Fixed'

Arithmetic used in the filter analysis, specified as 'double', 'single', or 'Fixed'. When the arithmetic input is not specified and the filter System object is unlocked, the analysis tool assumes a double-precision filter. When the arithmetic input is not specified and the System object is locked, the function performs the analysis based on the data type of the locked input.

The 'Fixed' value applies to filter System objects with fixed-point properties only.

When the 'Arithmetic' input argument is specified as 'Fixed' and the filter object has the data type of the coefficients set to 'Same word length as input', the arithmetic analysis depends on whether the System object is unlocked or locked.

- **unlocked** -- The analysis object function cannot determine the coefficients data type. The function assumes that the coefficients data type is signed, has a 16-bit word length, and is auto scaled. The function performs fixed-point analysis based on this assumption.
- **locked** -- When the input data type is 'double' or 'single', the analysis object function cannot determine the coefficients data type. The function assumes that the data type of the coefficients is signed, has a 16-bit word length, and is auto scaled. The function performs fixed-point analysis based on this assumption.

To check if the System object is locked or unlocked, use the `isLocked` function.



When the arithmetic input is specified as 'Fixed' and the filter object has the data type of the coefficients set to a custom numeric type, the object function performs fixed-point analysis based on the custom numeric data type.

## Output Arguments

### **phi** — Phase response

vector

Phase response vector of length  $n$ . If  $n$  is not specified, the function uses a default value of 8192. The phase response is evaluated at  $n$  points equally spaced around the upper half of the unit circle.

### **w** — frequencies

vector

Frequency vector of length  $n$ , in radians/sample.  $w$  consists of  $n$  points equally spaced around the upper half of the unit circle (from 0 to  $\pi$  radians/sample). If  $n$  is not specified, the function uses a default value of 8192.

## Version History

Introduced in R2011a

## See Also

### Functions

phasez

### Topics

“Analysis Methods for Filter System Objects” on page 3-2

## plot

**Package:** dsp

(To be removed) Plot pulse signal and metrics

---

**Note** `dsp.PulseMetrics` and `dsp.TransitionMetrics` objects will be removed in a future release. Use the functions from “Pulse and Transition Metrics” instead. To plot the metrics, call the functions with no output arguments. For more information, see “Compatibility Considerations”.

---

### Syntax

```
plot(pm)
```

### Description

`plot(pm)` plots the signal and metrics resulting from the last call of the object algorithm.

By default `plot` displays:

- the low- and high-state levels and the state-level boundaries defined by the `PercentStateLevelTolerance` property.
- the lower-, middle-, and upper-reference levels.
- the locations of the mid-reference level crossings of the positive (+) and negative (-) transitions of each detected pulse.

When the `TransitionOutputPort` property of the object is set to `true`, the locations of the upper and lower crossings are also plotted. When the `PreshootOutputPort` or `PostShootOutputPort` properties are set to `true`, the corresponding overshoots and undershoots are plotted as inverted or noninverted triangles. When the `SettlingOutputPort` property is set to `true`, the locations where the signal enters and remains within the lower- and upper-state boundaries over the specified seek duration are plotted.

### Examples

#### Slew Rates for 2.3 V Digital Clock

Find the slew rates of the leading and trailing edges of a 2.3 V digital clock sampled at 4 MHz.

```
load('pulseex.mat','x','t');
```

Construct the `dsp.PulseMetrics` object. Set the `TransitionOutputPort` property to `true` to report transition metrics for the initial and final transitions. Set the `StateLevelsSource` property to `'Auto'` to estimate the state levels from the data.

```
pm = dsp.PulseMetrics('SampleRate',4e6, ...
    'TransitionOutputPort', true, ...
    'StateLevelsSource', 'Auto');
```

Compute the pulse and transition metrics and plot the result.

```
[pulse,transition] = pm(x);  
plot(pm);
```

## Input Arguments

**pm** — Signal and metrics object

`dsp.PulseMetrics` | `dsp.TransitionMetrics`

Signal and metrics object, specified as one of the following:

- `dsp.PulseMetrics`
- `dsp.TransitionMetrics`

## Version History

**Introduced in R2012a**

**R2021b: `dsp.PulseMetrics` and `dsp.TransitionMetrics` System objects will be removed**

*Warns starting in R2021b*

`dsp.PulseMetrics` and `dsp.TransitionMetrics` System objects will be removed in a future release. Use the functions from “Pulse and Transition Metrics” instead. To plot the metrics, call these functions with no output arguments.

### Update Code

This table shows typical usage of the System object and explains how to update existing code to use these functions.

| Discouraged Usage  | Recommended Replacement  |
|--|--|
| <pre>load('clockx.mat','x','t'); pm = dsp.PulseMetrics('SampleRate',4e6,...     'TransitionOutputPort',true,...     'StateLevelsSource','Auto',...     'CycleOutputPort',true); [pulse, cycle, transition] = step(pm,x)  pulse =      struct with fields:          PositiveCross: [4x1 double]         NegativeCross: [4x1 double]            Width: [4x1 double]         RiseTime: [4x1 double]         FallTime: [4x1 double]  cycle =      struct with fields:          Period: [4x1 double]         Frequency: [4x1 double]         Separation: [4x1 double]            Width: [4x1 double]         DutyCycle: [4x1 double]  transition =      struct with fields:          Duration: [4x2 double]         Polarity: [4x2 double]         SlewRate: [4x2 double]         MiddleCross: [4x2 double]         LowerCross: [4x2 double]         UpperCross: [4x2 double]</pre> | <p>MATLAB code using the replacement functions is shown below.</p> |
| <p>If you are using a release prior to R2016b, replace <code>pm(x)</code> with <code>step(pm,x)</code>.</p>  |  |

| <b>Discouraged Usage</b>                       | <b>Recommended Replacement</b>  |
|--|---|
| <b>Plot the pulse</b><br><code>plot(pm)</code> | <b>Plot the pulse</b><br>To plot the individual pulse metrics, call the metrics functions without an output argument.<br>Some examples below:<br><code>dutycycle(x,t)</code><br><code>midcross(x,t)</code><br><code>pulseperiod(x,t)</code><br><code>pulsesep(x,t)</code><br><code>pulsewidth(x,t)</code> |

## plot

**Package:** dsp

(To be removed) Plot signal, state levels, and histogram

---

**Note** `dsp.StateLevels` will be removed in a future release. Use `statelevels` instead. To plot the state levels, call the `statelevels` function with no output arguments. For more information, see “Compatibility Considerations”.

---

### Syntax

```
plot(sl)
```

### Description

`plot(sl)` plots the signal and state levels computed in the last call to the algorithm. If the `Method` property of the state levels object is set to 'Histogram mode' or 'Histogram Mean', the histogram is plotted in a subplot below the signal.

### Examples

#### State Levels of 2.3 V Underdamped Noisy Clock

Compute and plot the state levels of a 2.3 V underdamped noisy clock. Load the clock data in the variable, `x`, and the sampling instants in the variable `t`.

```
load('clockex.mat','x','t');
```

Estimate the state levels.

```
sl = dsp.StateLevels;  
levels = sl(x);
```

Plot the clock data along with the estimated state levels and histograms.

```
plot(sl)
```

### Input Arguments

**sl** — State levels object

`dsp.StateLevels`

State levels object, specified as a `dsp.StateLevels` System object.

### Version History

Introduced in R2012a

**R2021b: dsp.StateLevels System object will be removed**

*Warns starting in R2021b*

`dsp.StateLevels` System object will be removed in a future release. To plot the state levels, call the `statelevels` function with no output arguments.

**Update Code**

This table shows typical usage of the System object and explains how to update existing code to use the `statelevels` function.

| Discouraged Usage   | Recommended Replacement   |
|---|---|
| <pre>load('clockex.mat', 'x', 't');<br/>slevel = dsp.StateLevels;</pre> <p><b>Plot state levels and the corresponding histogram</b></p> <pre>figure;<br/>plot(slevel)</pre> | <p><b>Plot state levels and the corresponding histogram</b></p> <pre>figure;<br/>statelevels(x)</pre> |

# polyphase

**Package:** dsp

Polyphase decomposition of multirate filter

## Syntax

```
p = polyphase(sysobj)
p = polyphase(sysobj, 'Arithmetic', arithType)
polyphase(sysobj)
```

## Description

`p = polyphase(sysobj)` returns the polyphase matrix `p` of the multirate filter System object `sysobj`. Each row in the matrix corresponds to a polyphase branch. The number of columns in `p` corresponds to the number of filter taps per polyphase branch.

`p = polyphase(sysobj, 'Arithmetic', arithType)` returns the polyphase matrix `p` in the precision set by the `arithType`.

`polyphase(sysobj)` launches the Filter Visualization Tool (`fvtool`) with all the polyphase subfilters to allow you to analyze each component subfilter individually.

## Examples

### Polyphase Matrix of an FIR Interpolator

When you create a multirate filter that uses polyphase decomposition, `polyphase` lets you analyze the component filters individually by returning the components as rows in a matrix. First, create an interpolate-by-three filter.

```
hs = dsp.FIRInterpolator
```

```
hs =
  dsp.FIRInterpolator with properties:
    InterpolationFactor: 3
    NumeratorSource: 'Property'
    Numerator: [0 -1.2906e-04 -2.2804e-04 0 5.5461e-04 8.0261e-04 0 -0.0015 -0.0020 0 0]
    Show all properties
```

In this syntax, the matrix `p` contains all of the subfilters for `hm`, one filter per matrix row.

```
p = polyphase(hs)
```

```
p = 3×24
```

```

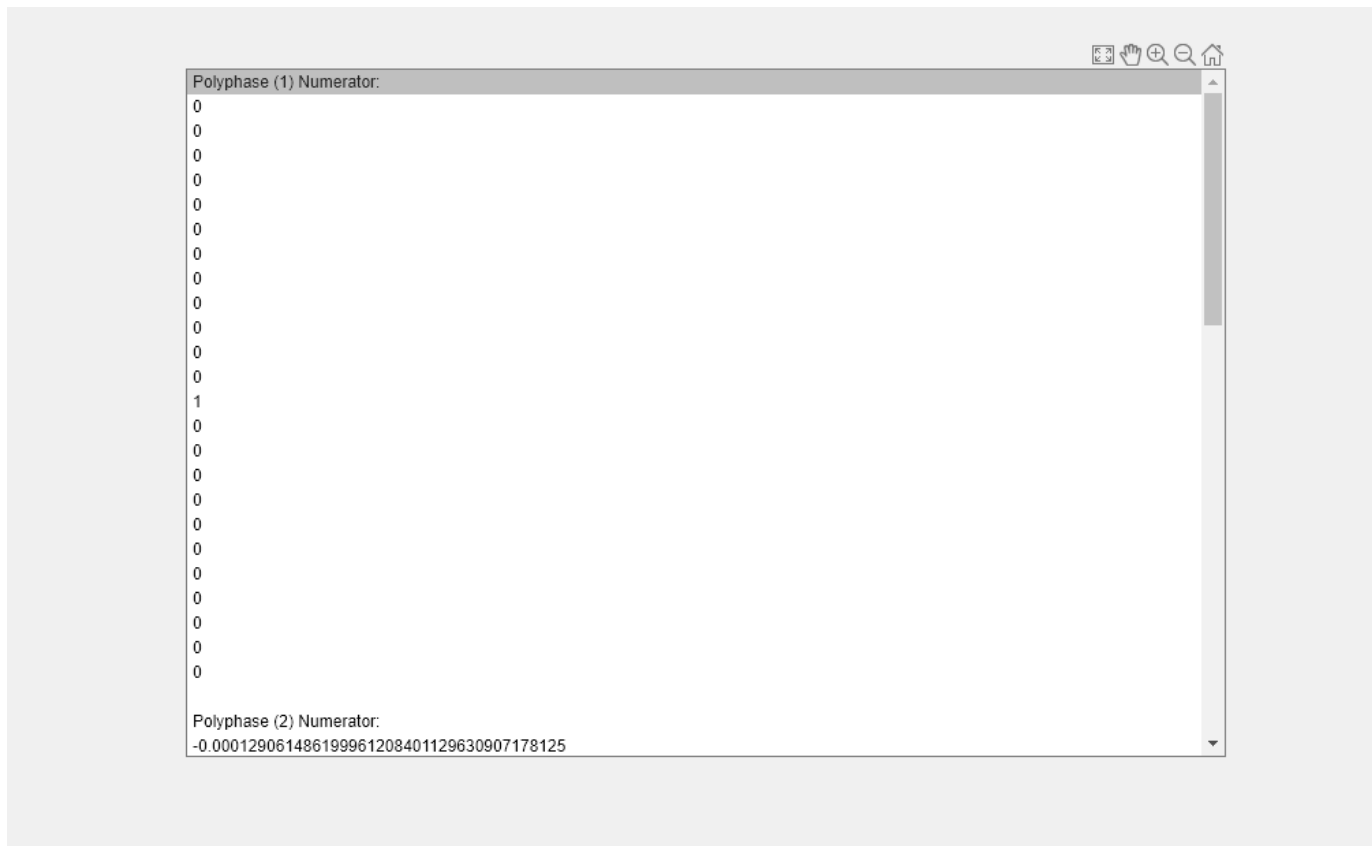
      0      0      0      0      0      0      0      0      0
-0.0001  0.0006 -0.0015  0.0034 -0.0067  0.0121 -0.0205  0.0332 -0.0530  0.0732  0.0928  0.1119  0.1305  0.1486  0.1662  0.1834  0.2001  0.2164  0.2322  0.2476  0.2626  0.2772  0.2914  0.3052  0.3186  0.3316  0.3442  0.3564  0.3682  0.3796  0.3906  0.4012  0.4114  0.4212  0.4306  0.4396  0.4482  0.4564  0.4642  0.4716  0.4786  0.4852  0.4914  0.4972  0.5026  0.5076  0.5122  0.5168  0.5212  0.5252  0.5288  0.5322  0.5352  0.5378  0.5402  0.5422  0.5438  0.5452  0.5462  0.5468  0.5472  0.5476  0.5478  0.5478  0.5476  0.5472  0.5468  0.5462  0.5452  0.5438  0.5422  0.5402  0.5378  0.5352  0.5322  0.5288  0.5252  0.5212  0.5168  0.5122  0.5076  0.5026  0.4972  0.4914  0.4852  0.4786  0.4716  0.4642  0.4564  0.4482  0.4396  0.4306  0.4212  0.4114  0.4012  0.3906  0.3796  0.3682  0.3564  0.3442  0.3316  0.3186  0.3052  0.2914  0.2772  0.2626  0.2476  0.2322  0.2164  0.2001  0.1834  0.1662  0.1486  0.1305  0.1119  0.0928  0.0732  0.0530  0.0332  0.0121  0.0067  0.0034  0.0015  0.0006  0.0001  0
```



-0.0002    0.0008    -0.0020    0.0043    -0.0083    0.0145    -0.0241    0.0388    -0.0620    0.1

Finally, using `polyphase` without an output argument opens the Filter Visualization Tool, ready for you to use the analysis capabilities of the tool to investigate the interpolator `hm`.

`polyphase(hs)`



The `fvtool` shows the coefficients of the subfilters. To see the magnitude response of the subfilters, click on the **Magnitude Response** button on the `fvtool` toolstrip.

## Input Arguments

### **sysobj** — Input filter

filter System object

Input filter, specified as as one of the following filter System objects:

- `dsp.CICCompensationDecimator`
- `dsp.CICCompensationInterpolator`
- `dsp.FIRDecimator`
- `dsp.FIRHalfbandDecimator`
- `dsp.FIRHalfbandInterpolator`

- `dsp.FIRInterpolator`
- `dsp.FIRRateConverter`
- `dsp.IIRHalfbandDecimator`
- `dsp.IIRHalfbandInterpolator`

### **arithType — Arithmetic type**

'double' (default) | 'single' | 'Fixed'

Specify the arithmetic used in computing the polyphase matrix. When you specify 'double' or 'single', the function performs double- or single-precision analysis. When you specify 'fixed', the arithmetic changes depending on the setting of the `CoefficientDataType` property and whether the System object is locked or unlocked.

#### **Details for Fixed-Point Arithmetic**

| <b>System Object State</b> | <b>Coefficient Data Type</b> | <b>Rule</b>  |
|----------------------------|------------------------------|--|
| Unlocked                   | 'Same as input'              | The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption.  |
| Unlocked                   | 'Custom'                     | The function performs fixed-point analysis based on the setting of the <code>CustomCoefficientsDataType</code> property.   |
| Locked                     | 'Same as input'              | When the input data type is 'double' or 'fixed', the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption. |
| Locked                     | 'Custom'                     | The function performs fixed-point analysis based on the setting of the <code>CustomCoefficientsDataType</code> property.   |

When you do not specify the arithmetic for non-CIC structures, the function uses double-precision arithmetic if the filter System object is in an unlocked state. If the System object is locked, the function performs analysis based on the locked input data type. CIC structures only support fixed-point arithmetic.

## **Output Arguments**

**p — polyphase matrix**

matrix

Polyphase matrix  $p$  of the multirate filter. Each row in the matrix corresponds to a polyphase branch. The first row of matrix  $p$  represents the first polyphase branch, the second row the second polyphase branch, and so on to the last polyphase branch. The number of columns in  $p$  corresponds to the number of filter taps per polyphase branch.

## **Version History**

**Introduced in R2011a**

## **See Also**

### **Topics**

“Analysis Methods for Filter System Objects” on page 3-2

# polyphase

**Package:** dsp

Return polyphase matrix

## Syntax

`p = polyphase(obj)`

## Description

`p = polyphase(obj)` returns the polyphase matrix used by the filter bank in `dsp.Channelizer` and `dsp.ChannelSynthesizer` System objects. Each row in the matrix corresponds to a polyphase branch. The number of columns in `p` corresponds to the number of filter taps per branch.

## Examples

### Polyphase Matrix of Filter Bank

Compute the polyphase matrix of the filter bank used by the channelizer.

Design a channelizer with the number of frequency bands or polyphase branches set to 8, the number of taps or coefficients per band set to 12, and stopband attenuation set to 80 dB.

```
channelizer = dsp.Channelizer;
p = polyphase(channelizer)
```

`p = 8×12`

|         |        |         |        |         |        |        |         |        |         |  |
|---------|--------|---------|--------|---------|--------|--------|---------|--------|---------|--|
|         | 0      | 0       | 0      | 0       | 0      | 0      | 0.1250  | 0      | 0       |  |
| -0.0000 | 0.0002 | -0.0007 | 0.0022 | -0.0056 | 0.0161 | 0.1216 | -0.0119 | 0.0045 | -0.0000 |  |
| -0.0000 | 0.0004 | -0.0015 | 0.0045 | -0.0117 | 0.0354 | 0.1118 | -0.0192 | 0.0073 | -0.0000 |  |
| -0.0001 | 0.0006 | -0.0023 | 0.0067 | -0.0172 | 0.0565 | 0.0966 | -0.0220 | 0.0085 | -0.0000 |  |
| -0.0001 | 0.0008 | -0.0029 | 0.0082 | -0.0210 | 0.0776 | 0.0776 | -0.0210 | 0.0082 | -0.0000 |  |
| -0.0001 | 0.0009 | -0.0031 | 0.0085 | -0.0220 | 0.0966 | 0.0565 | -0.0172 | 0.0067 | -0.0000 |  |
| -0.0001 | 0.0008 | -0.0027 | 0.0073 | -0.0192 | 0.1118 | 0.0354 | -0.0117 | 0.0045 | -0.0000 |  |
| -0.0001 | 0.0005 | -0.0017 | 0.0045 | -0.0119 | 0.1216 | 0.0161 | -0.0056 | 0.0022 | -0.0000 |  |

Each row in the matrix corresponds to a polyphase branch. The number of columns in the matrix corresponds to the number of filter taps per branch.

## Input Arguments

**obj** — Input filter System object

`dsp.Channelizer` | `dsp.ChannelSynthesizer`

Input filter, specified as either a `dsp.Channelizer` or `dsp.ChannelSynthesizer` System object.

Example: `channelizer = dsp.Channelizer;`

Example: `channelizer = dsp.ChannelSynthesizer`

## Output Arguments

### **p** — Polyphase matrix

matrix

Polyphase matrix of the filter bank, returned as a matrix of size  $[NFBands, NTPerBand]$ . The dimensions of the matrix depend on the type of System object in the `obj` argument:

- `dsp.Channelizer` -- *NFBands* is the value you specify in the `NumFrequencyBands` property, and *NTPerBand* is the value you specify in the `NumTapsPerBand` property.
- `dsp.ChannelSynthesizer` -- *NFBands* is the number of narrowband signals or the number of columns in the input signal, and *NTPerBand* is the value you specify in the `NumTapsPerBand` property.

## Version History

**Introduced in R2016b**

## See Also

### Functions

`tf` | `fvtool` | `freqz` | `coeffs` | `bandedgeFrequencies` | `centerFrequencies` | `getFilters`

### Objects

`dsp.Channelizer` | `dsp.ChannelSynthesizer`

## qreport

Most recent fixed-point filtering operation report

### Syntax

```
rlog = qreport(h)
```

### Description

`rlog = qreport(h)` returns the logging report stored in the filter object `h` in the object `rlog`. The ability to log features of the filtering operation is integrated in the fixed-point filter object and the `filter` method.

Each time you filter a signal with `h`, new log data overwrites the results in the filter from the previous filtering operation. To save the log from a filtering simulation, change the name of the output argument for the operation before subsequent filtering runs.

---

**Note** `qreport` requires Fixed-Point Designer software and that filter `h` is a fixed-point filter. Data logging for `fi` operations is a preference you set for each MATLAB session. To learn more about logging, `LoggingMode`, and `fi` object preferences, refer to `fipref` in the Fixed-Point Designer documentation.

---

Also, you cannot use `qreport` to log the filtering operations from a fixed-point Farrow filter.

---

Enable logging during filtering by setting `LoggingMode` to `on` for `fi` objects for your MATLAB session. Trigger logging by setting the `Arithmetic` property for `h` to `fixed`, making `h` a fixed-point filter and filtering an input signal.

### Using Fixed-Point Filtering Logging

Filter operation logging with `qreport` requires some preparation in MATLAB. Complete these steps before you use `qreport`.

- 1 Set the fixed-point object preference for `LoggingMode` to `on` for your MATLAB session. This setting enables data logging.  

```
fipref('LoggingMode','on')
```
- 2 Create your fixed-point filter.
- 3 Filter a signal with the filter.
- 4 Use `qreport` to return the filtering information stored in the filter object.

`qreport` provides a way to instrument your fixed-point filters and the resulting data log offers insight into how the filter responds to a particular input data signal.

Report object `rlog` contains a filter-structure-specific list of internal signals for the filter. Each signal contains

- Minimum and maximum values that were recorded during the last simulation. Minimum and maximum values correspond to values before quantization.

- Representable numerical range of the word length and fraction length format
- Number of overflows during filtering for that signal.

## Examples

### View the Logging Report

This example shows how to use `qreport` to log the results of filtering a sinusoidal signal with a fixed-point direct-form FIR filter, `firfilt`. To use the `qreport`, set the `LoggingMode` of fixed-point objects to 'on'.

```
fipref('loggingmode','on');
hd = design(fdesign.lowpass,'equiripple');
hd.arithmetic = 'fixed';
hd.InputWordLength = 32;
fs = 1000;           % Input sampling frequency.
t = 0:1/fs:1.5;     % Signal length = 1501 samples.
x = sin(2*pi*10*t); % Amplitude = 1 sinusoid.
y = filter(hd,x);
rlog = qreport(hd)
```

rlog =

| Fixed-Point Report |             |            |         |        |
|--------------------|-------------|------------|---------|--------|
|                    | Min         | Max        | Range   | Number |
| Input:             | -1          | 1          | -65536  | 65536  |
| Output:            | -1.02325    | 1.02325    | -131072 | 131072 |
| Product:           | -0.48538208 | 0.48538208 | -32768  | 32768  |
| Accumulator:       | -1.0852075  | 1.0852075  | -131072 | 131072 |

Restore the default logging mode preference.

```
fipref('loggingmode','off');
```

## Version History

Introduced in R2011a

## ccdf

Get coordinates of CCDF curves

### Syntax

```
[relpower,prob] = ccdf(meter)
```

### Description

[relpower,prob] = ccdf(meter) returns the coordinates of the CCDF curves.

### Examples

#### Plot Coordinates of CCDF Curves

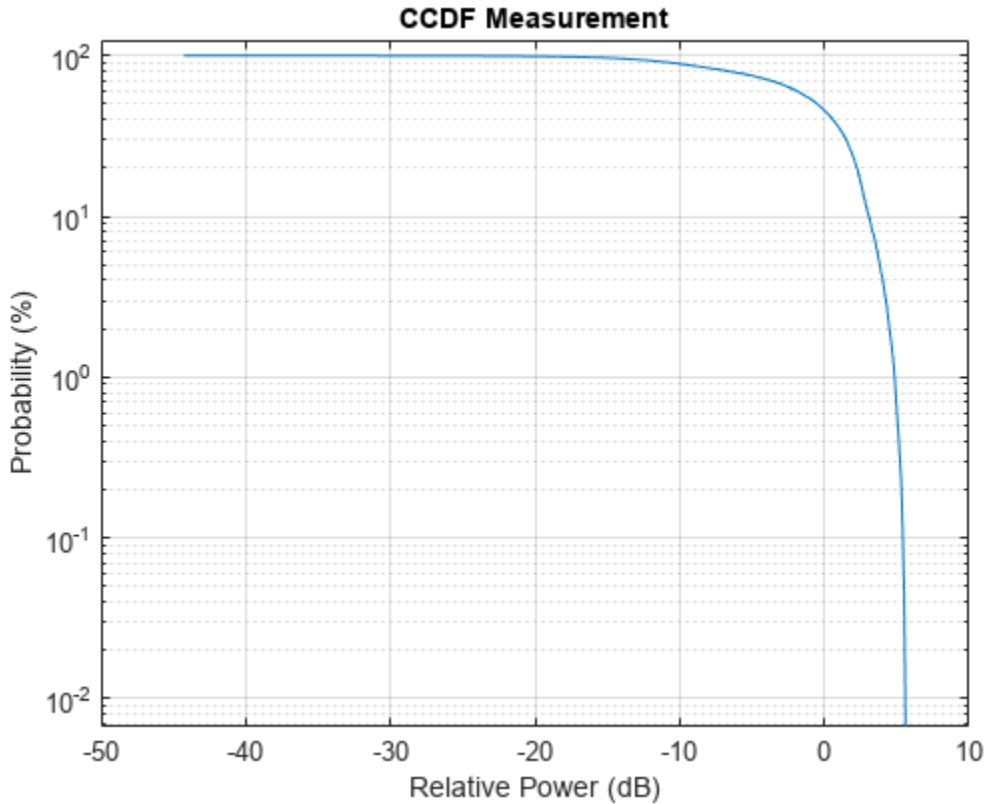
Compute the average power measurement of the generated voltage signal.

```
x = complex(rand(15000,1)-0.2,rand(15000,1)-0.2);  
pm = powermeter(ComputeCCDF=true);  
averagePower = pm(x); % power in dB
```

Plot the computed coordinates of the CCDF curve.

```
[relpower,prob] = ccdf(pm);  
semilogy(relpower,prob)  
xlabel('Relative Power (dB)');  
ylabel('Probability (%)');  
ylim([0 125]);  
grid on;  
title('CCDF Measurement');
```





## Input Arguments

### meter — Power meter

powermeter system object

Power meter, specified as a powermeter System object.

## Output Arguments

### relpower — Relative power

$N$ -by- $M$  matrix

Relative power in dB, returned as an  $N$ -by- $M$  matrix.  $M$  equals the number of columns in the signals provided and  $N$  equals  $\text{ceil}(\text{PowerRange}/\text{PowerResolution}) + 1$ .

Data Types: double

### prob — Probability

matrix

Probability in percentage, returned as a matrix. The probability that the instantaneous signal power of the  $j^{\text{th}}$  channel is above its average signal power by  $\text{relpower}(i,j)$ , in dB, is  $\text{prob}(i)/100$ .

Data Types: double

## **Version History**

Introduced in R2022a

### **See Also**

#### **Objects**

powermeter

#### **Functions**

plotCCDF | probability | relativePower

# plotCCDF

Plot CCDF curves

## Syntax

```
H = plotCCDF(meter)
H = plotCCDF(meter,ax)
H = plotCCDF(meter,GaussianReference=true)
H = plotCCDF( ____,Name=Value)
```

## Description

`H = plotCCDF(meter)` plots CCDF measurements in the `powermeter` System object `meter` and returns the plot handles as an output `H`. The number of curves plotted equals the number of input channels.

`H = plotCCDF(meter,ax)` plots CCDF measurements using the axes object `axis`.

`H = plotCCDF(meter,GaussianReference=true)`, uses the optional name-value argument `GaussianReference=true` to add a Gaussian reference curve, representing the power of a complex white Gaussian noise calculated as a chi-squared distribution, on the CCDF plot. By default `GaussianReference=false`.

`H = plotCCDF( ____,Name=Value)`, additionally you can use the name-value arguments of the `plot` function to customize the configuration of the CCDF plot. For example, `plotCCDF(meter,LineWidth=2)` plots the CCDF curves with a line width of 2 points.

## Examples

### Find Relative Power for Specified Probability

Compute the average power measurement of the generated voltage signal.

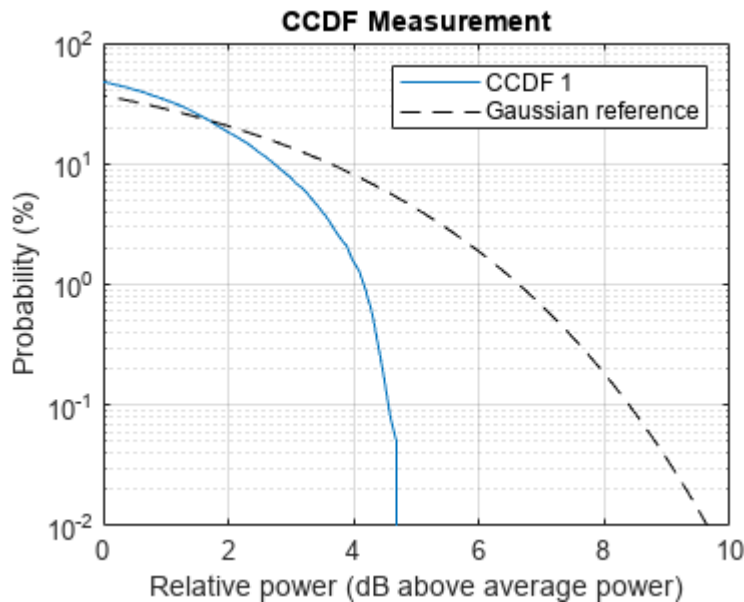
```
x = complex(rand(10000,1)-0.5,rand(10000,1)-0.5);
pm = powermeter(ComputeCCDF=true);
averagePower = pm(x); % power in dB
```

Compute the probability that the instantaneous power of the signal is 3 dB above the average power.

```
prob = probability(pm,3) % probability in percentage
prob = 7.6009
```

Plot the CCDF curve.

```
plotCCDF(pm,GaussianReference=true)
```



Find the relative power for the given probability.

```
relativepwr = relativePower(pm,prob)
relativepwr = 3.0000
```

## Input Arguments

### meter — Power meter

powermeter system object

Power meter, specified as a powermeter System object.

### ax — Axes object

axes object

Target axes, specified as an axes object. If you do not specify the axes, MATLAB plots into the current axes. The x-axis represents the relative power in dB, the ratio of instantaneous signal power to the average signal power. The y-axis represents the probability, in percentage, that the relative power exceeds a specified dB level.

## Output Arguments

### H — Plot handle output

line object

Plot handle output, returned as a line object. For more information about plot handle properties, see Line Properties.

## Version History

Introduced in R2022a

**See Also****Objects**

powermeter

**Functions**

ccdf | probability | relativePower

## probability

Use CCDF to find probability for specified relative power

### Syntax

```
prob = probability(meter,relpower)
```

### Description

`prob = probability(meter,relpower)` finds the probability `prob` in percent that the instantaneous signal power is above the average signal power by `relpower` dB.

### Examples

#### Find Relative Power for Specified Probability

Compute the average power measurement of the generated voltage signal.

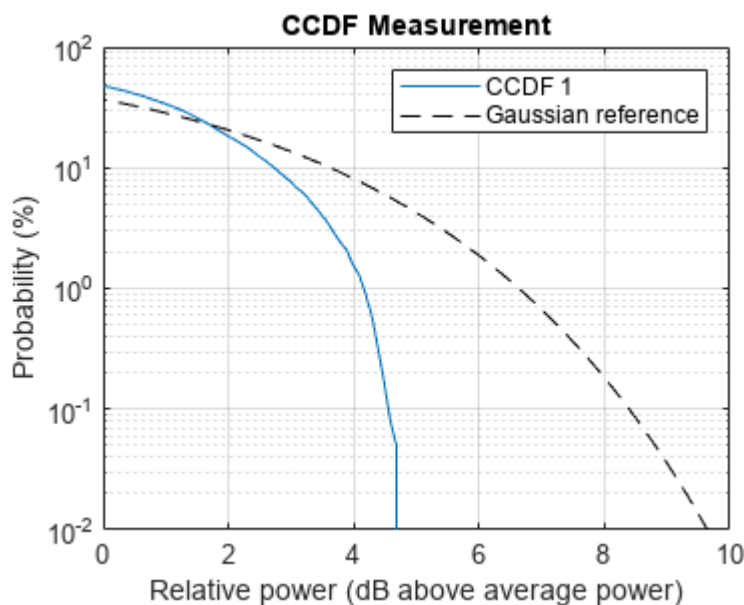
```
x = complex(rand(10000,1)-0.5,rand(10000,1)-0.5);
pm = powermeter(ComputeCCDF=true);
averagePower = pm(x); % power in dB
```

Compute the probability that the instantaneous power of the signal is 3 dB above the average power.

```
prob = probability(pm,3) % probability in percentage
prob = 7.6009
```

Plot the CCDF curve.

```
plotCCDF(pm,GaussianReference=true)
```



Find the relative power for the given probability.

```
relativepwr = relativePower(pm,prob)
```

```
relativepwr = 3.0000
```

## Input Arguments

### **meter** — Power meter

powermeter system object

Power meter, specified as a powermeter System object.

### **relpower** — Relative power

column vector

Relative power in dB, specified as a column vector.

Data Types: single | double

## Output Arguments

### **prob** — Probability

*N*-by-*M* matrix

Probability in percentage, returned as an *N*-by-*M* matrix. The probability that the instantaneous signal power of the *j*<sup>th</sup> channel is above its average signal power by `relpower(i,j)`, in dB, is `prob(i)/100`. *N* equals the number of rows in `relpower` and *M* equals the number of signals provided to the powermeter object.

If the `PowerRange` property is not set sufficiently large, the computed CCDF curve may not reach some relative power values in `relpower` and the corresponding values in `prob` is NaN.

Data Types: double

## Version History

Introduced in R2022a

## See Also

### **Objects**

powermeter

### **Functions**

plotCCDF | ccdf | relativePower

## relativePower

Use CCDF to find relative power for specified probability

### Syntax

```
relpower = relativePower(meter,prob)
```

### Description

`relpower = relativePower(meter,prob)` computes the relative power `relpower` for a specified probability `prob`, such that the instantaneous signal power is above the average signal power by `relpower` dB with probability `prob/100`. To enable this syntax, set the `computeCCDF` property to `true` in the `powermeter` System object.

### Examples

#### Find Relative Power for Specified Probability

Compute the average power measurement of the generated voltage signal.

```
x = complex(rand(10000,1)-0.5,rand(10000,1)-0.5);  
pm = powermeter(ComputeCCDF=true);  
averagePower = pm(x); % power in dB
```

Compute the probability that the instantaneous power of the signal is 3 dB above the average power.

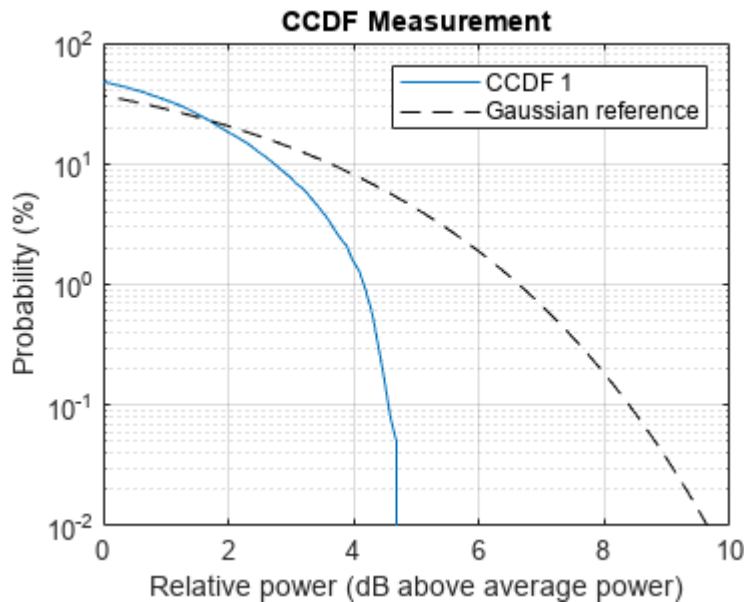
```
prob = probability(pm,3) % probability in percentage
```

```
prob = 7.6009
```

Plot the CCDF curve.

```
plotCCDF(pm,GaussianReference=true)
```





Find the relative power for the given probability.

```
relativepwr = relativePower(pm,prob)
relativepwr = 3.0000
```

## Input Arguments

### meter — Power meter

powermeter system object

Power meter, specified as a powermeter System object.

### prob — Probability

nonnegative column vector

Probability in percentage, specified as a nonnegative column vector. The probability that the instantaneous signal power of the  $j^{\text{th}}$  channel is above its average signal power by `relpower(i,j)`, in dB, is `prob(i)/100`.

If the `PowerRange` property is not set sufficiently large, the computed CCDF curve may not reach some probability values in `prob` and the corresponding values in `relpower` is NaN. If no `prob` value is provided, `relativePower` returns `relpower` for `prob = [10 1 0.1 0.01 0.001 0.0001]'`.

Data Types: single | double

## Output Arguments

### relpower — Relative power

$N$ -by- $M$  matrix

Relative power in dB, returned as an  $N$ -by- $M$  matrix.  $N$  equals the number of rows in `prob` and  $M$  equals the number of signals provided.

Data Types: double

## **Version History**

**Introduced in R2022a**

### **See Also**

#### **Objects**

powermeter

#### **Functions**

plotCCDF | ccdf | probability

# realizemdl

**Package:** dsp

Simulink subsystem block for filter

## Syntax

```
realizemdl(sysobj)
realizemdl(sysobj,Name,Value)
```

## Description

`realizemdl(sysobj)` generates a model of filter System object in a Simulink subsystem block using sum, gain, and delay blocks from Simulink. The properties and values of `sysobj` define the resulting subsystem block parameters.

`realizemdl` requires Simulink. To accurately realize models of quantized filters, use Fixed-Point Designer.

`realizemdl(sysobj,Name,Value)` generates the model for `sysobj` with additional options specified by one or more `Name,Value` pair arguments. Using name-value pair arguments lets you control more fully the way the block subsystem model gets built. You can specify such details as where the block goes, what the name is, or how to optimize the block structure.

## Examples

### Realize Simulink Model of a Lowpass Butterworth Filter

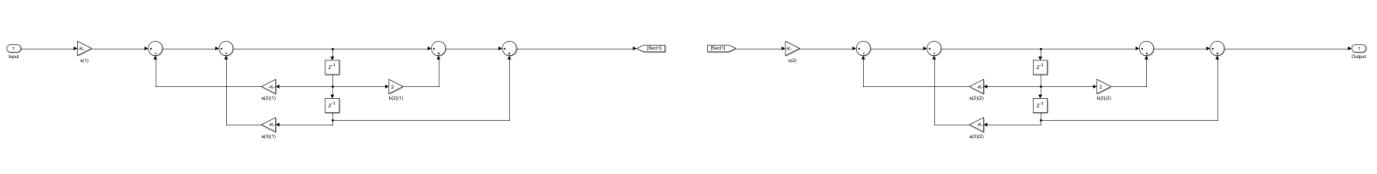
```
d = fdesign.lowpass('N,F3dB',4,0.25);
filterobject = design(d,'butter','systemobject',true);
```

Create a new model, `LPFilter.slx`, and realize the subsystem block in this model.

```
new_system('LPFilter');
realizemdl(filterobject);
```



View the block diagram by clicking on the subsystem block.

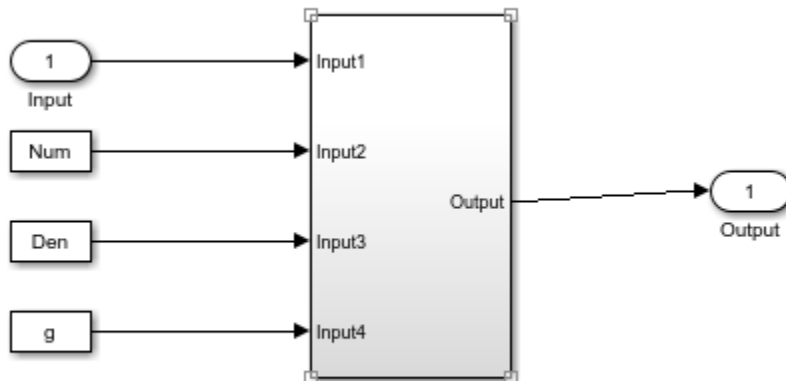


Create a new model, `LPFilterMapping.slx`, and realize the subsystem block, with coefficients mapped to ports, in this model.

```
new_system('LPFilterMapping');
realizemdl(filterobject, 'MapCoeffsToPorts', 'on');
```



View the block diagram by clicking on the subsystem block.



In this case, the filter is an IIR filter with a direct form II second-order sections structure. Setting `MapCoeffstoPorts` to 'on' exports the numerator coefficients, the denominator coefficients, and the gains to the MATLAB® workspace using the default variable names `Num`, `Den`, and `g`. Each column of `Num` and `Den` represents one second-order section. You can modify the filter coefficients directly in the MATLAB workspace providing tunability to the realized Simulink model.

## Input Arguments

### **sysobj** – Filter System object

filter System object

List of filter system objects that the function supports:

| Filter System objects                        |
|--|
| <code>dsp.AllpassFilter</code>               |
| <code>dsp.AllpoleFilter</code>               |
| <code>dsp.BiquadFilter</code>                |
| <code>dsp.CICCompensationDecimator</code>    |
| <code>dsp.CICCompensationInterpolator</code> |

| <b>Filter System objects</b> |
|------------------------------|
| dsp.CoupledAllpassFilter     |
| dsp.FarrowRateConverter      |
| dsp.FilterCascade            |
| dsp.FIRFilter                |
| dsp.FIRInterpolator          |
| dsp.FIRDecimator             |
| dsp.FIRRateConverter         |
| dsp.FIRHalfbandDecimator     |
| dsp.FIRHalfbandInterpolator  |
| dsp.FourthOrderSectionFilter |
| dsp.HighpassFilter           |
| dsp.IIRFilter                |
| dsp.IIRHalfbandDecimator     |
| dsp.IIRHalfbandInterpolator  |
| dsp.LowpassFilter            |
| dsp.NotchPeakFilter          |
| dsp.SOSFilter                |

### **Name-Value Pair Arguments**

```
Example: d = fdesign.lowpass('N,F3dB',4,0.25); filterobject =
design(d,'butter','systemobject',true);
realizemdl(filterobject,'MapCoeffsToPorts','on');
```

### **Destination – Destination choices**

'current' (default) | 'new' | character vector | string scalar

Specify whether to add the block to your current Simulink model or create a new model to contain the block. If you provide the name of a current subsystem as a character vector or a string scalar, the `realizemdl` function adds the new block to the specified subsystem.

### **BlockName – Name of the block**

'filter' (default) | character vector | string scalar

Provide the name for the new subsystem block. By default the block is named Filter.

### **MapCoeffsToPorts – Map coefficients**

'off' (default) | 'on'

Specify whether to map the coefficients of the filter to the ports of the block.

### **MapStates – Apply current filter states**

'off' (default) | 'on'

Specify whether to apply the current filter states to the realized model. Such specification allows you to save states from a filter object you may have used or configured in a specific way. The default

setting of 'off' means the states are not transferred to the model. Setting the property to 'on' preserves the current filter states in the realized model.

**OverwriteBlock — Overwrite existing block**

'off' (default) | 'on'

Specify whether to overwrite an existing block with the same name or create a new block.

**OptimizeZeros — Remove zero-gain blocks**

'off' (default) | 'on'

Specify whether to remove zero-gain blocks.

**OptimizeOnes — Replace unity-gain blocks**

'off' (default) | 'on'

Specify whether to replace unity-gain blocks with direct connections.

**OptimizeNegOnes — Replace negative unity-gain blocks**

'off' (default) | 'on'

Specify whether to replace negative unity-gain blocks with a sign change at the nearest sum block.

**OptimizeDelayChains — Replace delay chains**

'off' (default) | 'on'

Specify whether to replace delay chains made up of  $n$  unit delays with a single delay by  $n$ .

**CoeffNames — Names of coefficients**

{'Num'} (default FIR) | {'Num', 'Den'} (default direct form IIR) | {'Num', 'Den', 'g'} (default IIR SOS) | {'Num\_1', 'Num\_2', 'Num\_3' . . .} (default multistage) | {'K'} (default form lattice)

Specify the coefficient variable name as a cell array of character vectors. `MapCoeffsToPorts` must be set to 'on' for this property to apply.

**InputProcessing — Possible input processing options**

'columnsaschannels' | 'elementsaschannels'

Specify sample-based ('elementsaschannels') or frame-based ('columnsaschannels') processing.

**RateOption — Rate options**

'enforcesinglerate' (default) | 'allowmultirate'

Specify how the block adjusts the rate at the output to accommodate the reduced number of samples.

**Dependencies**

This parameter applies only when `InputProcessing` is 'columnsaschannels'.

**Arithmetic — Value types**

'double' | 'single'

The arithmetic for System object inputs must be 'double' or 'single'.

## Limitations

The destination must be a Simulink model. The function does not support a library file destination.

## Version History

Introduced in R2011a

## See Also

### Functions

design | fdesign

## rebuffer\_delay

Number of samples of delay introduced by buffering and unbuffering operations

### Syntax

```
d = rebuffer_delay(f,n,v)
d = rebuffer_delay(f,n,v,'mode')
```

### Description

`d = rebuffer_delay(f,n,v)` returns the delay, in samples, introduced by the Buffer or Unbuffer block in multitasking operations.

`d = rebuffer_delay(f,n,v,'mode')` returns the delay, in samples, introduced by the Buffer or Unbuffer block in the specified tasking mode.

### Input Arguments

**f**

Frame size of the input to the Buffer or Unbuffer block.

**n**

Size of the output buffer. Specify one of the following:

- The value of the **Output buffer size** parameter, if you are computing the delay introduced by a Buffer block.
- 1, if you are computing the delay introduced by an Unbuffer block.

**v**

Amount of buffer overlap. Specify one of the following:

- The value of the **Buffer overlap** parameter, if you are computing the delay introduced by a Buffer block.
- 0, if you are computing the delay introduced by an Unbuffer block.

**'mode'**

The tasking mode of the model. Specify one of the following options:

- 'singletasking'
- 'multitasking'

**Default:** 'multitasking'

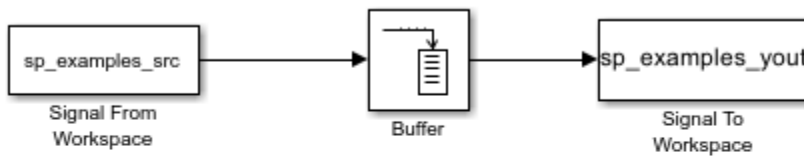
### Examples



### Compute Delay Introduced by Buffer Block

Compute the delay introduced by a Buffer block in a multitasking model.

Open the `ex_buffer_tut4` model.



Copyright 2019 The Mathworks, Inc

**Note:** This model creates the workspace variables "sp\_examples\_src" and "sp\_examples\_yout". Closing the model clears both variables from your workspace.

Double-click the Buffer block to open the block mask. Verify that you have the following settings:

- **Output buffer size** = 3
- **Buffer overlap** = 1
- **Initial conditions** = 0

Based on these settings, two of the required inputs to the `rebuffer_delay` function are as follows:

- $n = 3$
- $v = 1$

To determine the frame size of the input signal to the Buffer block, open the Signal From Workspace block mask. Verify that you have the following settings:

- **Signal** = `sp_examples_src`
- **Sample time** = 1
- **Samples per frame** = 4

Because **Samples per frame** = 4, you know the `f` input to the `rebuffer_delay` function is 4. After you verify the values of all the inputs to the `rebuffer_delay` function, determine the delay that the Buffer block introduces in this multitasking model. To do so, type the following at the MATLAB command line:

```
rebuffer_delay(4,3,1)
```

```
ans =
```

```
8
```

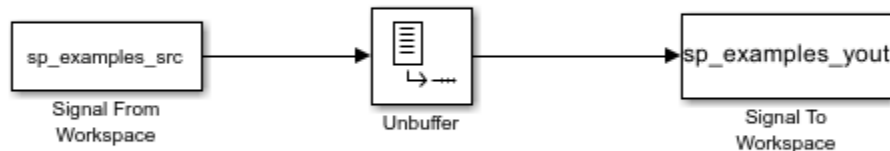
### Compute Delay Introduced by Unbuffer Block

Compute the delay introduced by an Unbuffer block in a multitasking model.

Open the `ex_unbuffer_ref1` model.

### Unbuffer Example 1

In this example, the Unbuffer block unbuffers a frame-based signal with frame size 3 to a sample-based signal.



**Note:** This model creates workspace variables called "sp\_examples\_src" and "sp\_examples\_yout". Closing the model clears both variables from your workspace.

To determine the frame size of the input to the Buffer block, open the Signal From Workspace block mask by double-clicking the block in your model. Verify that you have the following settings:

- **Signal** = `sp_examples_src`
- **Sample time** = 1
- **Samples per frame** = 3

Because **Samples per frame** = 3, you know the `f` input to the `rebuffer_delay` function is 3. Use the `rebuffer_delay` function to determine the amount of delay that the Unbuffer block introduces in this multitasking model. To compute the delay introduced by the Unbuffer block, use `f = 3`, `n = 1` and `v = 0`.

```
rebuffer_delay(3,1,0)
```

```
ans =
```

```
3
```

## More About

### Multitasking

When you run a model in `MultiTasking` mode, Simulink processes groups of blocks with the same execution priority through each stage of simulation based on task priority. Multitasking mode helps to create valid models of real-world multitasking systems, where sections of your model represent concurrent tasks. The **Treat each discrete rate as a separate task** parameter on the Solver (Simulink) pane of the Configuration Parameters dialog box controls this setting.

## Singletasking

When you run a model in `SingleTasking` mode, Simulink processes all blocks through each stage of simulation together. The **Treat each discrete rate as a separate task** parameter on the Solver (Simulink) pane of the Configuration Parameters dialog box controls this setting.

## Version History

Introduced before R2006a

## See Also

Buffer | Unbuffer

## Topics

“Buffer Delay and Initial Conditions”

## reffilter

Reference filter for fixed-point or single-precision filter

### Syntax

```
href = reffilter(hd)
```

### Description

`href = reffilter(hd)` returns a new filter `href` that has the same structure as `hd`, but uses the reference coefficients and has its arithmetic property set to `double`. Note that `hd` can be either a fixed-point filter (arithmetic property set to `'fixed'`), or a single-precision floating-point filter whose arithmetic property is `'single'`).

`reffilter(hd)` differs from `double(hd)` in that

- the filter `href` returned by `reffilter` has the reference coefficients of `hd`.
- `double(hd)` returns the quantized coefficients of `hd` represented in double-precision.

To check the performance of your fixed-point filter, use `href = reffilter(hd)` to quickly have the floating-point, double-precision version of `hd` available for comparison.

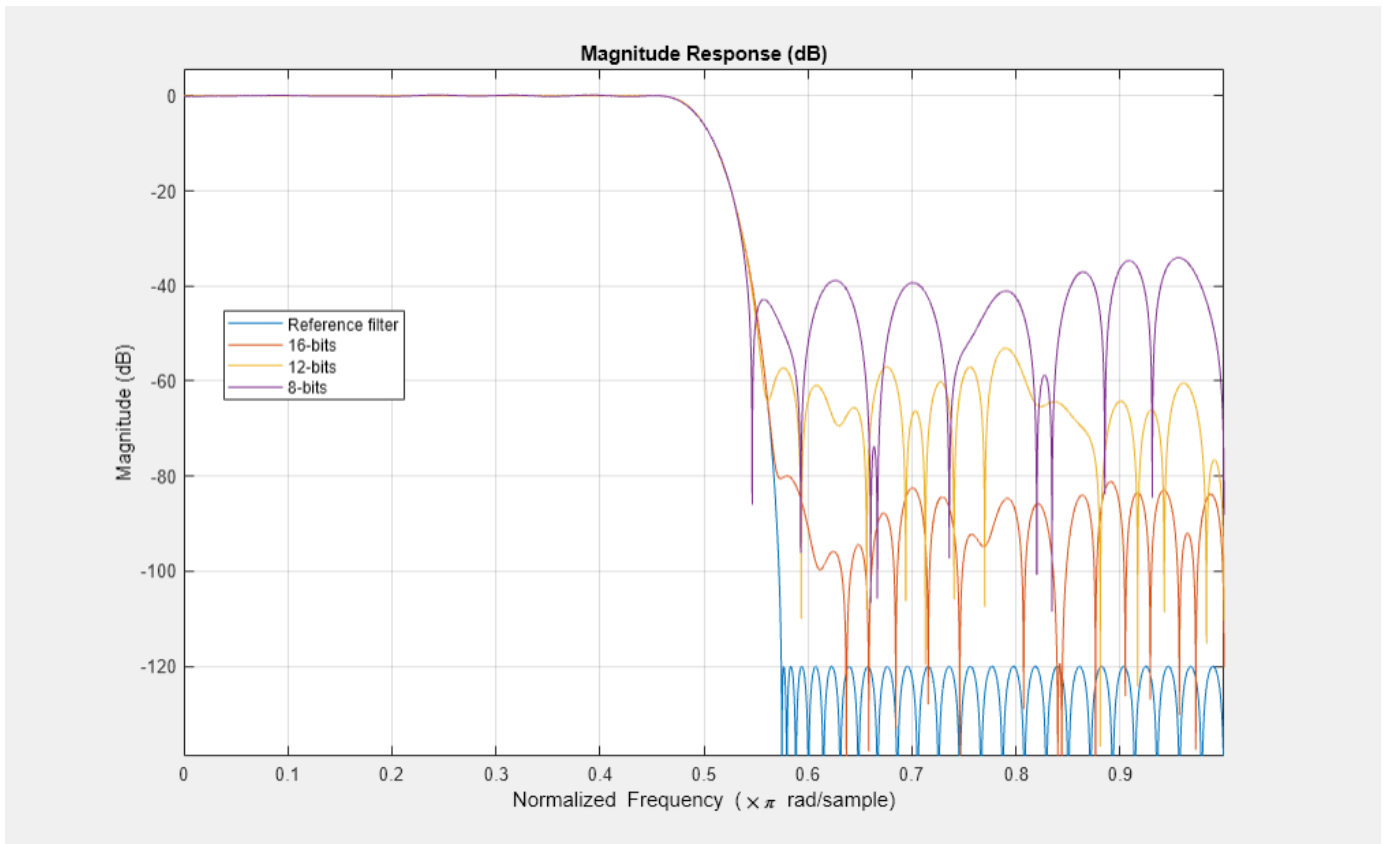
### Examples

#### Compare Fixed-point Quantizations of a Filter

Compare several fixed-point quantizations of a filter with the same double-precision floating-point version of the filter.

```
h = dfilt.dffir(firceqrip(87,.5,[1e-3,1e-6])); % Lowpass filter.
h1 = copy(h); h2 = copy(h); % Create copies of h.
h.arithmetic = 'fixed'; % Set h to filter using fixed-point...
                        % arithmetic.
h1.arithmetic = 'fixed'; % Same for h1.
h2.arithmetic = 'fixed'; % Same for h2.
h.CoeffWordLength = 16; % Use 16 bits to represent the...
                        % coefficients.
h1.CoeffWordLength = 12; % Use 12 bits to represent the...
                        % coefficients.
h2.CoeffWordLength = 8; % Use 8 bits to represent the...
                        % coefficients.

href = reffilter(h);
hfvt = fvtool(href,h,h1,h2);
set(hfvt,'ShowReference','off'); % Reference displayed once
                                % already.
legend(hfvt,'Reference filter','16-bits','12-bits','8-bits');
```



The fvtool shows href, the reference filter, and the effects of using three different word lengths to represent the coefficients.

## Version History

Introduced in R2011a

## reorder

**Package:** dsp

Reorder second-order sections of biquadratic filter System object

### Syntax

```
reorder(sysobj,order)
sysobjnew = reorder(sysobj,order)
reorder(sysobj,numorder,denorder)
reorder(sysobj,numorder,denorder,svorder)
reorder(sysobj,filter_type)
reorder(sysobj,dir_flag)
reorder(sysobj,dir_flag,sv)
reorder( __ , 'Arithmetic',arithType)
```

### Description

`reorder(sysobj,order)` rearranges the sections of the biquadratic filter object using the vector of indices provided in `order`.

`sysobjnew = reorder(sysobj,order)` generates a new biquadratic filter System object that contains the reordered sections. In this case, the original filter `sysobj` does not change.

`reorder(sysobj,numorder,denorder)` reorders the numerator and denominator of the filter separately using the vector of indices in `numorder` and `denorder`, respectively. These vectors must be of the same length.

`reorder(sysobj,numorder,denorder,svorder)` independently reorders the scale values of the biquadratic filter System object. If `svorder` is not specified, the scale values of the biquadratic filter are reordered in the same way as the numerator. The output scale value always remains at the end and `numorder` is used to reorder the scale values.

`reorder(sysobj,filter_type)` reorders `sysobj` in a way suitable for the specified filter type. This mode is intended for fixed-point implementations where the ordering of the sections can have a significant impact on the filter performance.

`reorder(sysobj,dir_flag)` rearranges the sections according to proximity to the origin of the poles of the sections.

`reorder(sysobj,dir_flag,sv)` reorders the scale values following the reordering of the poles when `sv` is set to 'poles', or following the reordering of the zeros when `sv` is set to 'zeros'. The scale values are not reordered when using the `dir_flag` option unless `sv` is specified.

`reorder( __ , 'Arithmetic',arithType)` assumes that the filter arithmetic is equal to `arithType`.

### Examples

## Reorder Biquadratic Filter

Being able to rearrange the order of the sections in a filter can be a powerful tool for controlling the filter design process. This example uses the `reorder` function to change the sections of a direct form II filter.

First use the automatic reordering option on a lowpass filter.

```
d = fdesign.lowpass('n,f3db',15,0.75);
sosFilt = design(d,'butter','SystemObject',true)
```

```
sosFilt =
  dsp.SOSFilter with properties:
    Structure: 'Direct form II'
    CoefficientSource: 'Property'
    Numerator: [8x3 double]
    Denominator: [8x3 double]
    HasScaleValues: true
    ScaleValues: [0.7948 0.7005 0.6306 0.5794 0.5430 0.5186 0.5046 0.7071 1]
```

Show all properties

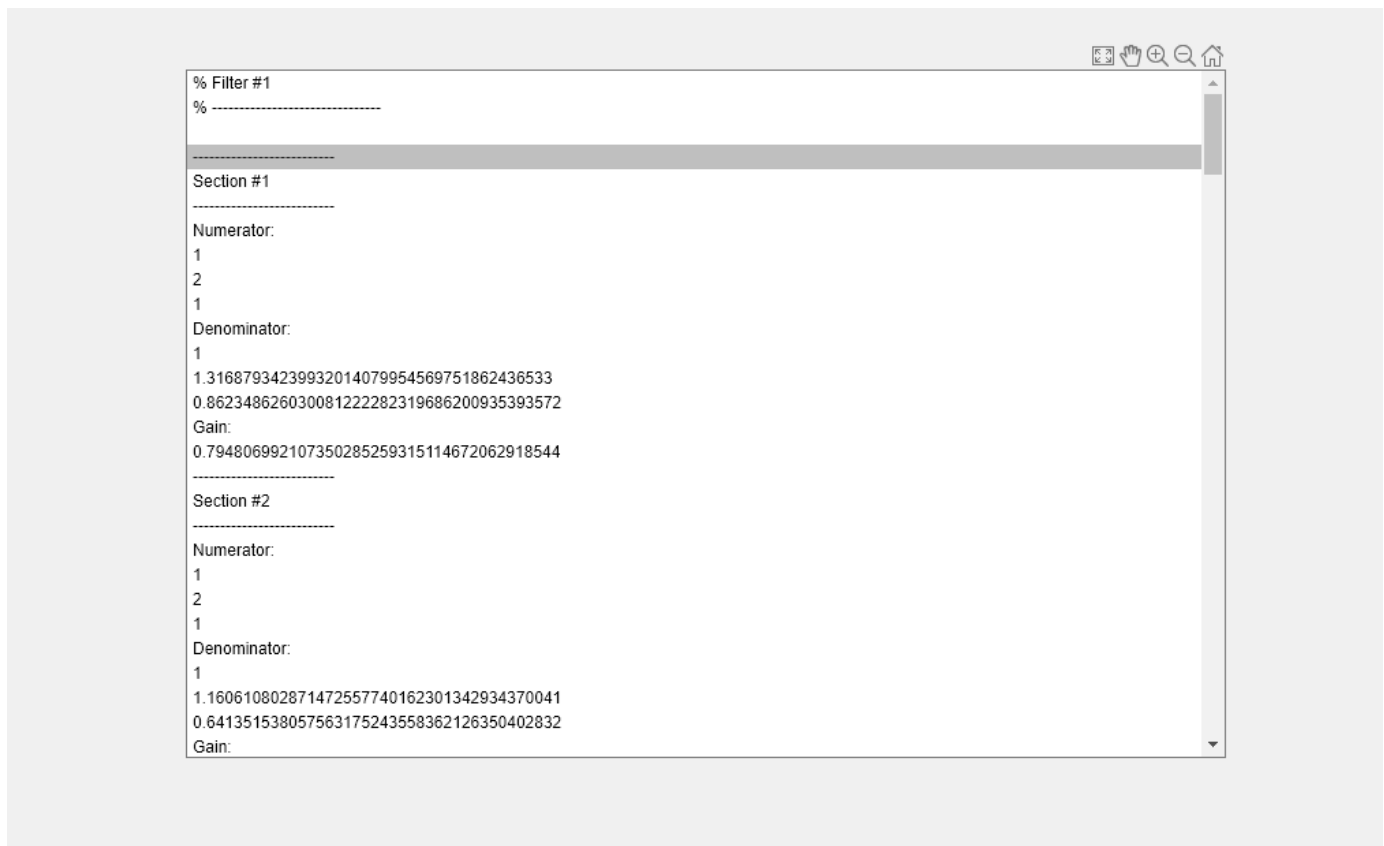
```
sosFiltreorder = reorder(sosFilt,'auto')
```

```
sosFiltreorder =
  dsp.SOSFilter with properties:
    Structure: 'Direct form II'
    CoefficientSource: 'Property'
    Numerator: [8x3 double]
    Denominator: [8x3 double]
    HasScaleValues: true
    ScaleValues: [0.6306 0.5046 0.5430 0.7948 0.5794 0.7071 0.5186 0.7005 1]
```

Show all properties

Now, use `reorder` to specify the new order for the sections.

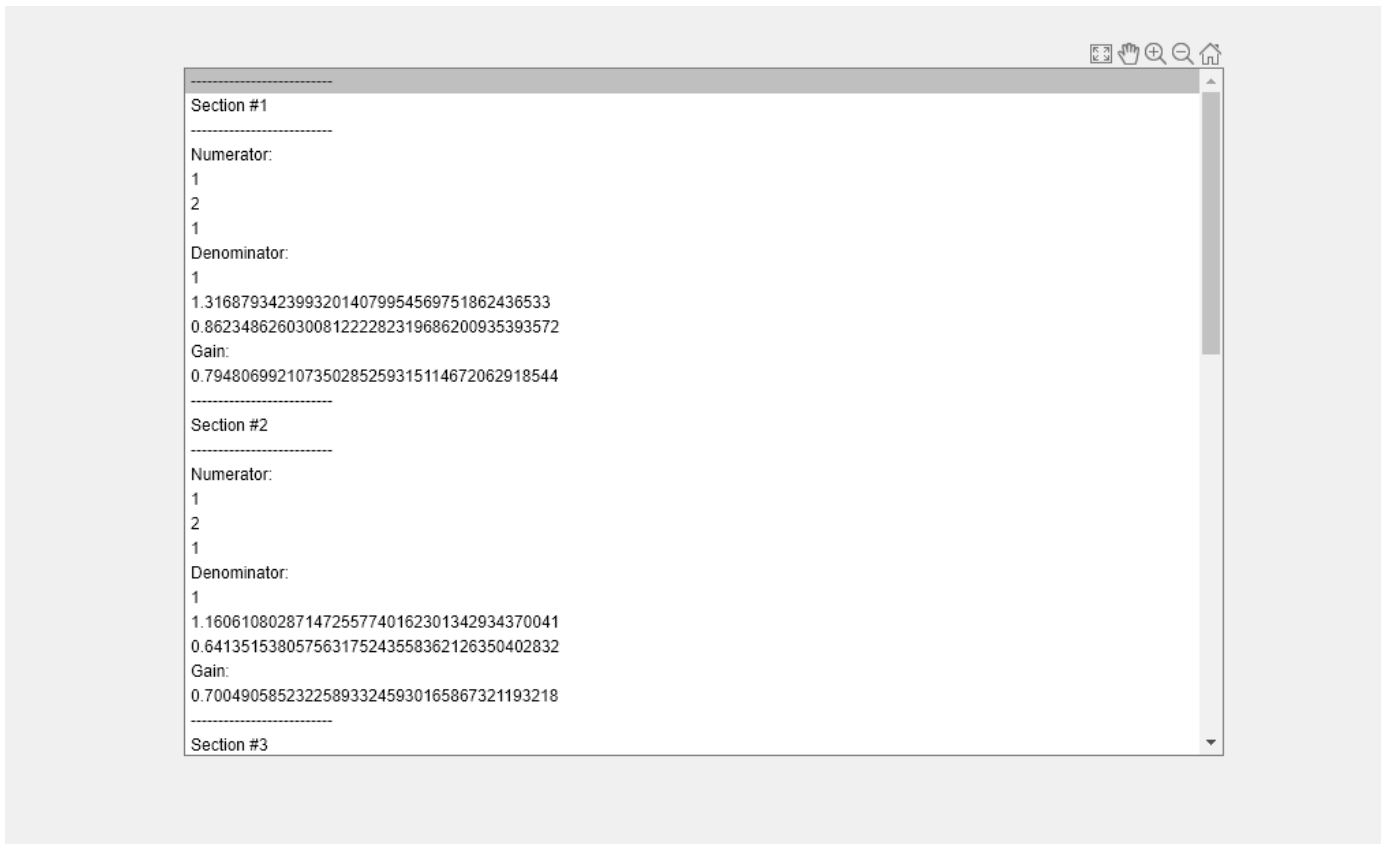
```
sosFiltspecifyreorder = reorder(sosFilt,[1 3:7 2 8]);
fvt = fvtool(sosFilt,sosFiltspecifyreorder,'analysis','coefficients');
```



Remove the third, fourth, and seventh sections.

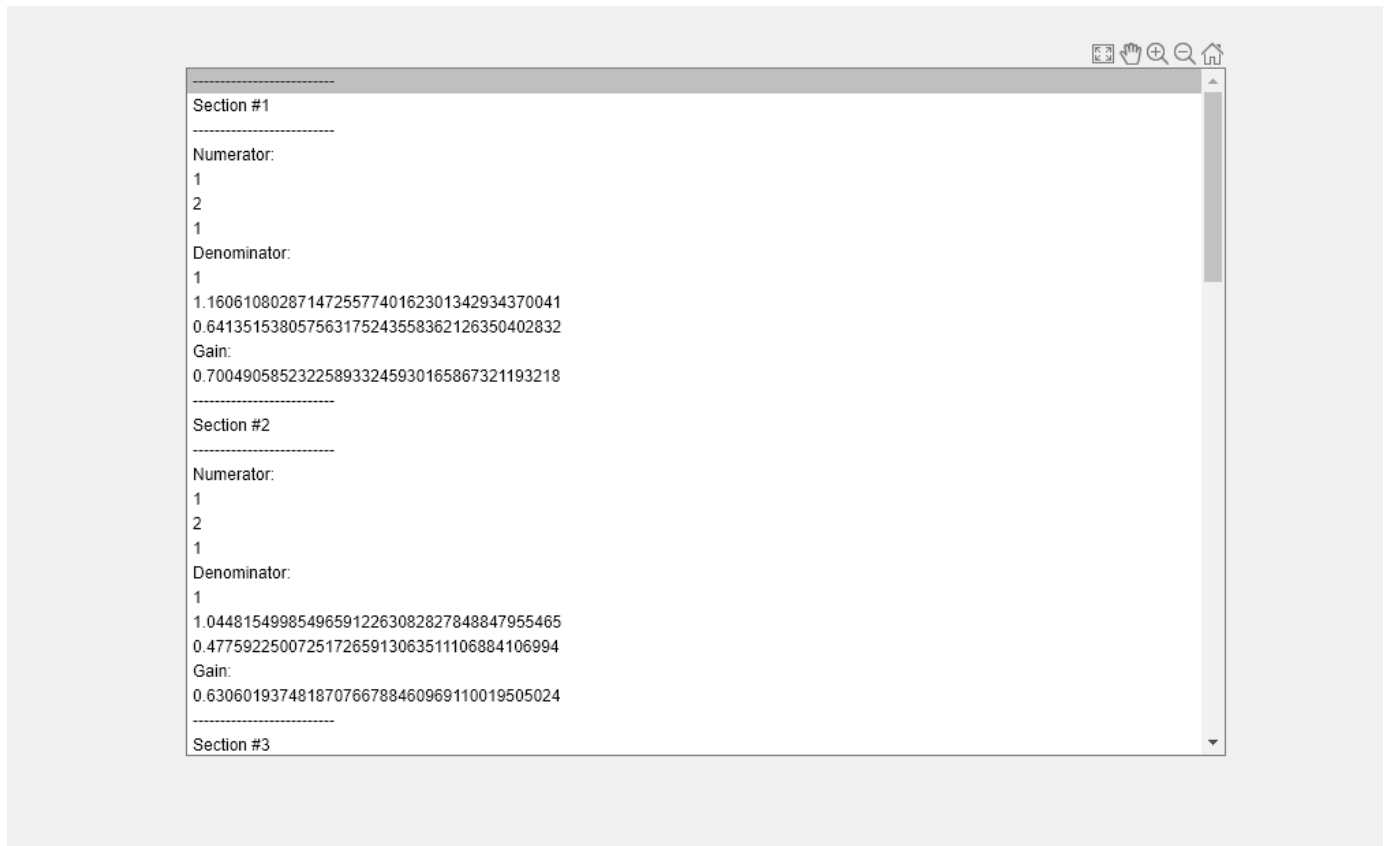
```
sosFiltclone1 = clone(sosFilt);
reorder(sosFiltclone1,logical([1 1 0 0 1 1 0 1]));
setfilter(fvt,sosFiltclone1);
```





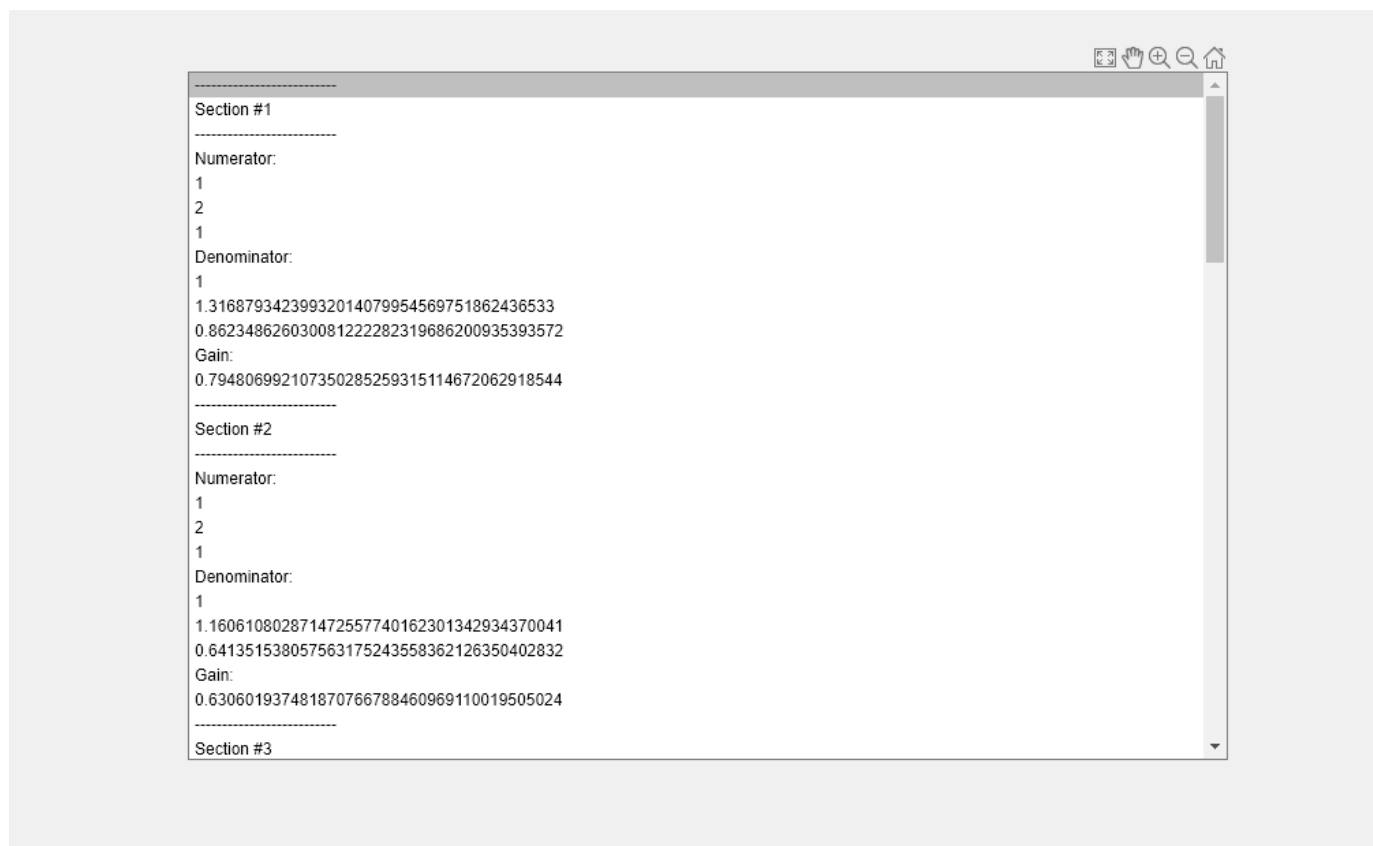
Move the first filter to the end and remove the eighth section.

```
sosFiltclone2 = clone(sosFilt);  
reorder(sosFiltclone2,[2:7 1]);  
setfilter(fvt,sosFiltclone2);
```



Move the numerator and denominator independently.

```
sosFiltclone3 = clone(sosFilt);  
reorder(sosFiltclone3, [1 3:8 2],(1:8));  
setfilter(fvt, sosFiltclone3);
```



## Input Arguments

### **sysobjj** — Biquadratic filter object

`dsp.BiquadFilter` System object | `dsp.SOSFilter` System object

Biquadratic filter object, specified as either `dsp.BiquadFilter` or `dsp.SOSFilter` objects.

### **order** — Vector of indices used to reorder filter sections

vector

Vector of indices used to reorder the filter sections. `order` does not need to contain all the indices of the filter. Omitting one or more filter section indices removes the omitted sections from the filter. You can use a logical array to remove sections from the filter, but not to reorder it.

When `order` is a vector of logicals, the function removes the sections of the filter that correspond to the index values where `order` is equal to `false`.

Data Types: `double` | `logical`

### **numorder** — Vector of indices used to reorder filter numerator

vector

Vector of indices used to reorder the numerator of the filter. The `numorder` and `denorder` vectors must be of the same length.

When `numorder` is a vector of logicals, the function removes the numerator coefficients that correspond to the index values where `numorder` is equal to `false`.

Data Types: `double` | `logical`

#### **denorder** — Vector of indices used to reorder filter denominator

vector

Vector of indices used to reorder the denominator of the filter. The `numorder` and `denorder` vectors must be of the same length.

When `denorder` is a vector of logicals, the function removes the denominator coefficients that correspond to the index values where `denorder` is equal to `false`.

Data Types: `double` | `logical`

#### **svorder** — Reorder scale values

vector

Independent reordering of scale values. When `svorder` is not specified, the scale values are reordered in the same way as the numerator. The output scale value always remains at the end when you use the argument `numorder` to reorder the scale values.

When `svorder` is a vector of logicals, the function removes the scale values that correspond to the index values where `svorder` is equal to `false`.

Data Types: `double` | `logical`

#### **filter\_type** — Filter type

'auto' | 'bandpass' | 'bandstop' | 'highpass' | 'lowpass'

Filter type. Automatic reordering only applies when `sysobj` was obtained using `fdesign`. With the 'auto' option as an input argument, `reorder` automatically rearranges the filter sections depending on the specification response type of the design (lowpass, highpass, etc).

#### **dir\_flag** — Pole direction flag

'down' | 'up'

Pole direction flag, specified as either:

- 'up' -- The first filter section contains the poles closest to the origin, and the last section contains the poles closest to the unit circle.
- 'down' -- The sections are ordered in the opposite direction. The zeros are always paired with the poles closest to them.

#### **sv** — Scale value options

'poles' | 'zeros'

Scale value options, specified as either:

- 'poles' -- Scale values are reordered following the reordering of the poles.
- 'zeros' -- Scale values are reordered following the reordering of the zeros.

The scale values are not reordered when using the `dir_flag` option unless `sv` is specified.

**arithType — Filter arithmetic**`'double' (default) | 'single' | 'fixed'`

Arithmetic type the `reorder` function assumes, specified as one of the following:

- `'double'` -- The function assumes a double precision filter when the arithmetic input is not specified and the filter System object is in an unlocked state.
- `'single'` -- The function assumes a single precision filter.
- `'fixed'` -- The overflow action used in the reorder operation is set to the action specified in the filter System object.

**Output Arguments****sysobjnew — Reordered biquadratic filter object**`dsp.BiquadFilter System object | dsp.SOSFilter System object`

Biquadratic filter object with reordered second-order sections, returned as a `dsp.BiquadFilter` or a `dsp.SOSFilter` object. The SOS matrices and the corresponding scale values in the output object are as per the reorder arguments run on the input `sysobj`.

**Version History**

Introduced in R2011a

**See Also**

`scale` | `scalecheck` | `scaleopts` | `cumsec` | `sos`

## scale

**Package:** dsp

Scale second-order sections

### Syntax

```
scale(sysobj)
sysobjnew = scale(sysobj)
scale(sysobj,pnorm)
scale(sysobj,pnorm,opts)
scale(sysobj,pnorm,Name,Value)
```

### Description

`scale(sysobj)` scales the biquadratic System object, `sysobj`, using peak magnitude response scaling (L-infinity, 'Linf'). This scaling reduces the possibility of overflows when the filter object operates in fixed-point arithmetic mode.

`sysobjnew = scale(sysobj)` generates a new filter System object, `sysobjnew`, with scaled second-order sections. The original filter System object, `sysobj`, is not changed.

`scale(sysobj,pnorm)` specifies the norm used to scale the filter. The variable `pnorm` can be either a discrete-time-domain norm or a frequency-domain norm. Valid time-domain norms are 'l1', 'l2', and 'linf'. Valid frequency-domain norms are 'L1', 'L2', and 'Linf'. Note that L2-norm is equal to l2-norm (Parseval's theorem) but the same is not true for other norms.

The different norms can be ordered in terms of how stringent they are as follows: 'l1' >= 'Linf' >= 'L2' = 'l2' >= 'L1' >= 'linf'.

Using the most stringent scaling, 'l1', the filter is the least prone to overflow, but also has the worst signal-to-noise ratio. Linf-scaling is the most commonly used scaling in practice.

`scale(sysobj,pnorm,opts)` uses an options object to specify the optional scaling parameters in lieu of specifying parameter-value pairs. The `opts` object can be created using the `scalexts` function: `opts = scalexts(sysobj)`.

`scale(sysobj,pnorm,Name,Value)` specifies optional scaling parameters via by one or more `Name,Value` pair arguments.

### Examples

#### Linf-norm Scaling of a Biquadratic SOS Filter

Demonstrate the Linf-norm scaling of a biquadratic SOS filter using the `scale` function.

```
Fs = 8000;
Fcutoff = 2000;
[z,p,k] = butter(10,Fcutoff/(Fs/2));
```

```
[sosMatrix,scaleValues] = zp2sos(z,p,k);
sosFilt = dsp.SOSFilter(Structure='Direct form I', ...
    Numerator=sosMatrix(:,1:3),Denominator=sosMatrix(:,4:6), ...
    HasScaleValues=true,ScaleValues=scaleValues)
```

```
sosFilt =
    dsp.SOSFilter with properties:

        Structure: 'Direct form I'
    CoefficientSource: 'Property'
        Numerator: [5x3 double]
        Denominator: [5x3 double]
    HasScaleValues: true
    ScaleValues: [0.0029 1 1 1 1 1]
```

Show all properties

```
scale(sosFilt,'linf',scalevalueconstraint='none',maxscalevalue=2)
```

## Input Arguments

### sysobj — Input filter

dsp.BiquadFilter System object | dsp.SOSFilter System object

Input filter, specified as one of the following System objects:

- dsp.BiquadFilter
- dsp.SOSFilter

Example: `biquad = dsp.BiquadFilter('Structure', 'Direct form I', ... 'SOSMatrix', s, 'ScaleValues', g);`

### pnorm — Discrete-time-domain norm or a frequency-domain norm

'Linf' (default) | 'L1' | 'L2' | 'l1' | 'l2' | 'linf'

Valid time-domain norm values for `pnorm` are 'l1', 'l2', and 'linf'. Valid frequency-domain norm values are 'L1', 'L2', and 'Linf'. The 'L2' norm is equal to the 'l2' norm (by Parseval's theorem), but this equivalency does not hold for other norms — 'l1' is not the same as 'L1' and 'Linf' is not the same as 'linf'.

Filter norms can be ordered in terms of how stringent they are, as follows from most stringent to least: 'l1', 'Linf', 'l2' ('L2'), 'linf'. Using 'l1', the most stringent scaling, produces a filter that is least likely to overflow, but has the worst signal-to-noise ratio performance. The default scaling 'Linf' (default) is the most commonly used scaling norm.

### opts — Scale options object

fdopts.sosscaling object

You can create an `fdopts.sosscaling` object, `opts`, using the `scalemopts` function.

The following table lists the properties of `opts`:

| Parameter            | Default    | Description and Valid Value  |
|----------------------|------------|--|
| sosReorder           | 'auto'     | Reorder section prior to scaling.<br>Valid options are 'auto' (default), 'none', 'up', 'down', 'lowpass', 'highpass', 'bandpass', and 'bandstop'.  |
| MaxNumerator         | 2          | Maximum allowed value for numerator coefficients.  |
| NumeratorConstraint  | 'none'     | Specifies whether and how to constrain numerator coefficient values. Options are 'none' (default), 'unit', 'normalize', and 'po2'.   |
| OverflowMode         | 'wrap'     | Sets the way the filter handles arithmetic overflow situations during scaling. Valid options are 'wrap' (default), 'saturate', and 'satall'.   |
| ScaleValueConstraint | 'unit'     | Specify whether to constrain the filter scale values, and how to constrain them. Valid options are 'unit' (default), 'none', and 'po2'.  |
| MaxScaleValue        | 'Not used' | Maximum allowed scale values. The filter applies the MaxScaleValue limit only when you set ScaleValueConstraint to a value other than unit. Setting MaxScaleValue to a numerical value automatically changes the ScaleValueConstraint setting to none. |

Example: `opts = scaleopts(biquad)`

### Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: `[z,p,k] = butter(10,2000/(8000/2)); [s,g] = zp2sos(z,p,k); biquad = dsp.BiquadFilter('Structure','Direct form I','SOSMatrix',s,'ScaleValues',g); scale(biquad,'linf','scalevalueconstraint','none','maxscalevalue',2)`

### Arithmetic – Arithmetic type

'double' (default) | 'single' | 'Fixed'

Arithmetic type used during analysis, specified as one of 'double', 'single', or 'fixed'. The scale method assumes a double precision filter when the arithmetic input is not specified and the



filter System object is in an unlocked state. If the System object is locked, the function performs analysis based on the locked input data type. If 'Arithmetic' is 'double' or 'single', the default values are used for all scaling options that are not specified as an input to the `scale` function. If 'Arithmetic' is 'fixed', the values used for the scaling options are set according to the settings in the filter System object. However, if a scaling option is specified that differs from the settings in `sysobj`, this option is used for scaling purposes but does not change the setting in `sysobj`. For example, if you do not specify the 'OverflowMode' scaling option, the `scale` method assumes that the 'OverflowMode' is equal to the value in the `OverflowAction` property of the filter object. If you do specify an 'OverflowMode' scaling option, then the `scale` function uses this overflow mode value regardless of the value in the `OverflowAction` property of the System object.

### **sosReorder — Reorder section prior to scaling**

'auto' (default) | 'none' | 'up' | 'down' | 'lowpass' | 'highpass' | 'bandpass' | 'bandstop'

Reorder filter sections prior to applying scaling. Possible options:

- 'auto'
- 'none'
- 'up'
- 'down'
- 'lowpass'
- 'highpass'
- 'bandpass'
- 'bandstop'

Automatic reordering takes effect when `sysobj` is obtained as a result from a design using `fdesign`. The sections are automatically reordered depending on the response type of the design.

### **MaxNumerator — Maximum value for numerator coefficients**

2 (default) | positive scalar

Maximum allowed value for numerator coefficients, specified as a positive scalar.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical

### **NumeratorConstraint — Method to constrain numerator coefficients**

'none' (default) | 'normalized' | 'po2' | 'unit'

Method to constrain numerator coefficient values, specified as one of the following:

- 'none'
- 'normalized'
- 'po2'
- 'unit'

### **OverflowMode — Overflow mode**

'wrap' (default) | 'saturate'

Sets the way the filter handles arithmetic overflow situations during scaling. If your device does not have guard bits available, and you are using saturation arithmetic for filtering, use 'sata11' instead of 'saturate'. The default is 'wrap'.

### **ScaleValueConstraint — Constrain scale values**

'unit' (default) | 'none' | 'po2'

Specify whether to constrain the filter scale values, and how to constrain them. Choosing 'unit' for the constraint disables the `MaxScaleValue` property setting. 'po2' constrains the scale values to be powers of 2, while 'none' removes any constraint on the scale values. 'unit' is the default value.

### **MaxScaleValue — Maximum value for scale values**

'Not Used' (default) | positive scalar

Maximum allowed scale values. The filter applies the `MaxScaleValue` limit only when you set `ScaleValueConstraint` to a value other than `unit` (the default setting). Setting `MaxScaleValue` to any numerical value automatically changes the `ScaleValueConstraint` setting to `none`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

## **Output Arguments**

### **sysobjnew — Scaled biquadratic filter object**

`dsp.BiquadFilter` System object | `dsp.SOSFilter` System object

Scaled biquadratic filter object, returned as one of the following System objects:

- `dsp.BiquadFilter`
- `dsp.SOSFilter`

The returned object contains the scaled second-order sections.

## **Version History**

**Introduced in R2011a**

## **References**

[1] Dehner, G.F. "Noise Optimized Digital Filter Design: Tutorial and Some New Aspects." *Signal Processing*. Vol. 83, Number 8, 2003, pp. 1565-1582.

## **See Also**

### **Functions**

`cumsec` | `reorder` | `scalecheck` | `scaleopts`

### **Objects**

`dsp.BiquadFilter` | `dsp.SOSFilter`

### **Topics**

"Analysis Methods for Filter System Objects" on page 3-2

# scalecheck

**Package:** dsp

Check scaling of biquadratic filter

## Syntax

```
s = scalecheck(sysobj,pnorm)
s = scalecheck(sysobj,pnorm,'Arithmetic',arithType)
```

## Description

`s = scalecheck(sysobj,pnorm)` checks the scaling of the input filter System object.

`s = scalecheck(sysobj,pnorm,'Arithmetic',arithType)` checks the scaling of the filter object with the arithmetic specified in `arithType`.

## Examples

### Linf-norm scaling of a filter

This example shows how to check the Linf-norm scaling of a filter.

Design an elliptic sos filter in the direct form II structure with default specifications.

```
EllipII = design(fdesign.lowpass, 'ellip', 'FilterStructure', 'df2sos',...
    'SystemObject',true)
```

```
EllipII =
    dsp.SOSFilter with properties:
        Structure: 'Direct form II'
        CoefficientSource: 'Property'
        Numerator: [3x3 double]
        Denominator: [3x3 double]
        HasScaleValues: true
        ScaleValues: [0.8629 2.0523 0.0127 1]
```

Show all properties

Check the scaling.

```
scalecheck(EllipII,'Linf')
ans = 2x3
    3.1678    15.0757    1.4974
    4.7360    52.6026    1.0000
```

Design an elliptic sos filter in the direct form I structure with default specifications.

```
EllipI = design(fdesign.lowpass('N,Fp,Ap,Ast',10,0.5,0.5,20), 'ellip',...
    'FilterStructure', 'dfllsos','SystemObject',true)
```

```
EllipI =
    dsp.SOSFilter with properties:

        Structure: 'Direct form I'
    CoefficientSource: 'Property'
        Numerator: [5x3 double]
        Denominator: [5x3 double]
    HasScaleValues: true
    ScaleValues: [0.9442 1.0014 1.0170 1.6551 0.1398 1]
```

Show all properties

Check the scaling.

```
scalecheck(EllipI,'Linf')
```

```
ans = 1x5
```

```
    1.7078    2.0807    2.6084    7.1467    1.0000
```

## Input Arguments

### sysobj — Input filter object

dsp.BiquadFilter System object | dsp.SOSFilter System object

Input filter, specified as one of the following System objects:

- dsp.BiquadFilter
- dsp.SOSFilter

### pnorm — Different types of norm

'l1' | 'l2' | 'linf' | 'L1' | 'L2' | 'Linf'

Discrete-time-domain norm or a frequency-domain norm.

Valid time-domain norm values for pnorm are 'l1', 'l2', and 'linf'. Valid frequency-domain norm values are 'L1', 'L2', and 'Linf'. The 'L2' norm is equal to the 'l2' norm (by Parseval's theorem), but this equivalency does not hold for other norms — 'l1' is not the same as 'L1' and 'Linf' is not the same as 'linf'.

### arithType — Arithmetic type

'double' (default) | 'single' | 'Fixed'

Arithmetic type used during analysis, specified as 'double', 'single', or 'fixed'. The function assumes a double precision filter when the arithmetic input is not specified and the filter System object is in an unlocked state.

## Output Arguments

### **s** — Filter scaling

scalar | row vector

Filter scaling for a given p-norm. An optimally scaled filter has partial norms equal to one. In such cases, **s** contains all ones.

For direct-form I (**df1sos**) and direct-form II transposed (**df2tsos**) filters, the function returns the p-norm of the filter computed from the filter input to the output of each second-order section. Therefore, the number of elements in **s** is one less than the number of sections in the filter. This p-norm computation does not include the trailing scale value of the filter, which you can find by entering `hd.scalevalue(end)` at the MATLAB prompt.

For direct-form II (**df2sos**) and direct-form I transposed (**df1tsos**) filters, the function returns a row vector whose elements contain the p-norm from the filter input to the input of the recursive part of each second-order section. This computation of the p-norm corresponds to the input to the multipliers in these filter structures. These inputs correspond to the locations in the signal flow where overflow should be avoided.

When **hd** has nontrivial scale values, that is, if any scale values are not equal to one, **s** is a two-row matrix, rather than a vector. The first row elements of **s** report the p-norm of the filter computed from the filter input to the output of each second-order section. The elements of the second row of **s** contain the p-norm computed from the input of the filter to the input of each scale value between the sections. For **df2sos** and **df1tsos** filter structures, the last numerator and the trailing scale value for the filter are not included when `scalecheck` checks the scaling.

Data Types: `double`

## Version History

Introduced in R2011a

## See Also

### Functions

`reorder` | `scale` | `scaleopts` | `cumsec`

### Objects

`dsp.BiquadFilter` | `dsp.SOSFilter`

### Topics

"Analysis Methods for Filter System Objects" on page 3-2

## scaleopts

**Package:** dsp

Create an options object for second-order section scaling

### Syntax

```
opts = scaleopts(sysobj)
opts = scaleopts(sysobj, 'Arithmetic', arithType)
```

### Description

`opts = scaleopts(sysobj)` uses the current settings in the filter System object to create an options object `opts` that contains specified scaling options for second-order section scaling. You can pass `opts` as an input to `scale` to apply scaling settings to a second-order filter.

`opts = scaleopts(sysobj, 'Arithmetic', arithType)` returns filter coefficients for the filter System object with the arithmetic specified in `arithType`.

### Examples

#### Options for scaling SOS filter

Create an options scaling object that contains the scaling options settings you require.

```
EllipI = design(fdesign.lowpass('N,Fp,Ap,Ast',10,0.5,0.5,20),...
    'ellip',...
    'FilterStructure', 'df1sos',...
    'SystemObject',true)
```

```
EllipI =
    dsp.SOSFilter with properties:

        Structure: 'Direct form I'
    CoefficientSource: 'Property'
        Numerator: [5x3 double]
        Denominator: [5x3 double]
    HasScaleValues: true
    ScaleValues: [0.9442 1.0014 1.0170 1.6551 0.1398 1]
```

Show all properties

```
opts = scaleopts(EllipI)

opts =

        sosReorder: 'auto'
    MaxNumerator: 2
    NumeratorConstraint: 'none'
    OverflowMode: 'wrap'
```

```
ScaleValueConstraint: 'unit'
MaxScaleValue: 'Not used'
```

## Input Arguments

### sysobj — Input filter

`dsp.BiquadFilter` System object | `dsp.SOSFilter` System object

Input filter, specified as one of the following System objects:

- `dsp.BiquadFilter`
- `dsp.SOSFilter`

Example: `biquad = dsp.BiquadFilter('Structure', 'Direct form I', ... 'SOSMatrix', s, 'ScaleValues', g);`

### arithType — Arithmetic type

'double' (default) | 'single' | 'Fixed'

Arithmetic type used during analysis, specified as 'double', 'single', or 'fixed'. The function assumes a double precision filter when the arithmetic input is not specified and the filter System object is in an unlocked state. The `scaleopts` function chooses the default values of the scaling options according to the 'Arithmetic' value and the System object settings.

## Output Arguments

### opts — Scale options object

`fdopts.sosscaling` object

You can create an `fdopts.sosscaling` object, `opts`, using the `scaleopts` function.

The following table lists the properties of `opts`:

| Parameter                        | Default | Description and Valid Value   |
|----------------------------------|---------|---|
| <code>sosReorder</code>          | 'auto'  | Reorder section prior to scaling.<br><br>Valid options are 'auto' (default), 'none', 'up', 'down', 'lowpass', 'highpass', 'bandpass', and 'bandstop'. |
| <code>MaxNumerator</code>        | 2       | Maximum allowed value for numerator coefficients.   |
| <code>NumeratorConstraint</code> | 'none'  | Specifies whether and how to constrain numerator coefficient values. Options are 'none' (default), 'unit', 'normalize', and 'po2'.                    |

| Parameter            | Default    | Description and Valid Value  |
|----------------------|------------|--|
| OverflowMode         | 'wrap'     | Sets the way the filter handles arithmetic overflow situations during scaling. Valid options are 'wrap' (default), 'saturate', and 'satall'.   |
| ScaleValueConstraint | 'unit'     | Specify whether to constrain the filter scale values, and how to constrain them. Valid options are 'unit' (default), 'none', and 'po2'.  |
| MaxScaleValue        | 'Not used' | Maximum allowed scale values. The filter applies the MaxScaleValue limit only when you set ScaleValueConstraint to a value other than unit. Setting MaxScaleValue to a numerical value automatically changes the ScaleValueConstraint setting to none. |

Example: `opts = scaleopts(biquad)`

## Version History

Introduced in R2011a

## See Also

### Functions

`cumsec` | `reorder` | `scale` | `scalecheck`

### Objects

`dsp.BiquadFilter` | `dsp.SOSFilter`

### Topics

"Analysis Methods for Filter System Objects" on page 3-2



# setCursorDataLabels

Customize data labels for cursor measurements

## Syntax

```
setCursorDataLabels(obj,labels)
```

## Description

setCursorDataLabels(obj,labels) customizes the data labels that appear in the tool tip of cursor measurements in the dsp.MatrixViewer.

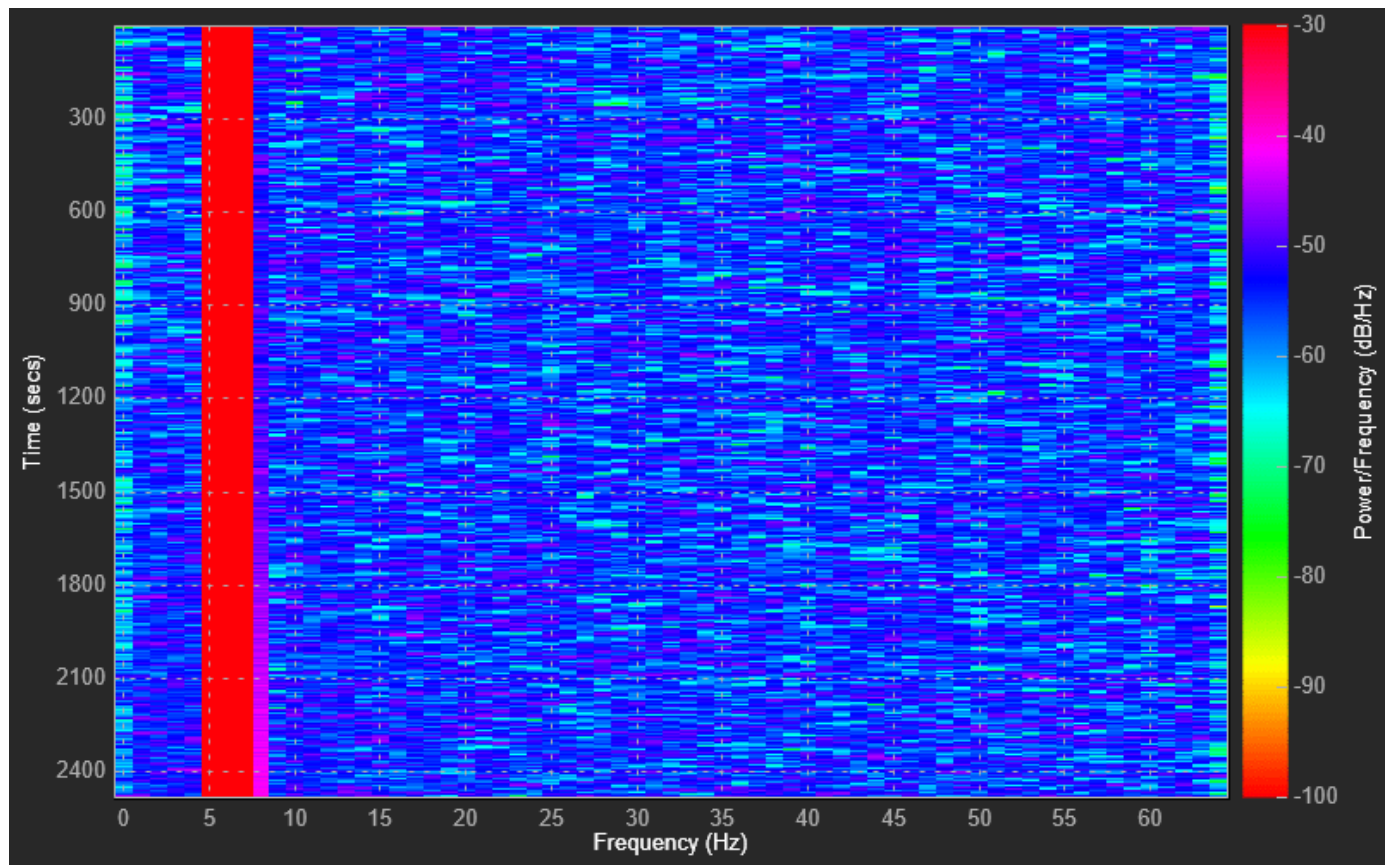
## Examples

### Set Cursor Labels on dsp.MatrixViewer

This example shows how to set the data labels for the cursor measurements on the dsp.MatrixViewer System object.

Display a chirp signal on the dsp.MatrixViewer scope.

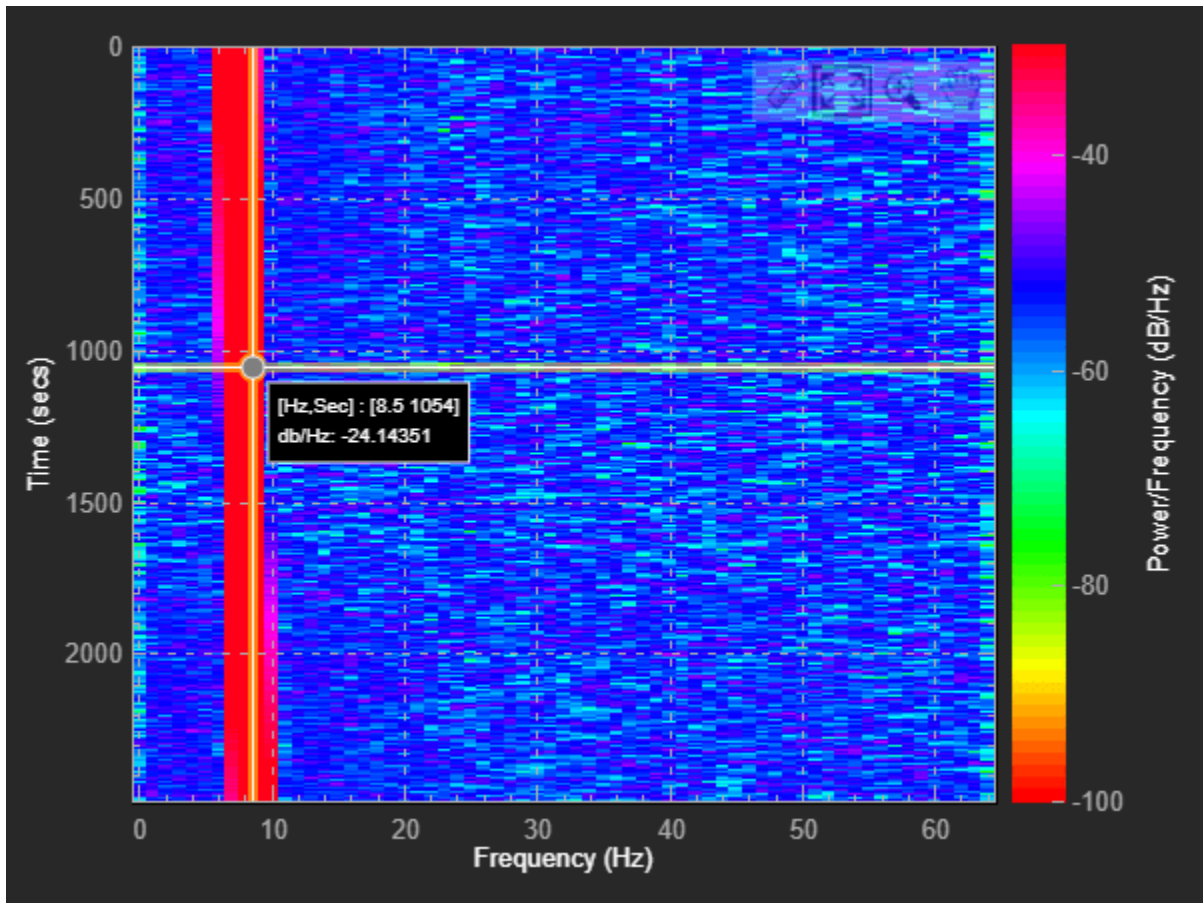
```
Fs = 233e3;
frameSize = 20e3;
chirp = dsp.Chirp("SampleRate",Fs,"SamplesPerFrame",frameSize,...
    "InitialFrequency",11e3,"TargetFrequency",11e3+55e3,...
    "Type","Quadratic");
scope = dsp.MatrixViewer(...
    "ColorBarLabel","Power/Frequency (dB/Hz)",...
    "XLabel","Frequency (Hz)",...
    "YLabel","Time (secs)",...
    "Colormap","hsv",...
    "ColorLimits",[-100,-30]);
y = chirp() + 0.05*randn(frameSize,1);
[~,~,~,Ps] = spectrogram(y,128,120,128,1e3);
val = 10*log10(abs(Ps)'+eps);
scope(val);
```



Change the data labels for the cursor measurements to match the type of data in the matrix.

```
setCursorDataLabels(scope, ["Hz", "Sec", "db/Hz"])
```

Enable the cursor measurements by selecting the cursor button in the axes toolbar. When you hover over the cursor, the tooltip shows you the data values with the new data labels.



## Input Arguments

### obj — Matrix viewer

`dsp.MatrixViewer` System object

Matrix viewer whose data labels you want to customize.

### labels — Label customization vector

`["X", "Y", "Value"]` (default) | 3-by-1 string array

Specify the label names as a three-element string array. The first element corresponds to the x-data, the second element to the y-data, and the third element to the matrix value.

Example: `["freq", "time", "power"]`

Data Types: `char` | `string`

## Version History

Introduced in R2019a

**See Also**

`dsp.MatrixViewer`

## set2int

Configure filter for integer filtering

### Syntax

```
set2int(h)
set2int(h,coeffwl)
set2int(...,inwl)
g = set2int(...)
```

### Description

This section applies to discrete-time (`dfilt`) filters.

`set2int(h)` scales the filter coefficients to integer values and sets the filter coefficient and input fraction lengths to zero.

`set2int(h,coeffwl)` uses the number of bits specified by `coeffwl` as the word length it uses to represent the filter coefficients.

`set2int(...,inwl)` uses the number of bits specified by `coeffwl` as the word length it uses to represent the filter coefficients and the number of bits specified by `inwl` as the word length to represent the input data.

`g = set2int(...)` returns the gain `g` introduced into the filter by scaling the filter coefficients to integers. `g` is always calculated to be a power of 2.

---

**Note** `set2int` does not work with CIC decimators or interpolators because they do not have coefficients.

---

## Examples

### Configure an FIR Filter for Integer Filtering

Two parts comprise this example. Part 1 compares the step response of an FIR filter in both the fractional and integer filter modes. Fractional mode filtering is essentially the opposite of integer mode. Integer mode uses a filter which has coefficients represented by integers. Fractional mode filters have coefficients represented in fractional form (nonzero fraction length).

The second part of the example depends on the following - after you filter a set of data, the input data and output data cover the same range of values, unless the filter process introduces gain in the output. Converting your filter object to integer form, and then filtering a set of data, does introduce gain into the system. When the examples refer to resetting the output to the same range as the input, the examples are accounting for this added gain feature.

```
b = rcosdesign(.25,4,25,'sqrt');
hd = dfilt.dffir(b);
hd.Arithmetic = 'fixed';
```

```
hd.InputFracLength = 0; % Integer inputs.  
x = ones(100,1);  
yfrac = filter(hd,x); % Fractional mode output.  
g = set2int(hd); % Convert to integer coefficients.  
yint = filter(hd,x); % Integer mode output.
```

Note that `yint` and `yfrac` are `fi` objects. Later in this example, use the `fi` object properties `WordLength` and `FractionLength` to work with the output data. Now use the gain `g` to rescale the output from the integer mode filter operation. Verify that the scaled integer output is equal to the fractional output.

```
yints = double(yint)/g;
```

Verify that the scaled integer output is equal to the fractional output.

```
max(abs(yints-double(yfrac)))
```

```
ans = 0
```

In part two, the example reinterprets the output binary data, putting the input and the output on the same scale by weighting the most significant bits in the input and output data equally.

```
WL = yint.WordLength;  
FL = yint.FractionLength + log2(g);  
yints2 = fi(zeros(size(yint)),true,WL,FL);  
yints2.bin = yint.bin;  
max(abs(double(yints2)-double(yfrac)))
```

```
ans = 0
```

## Version History

Introduced in R2011a

## setspecs

Specifications for filter specification object

### Syntax

```
setspecs(D,specvalue1,specvalue2,...)
setspecs(D,Specification,specvalue1,specvalue2,...)
setspecs(...Fs)
setspecs(...,MAGUNITS)
```

### Description

`setspecs(D,specvalue1,specvalue2,...)` sets the specifications in filter specification object, `D`, in the same order they appear in the `Specification` property.

`setspecs(D,Specification,specvalue1,specvalue2,...)` changes the specifications for an existing filter specification object and sets values for the new `Specification` property.

`setspecs(...Fs)` specifies the sampling frequency, `Fs`, in Hz. The sampling frequency must be a scalar trailing all other specifications. Entering a sampling frequency causes all other frequency specifications to be in Hz.

`setspecs(...,MAGUNITS)` specifies the units for any magnitude specifications. `MAGUNITS` can be one of the following: 'linear', 'dB', or 'squared'. The default is 'dB'. The magnitude specifications are always converted and stored in dB regardless of how the units are specified.

Use `SET(D,'SPECIFICATION')` to get the list of all available specification types for the filter specification object, `D`.

### Examples

#### Set the Filter Order and Cutoff Frequency Using `setspecs`

Construct a lowpass filter with specifications for the filter order and cutoff frequency (-6 dB). Use `setspecs` after construction to set the values of the filter order and cutoff frequency. Display the values in the MATLAB® command window.

```
D = fdesign.lowpass('N,Fc');
setspecs(D,10,0.2);
```

```
D.FilterOrder
```

```
ans = 10
```

```
D.Fcutoff
```

```
ans = 0.2000
```

### **Set the Specifications of a Highpass Filter Using setspecs**

Construct a highpass filter with specifications for the numerator order, denominator order, and 3-dB frequency. Assume the sampling frequency is 1 kHz. Use `setspecs` to set the numerator and denominator orders to 6. Set the 3-dB frequency to 250 Hz. In order to use frequency specifications in Hz, specify the sampling frequency as a trailing scalar.

```
D = fdesign.highpass('Nb,Na,F3dB');  
setspecs(D,6,6,250,1000);
```

## **Version History**

**Introduced in R2011b**

### **See Also**

`design` | `designmethods` | `designopts` | `fdesign`



# show

## Package:

Display scope window

## Syntax

```
show(scope)
```

## Description

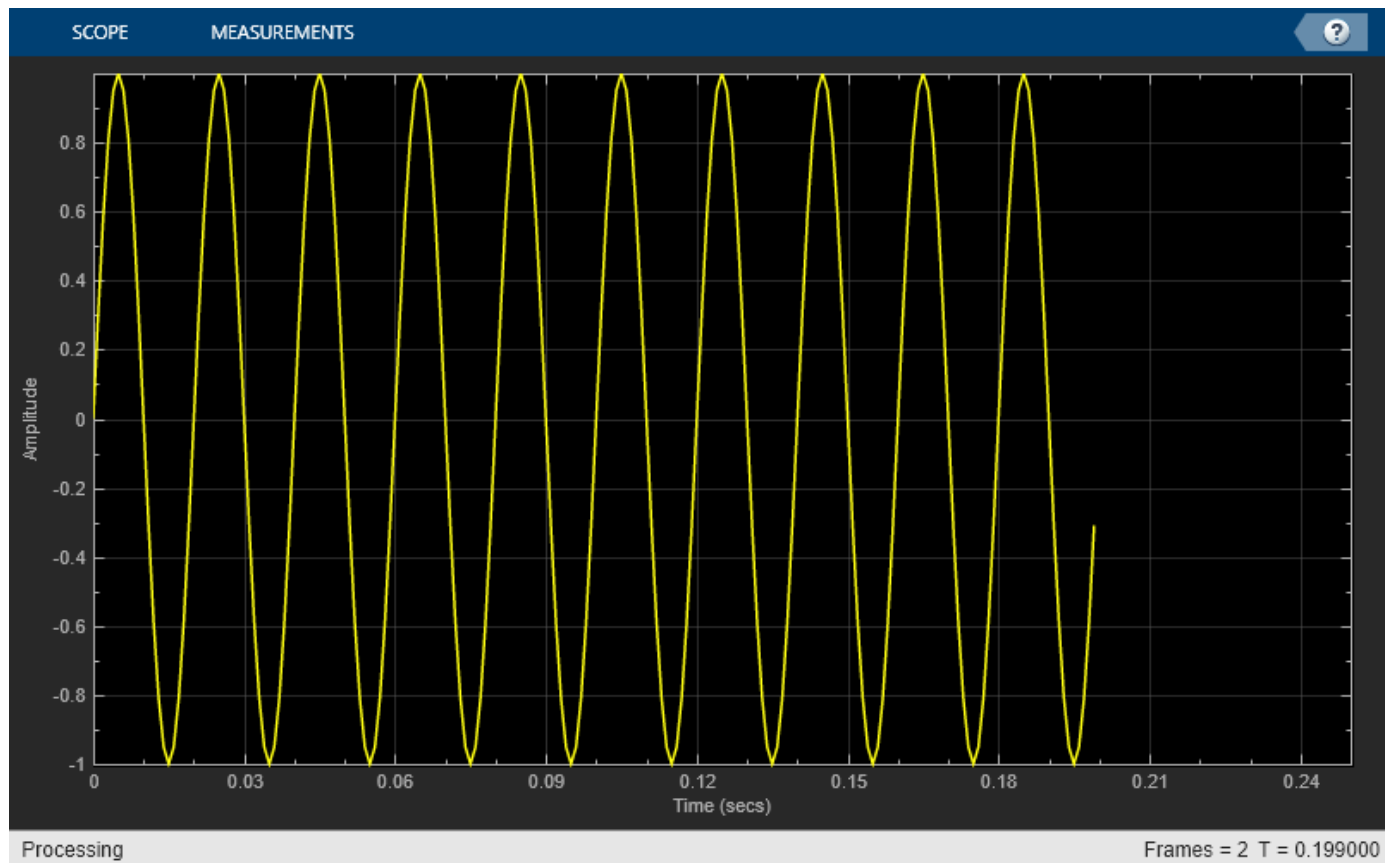
show(scope) shows the window of the scope.

## Examples

### Hide and Show Time Scope

Create a sine wave signal and view it in the scope.

```
Fs = 1000; % Sample rate
signal = dsp.SineWave(Frequency=50, SampleRate=Fs, ...
    SamplesPerFrame=100);
scope = timescope(SampleRate=Fs, TimeSpanSource="property", ...
    TimeSpan=0.25, YLimits=[-1 1]);
for ii = 1:2
    xsine = signal();
    scope(xsine)
end
```

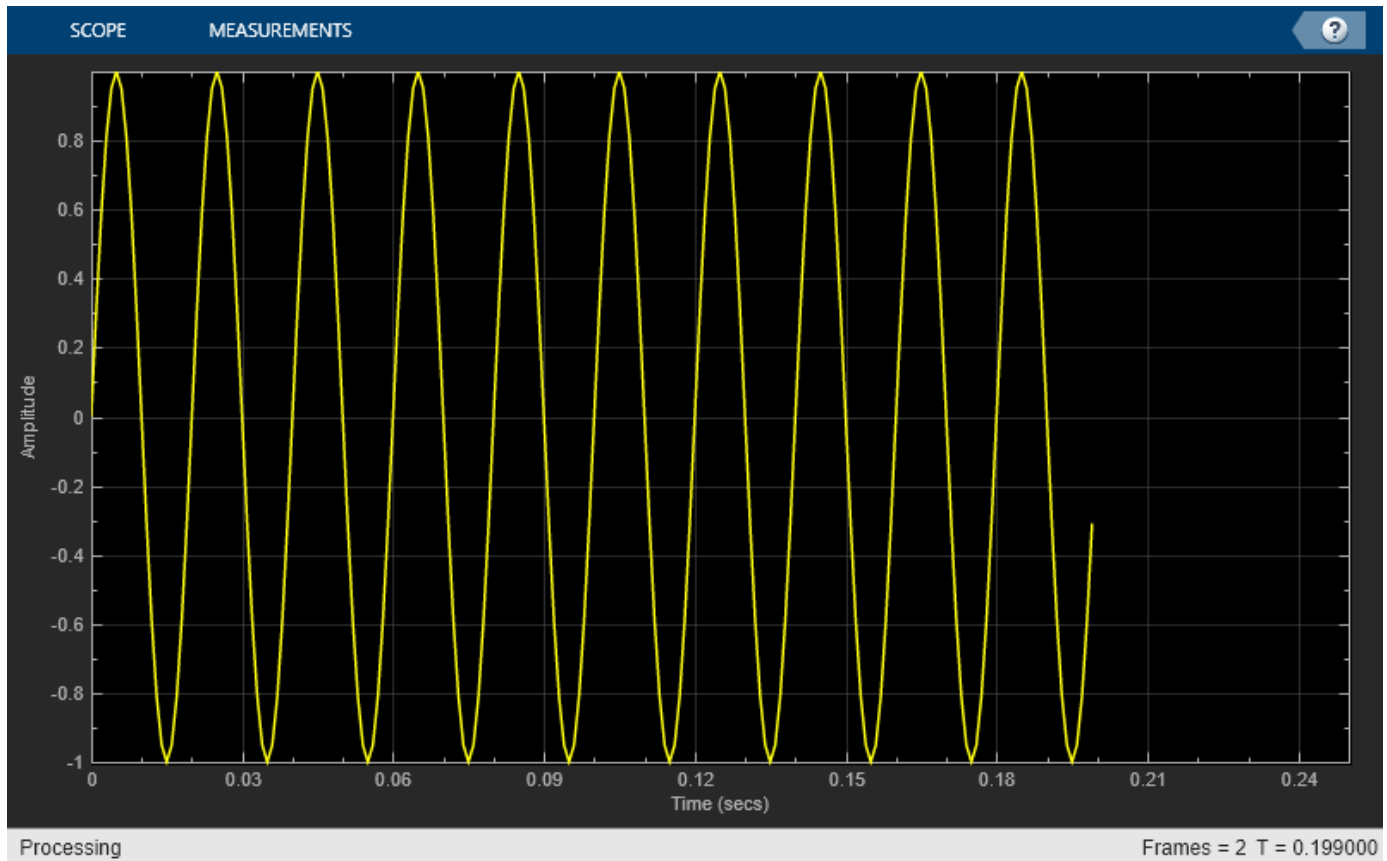


Hide the scope window.

```
if(isVisible(scope))  
    hide(scope)  
end
```

Show the scope window.

```
if(~isVisible(scope))  
    show(scope)  
end
```



Clean up workspace variables.

```
clear scope Fs sine ii xsine
```

## Input Arguments

### scope — Scope object

scope object

Scope object, specified as one of the following:

- spectrumAnalyzer object
- dsp.ArrayPlot object
- dsp.LogicAnalyzer System object
- timescope object
- dsp.DynamicFilterVisualizer object

Example: myScope = spectrumAnalyzer; show(myScope)

## Version History

Introduced in R2011a

**R2022a: Support for `dsp.SpectrumAnalyzer` object will be removed**

*Not recommended starting in R2022a*

The `show` function will stop supporting the `dsp.SpectrumAnalyzer` object in a future release. Use the `spectrumAnalyzer` object instead.

**Update Code**

No updates to your code are required except replacing instances of `dsp.SpectrumAnalyzer` with `spectrumAnalyzer`.

This table shows how the `show` function typically uses the `dsp.SpectrumAnalyzer System` object and explains how to update existing code to use the `spectrumAnalyzer` object.

| Discouraged Usage  | Recommended Replacement                                |
|--|--|
| <code>scope = dsp.SpectrumAnalyzer;<br/>show(scope)</code> | <code>scope = spectrumAnalyzer;<br/>show(scope)</code> |

**See Also****Functions**

`hide` | `isVisible` | `step`

**Objects**

`dsp.DynamicFilterVisualizer` | `timescope` | `dsp.ArrayPlot` | `dsp.LogicAnalyzer` | `spectrumAnalyzer`

# showsignalblockdatatypetable

Launch DSP System Toolbox data type support table

## Syntax

```
showsignalblockdatatypetable
```

## Description

The `showsignalblockdatatypetable` function shows a table of characteristics for the DSP System Toolbox blocks. The table lists capabilities and limitations of code generation, variable size, and supported data types for each block.

If a cell has an:

- "X" -- The corresponding block supports the data type or capability indicated by the column heading.
- "s" or a "u" -- The block supports signed or unsigned varieties of that data type, respectively.
- (#) -- Refer to the numbered footnotes at the bottom of the table window for further details. The (#) is hyperlinked to the footnotes at the bottom.

## Examples

### Show Block Characteristics for DSP System Toolbox

The `showsignalblockdatatypetable` function returns a table of block characteristics for DSP System Toolbox™.

Run the function in the MATLAB™ command prompt. The table opens in a separate window.

```
showsignalblockdatatypetable
```

```
Loading DSP System Toolbox Library.
```

## Version History

Introduced in R2008b

## See Also

### Topics

“Simulink Blocks in DSP System Toolbox that Support Fixed-Point Design”

“Variable-Size Signal Support DSP System Objects”

“Understanding C Code Generation in DSP System Toolbox”

## SOS

**Package:** dsp

Convert to second order sections

### Syntax

```
sosFilt = sos(sysobj)
sosFilt = sos(sysobj,dir_flag)
```

### Description

`sosFilt = sos(sysobj)` converts a IIR discrete-time filter to a second-order section (SOS) filter. The `sosFilt` output is a `dsp.SOSFilter` System object.

`sosFilt = sos(sysobj,dir_flag)` rearranges the second-order sections according to proximity to the origin of the poles of the sections.

### Examples

#### Convert IIR Discrete-time Filter to SOS Form

Convert an IIR discrete-time filter object to an SOS filter object that contains the corresponding SOS sections.

Design a 10th order lowpass Butterworth filter using the `butter` function.

```
N = 10;
Fc = 0.4;
[b,a] = butter(N,Fc);
```

Create a `dsp.IIRFilter` object and assign the designed coefficients to the `Numerator` and the `Denominator` properties of the object.

```
iir = dsp.IIRFilter(Numerator=b,Denominator=a)
```

```
iir =
  dsp.IIRFilter with properties:
    Structure: 'Direct form II transposed'
    Numerator: [4.9945e-04 0.0050 0.0225 0.0599 0.1049 0.1259 0.1049 0.0599 0.0225 0.0050]
    Denominator: [1 -1.9924 3.0195 -2.8185 2.0387 -1.0545 0.4144 -0.1157 0.0225 -0.0027 1.4]
    InitialConditions: 0
```

```
Show all properties
```

Use the `sos` function to convert the IIR filter object into an SOS filter that contains the corresponding SOS sections.

```
sosFilt = sos(iir)
```

```

sosFilt =
  dsp.SOSFilter with properties:

    Structure: 'Direct form II transposed'
    CoefficientSource: 'Property'
    Numerator: [5x3 double]
    Denominator: [5x3 double]
    HasScaleValues: true
    ScaleValues: [4.9945e-04 1 1 1 1 1]

```

Show all properties

You can obtain the SOS matrix and scale values from the SOS filter object.

```
sMatrix = [sosFilt.Numerator,sosFilt.Denominator]
```

```
sMatrix = 5x6
```

|        |        |        |        |         |        |
|--------|--------|--------|--------|---------|--------|
| 1.0000 | 2.0958 | 1.0984 | 1.0000 | -0.3187 | 0.0313 |
| 1.0000 | 2.0576 | 1.0602 | 1.0000 | -0.3345 | 0.0826 |
| 1.0000 | 1.9981 | 1.0005 | 1.0000 | -0.3695 | 0.1958 |
| 1.0000 | 1.9412 | 0.9436 | 1.0000 | -0.4317 | 0.3969 |
| 1.0000 | 1.9073 | 0.9097 | 1.0000 | -0.5380 | 0.7410 |

```
sValues = sosFilt.ScaleValues
```

```
sValues = 1x6
```

|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
| 0.0005 | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 1.0000 |
|--------|--------|--------|--------|--------|--------|

Compare these values to the values obtained using the `tf2sos` function and verify that they are the same.

```
[sMatrixfn,g] = tf2sos(b,a)
```

```
sMatrixfn = 5x6
```

|        |        |        |        |         |        |
|--------|--------|--------|--------|---------|--------|
| 1.0000 | 2.0958 | 1.0984 | 1.0000 | -0.3187 | 0.0313 |
| 1.0000 | 2.0576 | 1.0602 | 1.0000 | -0.3345 | 0.0826 |
| 1.0000 | 1.9981 | 1.0005 | 1.0000 | -0.3695 | 0.1958 |
| 1.0000 | 1.9412 | 0.9436 | 1.0000 | -0.4317 | 0.3969 |
| 1.0000 | 1.9073 | 0.9097 | 1.0000 | -0.5380 | 0.7410 |

```
g = 4.9945e-04
```

```
isequal(sMatrix,sMatrixfn)
```

```
ans = logical
     1
```

```
isequal(sValues(1),g)
```

```
ans = logical
     1
```

## Input Arguments

### **sysobj** — IIR discrete-time filter object

`dsp.IIRFilter` object

IIR discrete-time filter object, specified as a `dsp.IIRFilter` System object.

### **dir\_flag** — Pole direction flag

'up' | 'down'

Pole direction flag, specified as either:

- 'up' -- The first row contains the poles closest to the origin, and the last row contains the poles closest to the unit circle.
- 'down' -- The sections are ordered in the opposite direction. The zeros are always paired with the poles closest to them.

## Output Arguments

### **sosFilt** — Second order section filter object

`dsp.SOSFilter` object

Second order section filter object, returned as a `dsp.SOSFilter` System object. The `sos` function converts the input IIR filter numerator and denominator coefficients into SOS matrix and scale values, and assigns these values to the `dsp.SOSFilter` object.

## Version History

### Introduced in R2011a

### **R2022b: Designs `dsp.SOSFilter` object instead of `dsp.BiquadFilter` object**

*Behavior changed in R2022b*

The `sos` function now generates a `dsp.SOSFilter` object instead of the `dsp.BiquadFilter` object.

Initialize a `dsp.IIRFilter` object. Use the `sos` function to convert the IIR filter object into a biquadratic filter that contains the corresponding SOS sections. The output is a `dsp.SOSFilter` filter object.

```
iirFilt = dsp.IIRFilter
convFilt = sos(iirFilt)
```

```
convFilt =
    dsp.SOSFilter with properties:
        Structure: 'Direct form II transposed'
        CoefficientSource: 'Property'
        Numerator: [1 1 0]
        Denominator: [1 0.1000 0]
        HasScaleValues: false

    Show all properties
```

To design a `dsp.BiquadFilter` object instead of the `dsp.SOSFilter` object, you can use the `UseLegacyBiquadFilter` flag and set it to `true`.



```
convFilt = sos(iirFilt,UseLegacyBiquadFilter=true)
convFilt =
    dsp.BiquadFilter with properties:
        Structure: 'Direct form II transposed'
        SOSMatrixSource: 'Property'
        SOSMatrix: [1 1 0 1 0.1000 0]
        ScaleValues: [2x1 double]
        InitialConditions: 0
        OptimizeUnityScaleValues: true

Show all properties
```

## See Also

### Objects

dsp.SOSFilter

### Functions

reorder | scale | scalecheck | scaleopts | cumsec

## SS

**Package:** dsp

Convert discrete-time filter System object to state-space representation

### Syntax

```
[A,B,C,D] = ss(sysobj)
[A,B,C,D] = ss(sysobj,'Arithmetic',arithType)
```

### Description

`[A,B,C,D] = ss(sysobj)` converts a filter System object to state-space representation given by:

$$\begin{aligned}x(k+1) &= Ax(k) + Bu(k), \\ y(k) &= Cx(k) + Du(k),\end{aligned}$$

where  $x$  is the state vector,  $u$  is the input vector, and  $y$  is the output vector.

`[A,B,C,D] = ss(sysobj,'Arithmetic',arithType)` analyzes the filter System object based on the arithmetic specified in `arithType`.

For more input options, see `ss` in Signal Processing Toolbox.

## Examples

### State-Space Representation of Biquad Filter

Design a fourth-order, lowpass SOS filter object with a normalized cutoff frequency of 0.4.

```
[z,p,k] = ellip(4,1,60,.4);    % Set up the filter
[sosMatrix,scaleValues] = zp2sos(z,p,k);
sosFilt = dsp.SOSFilter(Structure='Direct form I',...
    Numerator=sosMatrix(:,1:3),Denominator=sosMatrix(:,4:6),...
    HasScaleValues=true,ScaleValues=scaleValues)
```

```
sosFilt =
    dsp.SOSFilter with properties:
        Structure: 'Direct form I'
    CoefficientSource: 'Property'
        Numerator: [2x3 double]
        Denominator: [2x3 double]
    HasScaleValues: true
        ScaleValues: [0.0351 1 1]
```

Show all properties

Convert the designed filter into state-space form using the `ss` function.

```
[A,B,C,D] = ss(sosFilt)
```

```
A = 8×8
```

```

    0         0         0         0         0         0         0         0
  1.0000     0         0         0         0         0         0         0
  1.8116    1.0000    1.0095   -0.3954     0         0         0         0
    0         0         1.0000     0         0         0         0         0
  1.8116    1.0000    1.0095   -0.3954     0         0         0         0
    0         0         0         0         1.0000     0         0         0
  1.8116    1.0000    1.0095   -0.3954    1.1484    1.0000    0.5581   -0.7823
    0         0         0         0         0         0         1.0000     0

```

```
B = 8×1
```

```

  0.0351
    0
  0.0351
    0
  0.0351
    0
  0.0351
    0

```

```
C = 1×8
```

```

  1.8116    1.0000    1.0095   -0.3954    1.1484    1.0000    0.5581   -0.7823

```

```
D = 0.0351
```

## Input Arguments

### sysobj — Input filter

filter System object

Input filter, specified as one of the following filter System objects:

- `dsp.BiquadFilter`
- `dsp.Differentiator`
- `dsp.FilterCascade`
- `dsp.FIRFilter`
- `dsp.FourthOrderSectionFilter`
- `dsp.HighpassFilter`
- `dsp.IIRFilter`
- `dsp.LowpassFilter`
- `dsp.NotchPeakFilter`
- `dsp.SOSFilter`
- `dsp.VariableBandwidthFIRFilter`

- `dsp.VariableBandwidthIIRFilter`

### **arithType — Arithmetic type**

'double' (default) | 'single' | 'Fixed'

Arithmetic used in the filter analysis, specified as 'double', 'single', or 'Fixed'. When the arithmetic input is not specified and the filter System object is unlocked, the analysis tool assumes a double-precision filter. When the arithmetic input is not specified and the System object is locked, the function performs the analysis based on the data type of the locked input.

The 'Fixed' value applies to filter System objects with fixed-point properties only.

When the 'Arithmetic' input argument is specified as 'Fixed' and the filter object has the data type of the coefficients set to 'Same word length as input', the arithmetic analysis depends on whether the System object is unlocked or locked.

- unlocked -- The analysis object function cannot determine the coefficients data type. The function assumes that the coefficients data type is signed, has a 16-bit word length, and is auto scaled. The function performs fixed-point analysis based on this assumption.
- locked -- When the input data type is 'double' or 'single', the analysis object function cannot determine the coefficients data type. The function assumes that the data type of the coefficients is signed, has a 16-bit word length, and is auto scaled. The function performs fixed-point analysis based on this assumption.

To check if the System object is locked or unlocked, use the `isLocked` function.

When the arithmetic input is specified as 'Fixed' and the filter object has the data type of the coefficients set to a custom numeric type, the object function performs fixed-point analysis based on the custom numeric data type.

## **Output Arguments**

### **A — State matrix**

matrix

State matrix, returned as an  $N$ -by- $N$  matrix, where  $N$  is the filter order.

Data Types: `double`

### **B — Input matrix**

column vector

Input matrix, returned as an  $N$ -by-1 column vector, indicating that the number of inputs to the linear system is 1.  $N$  is the filter order.

Data Types: `double`

### **C — Output matrix**

row vector

Output matrix, returned as a 1-by- $N$  row vector, indicating that the number of outputs of the linear system is 1.  $N$  is the filter order.

Data Types: `double`

**D — Feedthrough matrix**

scalar

Feedthrough matrix, returned as a scalar, indicating that the number of inputs and outputs of the linear system is 1.

Data Types: double

**Version History**

**Introduced in R2011a**

**See Also****Functions**

SS

**Topics**

“Analysis Methods for Filter System Objects” on page 3-2

## step

**Package:** dsp

Display time-varying magnitude response

### Syntax

```
step(dfv, filt)
step(dfv, B, A)
```

### Description

`step(dfv, filt)` displays the time-varying magnitude response of the object filter, `filt`, in the Dynamic Filter Visualizer figure, as long as `filt` has a valid `freqz()` implementation.

`step(dfv, B, A)` displays the magnitude response for the digital filters with numerator and denominator polynomial coefficients stored in  $B_1$  and  $A_1$ ,  $B_2$  and  $A_2$ , ..., and  $B_N$  and  $A_N$ , respectively.

### Examples

#### Plot Time-Varying Magnitude and Phase Response of FIR Filter

Design an FIR filter with a time-varying magnitude and phase response. Plot this varying response on a dynamic filter visualizer in normalized frequency units.

Create a `dsp.DynamicFilterVisualizer` object. Set the `PlotAsMagnitudePhase` and the `NormalizedFrequency` properties to `true`.

```
dfv = dsp.DynamicFilterVisualizer(PlotAsMagnitudePhase=1, ...
    NormalizedFrequency=true, ShowLegend=true, ...
    Title='Magnitude and Phase Response', ...
    FilterNames="FIR Filter")
```

```
dfv =
    dsp.DynamicFilterVisualizer with properties:
```

```
        FFTLength: 2048
    NormalizedFrequency: 1
        FrequencyRange: [0 1]
            XScale: 'Linear'
    MagnitudeDisplay: 'Magnitude (dB)'
    PlotAsMagnitudePhase: 1
            PlotType: 'Line'
        AxesScaling: 'Auto'
```

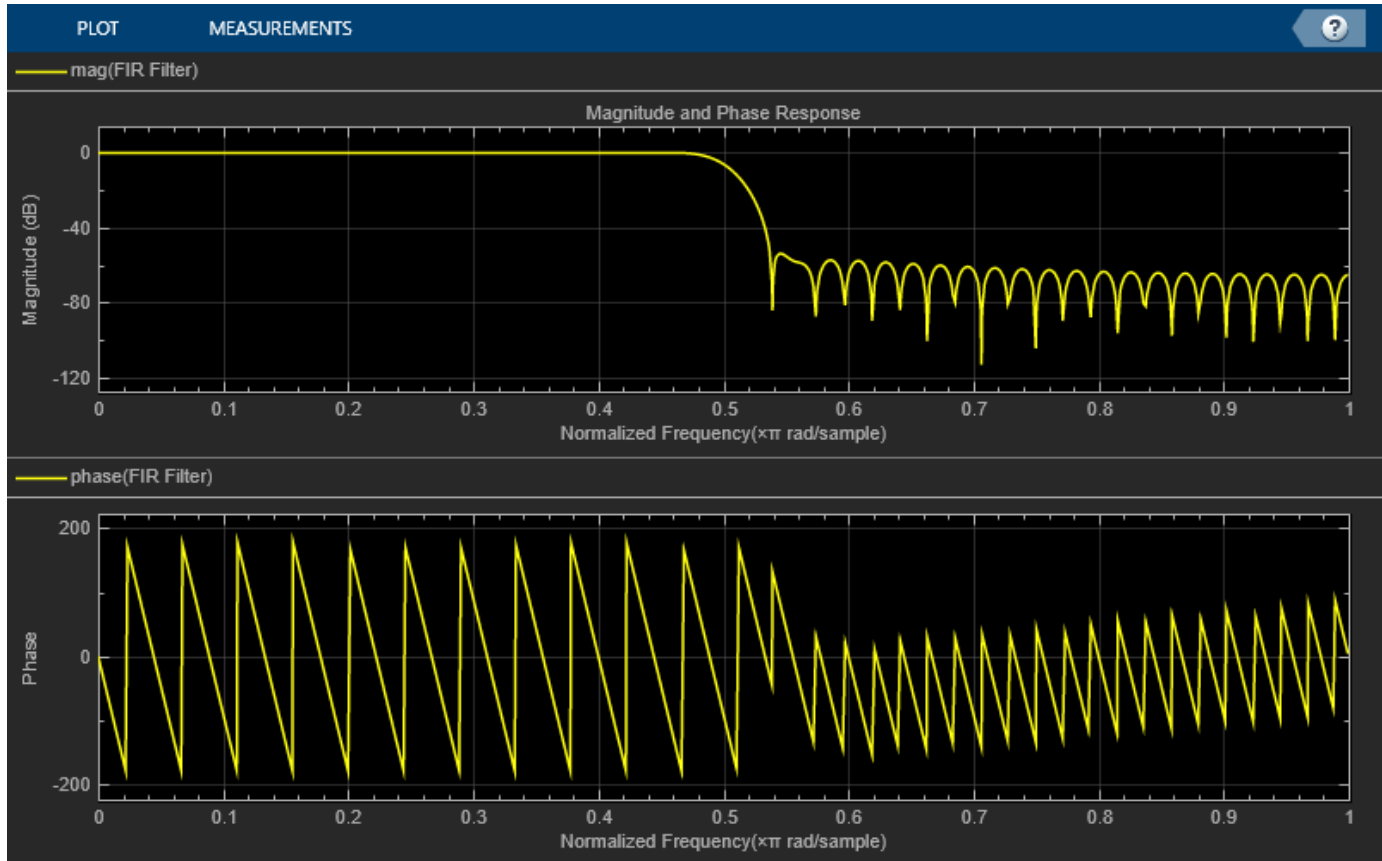
Show all properties

Vary the cutoff frequency of the FIR filter  $k$  from 0.1 to 0.5 in increments of 0.001. View the varying magnitude and phase response using the dynamic filter visualizer.

```

for k = 0.1:0.001:0.5
    b = fir1(90,k);
    dfv(b,1);
end

```



### Plot Time-Varying Magnitude Response of Variable Bandwidth FIR Filter

Visualize the varying magnitude response of the variable bandwidth FIR filter using the dynamic filter visualizer.

Create a `dsp.DynamicFilterVisualizer` object.

```

dfv = dsp.DynamicFilterVisualizer(YLimits=[-160 10],...
    FilterNames="Variable Bandwidth FIR Filter")

```

`dfv =`  
`dsp.DynamicFilterVisualizer` with properties:

```

    FFTLength: 2048
  NormalizedFrequency: 0
    SampleRate: 44100
   FrequencyRange: [0 22050]
        XScale: 'Linear'
  MagnitudeDisplay: 'Magnitude (dB)'

```

```

PlotAsMagnitudePhase: 0
    PlotType: 'Line'
    AxesScaling: 'Manual'

```

Show all properties

Design a bandpass variable bandwidth FIR filter with a center frequency of 5 kHz and a bandwidth of 4 kHz.

```

Fs = 44100;
vbw = dsp.VariableBandwidthFIRFilter(FilterType='Bandpass',...
    FilterOrder=100,...
    SampleRate=Fs,...
    CenterFrequency=5e3,...
    Bandwidth=4e3);

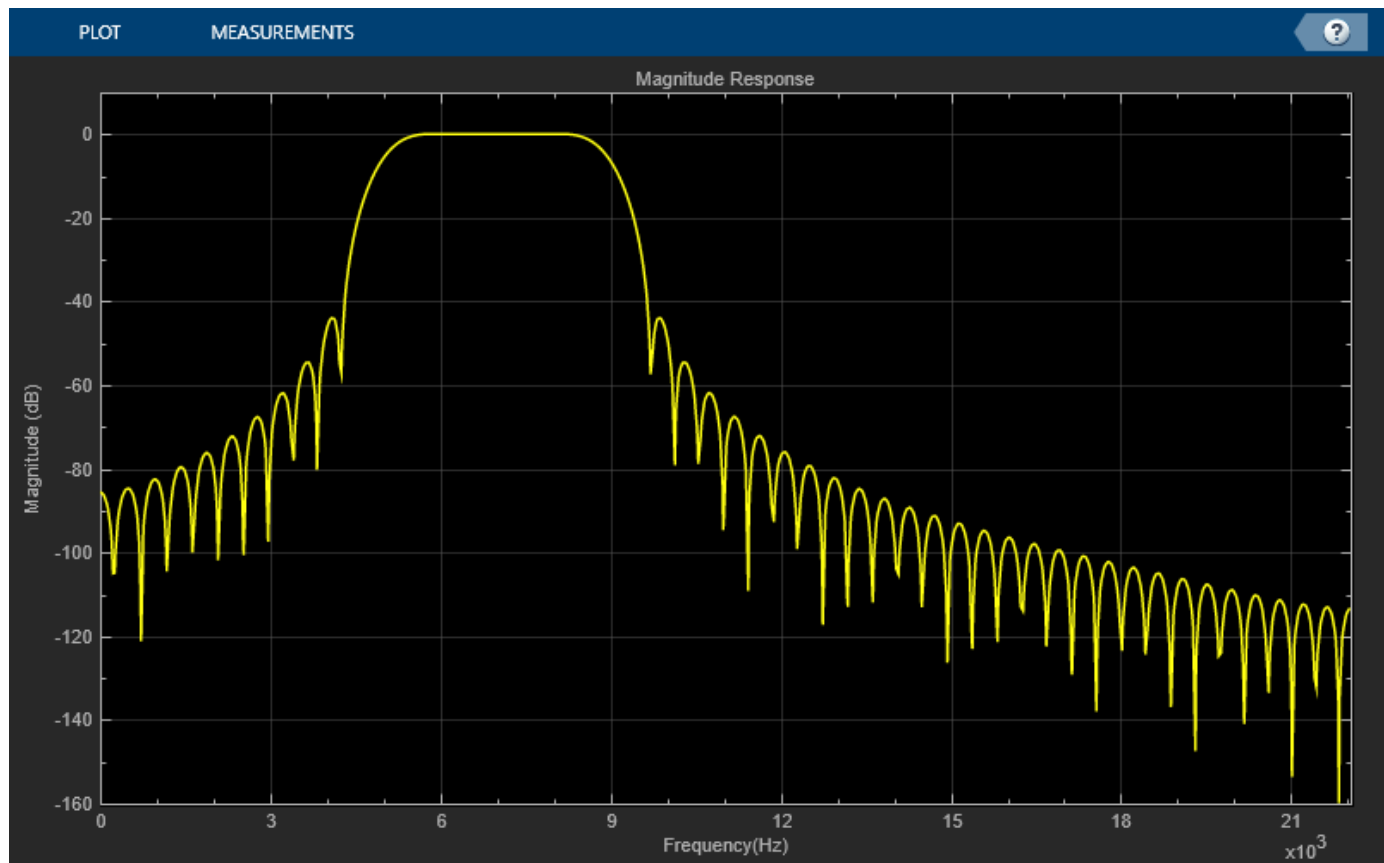
```

Vary the center frequency of the filter. Visualize the varying magnitude response of the filter using the `dsp.DynamicFilterVisualizer` object.

```

for idx = 1:100
    dfv(vbw);
    vbw.CenterFrequency = vbw.CenterFrequency + 20;
end

```





## Input Arguments

### **dfv** — Dynamic filter visualizer

`dsp.DynamicFilterVisualizer` object

Dynamic filter visualizer, specified as a `dsp.DynamicFilterVisualizer` object.

### **filt** — Filter

filter System object

Filter System object with a valid `freqz()` implementation.

### **B** — Numerator polynomial coefficients

row vector

Numerator polynomial coefficients, specified as a row vector.

Data Types: `single` | `double`

### **A** — Denominator polynomial coefficients

scalar | row vector

Denominator polynomial coefficients, specified as a:

- scalar -- The filter is an FIR filter.
- row vector -- The filter is an IIR filter.

Data Types: `single` | `double`

## Version History

Introduced in R2018b

## See Also

### Functions

`show` | `hide`

### Objects

`dsp.DynamicFilterVisualizer`

# specifyall

**Package:** dsp

Fully specify fixed-point filter System object settings

## Syntax

```
specifyall(sysobj)
specifyall(sysobj,false)
specifyall(sysobj,true)
```

## Description

`specifyall(sysobj)` sets all the data type fixed-point properties of the filter System object to 'Custom' so that you can easily specify all the fixed-point settings. If the object has a `FullPrecisionOverride` property, its value is set to `false`. `specifyall` is intended as a shortcut to changing all the fixed-point properties.

`specifyall(sysobj, false)` sets all fixed-point properties of the filter System object to their default values and sets the filter to full-precision mode, if one is available.

`specifyall(sysobj, true)` is equivalent to `specifyall(sysobj)`.

## Examples

### Specify All Fixed-point Settings of FIRFilter

Use `specifyall` to access all the fixed-point settings of an FIR filter implemented with a direct-form structure. Using `specifyall` disables all automatic filter scaling and resets the mode values.

```
b = firband(12,[0 0.4 0.5 1],[1 1 0 0],[1 0.2],{'w' 'c'});
firFilter = dsp.FIRFilter('Numerator',b);
get(firFilter)
```

*ans = struct with fields:*

```

    Numerator: [0.0164 0.1031 -0.0632 -0.0907 0.0467 0.3139 0.4526 0.3139 -0.0907 -0.0632 -0.1031 -0.0164]
  ReflectionCoefficients: [0.5000 0.5000]
    InitialConditions: 0
    NumeratorSource: 'Property'
  ReflectionCoefficientsSource: 'Property'
        Structure: 'Direct form'
  FullPrecisionOverride: 1
    RoundingMethod: 'Floor'
    OverflowAction: 'Wrap'
    CoefficientsDataType: 'Same word length as input'
  ReflectionCoefficientsDataType: 'Same word length as input'
    CustomCoefficientsDataType: [1x1 embedded.numerictype]
  CustomReflectionCoefficientsDataType: [1x1 embedded.numerictype]
        ProductDataType: 'Full precision'
    CustomProductDataType: [1x1 embedded.numerictype]
```

```

        AccumulatorDataType: 'Full precision'
    CustomAccumulatorDataType: [1x1 embedded.numerictype]
        StateDataType: 'Same as accumulator'
    CustomStateDataType: [1x1 embedded.numerictype]
        OutputDataType: 'Same as accumulator'
    CustomOutputDataType: [1x1 embedded.numerictype]

```

The `specifyall` function sets all the data type fixed-point properties of the FIR filter to 'Custom'.

```

specifyall(firFilter)
get(firFilter)

```

*ans = struct with fields:*

```

        Numerator: [0.0164 0.1031 -0.0632 -0.0907 0.0467 0.3139 0.4526 0.3139 0.1031 0.0164]
    ReflectionCoefficients: [0.5000 0.5000]
        InitialConditions: 0
        NumeratorSource: 'Property'
    ReflectionCoefficientsSource: 'Property'
        Structure: 'Direct form'
    FullPrecisionOverride: 0
        RoundingMethod: 'Floor'
        OverflowAction: 'Wrap'
        CoefficientsDataType: 'Custom'
    ReflectionCoefficientsDataType: 'Custom'
    CustomCoefficientsDataType: [1x1 embedded.numerictype]
    CustomReflectionCoefficientsDataType: [1x1 embedded.numerictype]
        ProductDataType: 'Custom'
    CustomProductDataType: [1x1 embedded.numerictype]
        AccumulatorDataType: 'Custom'
    CustomAccumulatorDataType: [1x1 embedded.numerictype]
        StateDataType: 'Custom'
    CustomStateDataType: [1x1 embedded.numerictype]
        OutputDataType: 'Custom'
    CustomOutputDataType: [1x1 embedded.numerictype]

```

## Input Arguments

### **sysobj** — Input filter object

filter System object

Input filter, specified as one of the following filter System objects:

- `dsp.AllpoleFilter`
- `dsp.BiquadFilter`
- `dsp.CICCompensationDecimator`
- `dsp.CICCompensationInterpolator`
- `dsp.CICDecimator`
- `dsp.CICInterpolator`
- `dsp.Differentiator`
- `dsp.FarrowRateConverter`

- `dsp.FIRDecimator`
- `dsp.FIRFilter`
- `dsp.FIRHalfbandDecimator`
- `dsp.FIRHalfbandInterpolator`
- `dsp.FIRInterpolator`
- `dsp.FIRRateConverter`
- `dsp.HighpassFilter`
- `dsp.IIRFilter`
- `dsp.LowpassFilter`
- `dsp.SOSFilter`

## **Version History**

**Introduced in R2011a**

# stepz

**Package:** dsp

Step response of discrete-time filter System object

## Syntax

```
[stepResp,t] = stepz(sysobj)
[stepResp,t] = stepz(sysobj,n)
[stepResp,t] = stepz(sysobj,n,fs)
[stepResp,t] = stepz(sysobj,[],fs)
[___] = stepz( ___, 'Arithmetic',arithType)
stepz(sysobj)
```

## Description

`[stepResp,t] = stepz(sysobj)` computes the step response of the filter System object and returns the response in the column vector `stepResp`, and a vector of times (or sample intervals) in `t`, where `t = [0 1 2 ...k-1]'`. `k` is the number of filter coefficients.

`[stepResp,t] = stepz(sysobj,n)` computes the step response at `floor(n)` 1-second intervals. The time vector `t` equals `(0:floor(n)-1)'`.

`[stepResp,t] = stepz(sysobj,n,fs)` computes the step response at `floor(n)` 1/fs-second intervals. The time vector `t` equals `(0:floor(n)-1)'/fs`.

`[stepResp,t] = stepz(sysobj,[],fs)` computes the step response at `k` 1/fs-second intervals, where `k` is the number of filter coefficients. The time vector `t` equals `(0:k-1)'/fs`.

`[___] = stepz( ___, 'Arithmetic',arithType)` analyzes the filter System object based on the arithmetic specified in `arithType` using any one of the previous syntaxes.

`stepz(sysobj)` launches `fvtool` and plots the step response of the filter System object.

For more input options, see `stepz` in Signal Processing Toolbox.

`stepz` works for both real and complex filters. When you omit the output arguments, the `stepz` function plots only the real part of the step response.

## Examples

### Step Response of Equiripple Lowpass FIR Filter

Design an equiripple lowpass FIR filter and compute the step response of the filter.

Use the `fdesign.lowpass` specification object to define the filter specifications. Run the `design` function on the specification object to create a `dsp.FIRFilter` object. Specify the passband frequency to be 20 kHz, stopband frequency to be 22.05 kHz, passband ripple to be 1 dB, and the stopband attenuation to be 80 dB. Specify the sampling frequency to be 96 kHz.

```
Fs = 96e3; filtSpecs = fdesign.lowpass(20e3,22.05e3,1,80,Fs);
firlp2 = design(filtSpecs,'equiripple','SystemObject',true)
```

```
firlp2 =
```

```
  dsp.FIRFilter with properties:
```

```
    Structure: 'Direct form'
```

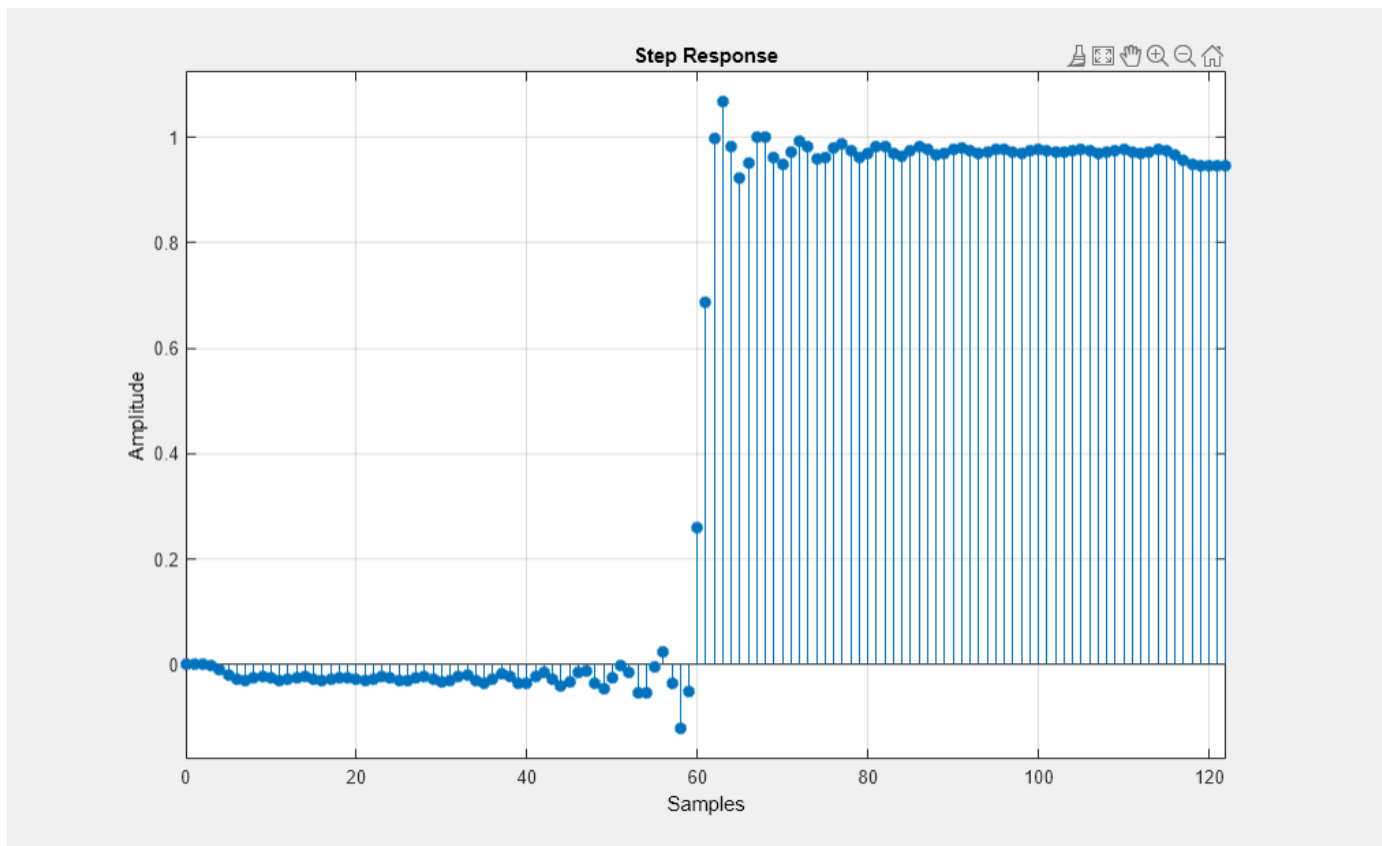
```
  NumeratorSource: 'Property'
```

```
    Numerator: [4.2157e-04 9.7407e-04 3.1143e-04 -0.0031 -0.0082 -0.0111 -0.0081 -8.9652e-05]
  InitialConditions: 0
```

```
Show all properties
```

Compute the step response of the filter using the `stepz` function.

```
stepz(firlp2)
```



## Input Arguments

### **sysobj** – Input filter

filter System object

Input filter, specified as one of the following filter System objects:

- `dsp.AllpassFilter`
- `dsp.AllpoleFilter`
- `dsp.BiquadFilter`
- `dsp.Channelizer`
- `dsp.CICCompensationDecimator`
- `dsp.CICCompensationInterpolator`
- `dsp.CICDecimator`
- `dsp.CICInterpolator`
- `dsp.CoupledAllpassFilter`
- `dsp.Differentiator`
- `dsp.FarrowRateConverter`
- `dsp.FilterCascade`
- `dsp.FIRDecimator`
- `dsp.FIRFilter`
- `dsp.FIRHalfbandDecimator`
- `dsp.FIRHalfbandInterpolator`
- `dsp.FIRInterpolator`
- `dsp.FIRRateConverter`
- `dsp.FourthOrderSectionFilter`
- `dsp.HighpassFilter`
- `dsp.IIRFilter`
- `dsp.IIRHalfbandDecimator`
- `dsp.IIRHalfbandInterpolator`
- `dsp.LowpassFilter`
- `dsp.NotchPeakFilter`
- `dsp.SOSFilter`
- `dsp.VariableBandwidthFIRFilter`
- `dsp.VariableBandwidthIIRFilter`

**n — Length of step response vector**

positive integer

Length of the step response vector, specified as a positive integer.

Data Types: `single` | `double`

**fs — Sampling frequency**

1 (default) | positive scalar

Sampling frequency used in computing the step response, specified as a positive scalar.

Data Types: `single` | `double`

**arithType — Arithmetic type**

'double' (default) | 'single' | 'Fixed'

Arithmetic used in the filter analysis, specified as 'double', 'single', or 'Fixed'. When the arithmetic input is not specified and the filter System object is unlocked, the analysis tool assumes a double-precision filter. When the arithmetic input is not specified and the System object is locked, the function performs the analysis based on the data type of the locked input.

The 'Fixed' value applies to filter System objects with fixed-point properties only.

When the 'Arithmetic' input argument is specified as 'Fixed' and the filter object has the data type of the coefficients set to 'Same word length as input', the arithmetic analysis depends on whether the System object is unlocked or locked.

- unlocked -- The analysis object function cannot determine the coefficients data type. The function assumes that the coefficients data type is signed, has a 16-bit word length, and is auto scaled. The function performs fixed-point analysis based on this assumption.
- locked -- When the input data type is 'double' or 'single', the analysis object function cannot determine the coefficients data type. The function assumes that the data type of the coefficients is signed, has a 16-bit word length, and is auto scaled. The function performs fixed-point analysis based on this assumption.

To check if the System object is locked or unlocked, use the `isLocked` function.

When the arithmetic input is specified as 'Fixed' and the filter object has the data type of the coefficients set to a custom numeric type, the object function performs fixed-point analysis based on the custom numeric data type.

## Output Arguments

### **stepResp** — Step response

vector

Step response, returned as an n-element vector. If n is not specified, the length of the step response vector equals the number of coefficients, k in the filter.

Data Types: double

### **t** — Time vector

vector

Time vector of length n in seconds. The vector t consists of n equally spaced points in the range  $(0 : \text{floor}(n) - 1) / fs$ . If n is not specified, the function uses the number of coefficients k in the filter.

Data Types: double

## Version History

Introduced in R2011a

### See Also

`freqz` | `impz` | `stepz`



## sysobj

Create filter System object from discrete-time filter

### Syntax

```
hs = sysobj(hfilt)
```

### Description

`hs = sysobj(hfilt)` creates a new filter System object `hs` from the `dfilt` object, `hfilt`.

The function supports a subset of `dfilt` objects. The following table lists supported filter structures for `hfilt` and the filter System object that the function creates.

| Single-rate   | Filter System object                  |
|---|---------------------------------------|
| Lattice AR( <code>dfilt.latticear</code> )  | <code>dsp.AllpoleFilter</code>        |
| Coupled-allpass, power-complementary lattice filter ( <code>dfilt.calatticepc</code> )        | <code>dsp.CoupledAllpassFilter</code> |
| Coupled-allpass, lattice filter ( <code>dfilt.calattice</code> )                              | <code>dsp.CoupledAllpassFilter</code> |
| Cascade of discrete time filters ( <code>dfilt.cascade</code> )                               | <code>dsp.CoupledAllpassFilter</code> |
| Direct Form I ( <code>dfilt.df1</code> )  | <code>dsp.IIRFilter</code>            |
| Direct Form I transposed ( <code>dfilt.df1t</code> )  | <code>dsp.IIRFilter</code>            |
| Direct Form II ( <code>dfilt.df2</code> )   | <code>dsp.IIRFilter</code>            |
| Direct Form II transposed ( <code>dfilt.df2t</code> )   | <code>dsp.IIRFilter</code>            |
| Direct-form FIR ( <code>dfilt.dffir</code> )  | <code>dsp.FIRFilter</code>            |
| Direct-form FIR transposed ( <code>dfilt.dffirt</code> )                                      | <code>dsp.FIRFilter</code>            |
| Direct-form symmetric FIR ( <code>dfilt.dfsymfir</code> )                                     | <code>dsp.FIRFilter</code>            |
| Direct-form antisymmetric FIR ( <code>dfilt.dfasymfir</code> )                                | <code>dsp.FIRFilter</code>            |
| Discrete-time, lattice, moving-average ( <code>dfilt.latticemamin</code> )                    | <code>dsp.FIRFilter</code>            |
| Discrete-time, second-order section, direct-form I ( <code>dfilt.df1sos</code> )              | <code>dsp.BiquadFilter</code>         |
| Discrete-time, second-order section, direct-form I transposed ( <code>dfilt.df1tsos</code> )  | <code>dsp.BiquadFilter</code>         |
| Discrete-time, second-order section, direct-form II ( <code>dfilt.df2sos</code> )             | <code>dsp.BiquadFilter</code>         |
| Discrete-time, second-order section, direct-form II transposed ( <code>dfilt.df2tsos</code> ) | <code>dsp.BiquadFilter</code>         |

## Input Arguments

### hfilt

Discrete-time filter (dfilt) object. The preceding table lists supported filter structures.

If hfilt is a discrete-time filter with the PersistentMemory property set to true, then the filter states are copied into the initial conditions properties of hs. Otherwise, initial conditions are ignored.

The function does not support some properties for SOS filter structures:

- If the CastBeforeSum property is set to false, the function issues a warning. dsp.BiquadFilter System objects always have a cast before a sum.
- If the Signed property is false, the function issues an error. dsp.BiquadFilter System objects do not support unsigned arithmetic.

## Output Arguments

### hs

Filter System object. The function maps almost all properties of hfilt into the filter System object. However, some properties are not mapped exactly:

- Filter System objects do not have a CoeffAutoScale property. The function specifies a word length and a fraction length regardless of whether the CoeffAutoScale property of hfilt is true or false.
- dsp.BiquadFilter System objects do not have a FullPrecisionOverride property. Full-precision values in hfilt are mapped to word and fraction lengths in hs. These settings correspond to the full-precision setting of the input data type.

## Examples

### Convert a discrete-time filter object to a System object

```
hfilt = dfilt.df1sos; %Direct-form I SOS
hs = sysobj(hfilt) %Biquadratic IIR filter

hs =
  dsp.BiquadFilter with properties:
      Structure: 'Direct form I'
  SOSMatrixSource: 'Property'
      SOSMatrix: [1 0 0 1 0 0]
      ScaleValues: [2x1 double]
  NumeratorInitialConditions: 0
  DenominatorInitialConditions: 0
  OptimizeUnityScaleValues: true
```

Show all properties

## **Version History**

**Introduced in R2012a**

## **tf**

**Package:** dsp

Return transfer function of overall prototype lowpass filter

### **Syntax**

```
[num,den] = tf(obj)
```

### **Description**

`[num,den] = tf(obj)` returns the vector of numerator coefficients, `num`, and the vector of denominator coefficients, `den`, for the overall prototype lowpass filter used for the filter bank in `dsp.Channelizer` and `dsp.ChannelSynthesizer` System objects.

### **Examples**

#### **Transfer Function of Prototype Lowpass Filter**

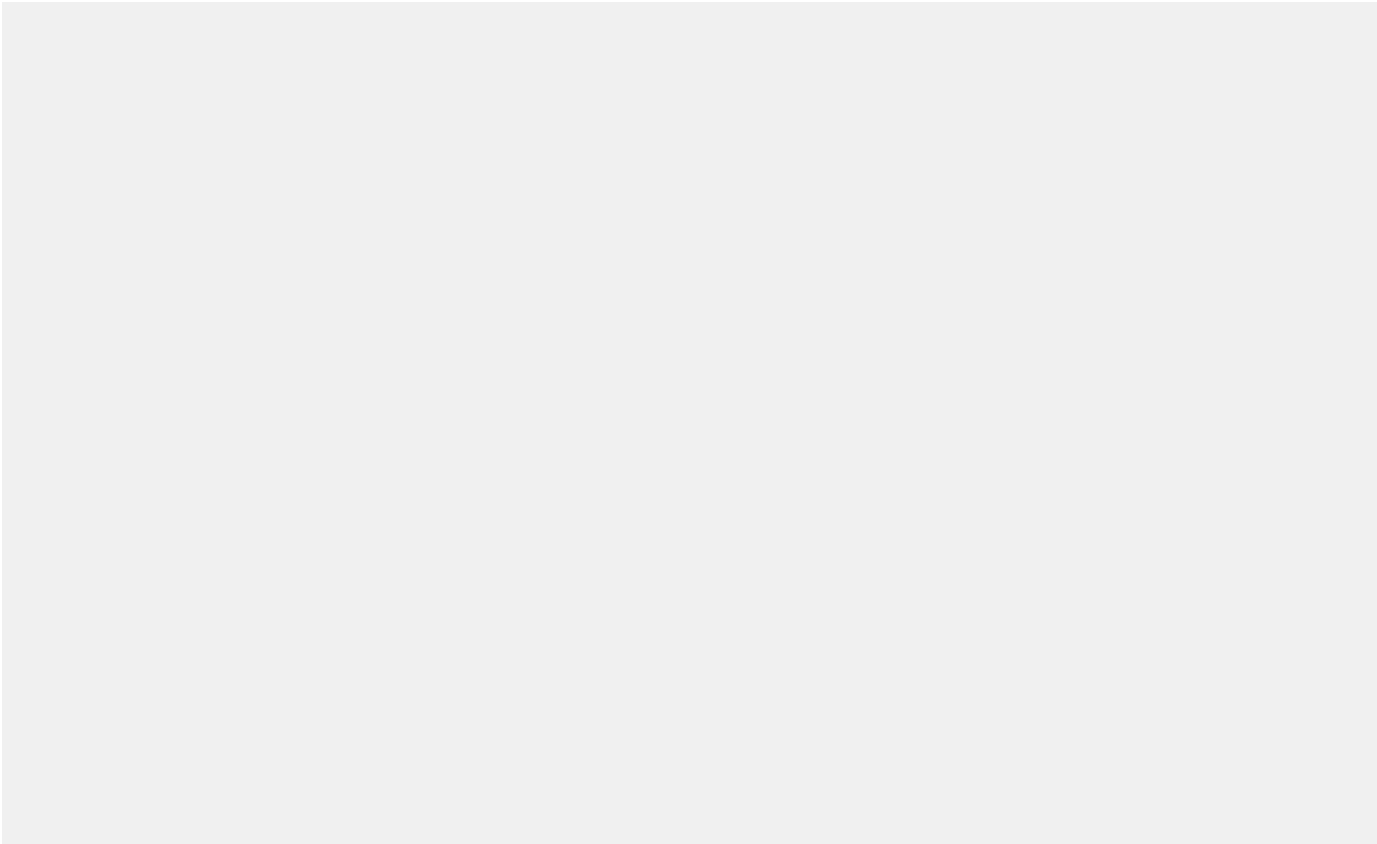
Determine the transfer function of the overall prototype lowpass filter used for the filter bank in the `dsp.Channelizer` System object.

Design a channelizer with the number of frequency bands or polyphase branches set to 8, the number of taps or coefficients per band set to 12, and stopband attenuation set to 80 dB. The function `tf` returns the transfer function of the prototype lowpass filter.

```
channelizer = dsp.Channelizer;  
[num,den] = tf(channelizer);
```

View the magnitude response of the prototype lowpass filter using `fvtool`.

```
fvtool(num,den);
```



## Input Arguments

### **obj** — Input filter System object

`dsp.Channelizer` | `dsp.ChannelSynthesizer`

Input filter, specified as either a `dsp.Channelizer` or a `dsp.ChannelSynthesizer` System object.

Example: `channelizer = dsp.Channelizer;`

Example: `channelizer = dsp.ChannelSynthesizer`

## Output Arguments

### **num** — Numerator coefficients

row vector

Numerator coefficients, returned as a row vector.

### **den** — Denominator coefficients

row vector

Denominator coefficients, returned as a row vector. For a finite Impulse response (FIR) filter, this value is 1.

## **Version History**

**Introduced in R2016b**

### **See Also**

#### **Functions**

polyphase | coeffs | fvtool | freqz | bandedgeFrequencies | centerFrequencies | getFilters

#### **Objects**

dsp.Channelizer | dsp.ChannelSynthesizer

# tf

**Package:** dsp

Convert discrete-time filter System object to transfer function

## Syntax

```
[num,den] = tf(sysobj)
[num,den] = tf(sysobj,'Arithmetic',arithType)
```

## Description

`[num,den] = tf(sysobj)` converts the discrete-time filter System object to numerator and denominator coefficient vectors of the equivalent transfer function.

`[num,den] = tf(sysobj,'Arithmetic',arithType)` analyzes the filter System object based on the arithmetic specified in `arithType`.

For more input options, see `tf` in Signal Processing Toolbox.

## Examples

### Transfer Function of SOS Filter

Compute the transfer function of an SOS filter using the `tf` function.

Create a fourth-order, lowpass SOS filter object with a normalized cutoff frequency of 0.4.

```
[z,p,k] = ellip(4,1,60,0.4); % Set up the filter
[sosMatrix,scaleValues] = zp2sos(z,p,k);
sosFilt = dsp.SOSFilter(Structure='Direct form I',...
    Numerator=sosMatrix(:,1:3),Denominator=sosMatrix(:,4:6),...
    HasScaleValues=true,ScaleValues=scaleValues)
```

```
sosFilt =
    dsp.SOSFilter with properties:
        Structure: 'Direct form I'
    CoefficientSource: 'Property'
        Numerator: [2x3 double]
        Denominator: [2x3 double]
    HasScaleValues: true
        ScaleValues: [0.0351 1 1]
```

Show all properties

Compute the transfer function of the designed SOS filter. The `tf` function returns the numerator and the denominator coefficient vectors of the filter.

```
[num,den] = tf(sosFilt)
```

```
num = 1×5
    0.0351    0.1038    0.1432    0.1038    0.0351

den = 1×5
    1.0000   -1.5676    1.7412   -1.0104    0.3093
```

## Input Arguments

### **sysobj** — Input filter

filter System object

Input filter, specified as one of the following filter System objects:

- `dsp.AllpassFilter`
- `dsp.AllpoleFilter`
- `dsp.BiquadFilter`
- `dsp.CICCompensationDecimator`
- `dsp.CICCompensationInterpolator`
- `dsp.CICDecimator`
- `dsp.CICInterpolator`
- `dsp.CoupledAllpassFilter`
- `dsp.Differentiator`
- `dsp.FarrowRateConverter`
- `dsp.FilterCascade`
- `dsp.FIRDecimator`
- `dsp.FIRFilter`
- `dsp.FIRHalfbandDecimator`
- `dsp.FIRHalfbandInterpolator`
- `dsp.FIRInterpolator`
- `dsp.FIRRateConverter`
- `dsp.FourthOrderSectionFilter`
- `dsp.HighpassFilter`
- `dsp.IIRFilter`
- `dsp.IIRHalfbandDecimator`
- `dsp.IIRHalfbandInterpolator`
- `dsp.LowpassFilter`
- `dsp.SOSFilter`
- `dsp.VariableBandwidthFIRFilter`
- `dsp.VariableBandwidthIIRFilter`



### **arithType — Arithmetic type**

'double' (default) | 'single' | 'Fixed'

Arithmetic used in the filter analysis, specified as 'double', 'single', or 'Fixed'. When the arithmetic input is not specified and the filter System object is unlocked, the analysis tool assumes a double-precision filter. When the arithmetic input is not specified and the System object is locked, the function performs the analysis based on the data type of the locked input.

The 'Fixed' value applies to filter System objects with fixed-point properties only.

When the 'Arithmetic' input argument is specified as 'Fixed' and the filter object has the data type of the coefficients set to 'Same word length as input', the arithmetic analysis depends on whether the System object is unlocked or locked.

- unlocked -- The analysis object function cannot determine the coefficients data type. The function assumes that the coefficients data type is signed, has a 16-bit word length, and is auto scaled. The function performs fixed-point analysis based on this assumption.
- locked -- When the input data type is 'double' or 'single', the analysis object function cannot determine the coefficients data type. The function assumes that the data type of the coefficients is signed, has a 16-bit word length, and is auto scaled. The function performs fixed-point analysis based on this assumption.

To check if the System object is locked or unlocked, use the `isLocked` function.

When the arithmetic input is specified as 'Fixed' and the filter object has the data type of the coefficients set to a custom numeric type, the object function performs fixed-point analysis based on the custom numeric data type.

## **Output Arguments**

### **num — Numerator coefficients**

row vector

Numerator coefficients of the filter, returned as a row vector.

Data Types: `double`

### **den — Denominator coefficients**

scalar | row vector

Denominator coefficients of the filter, returned as a row vector.

Data Types: `double`

## **Version History**

Introduced in R2011a

## **See Also**

### **Functions**

tf

**Topics**

“Analysis Methods for Filter System Objects” on page 3-2

## tf2ca

Transfer function to coupled allpass

### Syntax

```
[d1,d2] = tf2ca(b,a)
[d1,d2] = tf2ca(b,a)
[d1,d2,beta] = tf2ca(b,a)
```

### Description

`[d1,d2] = tf2ca(b,a)` where **b** is a real, symmetric vector of numerator coefficients and **a** is a real vector of denominator coefficients, corresponding to a stable digital filter, returns real vectors **d1** and **d2** containing the denominator coefficients of the allpass filters  $H1(z)$  and  $H2(z)$  such that

$$H(z) = \frac{B(z)}{A(z)} = \left(\frac{1}{2}\right)[H1(z) + H2(z)]$$

representing a coupled allpass decomposition.

`[d1,d2] = tf2ca(b,a)` where **b** is a real, antisymmetric vector of numerator coefficients and **a** is a real vector of denominator coefficients, corresponding to a stable digital filter, returns real vectors **d1** and **d2** containing the denominator coefficients of the allpass filters  $H1(z)$  and  $H2(z)$  such that

$$H(z) = \frac{B(z)}{A(z)} = \left(\frac{1}{2}\right)[H1(z) - H2(z)]$$

In some cases, the decomposition is not possible with real  $H1(z)$  and  $H2(z)$ . In those cases a generalized coupled allpass decomposition may be possible, as described in the following syntax.

`[d1,d2,beta] = tf2ca(b,a)` returns complex vectors **d1** and **d2** containing the denominator coefficients of the allpass filters  $H1(z)$  and  $H2(z)$ , and a complex scalar **beta**, satisfying  $|\text{beta}| = 1$ , such that

$$H(z) = \frac{B(z)}{A(z)} = \left(\frac{1}{2}\right)[\bar{\beta} \cdot H1(z) + \beta \cdot H2(z)]$$

representing the generalized allpass decomposition.

In the above equations,  $H1(z)$  and  $H2(z)$  are real or complex allpass IIR filters given by

$$H1(z) = \frac{\text{fliplr}(\overline{D1(z)})}{D1(z)}, H2(z) = \frac{\text{fliplr}(\overline{D2(1)(z)})}{D2(1)(z)}$$

where  $D1(z)$  and  $D2(z)$  are polynomials whose coefficients are given by **d1** and **d2**.

---

**Note** A coupled allpass decomposition is not always possible. Nevertheless, Butterworth, Chebyshev, and Elliptic IIR filters, among others, can be factored in this manner. For details, refer to Signal Processing Toolbox User's Guide.

---

## Examples

```
[b,a]=cheby1(9,.5,.4);  
[d1,d2]=tf2ca(b,a); % TF2CA returns denominators of the allpass.  
num = 0.5*conv(fliplr(d1),d2)+0.5*conv(fliplr(d2),d1);  
den = conv(d1,d2); % Reconstruct numerator and denominator.  
MaxDiff=max([max(b-num),max(a-den)]); % Compare original and reconstructed  
% numerator and denominators.
```

## Version History

Introduced in R2011a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

All inputs must be constant. Expressions or variables are allowed if their values do not change.

### See Also

[ca2tf](#) | [cl2tf](#) | [iirpowcomp](#) | [latc2tf](#) | [tf2latc](#)

## tf2cl

Transfer function to coupled allpass lattice

### Syntax

```
[k1,k2] = tf2cl(b,a)
[k1,k2] = tf2cl(b,a)
[k1,k2,beta] = tf2cl(b,a)
```

### Description

`[k1,k2] = tf2cl(b,a)` where `b` is a real, symmetric vector of numerator coefficients and `a` is a real vector of denominator coefficients, corresponding to a stable digital filter, will perform the coupled allpass decomposition

$$H(z) = \frac{B(z)}{A(z)} = \left(\frac{1}{2}\right)[H1(z) + H2(z)]$$

of a stable IIR filter  $H(z)$  and convert the allpass transfer functions  $H1(z)$  and  $H2(z)$  to a coupled lattice allpass structure with coefficients given in vectors `k1` and `k2`.

`[k1,k2] = tf2cl(b,a)` where `b` is a real, antisymmetric vector of numerator coefficients and `a` is a real vector of denominator coefficients, corresponding to a stable digital filter, performs the coupled allpass decomposition

$$H(z) = \frac{B(z)}{A(z)} = \left(\frac{1}{2}\right)[H1(z) - H2(z)]$$

of a stable IIR filter  $H(z)$  and converts the allpass transfer functions  $H1(z)$  and  $H2(z)$  to a coupled lattice allpass structure with coefficients given in vectors `k1` and `k2`.

In some cases, the decomposition is not possible with real  $H1(z)$  and  $H2(z)$ . In those cases, a generalized coupled allpass decomposition may be possible, using the syntax described below.

`[k1,k2,beta] = tf2cl(b,a)` performs the generalized allpass decomposition of a stable IIR filter  $H(z)$  and converts the complex allpass transfer functions  $H1(z)$  and  $H2(z)$  to corresponding lattice allpass filters

$$H(z) = \frac{B(z)}{A(z)} = \left(\frac{1}{2}\right)[\bar{\beta} \cdot H1(z) + \beta \cdot H2(z)]$$

where `beta` is a complex scalar of magnitude equal to 1.

---

**Note** Coupled allpass decomposition is not always possible. Nevertheless, Butterworth, Chebyshev, and Elliptic IIR filters, among others, can be factored in this manner. For details, refer to Signal Processing Toolbox User's Guide.

---

### Examples

```
[b,a]=cheby1(9,.5,.4);
[k1,k2]=tf2cl(b,a); % Get the reflection coeffs. for the lattices.
```

```
[num1,den1]=latc2tf(k1,'allpass'); % Convert each allpass lattice
[num2,den2]=latc2tf(k2,'allpass'); % back to transfer function.
num = 0.5*conv(num1,den2)+0.5*conv(num2,den1);
den = conv(den1,den2); % Reconstruct numerator and denominator.
MaxDiff=max([max(b-num),max(a-den)]); % Compare original and reconstructed
% numerator and denominators.
```

## Version History

Introduced in R2011a

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

All inputs must be constant. Expressions or variables are allowed if their values do not change.

### See Also

[ca2tf](#) | [cl2tf](#) | [iirpowcomp](#) | [latc2tf](#) | [tf2ca](#) | [tf2latc](#)

# validstructures

Structures for specification object with design method

## Syntax

```
filtstruct = validstructures(designSpecs,'Systemobject',true)
filtstruct = validstructures(designSpecs,method,'Systemobject',true)
```

## Description

`filtstruct = validstructures(designSpecs,'Systemobject',true)` returns a structure of cell arrays, `filtstruct`, which contains a set of valid filter structures for the filter specification object `designSpecs`. When you set `'Systemobject'` to `true`, `validstructures` returns a list of structures that support filter System objects. Each field in `filtstruct` lists a set of filter structures for the design method specified.

`filtstruct = validstructures(designSpecs,method,'Systemobject',true)` returns the valid structures for the filter specification object, `designSpecs`, and the design method, `method`, in a cell array of character vectors.

## Examples

### Valid Filter Structures

Design a default lowpass filter specification object. Use the `validstructures` function to obtain valid design methods and structures in a structure array. Display the fieldnames to see all valid design methods. Display the valid filter structures for the `equiripple` field.

```
D = fdesign.lowpass;
filtstruct = validstructures(D,'SystemObject',true);
```

```
fn = fieldnames(filtstruct)
```

```
fn = 8x1 cell
    {'butter'   }
    {'cheby1'  }
    {'cheby2'  }
    {'ellip'   }
    {'equiripple'}
    {'ifir'    }
    {'multistage'}
    {'kaiserwin' }
```

```
strs = eval(['filtstruct.' fn{5}])
```

```
strs = 1x3 cell
    {'dffir'}    {'dffirt'}    {'dfsymfir'}
```

Create a highpass filter of order 50 with a 3-dB frequency of 0.2. Obtain the available structures for a Butterworth design.

```
D = fdesign.highpass('N,F3dB',50,0.2);
C = validstructures(D,'butter','SystemObject',true)

C = 1x6 cell
    {'df1sos'}    {'df2sos'}    {'df1tsos'}    {'df2tsos'}    {'cascadeallpass'}    {'cascadewdf'}
```

## Input Arguments

### **designSpecs** — Filter specification object

object

Filter specification object, specified as one of the `fdesign` functions.

### **method** — Design method

character vector

Design method, specified as a character vector. You can pick a design method from the available methods given by the `designmethods` function.

## Output Arguments

### **filtstruct** — Available filter structures

structure

Available filter structures, returned as a structure with the fields determined by the input filter specification object, `designSpecs`, and the chosen design method.

Data Types: `struct`

## Version History

Introduced in R2009a

### See Also

`design` | `designmethods` | `designopts` | `fdesign`



# visualizeFilterStages

**Package:** dsp

Visualize filter stages

## Syntax

```
visualizeFilterStages(sysobj)
```

## Description

`visualizeFilterStages(sysobj)` shows the response of each individual filter stage of the filter System object using FVTool.

## Examples

### Filter Signal Through Complex Bandpass Decimator

Filter an input signal through a complex bandpass decimator and visualize the filtered spectrum in a spectrum analyzer.

#### Initialization

Create a `dsp.ComplexBandpassDecimator` System object™ with center frequency set to 2000 Hz, bandwidth of interest set to 1000 Hz, and sample rate set to 48 kHz. The decimation factor is computed as the ratio of the sample rate to the bandwidth of interest. The input to the decimator is a sine wave with a frame length of 1200 samples with tones at 1625 Hz, 2000 Hz, and 2125 Hz. Create a `spectrumAnalyzer` scope to visualize the signal spectrum.

```
Fs = 48e3;
CF = 2000;
BW = 1000;
D = Fs/BW;
FrameLength = 1200;
bpdecim = dsp.ComplexBandpassDecimator(D,CF,Fs);

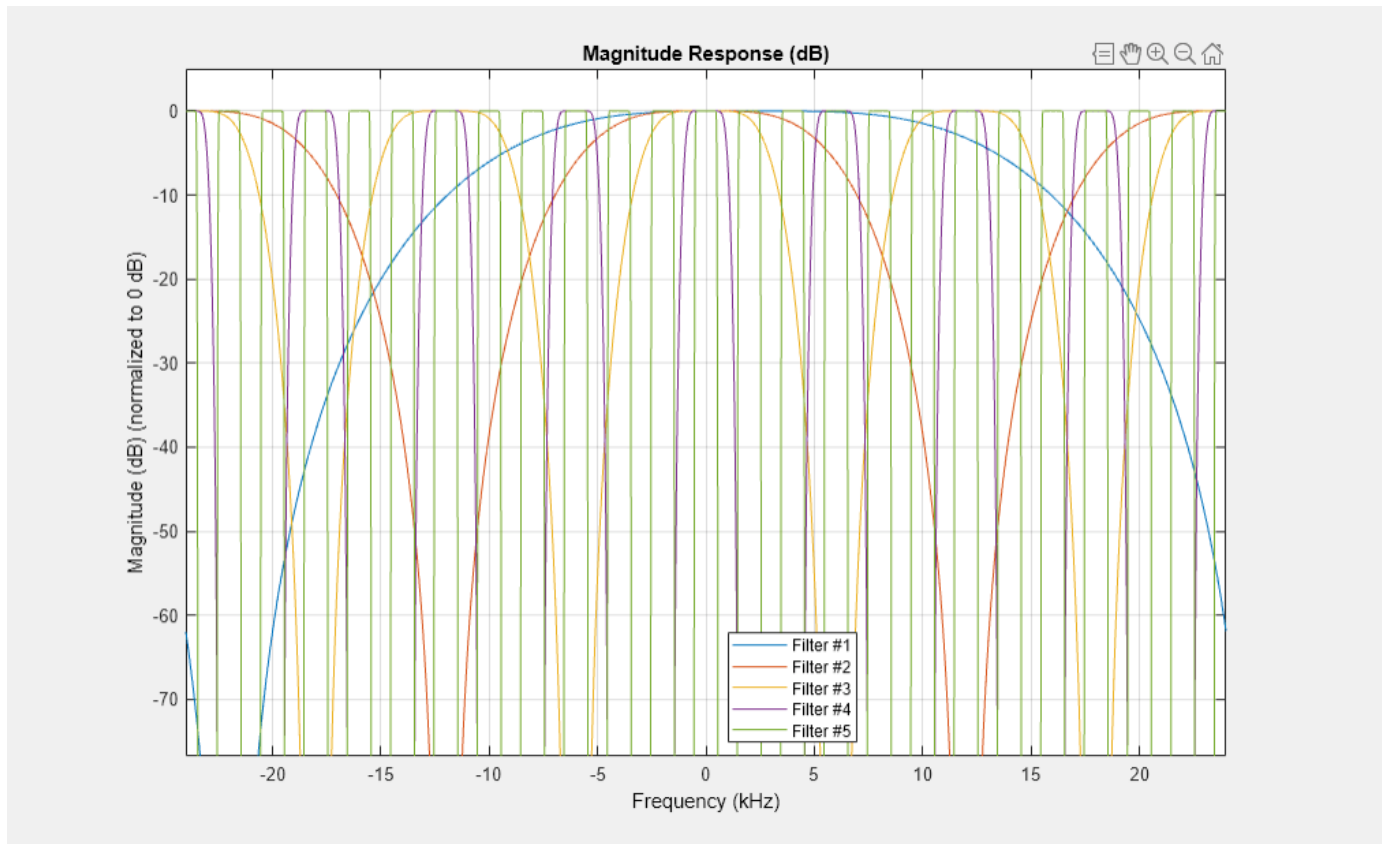
sa = spectrumAnalyzer('SampleRate',Fs/D,...
    'Method','welch',...
    'YLimits',[-120 40],...
    'FrequencyOffset',CF);

tones = [1625 2000 2125];
sin = dsp.SineWave('SampleRate',Fs,'Frequency',tones,...
    'SamplesPerFrame',FrameLength);
```

#### Visualize Filter Stages

Using the `visualizeFilterStages` function, you can visualize the response of each individual filter stage using FVTool.

```
visualizeFilterStages(bpdecim)
```



### Display Filter info

The `info` function displays information about the bandpass decimator.

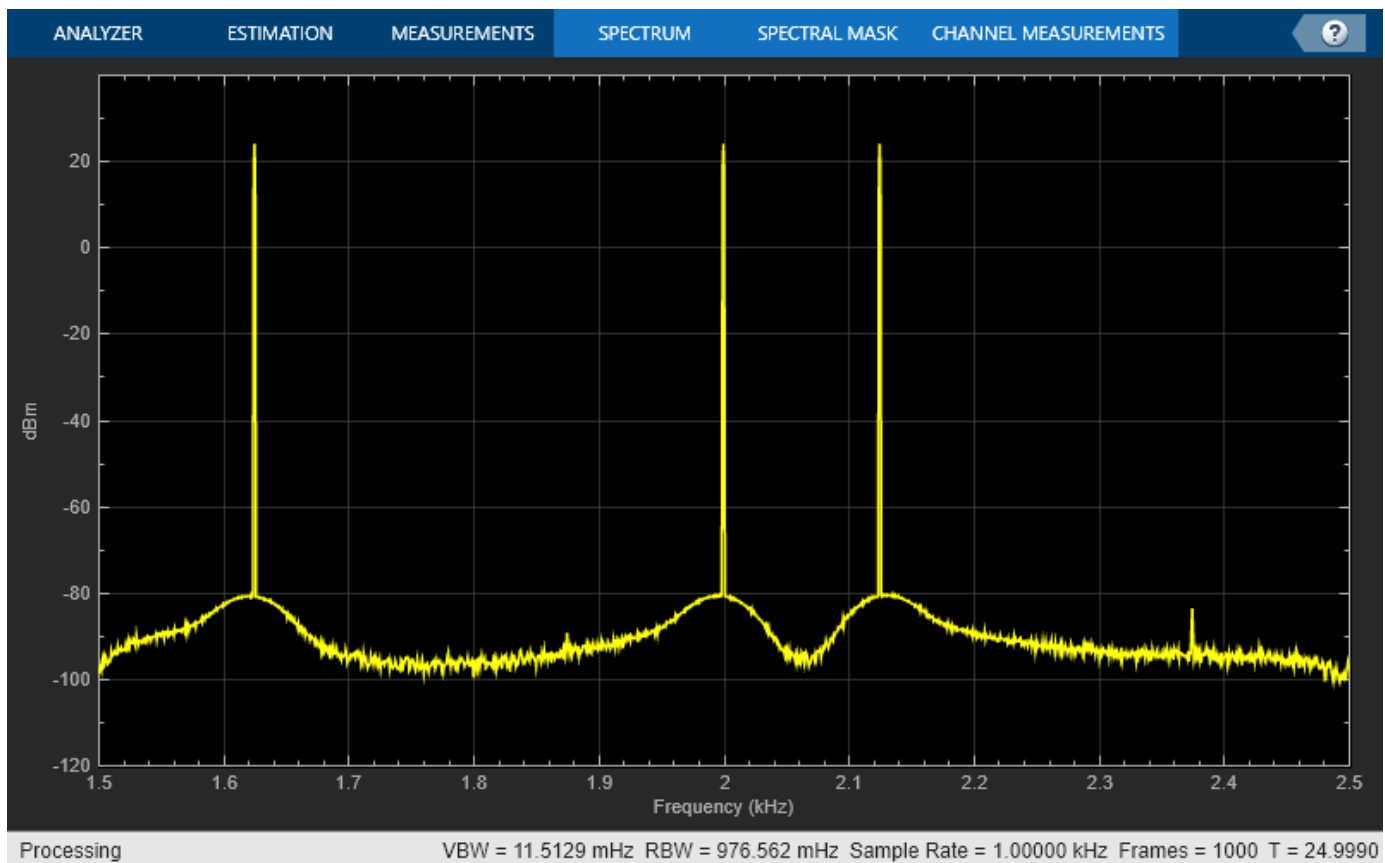
```
fprintf('%s',info(bpdecim))
```

```
Overall Decimation Factor      : 48
Bandwidth                     : 1000 Hz
Number of Filters             : 5
Real multiplications per Input Sample: 14.708333
Real additions per Input Sample  : 13.833333
Number of Coefficients        : 89
Filters:
  Filter 1:
  dsp.FIRDecimator - Decimation Factor : 2
  Filter 2:
  dsp.FIRDecimator - Decimation Factor : 2
  Filter 3:
  dsp.FIRDecimator - Decimation Factor : 2
  Filter 4:
  dsp.FIRDecimator - Decimation Factor : 3
  Filter 5:
  dsp.FIRDecimator - Decimation Factor : 2
```

## Stream In and Filter Signal

Construct a for-loop to run for 1000 iterations. In each iteration, stream in 1200 samples (one frame) of the noisy sine wave and apply the complex bandpass decimator on each frame of the input signal. Visualize the input and output spectrum in the spectrum analyzer, sa.

```
for index = 1:1000
    x = sum(sin(),2) + 1e-4*randn(FrameLength,1);
    z = bpdecim(x);
    sa(z);
end
```



The bandpass decimator with center frequency at 2000 Hz and a bandwidth of 1000 Hz passes the three sine wave tones at 1625 Hz, 2000 Hz, and 2125 Hz.

Change the center frequency of the decimator to 2400 Hz and filter the signal.

```
release(bpdecim);
bpdecim.CenterFrequency = 2400

bpdecim =
    dsp.ComplexBandpassDecimator with properties:

        CenterFrequency: 2400
        Specification: 'Decimation factor'
        DecimationFactor: 48
        StopbandAttenuation: 80
```

```

TransitionWidth: 100
MinimizeComplexCoefficients: true
SampleRate: 48000

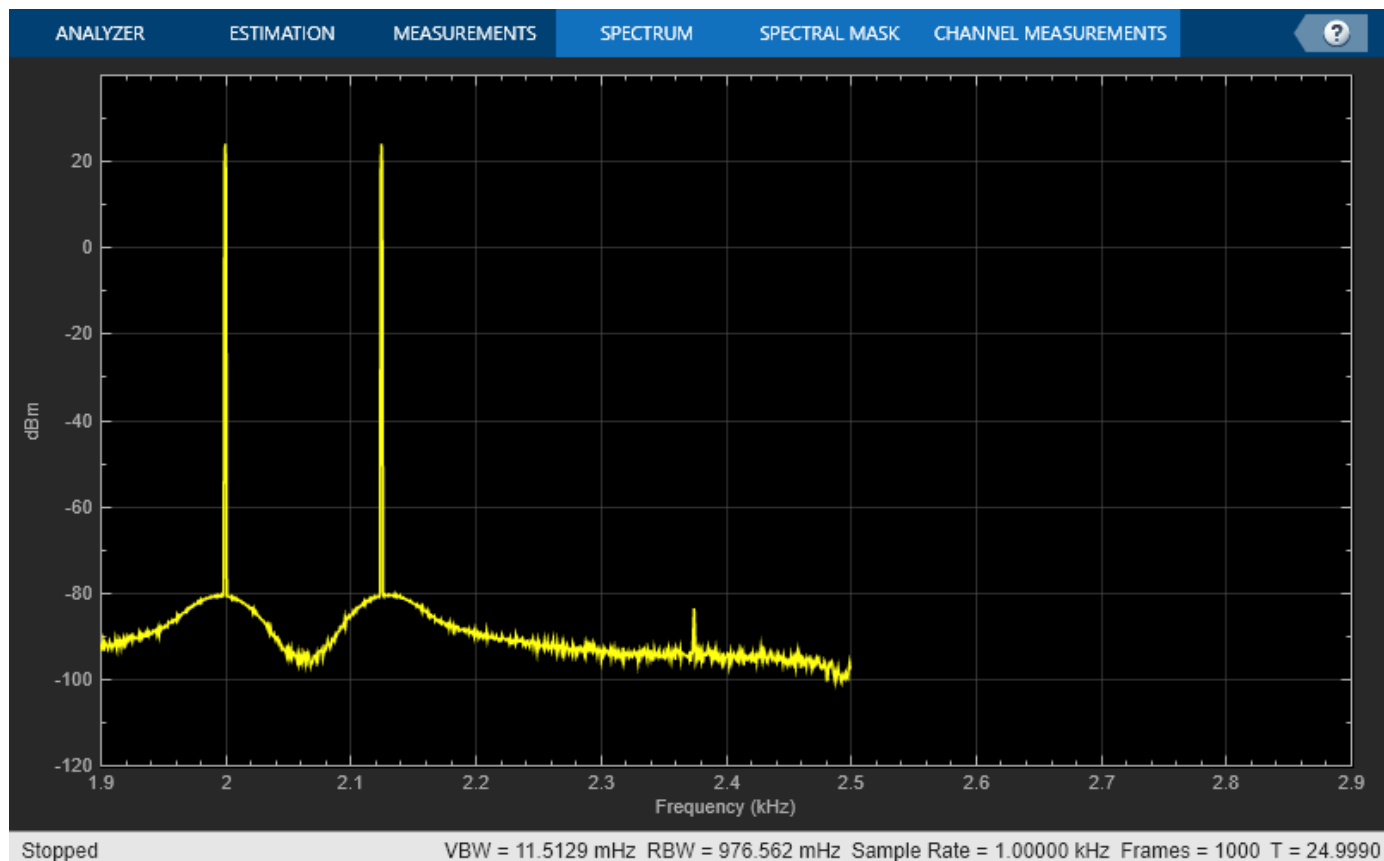
```

Configure the spectrum analyzer to show the bandwidth of interest, [-1900, 2900] Hz.

```

release(sa)
sa.FrequencyOffset = 2400;

```

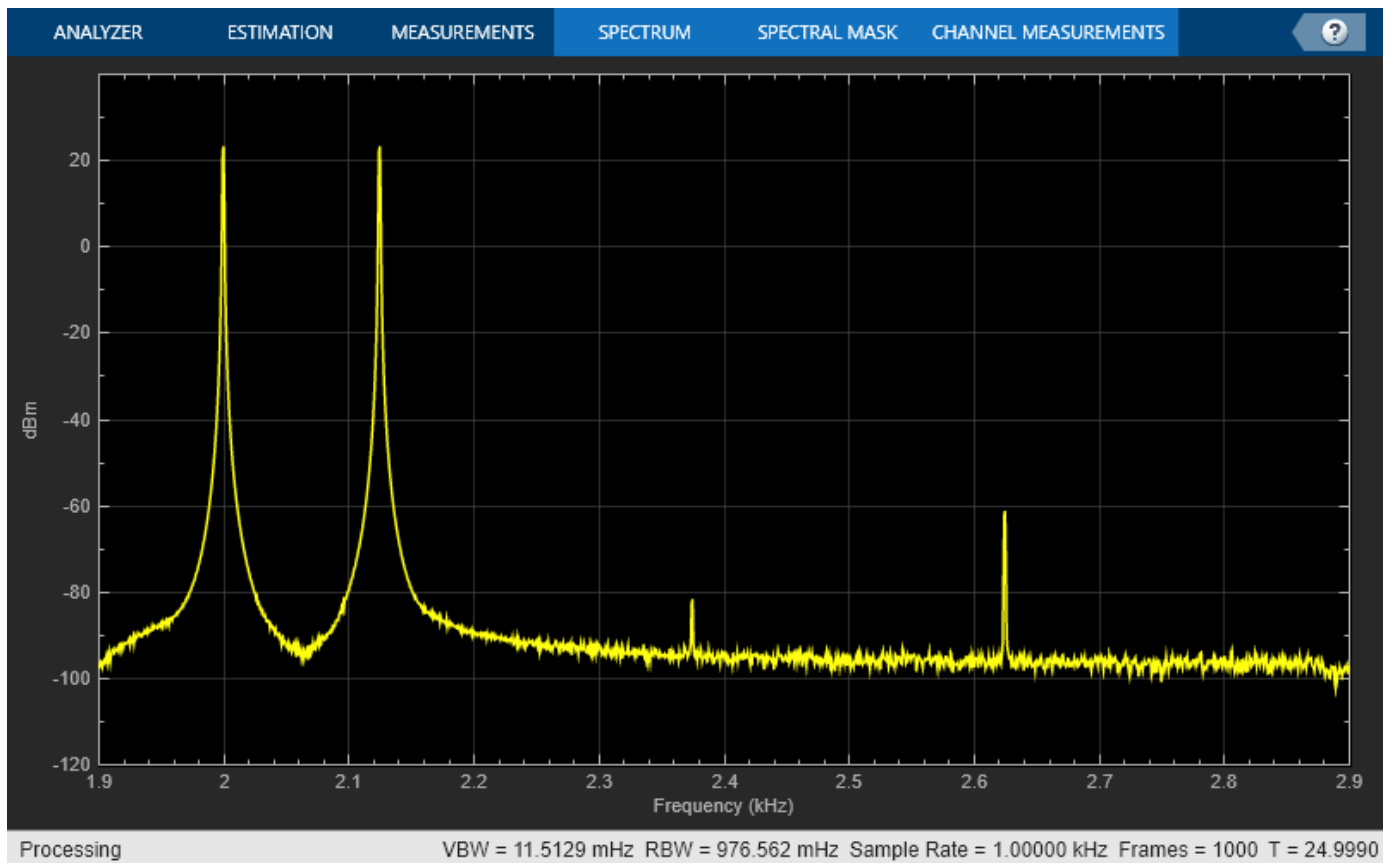


Stream in the data and filter the signal.

```

for index = 1:1000
    x = sum(sin(),2) + 1e-4 * randn(FrameLength,1);
    z = bpdecim(x);
    sa(z);
end

```

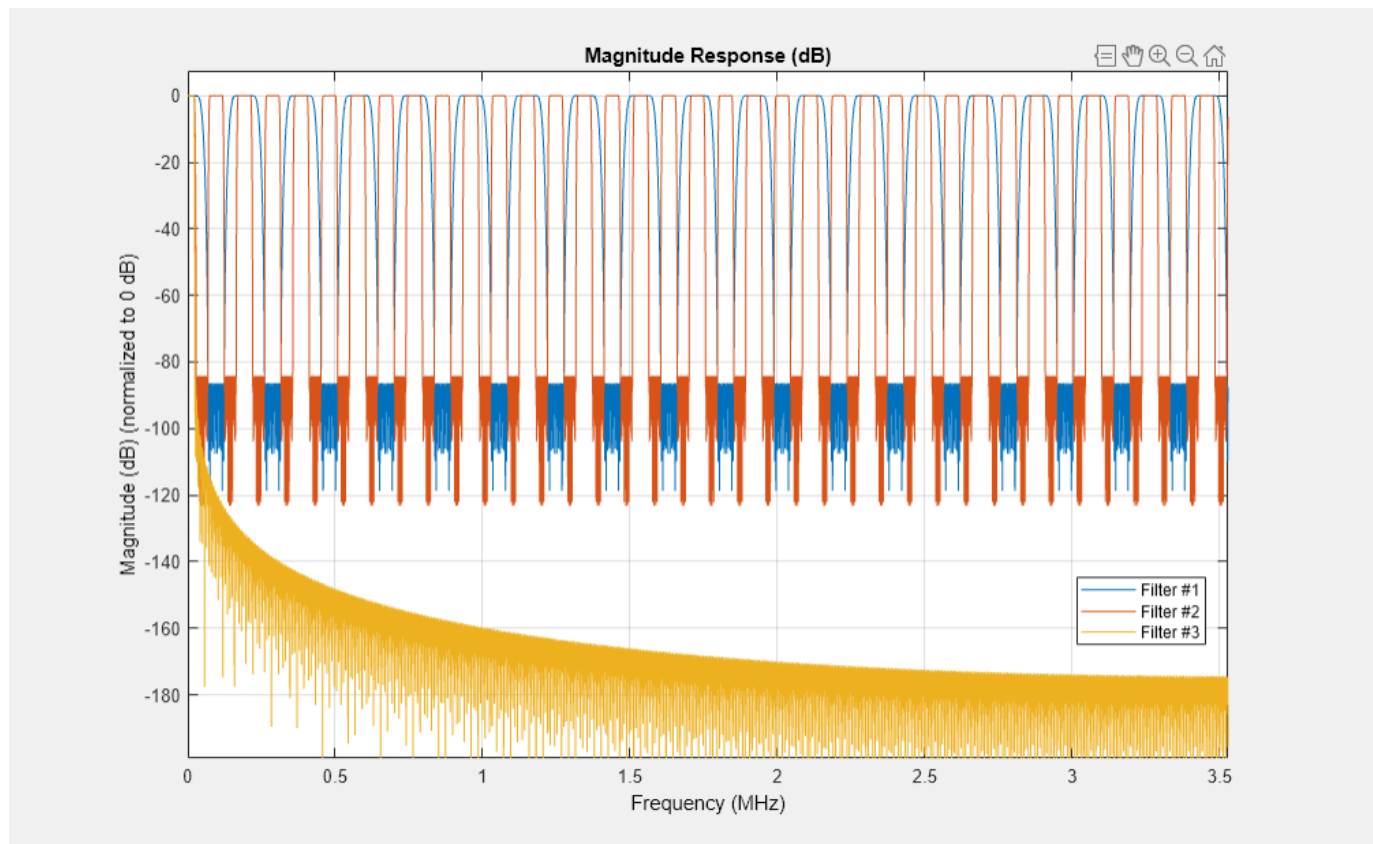


The tones at 2000 Hz and 2125 Hz are passed through the decimator, while the tone at 1625 Hz is filtered out.

### Sample Rate Converter Stages

Create a multistage sample rate converter with default properties, corresponding to the combined three filter stages used to convert from 192 kHz to 44.1 kHz. Visualize the stages.

```
src = dsp.SampleRateConverter;
visualizeFilterStages(src)
```



## Input Arguments

### **sysobj** — Filter System object

filter System object

Filter System object, specified as one of the following:

- `dsp.ComplexBandpassDecimator`
- `dsp.SampleRateConverter`

## Version History

Introduced in R2012a

## See Also

### Functions

`cost` | `freqz` | `info` | `getActualOutputRate` | `getFilters` | `getRateChangeFactors`

### Topics

"Design and Analysis of a Digital Down Converter"

"Digital Up and Down Conversion for Family Radio Service in MATLAB"

“IF Subsampling with Complex Multirate Filters”

## visualizeFilterStages

**Package:** dsp

Display response of digital down converter or digital up converter filter cascade

### Syntax

```
visualizeFilterStages(Conv)
visualizeFilterStages(Conv,'Arithmetic',arithType)
fvt = visualizeFilterStages(Conv)
```

### Description

`visualizeFilterStages(Conv)` plots the magnitude response of the filter stages and the cascade response of a digital down converter or digital up converter, `Conv`. The function plots the response of the filters up to the second CIC null frequency (or to the first when only one CIC null exists).

`visualizeFilterStages(Conv,'Arithmetic',arithType)` specifies the arithmetic type of the filter stages. Set input `arithType` to 'double', 'single', or 'fixed-point'. When the `Conv` object is in an unlocked state, you must specify the arithmetic type. When the `Conv` object is in a locked state, the object ignores the arithmetic input argument.

`fvt = visualizeFilterStages(Conv)` returns the handle to the `FVTool` object.

### Examples

#### Magnitude Response of Digital Down Converter

Plot the magnitude response of the digital down converter using the `fvtool` function and the `visualizeFilterStages` function.

Create a `dsp.DigitalDownConverter` System object with the default settings. Using the `fvtool` function, plot the magnitude response of the overall filter cascade. The `visualizeFilterStages` function in addition plots the magnitude response of the individual filters stages.

```
dwnConv = dsp.DigitalDownConverter

dwnConv =
  dsp.DigitalDownConverter with properties:

    DecimationFactor: 100
    MinimumOrderDesign: true
    Bandwidth: 200000
    StopbandFrequencySource: 'Auto'
    PassbandRipple: 0.1000
    StopbandAttenuation: 60
    Oscillator: 'Sine wave'
    CenterFrequency: 14000000
    SampleRate: 30000000
```

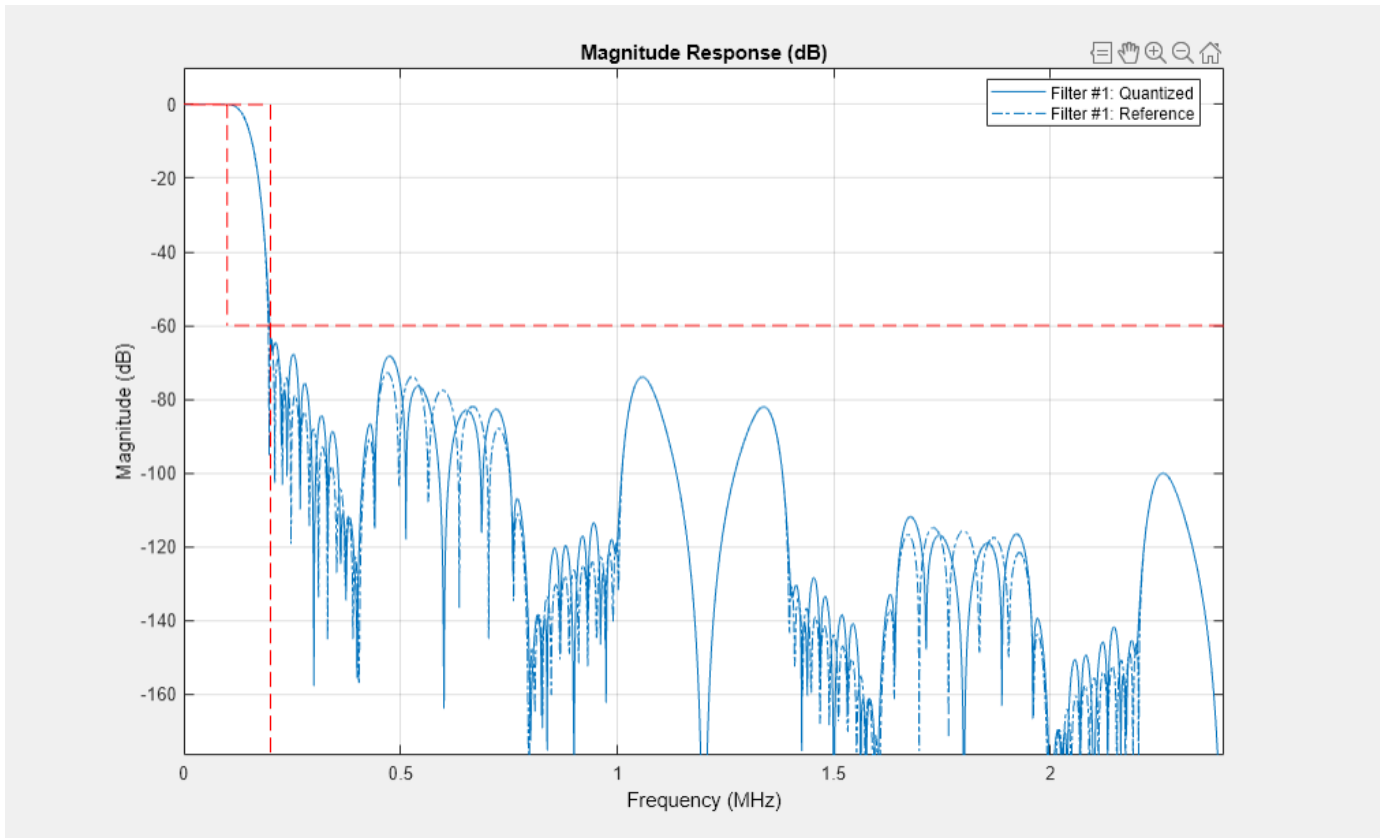


Show all properties

## Using fvtool

If the System object is unlocked, you must specify the filter arithmetic through the 'Arithmetic' input of the fvtool function. If the System object is locked, the arithmetic input is ignored.

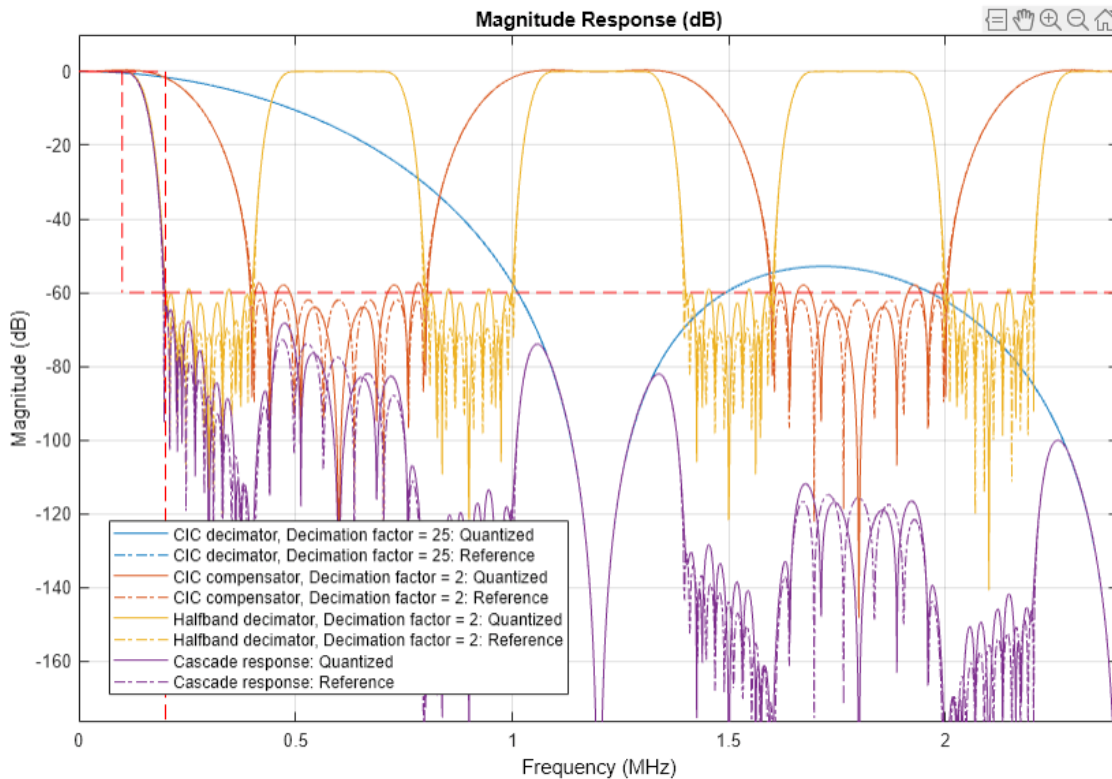
```
fvtool(dwnConv, 'Arithmetic', 'fixed-point')
```



## Using visualizeFilterStages

To view the magnitude response of the individual filter stages, call the visualizeFilterStages function.

```
visualizeFilterStages(dwnConv, 'Arithmetic', 'fixed-point')
```



## Input Arguments

### Conv — Digital down converter or digital up converter

`dsp.DigitalDownConverter` | `dsp.DigitalUpConverter`

Digital down converter or digital up converter, specified as a `dsp.DigitalDownConverter` or `dsp.DigitalUpConverter` System object.

### arithType — Arithmetic type

'double' (default) | 'single' | 'fixed-point'

Arithmetic type of the filter stages, specified as 'double', 'single', or 'fixed-point'. When the Conv object is in an unlocked state, you must specify the arithmetic type. When the Conv object is in a locked state, the object ignores the arithmetic input argument.

## Version History

Introduced in R2012a

## See Also

### Functions

`getInterpolationFactors` | `getDecimationFactors` | `getFilterOrders` | `fvtool` | `groupDelay` | `getFilters`

**Objects**

dsp.DigitalDownConverter | dsp.DigitalUpConverter

## wdf2allpass

Wave Digital Filter to allpass coefficient transformation

### Syntax

```
a = wdf2allpass(w)
A = wdf2allpass(W)
```

### Description

`a = wdf2allpass(w)` accepts a vector of transformed real allpass coefficients, `w`, and returns the conventional allpass polynomial version `a`. `w` is used by allpass filter objects such as `dsp.AllpassFilter`, and `dsp.CoupledAllpassFilter`, with `Structure` set to 'Wave Digital Filter'.

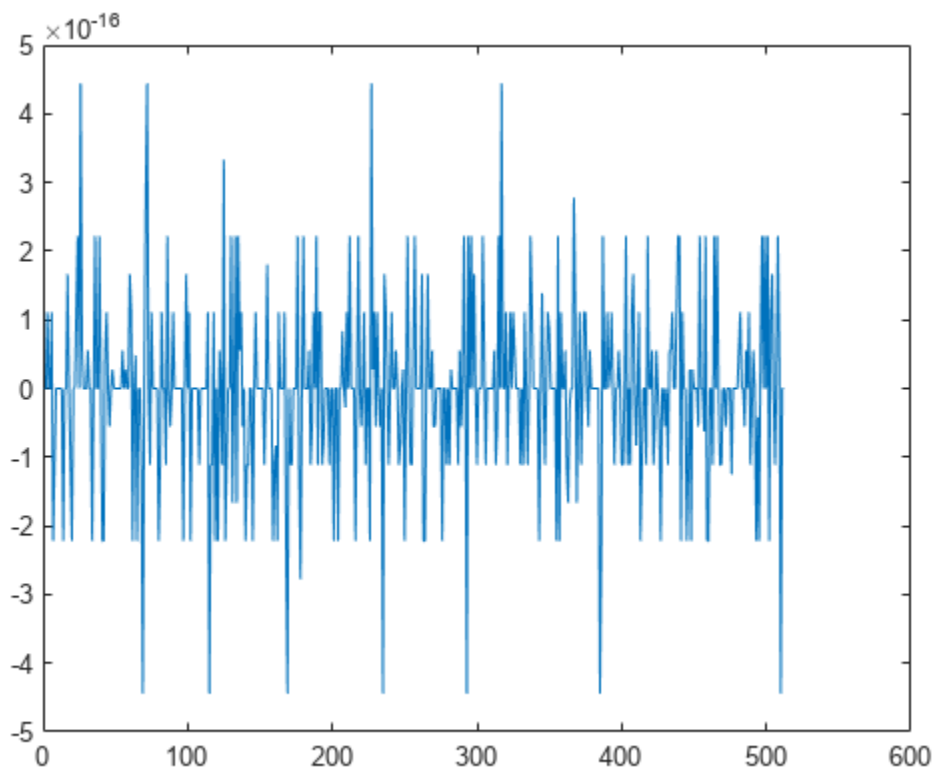
`A = wdf2allpass(W)` accepts the cell array of transformed allpass coefficient vectors `W`. Each cell of `W` contains the transformed real coefficients of a section of a cascade allpass filter. The output `A` is also a cell array, and each cell of `A` contains the conventional polynomial version of the corresponding cell of `W`. `W` is used by allpass filter objects such as `dsp.AllpassFilter` and `dsp.CoupledAllpassFilter`, with `Structure` set to 'Wave Digital Filter'. Every cell of `W` must contain a real vector of length 1, 2, or 4. When the length is 4, the second and fourth components must both be zero. `W` can be a row or column vector of cells while `A` is always returned as column.

### Examples

#### Convert Wave Digital Filter Coefficients to Allpass Polynomial Coefficients

Create a second order allpass filter with wave digital filter coefficients `w = [0.5 0]`. Convert these coefficients into polynomial form using `wdf2allpass`. Assign the polynomial coefficients to an allpass filter using the 'Minimum multiplier' structure. Pass a random input to both these filters and compare the outputs.

```
w = [0.5 0];
allpasswdf = dsp.AllpassFilter('Structure', 'Wave Digital Filter', ...
    'WDFCoefficients', w);
a = wdf2allpass(w);
allpass = dsp.AllpassFilter('AllpassCoefficients', a);
in = randn(512, 1);
outputallpasswdf = allpasswdf(in);
outputallpass = allpass(in);
plot(outputallpasswdf-outputallpass)
```



The difference between the two outputs is very small.

## Input Arguments

### **w** — Transformed Wave Digital Filter allpass coefficients

scalar | vector of real numbers

Numeric vector of transformed Wave Digital Filter allpass coefficients, specified as a real number. *w* can have only length equal to 1, 2, and 4. When the length is 4, the second and fourth components must both be zero. *w* can be a row or a column vector.

Example: `[0.3, -0.2]`

Data Types: `double` | `single`

### **W** — Transformed Wave Digital Filter allpass coefficients

scalar cell | vector of cells

Cascade of allpass filter coefficients in transformed Wave Digital Filter form, specified as a cell vector. Every cell of *W* must contain a real vector of length 1, 2, or 4. When the length is 4, the second and fourth components must both be zero. *W* can be a row or a column vector of cells.

Example: `{[0.3, -0.2]; 0.5}`

## Output Arguments

### **a** — allpass filter coefficients

vector of real numbers

Numeric vector of polynomial allpass coefficients, determined as a numeric row vector.

Data Types: `double` | `single`

### **A** — allpass filter coefficients

vector cell array

Cascade of allpass filter coefficient, determined as a column of cells, each containing a vector of length 1, 2, or 4.

Example: `{0.3 5.0 0.2}`

Data Types: `double` | `single`

## Algorithms

`wdf2allpass` provides the inverse operation of `allpass2wdf`, by transforming the transformed cascade of allpass coefficients  $W$  into their conventional polynomial representation  $A$ . Please refer to the reference page for `allpass2wdf` for more details about the two representations.

$W$  defines a multisection allpass filter, and `wdf2allpass` applies separately to each section, with the same transformation used in the single-section case. In this case, the numeric coefficients vector  $w$  can have order 1, 2, or 4.

The relations between the vector of section coefficients  $a$  and  $w$  respectively depend on the order, as follows:

*for order 1:*

$$a_1 = w_1$$

*for order 2:*

$$a_1 = w_2(1 + w_1)$$

$$a_2 = w_1$$

*for order 4:*

$$a_2 = w_3(1 + w_1)$$

$$a_4 = w_1$$

$$a_1 = a_3 = 0$$

## Version History

Introduced in R2014a

## References

- [1] M. Lutovac, D. Tomic, B. Evans, *Filter Design for Signal Processing using MATLAB and Mathematica*. Prentice Hall, 2001.

**See Also**

`allpass2wdf` | `ca2tf` | `latc2tf` | `dsp.AllpassFilter` | `dsp.CoupledAllpassFilter`

## window

FIR filter using windowed impulse response

### Syntax

```
h = window(d,fcnhdl,fcnarg,'SystemObject',true)
h = window(d,win,'SystemObject',true)
```

### Description

`h = window(d,fcnhdl,fcnarg,'SystemObject',true)` designs a single-rate digital filter System object using the specifications in filter specification object `d`.

`fcnhdl` is a handle to a filter design function that returns a window vector, such as the `hamming` or `blackman` functions. `fcnarg` is an optional argument that returns a window. You pass the function to `window`.

`h = window(d,win,'SystemObject',true)` designs a filter using the vector you supply in `win`. The length of vector `win` must be the same as the impulse response of the filter, which is equal to the filter order plus one.

### Examples

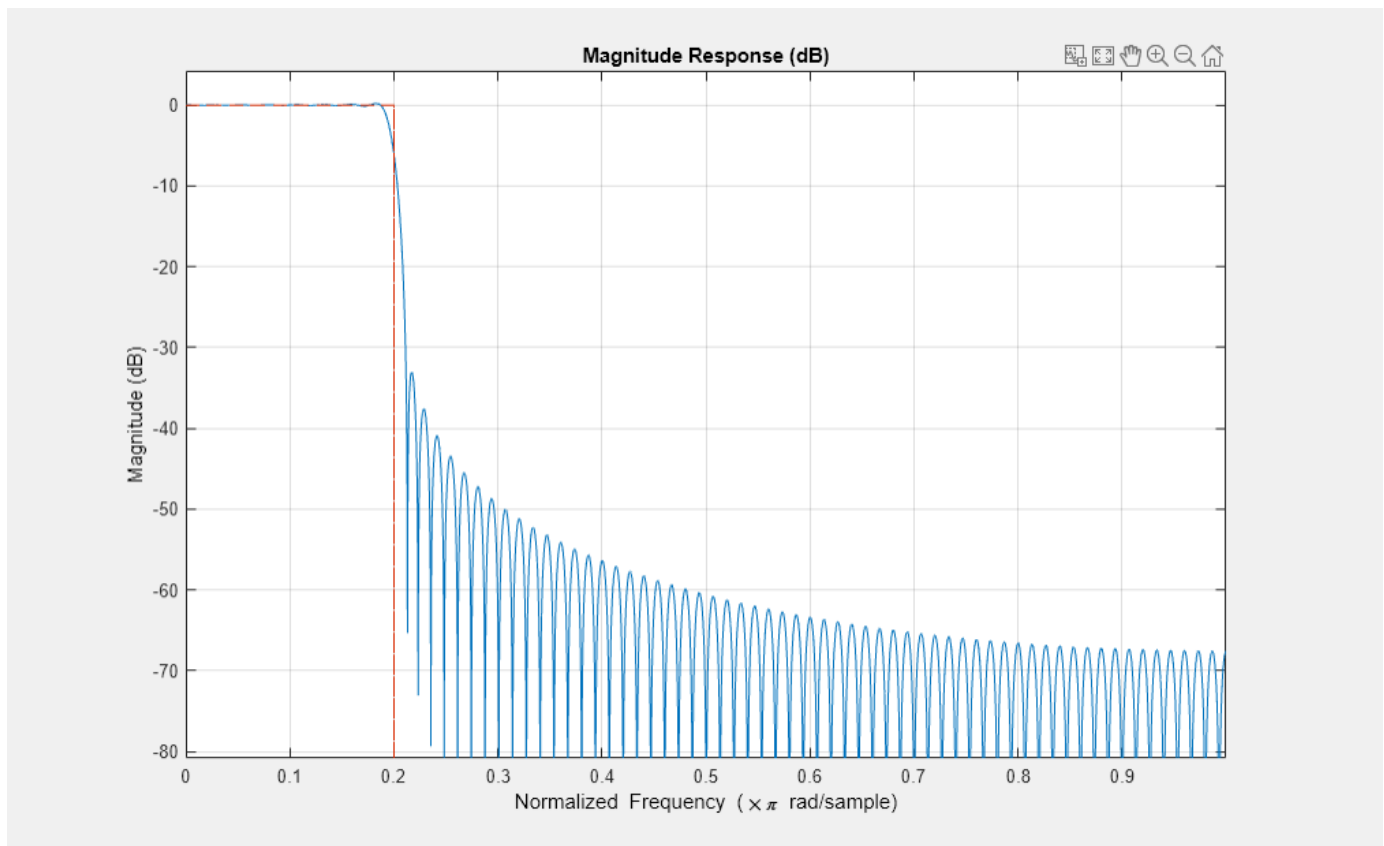
#### Design a Nyquist Filter Using Kaiser Window

This example designs a filter using the two design techniques of specifying a function handle and passing a window vector as an input argument.

Use a window vector provided by the `kaiser` window function to design a Nyquist filter. The window length must be the filter order plus one.

```
d = fdesign.nyquist(5,'n',150);
% Kaiser window with beta parameter 2.5
nyqFilter = window(d,'window',kaiser(151,2.5),'SystemObject',true);
fvtool(nyqFilter)
```





## Version History

Introduced in R2011a

### See Also

firls | kaiserwin

## zerophase

**Package:** dsp

Zero-phase response of discrete-time filter System object

### Syntax

```
[zphase,w] = zerophase(sysobj)
[zphase,w] = zerophase(sysobj,n)
[zphase,w] = zerophase( __ , 'Arithmetic',arithType)
zerophase(sysobj)
```

### Description

`[zphase,w] = zerophase(sysobj)` returns the zero-phase response vector, `zphase`, and the corresponding frequency vector, `w`, in radians/sample of the filter System object based on the current filter coefficients.

`[zphase,w] = zerophase(sysobj,n)` returns the `n`-point zero-phase response vector and the corresponding `n`-point frequency vector in radians/sample of the filter System object.

`[zphase,w] = zerophase( __ , 'Arithmetic',arithType)` analyzes the filter System object based on the arithmetic specified in `arithType` using either of the previous syntaxes.

`zerophase(sysobj)` launches `fvtool` and shows the zero-phase response of the filter System object.

For more input options, see `zerophase` in Signal Processing Toolbox.

### Examples

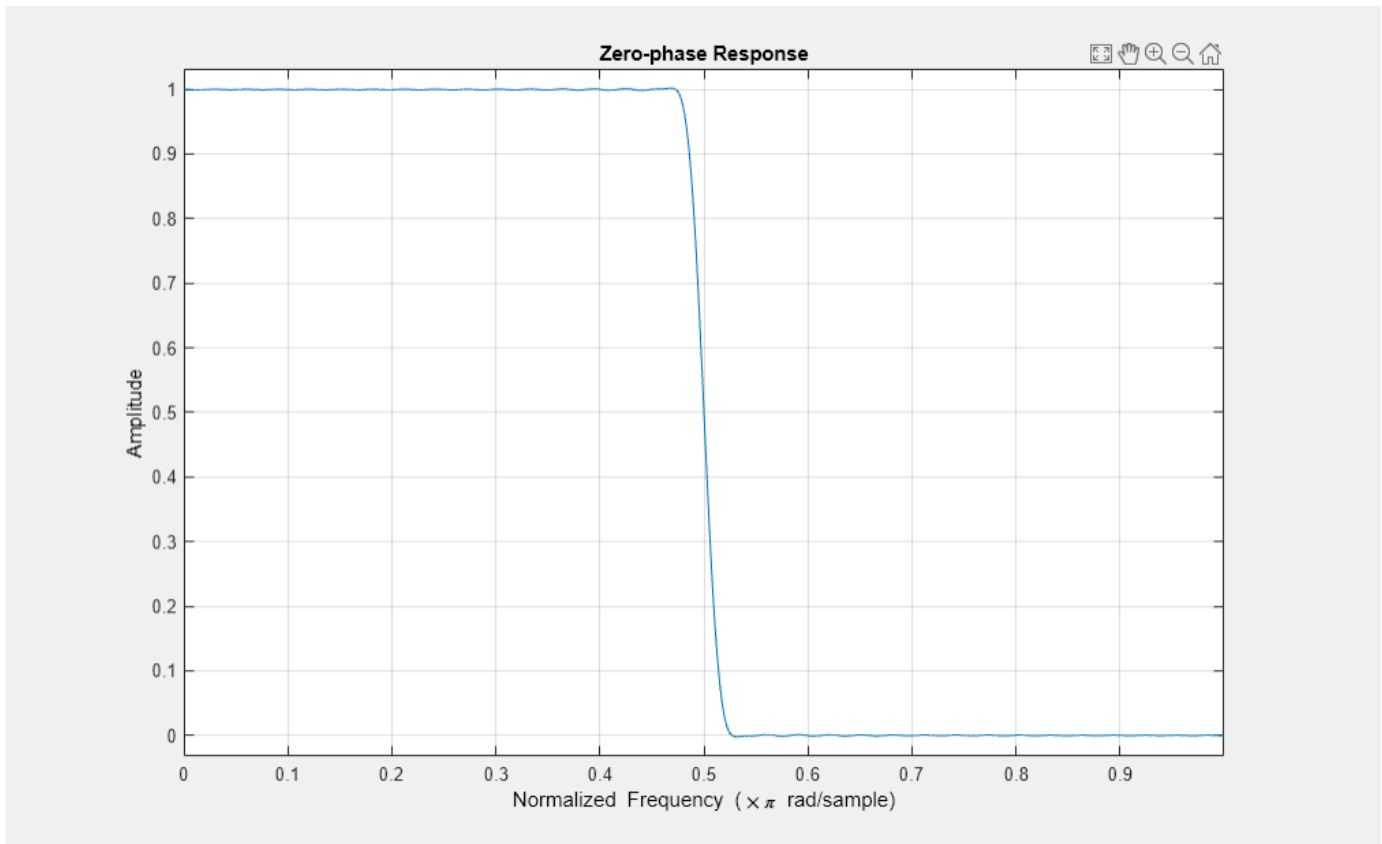
#### Zero-phase Response of Discrete-Time Filter

Create a `dsp.FIRFilter` System object™ and compute the zero-phase response of the filter.

```
Fs = 8000;
Fcutoff = 2000;
FIRfilt = dsp.FIRFilter('Numerator', fir1(130,Fcutoff/(Fs/2)));
```

The `zerophase` function computes the zero-phase response of the filter and displays the response using `fvtool`.

```
zerophase(FIRfilt)
```



## Input Arguments

### **sysobj** — Input filter

filter System object

Input filter, specified as one of the following filter System objects:

- `dsp.AllpassFilter`
- `dsp.AllpoleFilter`
- `dsp.BiquadFilter`
- `dsp.Channelizer`
- `dsp.CICCompensationDecimator`
- `dsp.CICCompensationInterpolator`
- `dsp.CICDecimator`
- `dsp.CICInterpolator`
- `dsp.CoupledAllpassFilter`
- `dsp.Differentiator`
- `dsp.FarrowRateConverter`
- `dsp.FilterCascade`

- `dsp.FIRDecimator`
- `dsp.FIRFilter`
- `dsp.FIRHalfbandDecimator`
- `dsp.FIRHalfbandInterpolator`
- `dsp.FIRInterpolator`
- `dsp.FIRRateConverter`
- `dsp.FourthOrderSectionFilter`
- `dsp.HighpassFilter`
- `dsp.IIRFilter`
- `dsp.IIRHalfbandDecimator`
- `dsp.IIRHalfbandInterpolator`
- `dsp.LowpassFilter`
- `dsp.NotchPeakFilter`
- `dsp.SOSFilter`
- `dsp.VariableBandwidthFIRFilter`
- `dsp.VariableBandwidthIIRFilter`

**n — Number of points over which zero-phase response is evaluated**

8192 (default) | positive integer

Number of points over which the zero-phase response is evaluated around the upper half of the unit circle. For an FIR filter where *n* is a power of two, the computation is done faster using FFTs.

Data Types: double

**arithType — Arithmetic type**

'double' (default) | 'single' | 'Fixed'

Arithmetic used in the filter analysis, specified as 'double', 'single', or 'Fixed'. When the arithmetic input is not specified and the filter System object is unlocked, the analysis tool assumes a double-precision filter. When the arithmetic input is not specified and the System object is locked, the function performs the analysis based on the data type of the locked input.

The 'Fixed' value applies to filter System objects with fixed-point properties only.

When the 'Arithmetic' input argument is specified as 'Fixed' and the filter object has the data type of the coefficients set to 'Same word length as input', the arithmetic analysis depends on whether the System object is unlocked or locked.

- unlocked -- The analysis object function cannot determine the coefficients data type. The function assumes that the coefficients data type is signed, has a 16-bit word length, and is auto scaled. The function performs fixed-point analysis based on this assumption.
- locked -- When the input data type is 'double' or 'single', the analysis object function cannot determine the coefficients data type. The function assumes that the data type of the coefficients is signed, has a 16-bit word length, and is auto scaled. The function performs fixed-point analysis based on this assumption.

To check if the System object is locked or unlocked, use the `isLocked` function.

When the arithmetic input is specified as 'Fixed' and the filter object has the data type of the coefficients set to a custom numeric type, the object function performs fixed-point analysis based on the custom numeric data type.

## Output Arguments

### **zphase** — Zero-phase response vector

vector

Zero-phase response vector, evaluated at  $n$  equally-spaced points around the upper half of the unit circle, returned as a vector. If  $n$  is not specified, the function uses a default value of 8192.

Data Types: double

### **w** — Frequency vector

vector

Frequency vector of length  $n$  in radians/sample.  $w$  consists of  $n$  equally-spaced points around the upper half of the unit circle (from 0 to  $\pi$  radians/sample). If  $n$  is not specified, the function uses a default value of 8192.

Data Types: double

## Version History

Introduced in R2011a

## See Also

### Functions

freqz | fvtool | grpdelay | impz | phasez | zplane | zerophase

## zpk

**Package:** dsp

Zero-pole-gain conversion of discrete-time filter System object

### Syntax

```
[z,p,k] = zpk(sysobj)
[z,p,k] = zpk(sysobj,'Arithmetic',arithType)
```

### Description

`[z,p,k] = zpk(sysobj)` returns the zeros, poles, and gain corresponding to the filter System object in vector `z`, vector `p`, and scalar `k`, respectively.

`[z,p,k] = zpk(sysobj,'Arithmetic',arithType)` analyzes the filter System object based on the arithmetic specified in `arithType`.

For more input options, see `zpk` in Signal Processing Toolbox.

### Examples

#### Highpass Filter in Zero-Pole-Gain Form

Compute the zero-pole-gain form of the highpass filter using the `zpk` function.

Design a minimum order highpass FIR filter with a passband frequency of 75 kHz and passband ripple of 0.2 dB. Specify a sample rate of 200 kHz.

```
hFilt = dsp.HighpassFilter('PassbandFrequency',75e3,'PassbandRipple',0.2,'SampleRate',200e3)
```

```
hFilt =
  dsp.HighpassFilter with properties:
```

```
      FilterType: 'FIR'
  DesignForMinimumOrder: true
      StopbandFrequency: 8000
      PassbandFrequency: 75000
  StopbandAttenuation: 80
      PassbandRipple: 0.2000
  NormalizedFrequency: false
      SampleRate: 200000
```

Show all properties

Find the zeros, poles, and the gain of the designed filter using the `zpk` function.

```
[z,p,k] = zpk(hFilt)
```

```
z = 8×1 complex
```

```
17.2236 + 0.0000i
-3.0709 + 0.0000i
 0.9732 + 0.2300i
 0.9732 - 0.2300i
 0.9954 + 0.0957i
 0.9954 - 0.0957i
-0.3256 + 0.0000i
 0.0581 + 0.0000i
```

```
p = 8×1
```

```
0
0
0
0
0
0
0
0
```

```
k = -0.0023
```

## Input Arguments

### **sysobj** — Input filter

filter System object

Input filter, specified as one of the following filter System objects:

- `dsp.AllpassFilter`
- `dsp.AllpoleFilter`
- `dsp.BiquadFilter`
- `dsp.CICCompensationDecimator`
- `dsp.CICCompensationInterpolator`
- `dsp.CICDecimator`
- `dsp.CICInterpolator`
- `dsp.CoupledAllpassFilter`
- `dsp.Differentiator`
- `dsp.FarrowRateConverter`
- `dsp.FilterCascade`
- `dsp.FIRDecimator`
- `dsp.FIRFilter`
- `dsp.FIRHalfbandDecimator`
- `dsp.FIRHalfbandInterpolator`
- `dsp.FIRInterpolator`
- `dsp.FIRRateConverter`

- `dsp.FourthOrderSectionFilter`
- `dsp.HighpassFilter`
- `dsp.IIRFilter`
- `dsp.IIRHalfbandDecimator`
- `dsp.IIRHalfbandInterpolator`
- `dsp.LowpassFilter`
- `dsp.NotchPeakFilter`
- `dsp.VariableBandwidthFIRFilter`
- `dsp.VariableBandwidthIIRFilter`

### **arithType – Arithmetic type**

'double' (default) | 'single' | 'Fixed'

Arithmetic used in the filter analysis, specified as 'double', 'single', or 'Fixed'. When the arithmetic input is not specified and the filter System object is unlocked, the analysis tool assumes a double-precision filter. When the arithmetic input is not specified and the System object is locked, the function performs the analysis based on the data type of the locked input.

The 'Fixed' value applies to filter System objects with fixed-point properties only.

When the 'Arithmetic' input argument is specified as 'Fixed' and the filter object has the data type of the coefficients set to 'Same word length as input', the arithmetic analysis depends on whether the System object is unlocked or locked.

- unlocked -- The analysis object function cannot determine the coefficients data type. The function assumes that the coefficients data type is signed, has a 16-bit word length, and is auto scaled. The function performs fixed-point analysis based on this assumption.
- locked -- When the input data type is 'double' or 'single', the analysis object function cannot determine the coefficients data type. The function assumes that the data type of the coefficients is signed, has a 16-bit word length, and is auto scaled. The function performs fixed-point analysis based on this assumption.

To check if the System object is locked or unlocked, use the `isLocked` function.

When the arithmetic input is specified as 'Fixed' and the filter object has the data type of the coefficients set to a custom numeric type, the object function performs fixed-point analysis based on the custom numeric data type.

## **Output Arguments**

### **z – Zeros**

column vector

Zeros of the filter, returned as a column vector.

Data Types: `double`

### **p – Poles**

column vector

Poles of the filter, returned as a column vector.



Data Types: double  
Complex Number Support: Yes

**k — Gain**

real scalar

Gain of the filter, returned as a real scalar.

Data Types: double

## Version History

Introduced in R2011a

### See Also

**Functions**

zpk

**Topics**

“Analysis Methods for Filter System Objects” on page 3-2

## zpkbpc2bpc

Zero-pole-gain complex bandpass frequency transformation

### Syntax

```
[Z2,P2,K2,AllpassNum,AllpassDen] = zpkbpc2bpc(Z,P,K,Wo,Wt)
```

### Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpkbpc2bpc(Z,P,K,Wo,Wt)` returns zeros,  $Z_2$ , poles,  $P_2$ , and gain factor,  $K_2$ , of the target filter transformed from the complex bandpass prototype by applying a first-order complex bandpass to complex bandpass frequency transformation.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The original lowpass filter is given with zeros,  $Z$ , poles,  $P$ , and gain factor,  $K$ .

This transformation effectively places two features of an original filter, located at frequencies  $W_{o1}$  and  $W_{o2}$ , at the required target frequency locations,  $W_{t1}$ , and  $W_{t2}$  respectively. It is assumed that  $W_{t2}$  is greater than  $W_{t1}$ . In most of the cases the features selected for the transformation are the band edges of the filter passbands. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter,  $F_1$  and  $F_2$ , with  $F_1$  preceding  $F_2$ . Feature  $F_1$  will still precede  $F_2$  after the transformation. However, the distance between  $F_1$  and  $F_2$  will not be the same before and after the transformation.

This transformation can also be used for transforming other types of filters; e.g., complex notch filters or resonators can be repositioned at two distinct desired frequencies at any place around the unit circle; e.g., in the adaptive system.

### Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
```

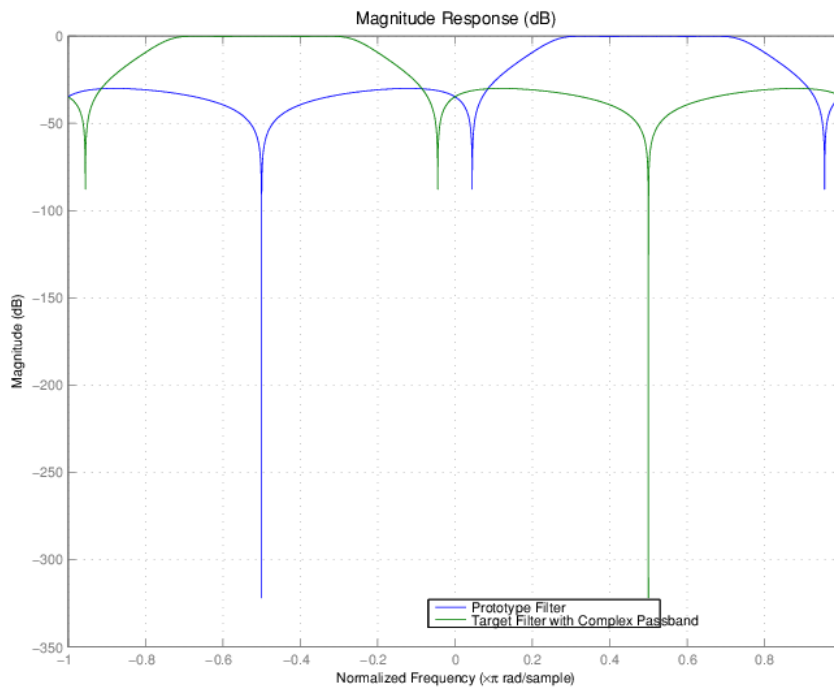
Create a complex passband from 0.25 to 0.75:

```
[b, a] = iirlp2bpc(b,a,0.5,[0.25,0.75]);
z = roots(b);
p = roots(a);
k = b(1);
[z2,p2,k2] = zpkbpc2bpc(z,p,k,[0.25, 0.75],[-0.75, -0.25]);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

Comparing the filters in FVTool shows the example results. Use the features in FVTool to check the filter coefficients, or other filter analyses.



## Arguments

| Variable     | Description   |
|--------------|---|
| $Z$          | Zeros of the prototype lowpass filter                       |
| $P$          | Poles of the prototype lowpass filter                       |
| $K$          | Gain factor of the prototype lowpass filter                 |
| $\omega_0$   | Frequency value to be transformed from the prototype filter |
| $\omega_t$   | Desired frequency location in the transformed target filter |
| $Z_2$        | Zeros of the target filter                                  |
| $P_2$        | Poles of the target filter                                  |
| $K_2$        | Gain factor of the target filter                            |
| $AllpassNum$ | Numerator of the mapping filter                             |
| $AllpassDen$ | Denominator of the mapping filter                           |

Frequencies must be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

## Version History

Introduced in R2011a

## See Also

zpkfttransf | allpassbpc2bpc | iirbpc2bpc

## zpkftransf

Zero-pole-gain frequency transformation

### Syntax

```
[Z2,P2,K2] = zpkftransf(Z,P,K,AllpassNum,AllpassDen)
```

### Description

`[Z2,P2,K2] = zpkftransf(Z,P,K,AllpassNum,AllpassDen)` returns zeros,  $Z_2$ , poles,  $P_2$ , and gain factor,  $K_2$ , of the transformed lowpass digital filter. The prototype lowpass filter is given with zeros,  $Z$ , poles,  $P$ , and gain factor,  $K$ . If `AllpassDen` is not specified it will default to 1. If neither `AllpassNum` nor `AllpassDen` is specified, then the function returns the input filter.

### Examples

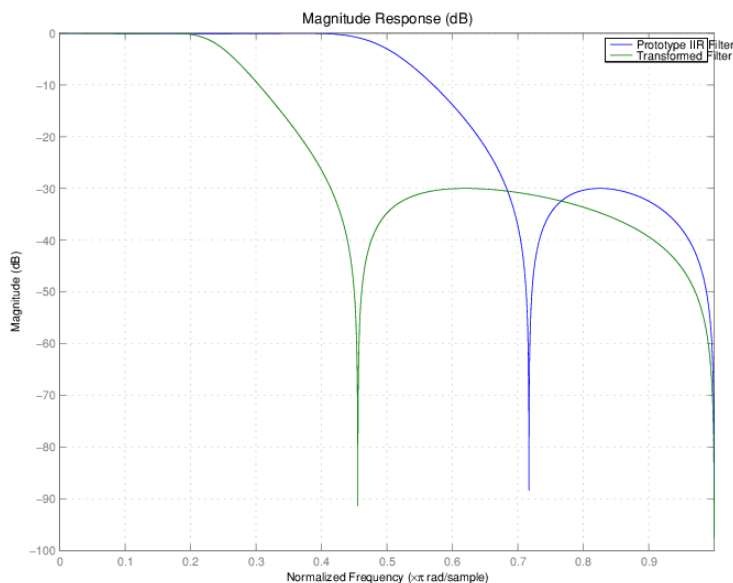
Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
[AlpNum, AlpDen] = allpasslp2lp(0.5, 0.25);
[z2, p2, k2] = zpkftransf(roots(b),roots(a),b(1),AlpNum,AlpDen);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

After transforming the filter, you get the response shown in the figure, where the passband has been shifted towards zero.



## Arguments

| Variable | Description                                 |
|----------|---|
| $Z$      | Zeros of the prototype lowpass filter       |
| $P$      | Poles of the prototype lowpass filter       |
| $K$      | Gain factor of the prototype lowpass filter |
| $FTFNum$ | Numerator of the mapping filter             |
| $FTFDen$ | Denominator of the mapping filter           |
| $Z2$     | Zeros of the target filter                  |
| $P2$     | Poles of the target filter                  |
| $K2$     | Gain factor of the target filter            |

## Version History

Introduced in R2011a

### See Also

iirftransf

## zpklp2bp

Zero-pole-gain lowpass to bandpass frequency transformation

### Syntax

```
[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2bp(Z,P,K,Wo,Wt)
```

### Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2bp(Z,P,K,Wo,Wt)` returns zeros,  $Z_2$ , poles,  $P_2$ , and gain factor,  $K_2$ , of the target filter transformed from the real lowpass prototype by applying a second-order real lowpass to real bandpass frequency mapping.

It also returns the numerator, `AllpassNum`, and the denominator `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with zeros,  $Z$ , poles,  $P$ , and gain factor,  $K$ .

This transformation effectively places one feature of an original filter, located at frequency  $-W_o$ , at the required target frequency location,  $W_{t1}$ , and the second feature, originally at  $+W_o$ , at the new location,  $W_{t2}$ . It is assumed that  $W_{t2}$  is greater than  $W_{t1}$ . This transformation implements the "DC Mobility," which means that the Nyquist feature stays at Nyquist, but the DC feature moves to a location dependent on the selection of  $W_t$ .

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter,  $F_1$  and  $F_2$ , with  $F_1$  preceding  $F_2$ . Feature  $F_1$  will still precede  $F_2$  after the transformation. However, the distance between  $F_1$  and  $F_2$  will not be the same before and after the transformation.

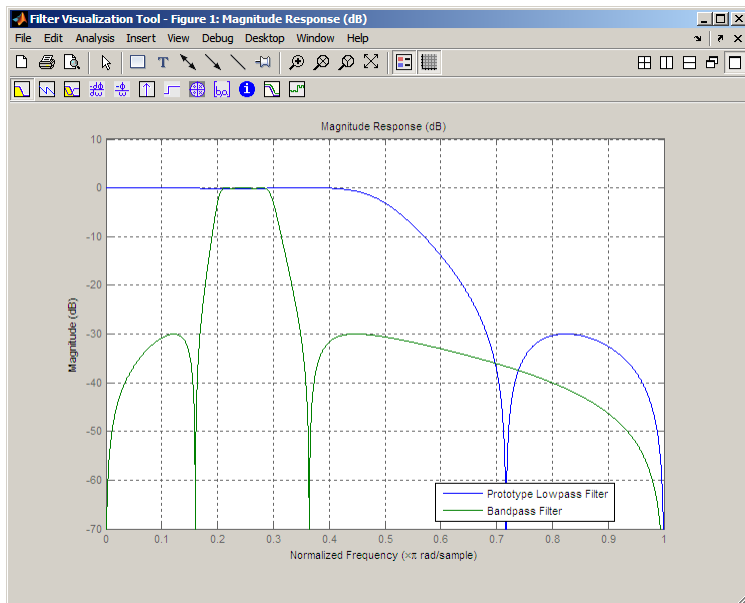
Choice of the feature subject to the lowpass to bandpass transformation is not restricted only to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Real lowpass to bandpass transformation can also be used for transforming other types of filters; e.g., real notch filters or resonators can be easily doubled and positioned at two distinct, desired frequencies.

### Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[B,A] = ellip(3,0.1,30,0.409);
Z = roots(B);
P = roots(A);
K = B(1);
[Z2,P2,K2] = zpklp2bp(Z,P,K, 0.5, [0.2 0.3]);
hfvtool(B,A,K2*poly(Z2),poly(P2));
legend(hfvtool,'Prototype Lowpass Filter', 'Bandpass Filter');
axis([0 1 -70 10]);
```



## Arguments

| Variable     | Description   |
|--------------|---|
| $Z$          | Zeros of the prototype lowpass filter                       |
| $P$          | Poles of the prototype lowpass filter                       |
| $K$          | Gain factor of the prototype lowpass filter                 |
| $\omega_0$   | Frequency value to be transformed from the prototype filter |
| $\omega_t$   | Desired frequency location in the transformed target filter |
| $Z_2$        | Zeros of the target filter                                  |
| $P_2$        | Poles of the target filter                                  |
| $K_2$        | Gain factor of the target filter                            |
| $AllpassNum$ | Numerator of the mapping filter                             |
| $AllpassDen$ | Denominator of the mapping filter                           |

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

## Version History

Introduced in R2011a

## References

Constantinides, A.G., "Spectral transformations for digital filters," *IEE Proceedings*, vol. 117, no. 8, pp. 1585-1590, August 1970.

Nowrouzian, B. and A.G. Constantinides, "Prototype reference transfer function parameters in the discrete-time frequency transformations," *Proceedings 33rd Midwest Symposium on Circuits and Systems*, Calgary, Canada, vol. 2, pp. 1078-1082, August 1990.

Nowrouzian, B. and L.T. Bruton, "Closed-form solutions for discrete-time elliptic transfer functions," *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, vol. 2, pp. 784-787, 1992.

Constantinides, A.G., "Design of bandpass digital filters," *IEEE Proceedings*, vol. 1, pp. 1129-1231, June 1969.

**See Also**

`zpkftransf` | `allpasslp2bp` | `iirlp2bp`



## zpklp2bpc

Zero-pole-gain lowpass to complex bandpass frequency transformation

### Syntax

```
[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2bpc(Z,P,K,Wo,Wt)
```

### Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2bpc(Z,P,K,Wo,Wt)` returns zeros,  $Z_2$ , poles,  $P_2$ , and gain factor,  $K_2$ , of the target filter transformed from the real lowpass prototype by applying a first-order real lowpass to complex bandpass frequency transformation.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with zeros,  $Z$ , poles,  $P$ , and gain factor,  $K$ .

This transformation effectively places one feature of an original filter, located at frequency  $-W_o$ , at the required target frequency location,  $W_{t1}$ , and the second feature, originally at  $+W_o$ , at the new location,  $W_{t2}$ . It is assumed that  $W_{t2}$  is greater than  $W_{t1}$ .

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter,  $F_1$  and  $F_2$ , with  $F_1$  preceding  $F_2$ . Feature  $F_1$  will still precede  $F_2$  after the transformation. However, the distance between  $F_1$  and  $F_2$  will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to bandpass transformation is not restricted only to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Lowpass to bandpass transformation can also be used for transforming other types of filters; e.g., real notch filters or resonators can be doubled and positioned at two distinct desired frequencies at any place around the unit circle forming a pair of complex notches/resonators. This transformation can be used for designing bandpass filters for radio receivers from the high-quality prototype lowpass filter.

### Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
z = roots(b);
p = roots(a);
k = b(1);
[z2,p2,k2] = zpklp2bpc(z, p, k, 0.5, [0.2 0.3]);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

## Arguments

| Variable     | Description  |
|--------------|--|
| $Z$          | Zeros of the prototype lowpass filter  |
| $P$          | Poles of the prototype lowpass filter  |
| $K$          | Gain factor of the prototype lowpass filter  |
| $W_o$        | Frequency value to be transformed from the prototype filter. It should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.   |
| $W_t$        | Desired frequency locations in the transformed target filter. They should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate. |
| $Z_2$        | Zeros of the target filter   |
| $P_2$        | Poles of the target filter   |
| $K_2$        | Gain factor of the target filter   |
| $AllpassNum$ | Numerator of the mapping filter  |
| $AllpassDen$ | Denominator of the mapping filter  |

## Version History

Introduced in R2011a

### See Also

`zpkftransf` | `allpasslp2bpc` | `iirlp2bpc`

## zpklp2bs

Zero-pole-gain lowpass to bandstop frequency transformation

### Syntax

```
[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2bs(Z,P,K,Wo,Wt)
```

### Description

[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2bs(Z,P,K,Wo,Wt) returns zeros,  $Z_2$ , poles,  $P_2$ , and gain factor,  $K_2$ , of the target filter transformed from the real lowpass prototype by applying a second-order real lowpass to real bandstop frequency mapping.

It also returns the numerator, AllpassNum, and the denominator, AllpassDen, of the allpass mapping filter. The prototype lowpass filter is given with zeros, Z, poles, P, and gain factor, K.

This transformation effectively places one feature of an original filter, located at frequency  $-W_o$ , at the required target frequency location,  $W_{t1}$ , and the second feature, originally at  $+W_o$ , at the new location,  $W_{t2}$ . It is assumed that  $W_{t2}$  is greater than  $W_{t1}$ . This transformation implements the "Nyquist Mobility," which means that the DC feature stays at DC, but the Nyquist feature moves to a location dependent on the selection of  $W_o$  and  $W_t$ s.

Relative positions of other features of an original filter change in the target filter. This means that it is possible to select two features of an original filter,  $F_1$  and  $F_2$ , with  $F_1$  preceding  $F_2$ . After the transformation feature  $F_2$  will precede  $F_1$  in the target filter. However, the distance between  $F_1$  and  $F_2$  will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to bandstop transformation is not restricted only to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

### Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
z = roots(b);
p = roots(a);
k = b(1);
[z2,p2,k2] = zpklp2bs(z, p, k, 0.5, [0.2 0.3]);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

### Arguments

| Variable | Description                           |
|----------|---------------------------------------|
| Z        | Zeros of the prototype lowpass filter |

| Variable     | Description   |
|--------------|---|
| $P$          | Poles of the prototype lowpass filter                       |
| $K$          | Gain factor of the prototype lowpass filter                 |
| $W_o$        | Frequency value to be transformed from the prototype filter |
| $W_t$        | Desired frequency location in the transformed target filter |
| $Z_2$        | Zeros of the target filter                                  |
| $P_2$        | Poles of the target filter                                  |
| $K_2$        | Gain factor of the target filter                            |
| $AllpassNum$ | Numerator of the mapping filter                             |
| $AllpassDen$ | Denominator of the mapping filter                           |

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

## Version History

Introduced in R2011a

## References

Constantinides, A.G., "Spectral transformations for digital filters," *IEEE Proceedings*, vol. 117, no. 8, pp. 1585-1590, August 1970.

Nowrouzian, B. and A.G. Constantinides, "Prototype reference transfer function parameters in the discrete-time frequency transformations," *Proceedings 33rd Midwest Symposium on Circuits and Systems*, Calgary, Canada, vol. 2, pp. 1078-1082, August 1990.

Nowrouzian, B. and L.T. Bruton, "Closed-form solutions for discrete-time elliptic transfer functions," *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, vol. 2, pp. 784-787, 1992.

Constantinides, A.G., "Design of bandpass digital filters," *IEEE Proceedings*, vol. 1, pp. 1129-1231, June 1969.

## See Also

zpkftransf | allpasslp2bs | iirlp2bs

## zpklp2bsc

Zero-pole-gain lowpass to complex bandstop frequency transformation

### Syntax

```
[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2bsc(Z,P,K,Wo,Wt)
```

### Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2bsc(Z,P,K,Wo,Wt)` returns zeros,  $Z_2$ , poles,  $P_2$ , and gain factor,  $K_2$ , of the target filter transformed from the real lowpass prototype by applying a first-order real lowpass to complex bandstop frequency transformation.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with zeros,  $Z$ , poles,  $P$ , and gain factor,  $K$ .

This transformation effectively places one feature of an original filter, located at frequency  $-W_o$ , at the required target frequency location,  $W_{t1}$ , and the second feature, originally at  $+W_o$ , at the new location,  $W_{t2}$ . It is assumed that  $W_{t2}$  is greater than  $W_{t1}$ . Additionally the transformation swaps passbands with stopbands in the target filter.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter,  $F_1$  and  $F_2$ , with  $F_1$  preceding  $F_2$ . Feature  $F_1$  will still precede  $F_2$  after the transformation. However, the distance between  $F_1$  and  $F_2$  will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to bandstop transformation is not restricted only to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Lowpass to bandpass transformation can also be used for transforming other types of filters; e.g., real notch filters or resonators can be doubled and positioned at two distinct desired frequencies at any place around the unit circle forming a pair of complex notches/resonators.

### Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
z = roots(b);
p = roots(a);
k = b(1);
[z2,p2,k2] = zpklp2bsc(z, p, k, 0.5, [0.2, 0.3]);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

## Arguments

| Variable     | Description   |
|--------------|---|
| $Z$          | Zeros of the prototype lowpass filter   |
| $P$          | Poles of the prototype lowpass filter   |
| $K$          | Gain factor of the prototype lowpass filter   |
| $W_o$        | Frequency value to be transformed from the prototype filter. It should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.     |
| $W_t$        | Desired frequency locations in the transformed target filter. They should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate. |
| $Z_2$        | Zeros of the target filter  |
| $P_2$        | Poles of the target filter  |
| $K_2$        | Gain factor of the target filter  |
| $AllpassNum$ | Numerator of the mapping filter   |
| $AllpassDen$ | Denominator of the mapping filter   |

## Version History

Introduced in R2011a

### See Also

`zpkftransf` | `allpasslp2bsc` | `iirlp2bsc`

## zpklp2hp

Zero-pole-gain lowpass to highpass frequency transformation

### Syntax

```
[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2hp(Z,P,K,Wo,Wt)
```

### Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2hp(Z,P,K,Wo,Wt)` returns zeros,  $Z_2$ , poles,  $P_2$ , and gain factor,  $K_2$ , of the target filter transformed from the real lowpass prototype by applying a first-order real lowpass to real highpass frequency mapping. This transformation effectively places one feature of an original filter, located at frequency  $W_o$ , at the required target frequency location,  $W_t$ , at the same time rotating the whole frequency response by half of the sampling frequency. Result is that the DC and Nyquist features swap places.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with zeros,  $Z$ , poles,  $P$ , and the gain factor,  $K$ .

Relative positions of other features of an original filter change in the target filter. This means that it is possible to select two features of an original filter,  $F_1$  and  $F_2$ , with  $F_1$  preceding  $F_2$ . After the transformation feature  $F_2$  will precede  $F_1$  in the target filter. However, the distance between  $F_1$  and  $F_2$  will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to highpass transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, or the deep minimum in the stopband, or other ones.

Lowpass to highpass transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can change their position in a simple way without designing them again.

### Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
z = roots(b);
p = roots(a);
k = b(1);
[z2,p2,k2] = zpklp2hp(z, p, k, 0.5, 0.25);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

### Arguments

| Variable | Description                           |
|----------|---------------------------------------|
| $Z$      | Zeros of the prototype lowpass filter |

| Variable     | Description   |
|--------------|---|
| $P$          | Poles of the prototype lowpass filter                       |
| $K$          | Gain factor of the prototype lowpass filter                 |
| $W_o$        | Frequency value to be transformed from the prototype filter |
| $W_t$        | Desired frequency location in the transformed target filter |
| $Z_2$        | Zeros of the target filter                                  |
| $P_2$        | Poles of the target filter                                  |
| $K_2$        | Gain factor of the target filter                            |
| $AllpassNum$ | Numerator of the mapping filter                             |
| $AllpassDen$ | Denominator of the mapping filter                           |

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

## Version History

Introduced in R2011a

## References

Constantinides, A.G., "Spectral transformations for digital filters," *IEE Proceedings*, vol. 117, no. 8, pp. 1585-1590, August 1970.

Nowrouzian, B. and A.G. Constantinides, "Prototype reference transfer function parameters in the discrete-time frequency transformations," *Proceedings 33rd Midwest Symposium on Circuits and Systems*, Calgary, Canada, vol. 2, pp. 1078-1082, August 1990.

Nowrouzian, B. and L.T. Bruton, "Closed-form solutions for discrete-time elliptic transfer functions," *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, vol. 2, pp. 784-787, 1992.

Constantinides, A.G., "Frequency transformations for digital filters," *Electronics Letters*, vol. 3, no. 11, pp. 487-489, November 1967.

## See Also

zpkftransf | allpasslp2hp | iirlp2hp



## zpklp2lp

Zero-pole-gain lowpass to lowpass frequency transformation

### Syntax

```
[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2lp(Z,P,K,Wo,Wt)
```

### Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2lp(Z,P,K,Wo,Wt)` returns zeros,  $Z_2$ , poles,  $P_2$ , and gain factor,  $K_2$ , of the target filter transformed from the real lowpass prototype by applying a first-order real lowpass to real lowpass frequency mapping. This transformation effectively places one feature of an original filter, located at frequency  $W_o$ , at the required target frequency location,  $W_t$ .

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with zeros,  $Z$ , poles,  $P$ , and gain factor,  $K$ .

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter,  $F_1$  and  $F_2$ , with  $F_1$  preceding  $F_2$ . Feature  $F_1$  will still precede  $F_2$  after the transformation. However, the distance between  $F_1$  and  $F_2$  will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to lowpass transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Lowpass to lowpass transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can change their position in a simple way without designing them again.

### Examples

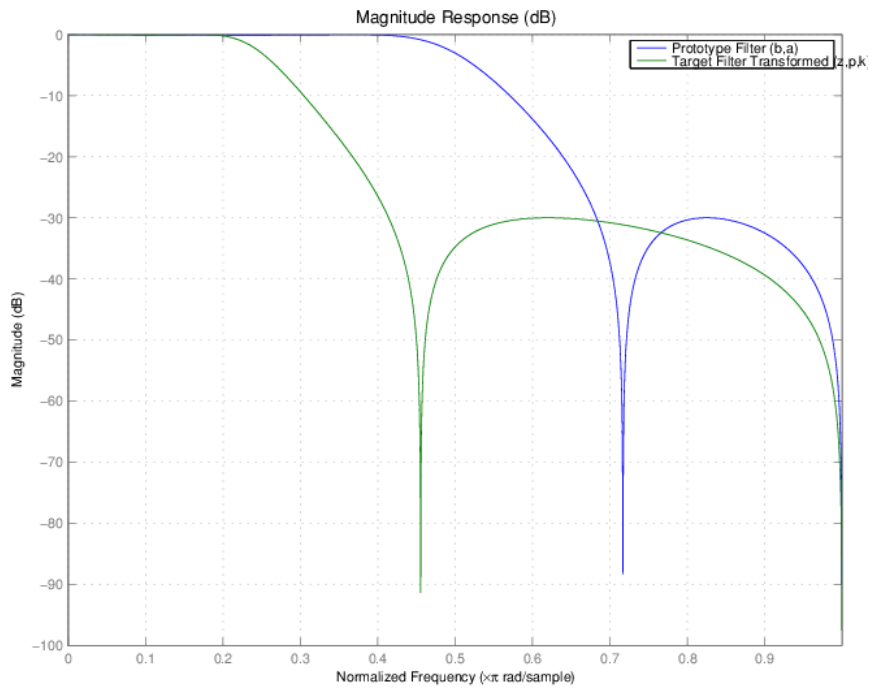
Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
z = roots(b);
p = roots(a);
k = b(1);
[z2,p2,k2] = zpklp2lp(z, p, k, 0.5, 0.25);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

Using `zpklp2lp` creates the desired half band IIR filter with the transformed features that you specify in the transformation function. This figure shows the results.



## Arguments

| Variable     | Description   |
|--------------|---|
| $Z$          | Zeros of the prototype lowpass filter                       |
| $P$          | Poles of the prototype lowpass filter                       |
| $K$          | Gain factor of the prototype lowpass filter                 |
| $\omega_0$   | Frequency value to be transformed from the prototype filter |
| $\omega_t$   | Desired frequency location in the transformed target filter |
| $Z_2$        | Zeros of the target filter                                  |
| $P_2$        | Poles of the target filter                                  |
| $K_2$        | Gain factor of the target filter                            |
| $AllpassNum$ | Numerator of the mapping filter                             |
| $AllpassDen$ | Denominator of the mapping filter                           |

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

## Version History

Introduced in R2011a

## References

Constantinides, A.G., "Spectral transformations for digital filters," *IEE Proceedings*, vol. 117, no. 8, pp. 1585-1590, August 1970.

Nowrouzian, B. and A.G. Constantinides, "Prototype reference transfer function parameters in the discrete-time frequency transformations," *Proceedings 33rd Midwest Symposium on Circuits and Systems*, Calgary, Canada, vol. 2, pp. 1078-1082, August 1990.

Nowrouzian, B. and L.T. Bruton, "Closed-form solutions for discrete-time elliptic transfer functions," *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, vol. 2, pp. 784-787, 1992.

Constantinides, A.G., "Frequency transformations for digital filters," *Electronics Letters*, vol. 3, no. 11, pp. 487-489, November 1967.

### **See Also**

[zpkftransf](#) | [allpasslp2lp](#) | [iirlp2lp](#)

## zpklp2mb

Zero-pole-gain lowpass to M-band frequency transformation

### Syntax

```
[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2mb(Z,P,K,Wo,Wt)
[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2mb(Z,P,K,Wo,Wt,Pass)
```

### Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2mb(Z,P,K,Wo,Wt)` returns zeros,  $Z_2$ , poles,  $P_2$ , and gain factor,  $K_2$ , of the target filter transformed from the real lowpass prototype by applying an Mth-order real lowpass to real multibandpass frequency mapping. By default the DC feature is kept at its original location.

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2mb(Z,P,K,Wo,Wt,Pass)` allows you to specify an additional parameter, `Pass`, which chooses between using the "DC Mobility" and the "Nyquist Mobility". In the first case the Nyquist feature stays at its original location and the DC feature is free to move. In the second case the DC feature is kept at an original frequency and the Nyquist feature is allowed to move.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with zeros,  $Z$ , poles,  $P$ , and gain factor,  $K$ .

This transformation effectively places one feature of an original filter, located at frequency  $W_o$ , at the required target frequency locations,  $W_{t1}, \dots, W_{tM}$ .

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter,  $F_1$  and  $F_2$ , with  $F_1$  preceding  $F_2$ . Feature  $F_1$  will still precede  $F_2$  after the transformation. However, the distance between  $F_1$  and  $F_2$  will not be the same before and after the transformation.

Choice of the feature subject to this transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can be easily replicated at a number of required frequency locations. A good application would be an adaptive tone cancellation circuit reacting to the changing number and location of tones.

### Examples

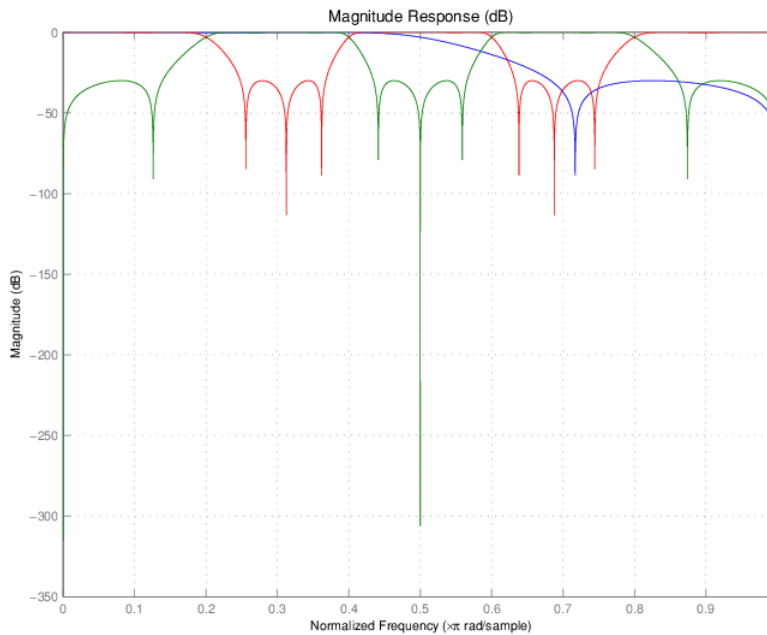
Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
z = roots(b);
p = roots(a);
k = b(1);
[z1,p1,k1] = zpklp2mb(z, p, k, 0.5, [2 4 6 8]/10, 'pass');
[z2,p2,k2] = zpklp2mb(z, p, k, 0.5, [2 4 6 8]/10, 'stop');
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k1*poly(z1), poly(p1), k2*poly(z2), poly(p2));
```

The resulting multiband filter that replicates features from the prototype appears in the figure shown. Note the accuracy of the replication process.



## Arguments

| Variable          | Description   |
|-------------------|---|
| $Z$               | Zeros of the prototype lowpass filter   |
| $P$               | Poles of the prototype lowpass filter   |
| $K$               | Gain factor of the prototype lowpass filter                                   |
| $w_0$             | Frequency value to be transformed from the prototype filter                   |
| $w_t$             | Desired frequency location in the transformed target filter                   |
| <i>Pass</i>       | Choice ('pass' / 'stop') of passband/stopband at DC, 'pass' being the default |
| $Z_2$             | Zeros of the target filter  |
| $P_2$             | Poles of the target filter  |
| $K_2$             | Gain factor of the target filter  |
| <i>AllpassNum</i> | Numerator of the mapping filter   |
| <i>AllpassDen</i> | Denominator of the mapping filter   |

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

## Version History

Introduced in R2011a

## References

Franchitti, J.C., "All-pass filter interpolation and frequency transformation problems," *MSc Thesis*, Dept. of Electrical and Computer Engineering, University of Colorado, 1985.

Feyh, G., J.C. Franchitti and C.T. Mullis, "All-pass filter interpolation and frequency transformation problem," *Proceedings 20th Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, California, pp. 164-168, November 1986.

Mullis, C.T. and R.A. Roberts, *Digital Signal Processing*, section 6.7, Reading, Massachusetts, Addison-Wesley, 1987.

Feyh, G., W.B. Jones and C.T. Mullis, *An extension of the Schur Algorithm for frequency transformations, Linear Circuits, Systems and Signal Processing: Theory and Application*, C. J. Byrnes et al Eds, Amsterdam: Elsevier, 1988.

## See Also

`zpkftransf` | `allpasslp2mb` | `iirlp2mb`

## zpklp2mbc

Zero-pole-gain lowpass to complex M-band frequency transformation

### Syntax

```
[Z2,P2,K2,AllpassNum,AllpassDen] = zpklpmbc(Z,P,K,Wo,Wt)
```

### Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpklpmbc(Z,P,K,Wo,Wt)` returns zeros,  $Z_2$ , poles,  $P_2$ , and gain factor,  $K_2$ , of the target filter transformed from the real lowpass prototype by applying an Mth-order real lowpass to complex multibandpass frequency transformation.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with zeros,  $Z$ , poles,  $P$ , and gain factor,  $K$ .

This transformation effectively places one feature of an original filter, located at frequency  $W_o$ , at the required target frequency locations,  $W_{t1}, \dots, W_{tM}$ .

Choice of the feature subject to this transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature, for example, the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter,  $F_1$  and  $F_2$ , with  $F_1$  preceding  $F_2$ . Feature  $F_1$  will still precede  $F_2$  after the transformation. However, the distance between  $F_1$  and  $F_2$  will not be the same before and after the transformation.

This transformation can also be used for transforming other types of filters; e.g., to replicate notch filters and resonators at any required location.

### Examples

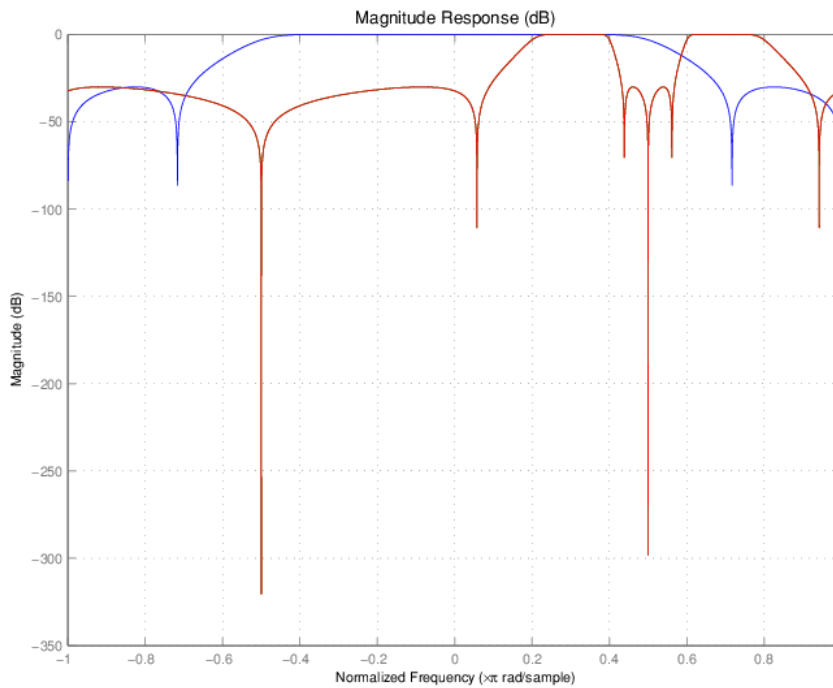
Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
z = roots(b);
p = roots(a);
k = b(1);
[z1,p1,k1] = zpklp2mbc(z, p, k, 0.5, [2 4 6 8]/10);
[z2,p2,k2] = zpklp2mbc(z, p, k, 0.5, [2 4 6 8]/10);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k1*poly(z1), poly(p1), k2*poly(z2), poly(p2));
```

You could review the coefficients to compare the filters, but the graphical comparison shown here is quicker and easier.



However, looking at the coefficients in FVTool shows the complex nature desired.

## Arguments

| Variable     | Description   |
|--------------|---|
| $Z$          | Zeros of the prototype lowpass filter   |
| $P$          | Poles of the prototype lowpass filter   |
| $K$          | Gain factor of the prototype lowpass filter   |
| $W_0$        | Frequency value to be transformed from the prototype filter. It should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.     |
| $W_t$        | Desired frequency locations in the transformed target filter. They should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate. |
| $Z_2$        | Zeros of the target filter  |
| $P_2$        | Poles of the target filter  |
| $K_2$        | Gain factor of the target filter  |
| $AllpassNum$ | Numerator of the mapping filter   |
| $AllpassDen$ | Denominator of the mapping filter   |

## Version History

Introduced in R2011a



**See Also**

zpkftransf | allpasslp2mbc | iirlp2mbc

## zpklp2xc

Zero-pole-gain lowpass to complex N-point frequency transformation

### Syntax

```
[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2xc(Z,P,K,Wo,Wt)
```

### Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2xc(Z,P,K,Wo,Wt)` returns zeros,  $Z_2$ , poles,  $P_2$ , and gain factor,  $K_2$ , of the target filter transformed from the real lowpass prototype by applying an Nth-order real lowpass to complex multipoint frequency transformation.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with zeros,  $Z$ , poles,  $P$ , and gain factor,  $K$ .

Parameter  $N$  also specifies the number of replicas of the prototype filter created around the unit circle after the transformation. This transformation effectively places  $N$  features of an original filter, located at frequencies  $W_{o1}, \dots, W_{oN}$ , at the required target frequency locations,  $W_{t1}, \dots, W_{tM}$ .

Relative positions of other features of an original filter are the same in the target filter for the Nyquist mobility and are reversed for the DC mobility. For the Nyquist mobility this means that it is possible to select two features of an original filter,  $F_1$  and  $F_2$ , with  $F_1$  preceding  $F_2$ . Feature  $F_1$  will still precede  $F_2$  after the transformation. However, the distance between  $F_1$  and  $F_2$  will not be the same before and after the transformation. For DC mobility feature  $F_2$  will precede  $F_1$  after the transformation.

Choice of the feature subject to this transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones. The only condition is that the features must be selected in such a way that when creating  $N$  bands around the unit circle, there will be no band overlap.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can be easily replicated at a number of required frequency locations. A good application would be an adaptive tone cancellation circuit reacting to the changing number and location of tones.

### Examples

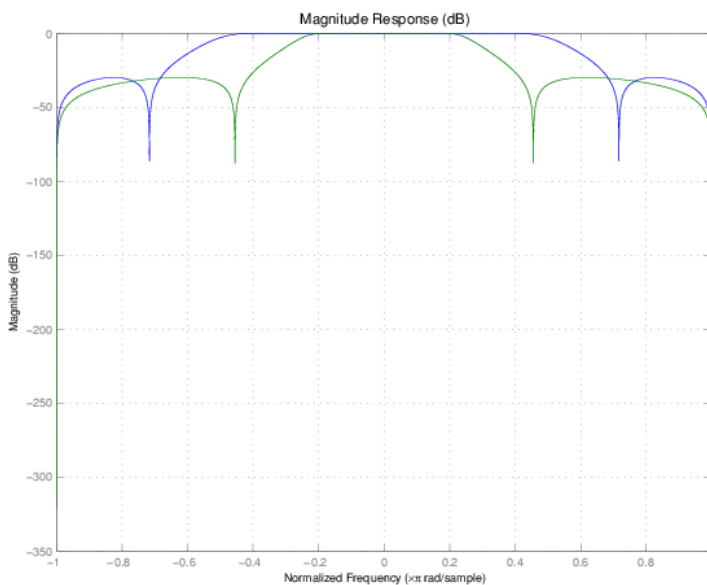
Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
z = roots(b);
p = roots(a);
k = b(1);
[z2,p2,k2] = zpklp2xc(z, p, k, [-0.5 0.5], [-0.25 0.25]);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

Plotting the filters on the same axes lets you compare the results graphically, shown here.



## Arguments

| Variable     | Description   |
|--------------|---|
| $Z$          | Zeros of the prototype lowpass filter   |
| $P$          | Poles of the prototype lowpass filter   |
| $K$          | Gain factor of the prototype lowpass filter   |
| $W_o$        | Frequency values to be transformed from the prototype filter. They should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.  |
| $W_t$        | Desired frequency locations in the transformed target filter. They should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate. |
| $Z_2$        | Zeros of the target filter  |
| $P_2$        | Poles of the target filter  |
| $K_2$        | Gain factor of the target filter  |
| $AllpassNum$ | Numerator of the mapping filter   |
| $AllpassDen$ | Denominator of the mapping filter   |

## Version History

Introduced in R2011a

## See Also

zpkftransf | allpasslp2xc | iirlp2xc

## zpklp2xn

Zero-pole-gain lowpass to N-point frequency transformation

### Syntax

```
[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2xn(Z,P,K,Wo,Wt)
[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2xn(Z,P,K,Wo,Wt,Pass)
```

### Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2xn(Z,P,K,Wo,Wt)` returns zeros,  $Z_2$ , poles,  $P_2$ , and gain factor,  $K_2$ , of the target filter transformed from the real lowpass prototype by applying an Nth-order real lowpass to real multipoint frequency transformation, where N is the number of features being mapped. By default the DC feature is kept at its original location.

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2xn(Z,P,K,Wo,Wt,Pass)` allows you to specify an additional parameter, `Pass`, which chooses between using the "DC Mobility" and the "Nyquist Mobility". In the first case the Nyquist feature stays at its original location and the DC feature is free to move. In the second case the DC feature is kept at an original frequency and the Nyquist feature is allowed to move.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with zeros,  $Z$ , poles,  $P$ , and gain factor,  $K$ .

Parameter N also specifies the number of replicas of the prototype filter created around the unit circle after the transformation. This transformation effectively places N features of an original filter, located at frequencies  $W_{o1}, \dots, W_{oN}$ , at the required target frequency locations,  $W_{t1}, \dots, W_{tM}$ .

Relative positions of other features of an original filter are the same in the target filter for the Nyquist mobility and are reversed for the DC mobility. For the Nyquist mobility this means that it is possible to select two features of an original filter,  $F_1$  and  $F_2$ , with  $F_1$  preceding  $F_2$ . Feature  $F_1$  will still precede  $F_2$  after the transformation. However, the distance between  $F_1$  and  $F_2$  will not be the same before and after the transformation. For DC mobility feature  $F_2$  will precede  $F_1$  after the transformation.

Choice of the feature subject to this transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones. The only condition is that the features must be selected in such a way that when creating N bands around the unit circle, there will be no band overlap.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can be easily replicated at a number of required frequency locations. A good application would be an adaptive tone cancellation circuit reacting to the changing number and location of tones.

### Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
z = roots(b);
```

```

p = roots(a);
k = b(1);
[z2,p2,k2] = zpklp2xn(z, p, k, [-0.5 0.5], [0 0.25], 'pass');
hfvt = fvtool(b, a, k2*poly(z2), poly(p2));
legend(hfvt, 'Original Filter', 'Half-band Filter');

```

As demonstrated by the figure, the target filter has the desired response shape and values replicated from the prototype.

## Arguments

| Variable          | Description   |
|-------------------|---|
| <i>Z</i>          | Zeros of the prototype lowpass filter   |
| <i>P</i>          | Poles of the prototype lowpass filter   |
| <i>K</i>          | Gain factor of the prototype lowpass filter                                   |
| <i>Wo</i>         | Frequency value to be transformed from the prototype filter                   |
| <i>Wt</i>         | Desired frequency location in the transformed target filter                   |
| <i>Pass</i>       | Choice ('pass' / 'stop') of passband/stopband at DC, 'pass' being the default |
| <i>Z2</i>         | Zeros of the target filter  |
| <i>P2</i>         | Poles of the target filter  |
| <i>K2</i>         | Gain factor of the target filter  |
| <i>AllpassDen</i> | Numerator of the mapping filter   |
| <i>AllpassDen</i> | Denominator of the mapping filter   |

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

## Version History

Introduced in R2011a

## References

Cain, G.D., A. Krukowski and I. Kale, "High Order Transformations for Flexible IIR Filter Design," *VII European Signal Processing Conference (EUSIPCO'94)*, vol. 3, pp. 1582-1585, Edinburgh, United Kingdom, September 1994.

Krukowski, A., G.D. Cain and I. Kale, "Custom designed high-order frequency transformations for IIR filters," *38th Midwest Symposium on Circuits and Systems (MWSCAS'95)*, Rio de Janeiro, Brazil, August 1995.

## See Also

zpkftransf | allpasslp2xn | iirlp2xn

## zpkrateup

Zero-pole-gain complex bandpass frequency transformation

### Syntax

```
[Z2,P2,K2,AllpassNum,AllpassDen] = zpkrateup(Z,P,K,N)
```

### Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpkrateup(Z,P,K,N)` returns zeros,  $Z_2$ , poles,  $P_2$ , and gain factor,  $K_2$ , of the target filter being transformed from any prototype by applying an  $N$ th-order rateup frequency transformation, where  $N$  is the upsample ratio. Transformation creates  $N$  equal replicas of the prototype filter frequency response.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The original lowpass filter is given with zeros,  $Z$ , poles,  $P$ , and gain factor,  $K$ .

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter,  $F_1$  and  $F_2$ , with  $F_1$  preceding  $F_2$ . Feature  $F_1$  will still precede  $F_2$  after the transformation. However, the distance between  $F_1$  and  $F_2$  will not be the same before and after the transformation.

### Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
z = roots(b);
p = roots(a);
k = b(1);
% Upsample the prototype filter 4 times
[z2,p2,k2] = zpkrateup(z, p, k, 4);
% Compare prototype filter with target filter
fvtool(b, a, k2*poly(z2), poly(p2));
```

### Arguments

| Variable | Description                                 |
|----------|---|
| $Z$      | Zeros of the prototype lowpass filter       |
| $P$      | Poles of the prototype lowpass filter       |
| $K$      | Gain factor of the prototype lowpass filter |
| $N$      | Integer upsampling ratio                    |
| $Z2$     | Zeros of the target filter                  |
| $P2$     | Poles of the target filter                  |
| $K2$     | Gain factor of the target filter            |

---

| Variable          | Description                       |
|-------------------|-----------------------------------|
| <i>AllpassNum</i> | Numerator of the mapping filter   |
| <i>AllpassDen</i> | Denominator of the mapping filter |

Frequencies must be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

## Version History

Introduced in R2011a

### See Also

[allpassrateup](#) | [iirrateup](#)

## zpkshift

Zero-pole-gain real shift frequency transformation

### Syntax

```
[Z2,P2,K2,AllpassNum,AllpassDen] = zpkshift(Z,P,K,Wo,Wt)
```

### Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpkshift(Z,P,K,Wo,Wt)` returns the zeros,  $Z2$ , poles,  $P2$ , and gain factor,  $K2$ , of the target filter transformed from the zeros, poles, and gain factor of real lowpass prototype by applying a second-order real shift frequency mapping. It also returns the numerator,  $AllpassNum$ , and the denominator,  $AllpassDen$  of the allpass mapping filter.

This transformation places one selected feature of an original filter, located at frequency  $W_o$ , at the required target frequency location,  $W_t$ . This transformation implements the "DC Mobility," which means that the Nyquist feature stays at Nyquist, but the DC feature moves to a location dependent on the selection of  $W_o$  and  $W_t$ .

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter,  $F_1$  and  $F_2$ , with  $F_1$  preceding  $F_2$ . Feature  $F_1$  will still precede  $F_2$  after the transformation. However, the distance between  $F_1$  and  $F_2$  will not be the same before and after the transformation.

Choice of the feature subject to the real shift transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

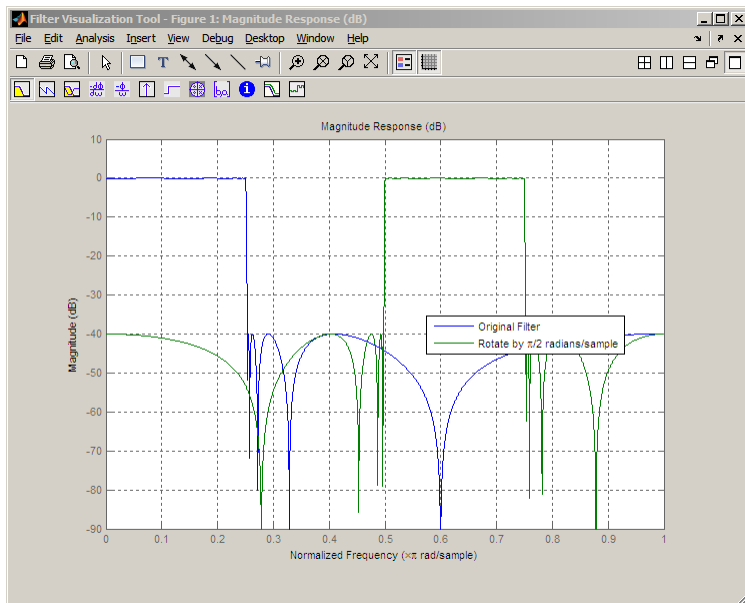
This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can change their position in a simple way without the need to design them again.

### Examples

Rotate frequency response by  $\pi/2$  radians/sample:

```
[B,A] = ellip(10,0.1,40,0.25);
% Elliptic lowpass filter with passband frequency 0.25*pi rad/sample
Z = roots(B); % get roots of numerator polynomial- filter zeros
P = roots(A); % get roots of denominator polynomial- filter poles
K = B(1);
[Z2,P2,K2] = zpkshift(Z,P,K,0.25,0.75); % shift by 0.25*pi rad/sample
Num = poly(Z2);
Den = poly(P2);
hfvtool(B,A,K2*Num,Den);
legend(hfvtool,'Original Filter','Rotate by \pi/2 radians/sample');
axis([0 1 -90 10]);
```





## Version History

Introduced in R2011a

### See Also

`zpkftransf` | `allpassshift` | `iirshift`

## zpkshifc

Zero-pole-gain complex shift frequency transformation

### Syntax

```
[Z2,P2,K2,AllpassNum,AllpassDen] = zpkshifc(Z,P,K,Wo,Wt)
[Num,Den,AllpassNum,AllpassDen] = zpkshifc(Z,P,K,0,0.5)
[Num,Den,AllpassNum,AllpassDen] = zpkshifc(Z,P,K,0,-0.5)
```

### Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpkshifc(Z,P,K,Wo,Wt)` returns zeros,  $Z_2$ , poles,  $P_2$ , and gain factor,  $K_2$ , of the target filter transformed from the real lowpass prototype by applying a first-order complex frequency shift transformation. This transformation rotates all the features of an original filter by the same amount specified by the location of the selected feature of the prototype filter, originally at  $W_o$ , placed at  $W_t$  in the target filter.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with zeros,  $Z$ , poles,  $P$ , and the gain factor,  $K$ .

`[Num,Den,AllpassNum,AllpassDen] = zpkshifc(Z,P,K,0,0.5)` performs the Hilbert transformation, i.e. a 90 degree counterclockwise rotation of an original filter in the frequency domain.

`[Num,Den,AllpassNum,AllpassDen] = zpkshifc(Z,P,K,0,-0.5)` performs the inverse Hilbert transformation, i.e. a 90 degree clockwise rotation of an original filter in the frequency domain.

### Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
z = roots(b);
p = roots(a);
k = b(1);
```

#### Rotation by $\pi/4$ Radians/Sample

Rotation by -0.25:

```
[z2,p2,k2] = zpkshifc(z, p, k, 0.5, 0.25);
fvtool(b, a, k2*poly(z2), poly(p2));
```

### Rotation by $\pi/2$ Radians/Sample

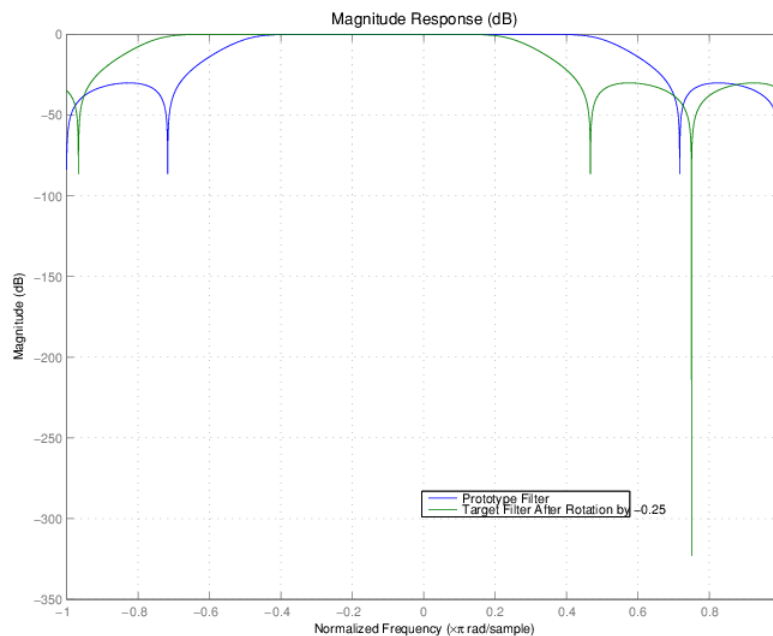
```
[z2,p2,k2] = zpkshifc(z, p, k, 0, 0.5);
fvtool(b, a, k2*poly(z2), poly(p2));
```

### Rotation by $-\pi/2$ Radians/Sample

```
[z2,p2,k2] = zpkshifc(z, p, k, 0, -0.5);
fvtool(b, a, k2*poly(z2), poly(p2));
```

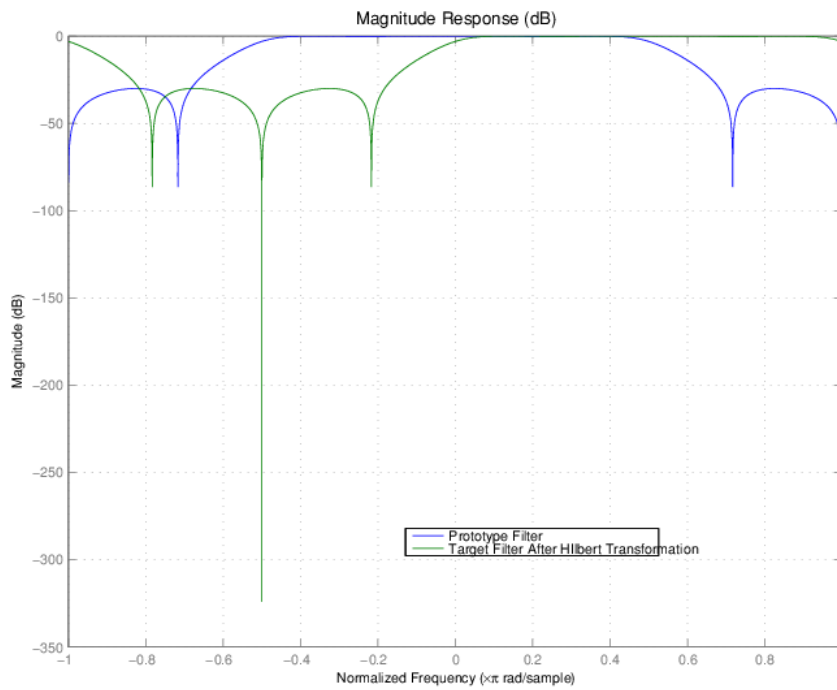
### Result of Example 1

After performing the rotation, the resulting filter shows the features desired.



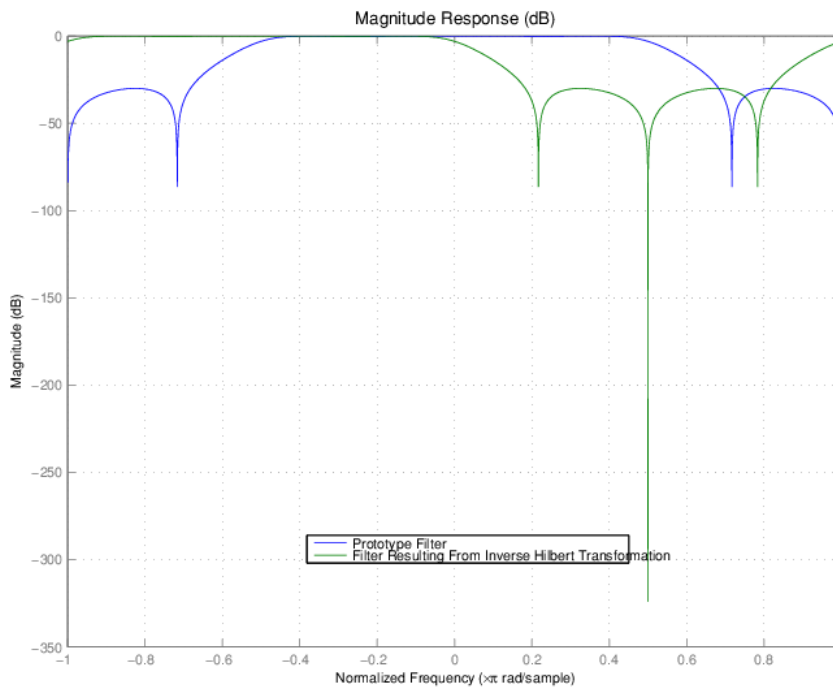
### Result of Example 2

Similar to the first example, performing the Hilbert transformation generates the desired target filter, shown here.



### Result of Example 3

Finally, using the inverse Hilbert transformation creates yet a third filter, as the figure shows.



## Arguments

| Variable     | Description   |
|--------------|---|
| $Z$          | Zeros of the prototype lowpass filter                       |
| $P$          | Poles of the prototype lowpass filter                       |
| $K$          | Gain factor of the prototype lowpass filter                 |
| $\omega_0$   | Frequency value to be transformed from the prototype filter |
| $\omega_t$   | Desired frequency location in the transformed target filter |
| $Z_2$        | Zeros of the target filter                                  |
| $P_2$        | Poles of the target filter                                  |
| $K_2$        | Gain factor of the target filter                            |
| $AllpassDen$ | Numerator of the mapping filter                             |
| $AllpassDen$ | Denominator of the mapping filter                           |

Frequencies must be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

## Version History

Introduced in R2011a

## References

Oppenheim, A.V., R.W. Schaffer and J.R. Buck, *Discrete-Time Signal Processing*, Prentice-Hall International Inc., 1989.

Dutta-Roy, S.C. and B. Kumar, "On digital differentiators, Hilbert transformers, and half-band low-pass filters," *IEEE Transactions on Education*, vol. 32, pp. 314-318, August 1989.

**See Also**

`zpkftransf` | `allpassshiftc` | `iirshiftc`

# zplane

**Package:** dsp

Z-plane zero-pole plot for discrete-time filter System object

## Syntax

```
zplane(sysobj)
zplane(sysobj,'Arithmetic',arithType)
[z,p,k] = zplane(sysobj)
```

## Description

`zplane(sysobj)` plots the zeros and poles of the filter System object, `sysobj`, with the unit circle for reference in the Filter Visualization Tool (FVTool). Each zero is represented with an 'o' and each pole with a 'x' on the plot. Multiple zeros and poles are indicated by the multiplicity number shown at the upper right of the zero or pole.

When you run the filter System object with a fixed-point input, the filter becomes a quantized fixed-point filter, `filtQuant`. When `filtQuant` is a quantized filter, `zplane(filtQuant)` plots the poles and zeros of the quantized and unquantized filters. The symbols  $\square$  and  $+$  represent the zeros and poles of the quantized filter `filtQuant`. The plot includes the unit circle for reference.

`zplane(sysobj,'Arithmetic',arithType)` also analyzes the filter System object, `sysobj`, based on the arithmetic specified in `arithType`.

`[z,p,k] = zplane(sysobj)` returns the zeros, poles, and the gain of the filter.

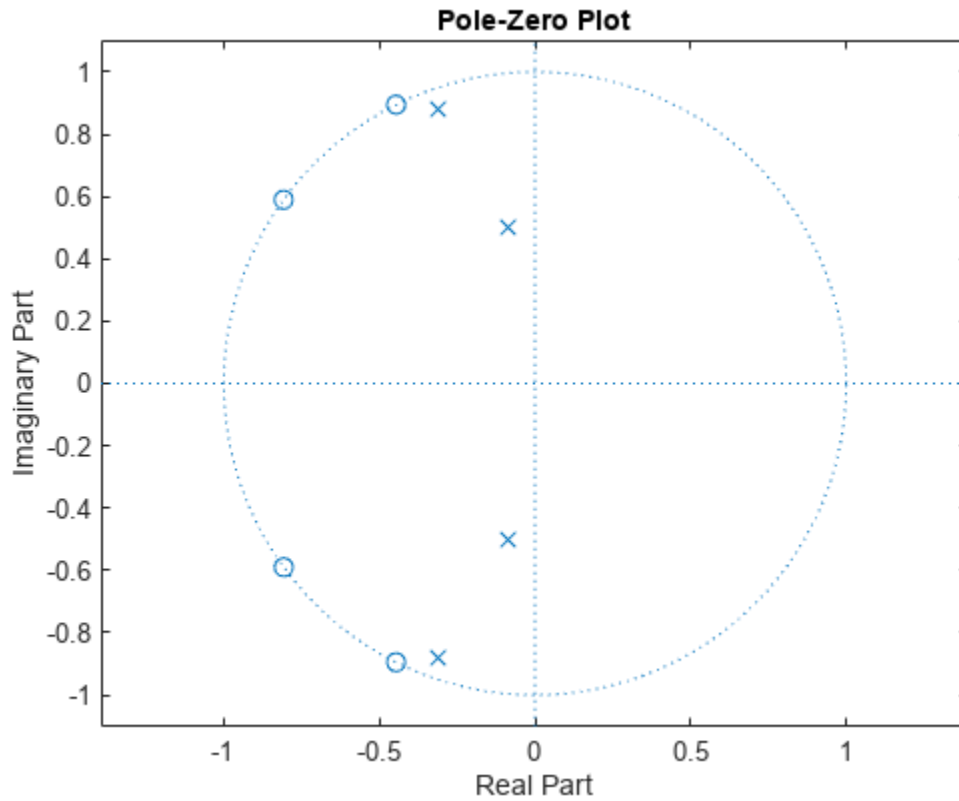
For more input options, see `zplane` in Signal Processing Toolbox.

## Examples

### Plot Poles and Zeros of Fourth-Order Filter

Create a fourth-order IIR digital filter with a cutoff frequency of 0.6. Plot the poles and zeros of this filter.

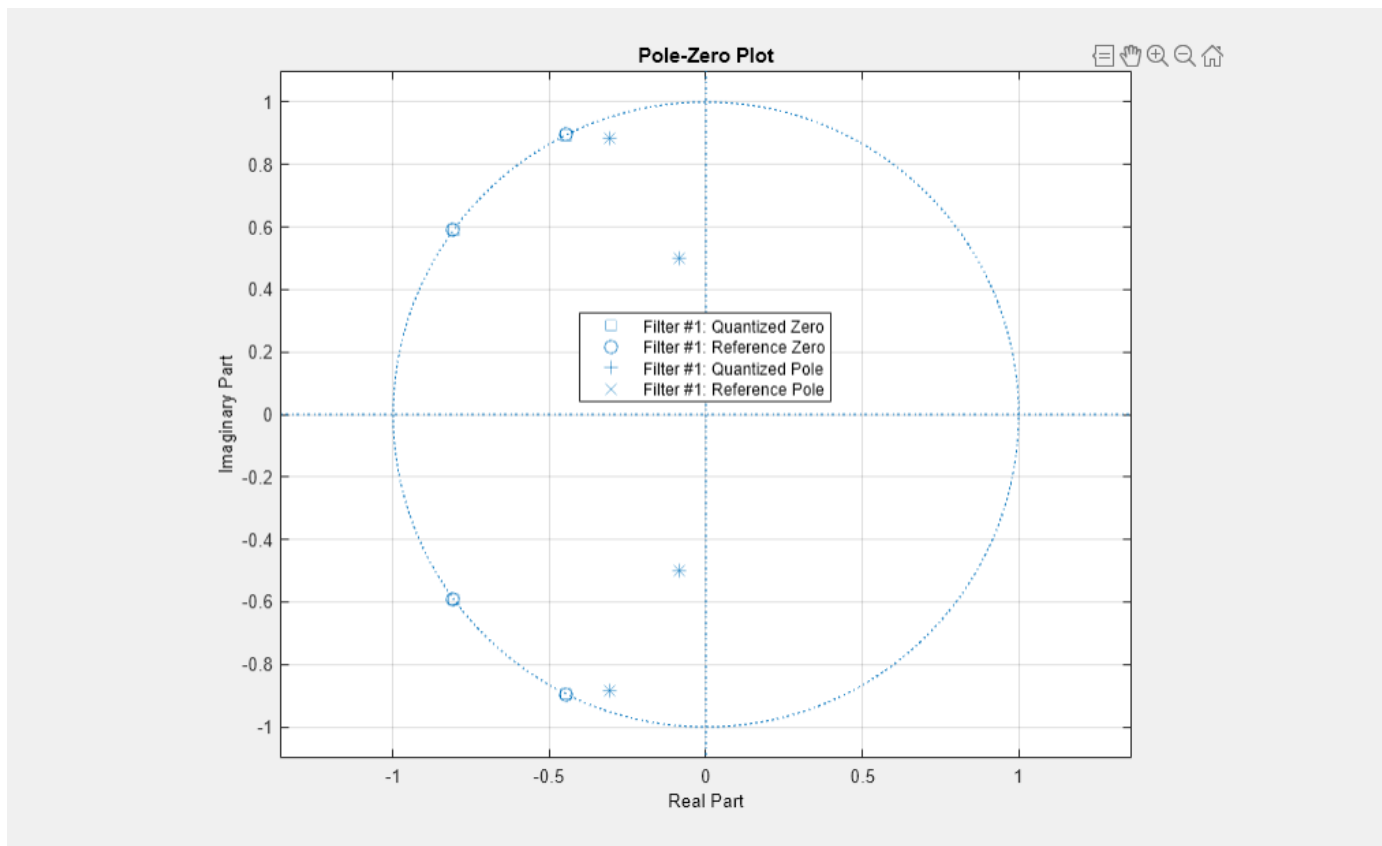
```
[b,a] = ellip(4,.5,20,.6);
zplane(b,a)
```



Quantize the filter by passing a fixed-point input through the filter algorithm. Plot the quantized and unquantized poles and zeros associated with this filter.

```
iirFilt = dsp.IIRFilter('Numerator',b,'Denominator',a);  
in = fi(randn(15,6),1,15,3);  
out = iirFilt(in);  
zplane(iirFilt)
```





## Input Arguments

### sysobj — Input filter

filter System object

Input filter, specified as one of the following filter System objects:

- `dsp.AllpassFilter`
- `dsp.AllpoleFilter`
- `dsp.BiquadFilter`
- `dsp.Channelizer`
- `dsp.CICCompensationDecimator`
- `dsp.CICCompensationInterpolator`
- `dsp.CICDecimator`
- `dsp.CICInterpolator`
- `dsp.CoupledAllpassFilter`
- `dsp.Differentiator`
- `dsp.FarrowRateConverter`
- `dsp.FilterCascade`

- `dsp.FIRDecimator`
- `dsp.FIRFilter`
- `dsp.FIRHalfbandDecimator`
- `dsp.FIRHalfbandInterpolator`
- `dsp.FIRInterpolator`
- `dsp.FIRRateConverter`
- `dsp.FourthOrderSectionFilter`
- `dsp.HighpassFilter`
- `dsp.IIRFilter`
- `dsp.IIRHalfbandDecimator`
- `dsp.IIRHalfbandInterpolator`
- `dsp.LowpassFilter`
- `dsp.NotchPeakFilter`
- `dsp.SOSFilter`
- `dsp.VariableBandwidthFIRFilter`
- `dsp.VariableBandwidthIIRFilter`

**arithType — Arithmetic type**

`'double'` (default) | `'single'` | `'Fixed'`

Arithmetic used in the filter analysis, specified as `'double'`, `'single'`, or `'Fixed'`. When the arithmetic input is not specified and the filter System object is unlocked, the analysis tool assumes a double-precision filter. When the arithmetic input is not specified and the System object is locked, the function performs the analysis based on the data type of the locked input.

The `'Fixed'` value applies to filter System objects with fixed-point properties only.

When the `'Arithmetic'` input argument is specified as `'Fixed'` and the filter object has the data type of the coefficients set to `'Same word length as input'`, the arithmetic analysis depends on whether the System object is unlocked or locked.

- `unlocked` -- The analysis object function cannot determine the coefficients data type. The function assumes that the coefficients data type is signed, has a 16-bit word length, and is auto scaled. The function performs fixed-point analysis based on this assumption.
- `locked` -- When the input data type is `'double'` or `'single'`, the analysis object function cannot determine the coefficients data type. The function assumes that the data type of the coefficients is signed, has a 16-bit word length, and is auto scaled. The function performs fixed-point analysis based on this assumption.

To check if the System object is locked or unlocked, use the `isLocked` function.

When the arithmetic input is specified as `'Fixed'` and the filter object has the data type of the coefficients set to a custom numeric type, the object function performs fixed-point analysis based on the custom numeric data type.

## Output Arguments

### **z — Location of zeros**

vector

Location of zeros of the filter transfer function on the zplane, returned as a column vector. The zeros are the roots of the polynomial whose coefficients are given by the numerator coefficients of the filter System object.

In case there are no zeros, the vector `z` is set to an empty matrix `[]`.

Data Types: `double`

### **p — Location of poles**

vector

Location of poles of the filter transfer function on the zplane, returned as a column vector. The poles are the roots of the polynomial whose coefficients are given by the denominator coefficients of the filter System object.

In case there are no poles, the vector `p` is set to an empty matrix `[]`.

Data Types: `double`

### **k — Gain of filter**

real scalar

Gain of the filter, returned as a real scalar.

Data Types: `double`

## Version History

Introduced in R2011a

## See Also

### Functions

`freqz` | `impz` | `zplane`



# Reference for the Properties of Filter Objects

---

## Multirate Filter Properties

### In this section...

“Compatibility” on page 6-2

“Property Summaries” on page 6-2

“Property Details for Multirate Filter Properties” on page 6-5

“References for Multirate Filters” on page 6-12

### Compatibility

`mfilt` objects will be removed in a future release. Refer to the reference page for a specific `mfilt` object to see its recommended replacement.

### Property Summaries

The following table summarizes the multirate filter properties and provides a brief description of each. Full descriptions of each property are given in the subsequent section.

| Name              | Values   | Default       | Description  |
|-------------------|--|---------------|--|
| BlockLength       | Positive integers  | 100           | Length of each block of data input to the FFT used in the filtering. <code>fftfirinterp</code> multirate filters include this property.  |
| DecimationFactor  | Any positive integer   | 2             | Amount to reduce the input sampling rate.  |
| DifferentialDelay | Any integer  | 1             | Sets the differential delay for the filter. Usually a value of one or two is appropriate.  |
| FilterInternals   | FullPrecision,<br>MinWordLengths,<br>SpecifyWordLengths,<br>SpecifyPrecision | FullPrecision | Controls whether the filter sets the output word and fraction lengths, and the accumulator word and fraction lengths automatically to maintain the best precision results during filtering. The default value, <code>FullPrecision</code> , sets automatic word and fraction length determination by the filter. <code>SpecifyPrecision</code> exposes the output and accumulator related properties so you can set your own word and fraction lengths for them. |

| Name                | Values                 | Default           | Description   |
|---------------------|------------------------|-------------------|---|
| FilterStructure     | mfilt structure        | None              | Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering — gains, delays, sums, products, and input/output. You cannot set this property — it is always read only and results from your choice of mfilt object.   |
| InputOffset         | Integers               | 0                 | Contains the number of input data samples processed without generating an output sample.<br>InputOffset = $\text{mod}(\text{length}(nx), m)$ where $nx$ is the number of input samples that have been processed so far and $m$ is the decimation factor.  |
| InterpolationFactor | Positive integers      | 2                 | Interpolation factor for the filter. $l$ specifies the amount to increase the input sampling rate.  |
| NumberOfSections    | Any positive integer   | 2                 | Number of sections used in the decimator, or in the comb and integrator portions of CIC filters.  |
| Numerator           | Array of double values | No default values | Vector containing the coefficients of the FIR lowpass filter used for interpolation.  |
| OverflowMode        | saturate, [wrap]       | wrap              | Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision. |

| Name               | Values   | Default               | Description   |
|--------------------|--|-----------------------|---|
| PolyphaseAccum     | Values depend on filter type. Either double, single, or fixed-point object | 0                     | Stores the value remaining in the accumulator after the filter processes the last input sample. The stored value for PolyphaseAccum affects the next output when PersistentMemory is true and InputOffset is not equal to 0. Always provides full precision values. Compare the AccumWordLength and AccumFracLength.  |
| PersistentMemory   | false or true  | false                 | Determines whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. PersistentMemory returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. |
| RateChangeFactors  | [l,m]  | [2,3] or [3,2]        | Reports the decimation (m) and interpolation (l) factors for the filter object. Combining these factors results in the final rate change for the signal. The default changes depending on whether the filter decimates or interpolates.   |
| States             | Any m+1-by-n matrix of double values                                       | 2-by-2 matrix, int32  | Stored conditions for the filter, including values for the integrator and comb sections. n is the number of filter sections and m is the differential delay. Stored in a filtstates object.   |
| SpecifyWordLengths | Vector of integers   | [16 16 16 16]<br>bits |   |



| Name                 | Values                                  | Default | Description   |
|----------------------|---|---------|---|
| WordLengthPerSection | Any integer or a vector of length $2*n$ | 16      | Defines the word length used in each section while accumulating the data in the integrator sections or while subtracting the data during the comb sections (using 'wrap' arithmetic). Enter <code>WordLengthPerSection</code> as a scalar or vector of length $2*n$ , where $n$ is the number of sections. When <code>WordLengthPerSection</code> is a scalar, the scalar value is applied to each filter section. The default is 16 for each section in the decimator. |

The following sections provide details about the properties that govern the way multirate filter work. Creating any multirate filter object puts in place a number of these properties. The following pages list the `mfilt` object properties in alphabetical order.

## Property Details for Multirate Filter Properties

### BitsPerSection

Any integer or a vector of length  $2*n$ .

Defines the bits per section used while accumulating the data in the integrator sections or while subtracting the data during the comb sections (using `wrap` arithmetic). Enter `bps` as a scalar or vector of length  $2*n$ , where  $n$  is the number of sections. When `bps` is a scalar, the scalar value is applied to each filter section. The default is 16 for each section in the decimator.

### BlockLength

Length of each block of input data used in the filtering.

`mfilt.fftfirinterp` objects process data in blocks whose length is determined by the value you set for the `BlockLength` property. By default the property value is 100. When you set the `BlockLength` value, try choosing a value so that  $[\text{BlockLength} + \text{length}(\text{filter order})]$  is a power of two.

Larger block lengths generally reduce the computation time.

### DecimationFactor

Decimation factor for the filter. `m` specifies the amount to reduce the sampling rate of the input signal. It must be an integer. You can enter any integer value. The default value is 2.

### DifferentialDelay

Sets the differential delay for the filter. Usually a value of one or two is appropriate. While you can set any value, the default is one and the maximum is usually two.

## FilterInternals

Similar to the FilterInternals pane in **Filter Designer**, this property controls whether the filter sets the output word and fraction lengths automatically, and the accumulator word and fraction lengths automatically as well, to maintain the best precision results during filtering. The default value, `FullPrecision`, sets automatic word and fraction length determination by the filter. Setting `FilterInternals` to `SpecifyPrecision` exposes the output and accumulator related properties so you can set your own word and fraction lengths for them.

### About FilterInternals Mode

There are four usage modes for this that you set using the `FilterInternals` property in multirate filters.

- `FullPrecision` — All word and fraction lengths set to  $B_{max} + 1$ , called  $B_{accum}$  by Fred Harris in [2]. Full precision is the default setting.
- `MinWordLengths` — Minimum Word Lengths
- `SpecifyWordLengths` — Specify Word Lengths
- `SpecifyPrecision` — Specify Precision

### Full Precision

In full precision mode, the word lengths of all sections and the output are set to  $B_{accum}$  as defined by

$$B_{accum} = \text{ceil}(N_{secs}(\text{Log}_2(D \times M)) + \text{InputWordLength})$$

where  $N_{secs}$  is the number of filter sections.

Section fraction lengths and the fraction length of the output are set to the input fraction length.

Here is the display looks for this mode.

```
FilterStructure: 'Cascaded Integrator-Comb Decimator'  
Arithmetic: 'fixed'  
DifferentialDelay: 1  
NumberOfSections: 2  
DecimationFactor: 4  
PersistentMemory: false
```

```
InputWordLength: 16  
InputFracLength: 15
```

```
FilterInternals: 'FullPrecision'
```

### Minimum Word Lengths

In minimum word length mode, you control the output word length explicitly. When the output word length is less than  $B_{accum}$ , roundoff noise is introduced at the output of the filter. Hogenauer's bit pruning theory (refer to [3] in the following References section) states that one valid design criterion is to make the word lengths of the different sections of the filter smaller than  $B_{accum}$  as well, so that the roundoff noise introduced by all sections does not exceed the roundoff noise introduced at the output.

In this mode, the design calculates the word lengths of each section to meet the Hogenauer criterion. The algorithm subtracts the number of bits computed using eq. 21 in Hogenauer's paper from  $B_{accum}$  to determine the word length each section.

To compute the fraction lengths of the different sections, the algorithm notes that the bits thrown out for this word length criterion are least significant bits (LSB), therefore each bit thrown out at a particular section decrements the fraction length of that section by one bit compared to the input fraction length. Setting the output word length for the filter automatically sets the output fraction length as well.

Here is the display for this mode:

```
FilterStructure: 'Cascaded Integrator-Comb Decimator'
Arithmetic: 'fixed'
DifferentialDelay: 1
NumberOfSections: 2
DecimationFactor: 4
PersistentMemory: false

InputWordLength: 16
InputFracLength: 15

FilterInternals: 'MinWordLengths'

OutputWordLength: 16
```

### Specify Word Lengths

In this mode, the design algorithm discards the LSBs, adjusting the fraction length so that unrecoverable overflow does not occur, always producing a reasonable output.

You can specify the word lengths for all sections and the output, but you cannot control the fraction lengths for those quantities.

To specify the word lengths, you enter a vector of length  $2*(NumberOfSections)$ , where each vector element represents the word length for a section. If you specify a scalar, such as  $B_{accum}$ , the full-precision output word length, the algorithm expands that scalar to a vector of the appropriate size, applying the scalar value to each section.

The CIC design does not check that the specified word lengths are monotonically decreasing. There are some cases where the word lengths are not necessarily monotonically decreasing, for example

```
hcic=mfilt.cicdecim;
hcic.FilterInternals='minwordlengths';
hcic.Outputwordlength=14;
```

which are valid CIC filters but the word lengths do not decrease monotonically across the sections.

Here is the display looks like for the SpecifyWordLengths mode.

```
FilterStructure: 'Cascaded Integrator-Comb Decimator'
Arithmetic: 'fixed'
DifferentialDelay: 1
NumberOfSections: 2
DecimationFactor: 4
PersistentMemory: false
```

```
InputWordLength: 16
InputFracLength: 15

FilterInternals: 'SpecifyWordLengths'

SectionWordLengths: [19 18 18 17]

OutputWordLength: 16
```

### Specify Precision

In this mode, you have full control over the word length and fraction lengths of all sections and the filter output.

When you elect the `SpecifyPrecision` mode, you must enter a vector of length  $2*(\text{NumberOfSections})$  with elements that represent the word length for each section. When you enter a scalar such as  $B_{accum}$ , the CIC algorithm expands that scalar to a vector of the appropriate size and applies the scalar value to each section and the output. The design does not check that this vector is monotonically decreasing.

Also, you must enter a vector of length  $2*(\text{NumberOfSections})$  with elements that represent the fraction length for each section as well. When you enter a scalar such as  $B_{accum}$ , the design applies scalar expansion as done for the word lengths.

Here is the `SpecifyPrecision` display.

```
FilterStructure: 'Cascaded Integrator-Comb Decimator'
Arithmetic: 'fixed'
DifferentialDelay: 1
NumberOfSections: 2
DecimationFactor: 4
PersistentMemory: false

InputWordLength: 16
InputFracLength: 15

FilterInternals: 'SpecifyPrecision'

SectionWordLengths: [19 18 18 17]
SectionFracLengths: [14 13 13 12]

OutputWordLength: 16
OutputFracLength: 11
```

### FilterStructure

Reports the type of filter object, such as a decimator or fractional integrator. You cannot set this property — it is always read only and results from your choice of `mfilt` object.

```
hm = mfilt.firdecim
```

```
hm =
```

```
    FilterStructure: 'Direct-Form FIR Polyphase Decimator'
           Arithmetic: 'double'
           Numerator: [1x48 double]
    DecimationFactor: 2
    PersistentMemory: false
```

## InputOffset

When you decimate signals whose length is not a multiple of the decimation factor  $M$ , the last samples —  $(nM + 1)$  to  $[(n+1)M - 1]$ , where  $n$  is an integer — are processed and used to track where the filter stopped processing input data and when to expect the next output sample. If you think of the filtering process as generating an output for a block of input data, `InputOffset` contains a count of the number of samples in the last incomplete block of input data.

---

**Note** `InputOffset` applies only when you set `PersistentMemory` to `true`. Otherwise, `InputOffset` is not available for you to use.

---

Two different cases can arise when you decimate a signal:

- 1 The input signal is a multiple of the filter decimation factor. In this case, the filter processes the input samples and generates output samples for all inputs as determined by the decimation factor. For example, processing 99 input samples with a filter that decimates by three returns 33 output samples.
- 2 The input signal is not a multiple of the decimation factor. When this occurs, the filter processes all of the input samples, generates output samples as determined by the decimation factor, and has one or more input samples that were processed but did not generate an output sample.

For example, when you filter 100 input samples with a filter which has decimation factor of 3, you get 33 output samples, and 1 sample that did not generate an output. In this case, `InputOffset` stores the value 1 after the filter run.

`InputOffset` equal to 1 indicates that, if you divide your input signal into blocks of data with length equal to your filter decimation factor, the filter processed one sample from a new (incomplete) block of data. Subsequent inputs to the filter are concatenated with this single sample to form the next block of length  $m$ .

One way to define the value stored in `InputOffset` is

```
InputOffset = mod(length(nx),m)
```

where  $nx$  is the number of input samples in the data set and  $m$  is the decimation factor.

Storing `InputOffset` in the filter allows you to stop filtering a signal at any point and start over from there, provided that the `PersistentMemory` property is set to `true`. Being able to resume filtering after stopping a signal lets you break large data sets in to smaller pieces for filtering. With `PersistentMemory` set to `true` and the `InputOffset` property in the filter, breaking a signal into sections of arbitrary length and filtering the sections is equivalent to filtering the entire signal at once.

```
xtot=[x,x];
ytot=filter(hm1,xtot)
ytot =
    0    -0.0003    0.0005   -0.0014    0.0028   -0.0054    0.0092
reset(hm1); % Clear history of the filter
hm1.PersistentMemory='true';
ysec=[filter(hm1,x) filter(hm1,x)]
ysec =
    0    -0.0003    0.0005   -0.0014    0.0028   -0.0054    0.0092
```

This test verifies that `ysec` (the signal filtered by sections) is equal to `ytot` (the entire signal filtered at once).

### **InterpolationFactor**

Amount to increase the sampling rate. Interpolation factor for the filter. It specifies the amount to increase the input sampling rate. It must be an integer. Two is the default value. You may use any positive value.

### **NumberOfSections**

Number of sections used in the multirate filter. By default multirate filters use two sections, but any positive integer works.

### **OverflowMode**

The `OverflowMode` property is specified as one of the following two character vectors indicating how to respond to overflows in fixed-point arithmetic:

- `'saturate'` — saturate overflows.

When the values of data to be quantized lie outside of the range of the largest and smallest representable numbers (as specified by the applicable word length and fraction length properties), these values are quantized to the value of either the largest or smallest representable value, depending on which is closest.

- `'wrap'` — wrap all overflows to the range of representable values.

When the values of data to be quantized lie outside of the range of the largest and smallest representable numbers (as specified by the data format properties), these values are wrapped back into that range using modular arithmetic relative to the smallest representable number. You can learn more about modular arithmetic in Fixed-Point Designer documentation.

These rules apply to the `OverflowMode` property.

- Applies to the accumulator and output data only.
- Does not apply to coefficients or input data. These always saturate the results.
- Does not apply to products. Products maintain full precision at all times. Your filters do not lose precision in the products.

### **Default value**

: `'saturate'`

---

**Note** Numbers in floating-point filters that extend beyond the dynamic range overflow to  $\pm\text{inf}$ .

---

### **PolyphaseAccum**

The idea behind `PolyphaseAccum` and `AccumWordLength/AccumFracLength` is to distinguish between the adders that always work in full precision (`PolyphaseAccum`) from the others [the adders that are controlled by the user (through `AccumWordLength` and `AccumFracLength`) and that may introduce quantization effects when you set property `FilterInternals` to `SpecifyPrecision`].

Given a product format determined by the input word and fraction lengths, and the coefficients word and fraction lengths, doing full precision accumulation means allowing enough guard bits to avoid overflows and underflows.

Property `PolyphaseAccum` stores the value that was in the accumulator the last time your filter ran out of input samples to process. The default value for `PolyphaseAccum` affects the next output only if `PersistentMemory` is `true` and `InputOffset` is not equal to 0.

`PolyphaseAccum` stores data in the format for the filter arithmetic. Double-precision filters store doubles in `PolyphaseAccum`. Single-precision filter store singles in `PolyphaseAccum`. Fixed-point filters store `fi` objects in `PolyphaseAccum`.

### PersistentMemory

Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. `PersistentMemory` returns to zero any state that the filter changes during processing. States that the filter does not change are not affected.

Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter object. `PersistentMemory` returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to `true` — the filter retains memory about filtering operations from one to the next. Maintaining memory lets you filter large data sets as collections of smaller subsets and get the same result.

```
xtot=[x,x];
ytot=filter(hm1,xtot)
ytot =

    0    -0.0003    0.0005   -0.0014    0.0028   -0.0054    0.0092
reset(hm1); % Clear history of the filter
hm1.PersistentMemory='true';
ysec=[filter(hm1,x) filter(hm1,x)]

ysec =

    0    -0.0003    0.0005   -0.0014    0.0028   -0.0054    0.0092
```

This test verifies that `ysec` (the signal filtered by sections) is equal to `ytot` (the entire signal filtered at once).

### RateChangeFactors

Reports the decimation (`m`) and interpolation (`l`) factors for the filter object when you create fractional integrators and decimators, although `m` and `l` are used as arguments to both decimators and integrators, applying the same meaning. Combining these factors as input arguments to the fractional decimator or integrator results in the final rate change for the signal.

For decimating filters, the default is [2,3]. For integrators, [3,2].

### States

Stored conditions for the filter, including values for the integrator and comb sections. `m` is the differential delay and `n` is the number of sections in the filter.

### About the States of Multirate Filters

In the `states` property you find the states for both the integrator and comb portions of the filter, stored in a `filtstates` object. `states` is a matrix of dimensions  $m+1$ -by- $n$ , with the states in CIC filters apportioned as follows:

- States for the integrator portion of the filter are stored in the first row of the state matrix.
- States for the comb portion fill the remaining rows in the state matrix.

In the state matrix, state values are specified and stored in `double` format.

`States` stores conditions for the delays between each interpolator phase, the filter states, and the states at the output of each phase in the filter, including values for the interpolator and comb states.

The number of states is  $(lh-1)*m+(l-1)*(lo+mo)$  where  $lh$  is the length of each subfilter, and  $l$  and  $m$  are the interpolation and decimation factors.  $lo$  and  $mo$ , the input and output delays between each interpolation phase, are integers from Euclid's theorem such that  $lo*l-mo*m = -1$  (refer to the reference for more details). Use `euclidfactors` to get  $lo$  and  $mo$  for an `mfilt.firfracdecim` object.

`States` defaults to a vector of zeros that has length equal to `nstates(hm)`

### References for Multirate Filters

- [1] Fliege, N.J., *Multirate Digital Signal Processing*, John Wiley and Sons, 1994.
- [2] Harris, Fredric J, *Multirate Signal Processing for Communication Systems*, Prentice Hall PTR, 2004.
- [3] Hogenauer, E. B., "An Economical Class of Digital Filters for Decimation and Interpolation," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. ASSP-29, No. 2, April 1981, pp. 155-162.
- [4] Lyons, Richard G., *Understanding Digital Signal Processing*, Prentice Hall PTR, 2004
- [5] Mitra, S.K., *Digital Signal Processing*, McGraw-Hill, 1998.
- [6] Orfanidis, S.J., *Introduction to Signal Processing*, Prentice-Hall, Inc., 1996.